

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2011/2012



PRACA DYPLOMOWA INŻYNIERSKA

Łukasz Szymon Sowa

Nowoczesne mechanizmy bezpieczeństwa w systemie Linux

Opiekun pracy:
dr inż. Tomasz Jordan Kruk

Ocena

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego



Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1989.02.06

Data rozpoczęcia studiów: 2008.10.01

Życiorys

Urodziłem się 6 lutego 1989 roku w Warszawie. W 2008 roku ukończyłem Społeczne Liceum Ogólnokształcące nr 5 w Milanówku. W tym samym roku rozpocząłem studia stacjonarne pierwszego stopnia na Wydziale Elektroniki i Technik Informatycznych Politechniki Warszawskiej na kierunku Informatyka. Przez pięć semestrów studiów otrzymywałem stypendium naukowe za bardzo dobre wyniki w nauce.

.....
Podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dniu

z wynikiem

Ogólny wynik studiów:

Dodatkowe wnioski i uwagi Komisji:

.....

.....

Streszczenie

W pracy przedstawiono przegląd nowoczesnych mechanizmów bezpieczeństwa systemu Linux na przykładzie dystrybucji Google Chrome OS. Szczególną uwagę poświęcono technikom kontroli dostępu do wywołań systemowych. Przeanalizowano korzyści płynące z zastosowania zabezpieczeń tego typu, a także omówiono istniejące rozwiązania dla systemu Linux. Zasadniczą częścią pracy jest opis procesu powstawania nowatorskiego mechanizmu kontroli wywołań systemowych, który stanowił praktyczną część pracowni dyplomowej. W tym fragmencie skupiono się na zaprezentowaniu specyfiki programowania w przestrzeni jądra, a także na szczegółowym przedstawieniu przebiegu prac.

Słowa kluczowe: *Linux, bezpieczeństwo, wywołania systemowe, ABI, programowanie w przestrzeni jądra, Chrome OS, kontenery.*

Modern security mechanisms in Linux

The following thesis presents an overview of modern security mechanisms in Linux, the case study of which is Google Chrome OS. Particular attention is paid to methods of system call interposition — benefits of their usage and existing implementations for Linux. The fundamental part of this paper is the description of development process of innovative system call interposition mechanism created as a practical assignment of the thesis. This part focuses on the peculiarity of kernel programming as well as detailed presentation of the work course.

Keywords: *Linux, security, system calls, ABI, system call interposition, kernel programming, Chrome OS, containers.*

Spis treści

Wstęp	1
1. Mechanizmy bezpieczeństwa na przykładzie systemu Google Chrome OS	3
1.1. Cele i motywacje	3
1.2. Mechanizmy zewnętrzne	4
1.2.1. Weryfikacja integralności i autentyczności — Verified Boot	4
1.2.2. Odzyskiwanie systemu — System Recovery	7
1.3. Mechanizmy wewnętrzne	10
1.3.1. Ochrona rootfs	10
1.3.2. Ochrona danych użytkownika — eCryptfs	11
1.3.3. Ograniczanie uprawnień — minijail	12
1.3.4. Ograniczanie zasobów — Control Groups	15
1.3.5. Kontrola dostępu — Medium Access Control	15
1.3.6. Systemowe API — przeglądarka Google Chrome	16
1.3.7. Usługa aktualizacji — Autoupdate	17
1.4. Podsumowanie i ocena rozwiązań	17
2. Kontrola dostępu do systemowego ABI	19
2.1. Korzyści	19
2.2. Zastosowania	21
2.2.1. Kontenery procesów	21
2.2.2. Serwery	21
2.2.3. Obliczenia rozproszone	22
2.2.4. Aplikacje uprzywilejowane	22
2.2.5. Sandboxing	23
2.3. Istniejące rozwiązania	23
2.3.1. ptrace	24
2.3.2. systrace	25
2.3.3. Capabilities	25
2.3.4. seccomp	26
2.3.5. Audit syscall	26
2.3.6. LD_PRELOAD	27
2.3.7. Chromium sandbox	27
2.3.8. Patch out	29
2.4. Podsumowanie i wnioski	29
3. Projekt	31
3.1. Wymagania	31
3.1.1. Wymagania funkcjonalne	32
3.1.2. Wymagania нефункционалне	33
3.2. Decyzje projektowe	34
3.2.1. Architektura mechanizmu	34
3.2.2. Model modułu	36
3.2.3. Sprawdzanie polityki	37
3.2.4. Technika realizacji modułu	39
3.2.5. Podsumowanie podjętych decyzji	41
4. Implementacja	43

4.1. Control Groups	43
4.1.1. Budowa	44
4.1.2. Implementacja podsystemu — API	45
4.2. Programowanie w przestrzeni jądra	48
4.2.1. Optymalizacja	49
4.2.2. Zarządzanie pamięcią	51
4.2.3. Programowanie defensywne	52
4.2.4. Synchronizacja	52
4.2.5. Wykorzystanie API	56
4.3. Wykorzystane narzędzia	57
4.3.1. Tworzenie kodu	57
4.3.2. Kompilacja	58
4.3.3. Uruchamianie	58
4.3.4. Debugowanie	59
4.3.5. Wstępne testowanie	60
4.4. Instrukcja użytkownika	61
4.4.1. Aplikacja łąty	61
4.4.2. Interfejs użytkownika	61
5. Testowanie	63
5.1. Testy systemowe	63
5.2. Testy wydajnościowe	65
5.2.1. Mikrotesty	66
5.2.2. Makrotesty	68
5.2.3. Podsumowanie	71
5.3. Przykładowe wdrożenia	72
5.3.1. Serwer DNS	72
5.3.2. Serwer zdalnej kompilacji i uruchamiania	72
5.4. Ocena realizacji wymagań	73
5.4.1. Wymagania funkcjonalne	73
5.4.2. Wymagania нефункционалне	74
Podsumowanie	76
A. Instrukcja modułu	78
A.1. Aplikacja łąty i kompilacja jądra	78
A.2. Przykładowe użycie podsystemu	78
Bibliografia	81

Wstęp

Burzliwy rozwój informatyki spowodował, że komputery znalazły zastosowanie w praktycznie każdej dziedzinie życia — od przemysłu i biznesu aż do rozrywki. Dynamiczny postęp w teleinformatyce sprawił natomiast, że w zasadzie każdy komputer (zarówno maszyna klasy *mainframe*, jak i niewielkie urządzenie mobilne) jest węzłem pewnego rodzaju sieci — najczęściej Internetu. Dzięki tym procesom komputery stały się najważniejszym środkiem wymiany informacji i są niezwykle istotnym elementem światowej gospodarki. Negatywnym zjawiskiem społecznym rozwijającym się wraz z powszechną komputeryzacją i dostępem do Internetu jest przestępczość komputerowa, która, niezależnie od motywacji, jest realnym zagrożeniem dla użytkowników. Naturalną konsekwencją tych procesów jest nieustanny rozwój mechanizmów, metod i procedur bezpieczeństwa, które mają zapobiec wszelkim atakom. Chronienie zasobów komputerowych jest zadaniem odpowiedzialnym i trudnym, gdyż to atakujący są na pozycji uprzywilejowanej — z góry znają istniejące zabezpieczenia i w ukryciu mogą przygotowywać coraz bardziej wyrafinowane ataki. Inżynierowie zajmujący się bezpieczeństwem systemów komputerowych muszą więc nieustannie przewidywać potencjalne zagrożenia i ciągle starać się być „o krok naprzód” w stosunku do cyberprzestępców. Bezpieczeństwo jest zatem interesującą i dynamicznie rozwijającą się dziedziną informatyki, wymagającą wyobraźni oraz analitycznego myślenia.

Ochrona większości systemów informatycznych sprowadza się do zastosowania odpowiednich metod i technik zabezpieczających poszczególne jego warstwy. Analizując bezpieczeństwo takiego systemu, można rozważać np. zabezpieczenia fizycznego dostępu do maszyn, zabezpieczenia protokołu komunikacji, zabezpieczenia aplikacji dostępowej itd. Oprogramowanie, które jest częścią praktycznie każdego systemu komputerowego, to system operacyjny, pod nadzorem którego pracują pozostałe aplikacje wykonawcze. Ze względu na takie nadrzędne umiejscowienie systemu operacyjnego, jego bezpieczeństwo jest fundamentalnym składnikiem niezawodności kompletnego systemu informatycznego. Z tego powodu bardzo często pierwsze działania związane z ochroną systemu komputerowego dotyczą zabezpieczenia warstwy systemu operacyjnego. W zależności od specyfiki projektu, podjęte czynności mogą mieć bardzo różny wymiar i np. mogą sprowadzać się wyłącznie do zaktualizowania oprogramowania systemowego. Istnieje jednak bardzo wiele przedsięwzięć, które dla zapewnienia należytego poziomu bezpieczeństwa systemu operacyjnego, wymagają zastosowania wyszukanych narzędzi i technik.

W związku z powyższymi spostrzeżeniami, w niniejszej pracy podjęto temat mechanizmów bezpieczeństwa. Systemy operacyjne znacznie się od siebie różnią, z tego też powodu stworzone dla nich narzędzia są zróżnicowane. Bardzo duża liczba systemów operacyjnych uniemożliwia zatem przeprowadzenie całościowej, a przy tym wnikliwej analizy wszystkich dostępnych mechanizmów, dlatego w pracy skoncentrowano się wyłącznie na nowoczesnych środkach bezpieczeństwa przygotowanych dla systemu *Linux*. System ten nigdy nie zyskał znaczącej popularności

na stacjach roboczych, jest jednak powszechnie obecny w zastosowaniach serwerowych, wbudowanych, a ostatnio także na urządzeniach mobilnych. Wysokie bezpieczeństwo składające się na niezawodność *Linuksa* ma zatem w wielu zastosowaniach istotne znaczenie przemysłowe, czyniąc opisywaną tematykę szczególnie interesującą.

Celem pracy jest przedstawienie procesu wytwarzania oprogramowania dla *Linuksa*, w wyniku czego powstał mechanizm zwiększający bezpieczeństwo tego systemu. Aby zrealizować sformułowane zadanie, w pierwszej kolejności przeanalizowano nowoczesną dystrybucję *Google Chrome OS* pod kątem zastosowanych w niej środków ochrony. Podczas eksploracji mechanizmów bezpieczeństwa zidentyfikowano zapotrzebowanie na narzędzia służące do ochrony ABI systemu. W kolejnym kroku przeprowadzono zatem analizę dostępnych rozwiązań powyższego problemu. Wnioski wyciągnięte na podstawie przeglądu dziedziny doprowadziły do sformułowania wymagań i zaprojektowania na tej podstawie nowego mechanizmu, który następnie został zaimplementowany i przetestowany.

Układ rozdziałów niniejszej pracy w naturalny sposób odzwierciedla przebieg powyższego procesu. W rozdziale 1 zaprezentowano analizę mechanizmów bezpieczeństwa na przykładzie dystrybucji *Google Chrome OS*, poprzedzoną określeniem celów i potrzeb, które doprowadziły do zaprojektowania konkretnych rozwiązań. Kolejny rozdział koncentruje się na zagadnieniu ochrony dostępu do ABI systemu *Linux*. W rozdziale tym pokazano korzyści wynikające z zastosowania mechanizmu oferującego tego typu usługi, a następnie dokonano przeglądu istniejących projektów podejmujących ten problem. Rozdział 3 dotyczy projektu nowego mechanizmu, zbudowanego na podstawie wniosków z wykonanego przeglądu. W pierwszej jego części sformułowano wymagania projektowe, zarówno funkcjonalne jak i nie-funkcjonalne. Na tej podstawie, przy wykorzystaniu wiedzy dziedzinowej, podjęto wstępne decyzje projektowe, które umożliwiły rozpoczęcie implementacji opisanej w kolejnym rozdziale. Podczas prezentowania procesu implementacji główny nacisk położono na przedstawieniu specyfiki programowania jądra *Linuksa*. Rozdział 4 przedstawia zatem typowe zagadnienia oraz charakterystyczne narzędzia towarzyszące wytwarzaniu takiego oprogramowania. Rozdział kończy się pełną instrukcją użytkownika opisującą instalację i interfejs stworzonego mechanizmu. Ostatni rozdział dotyczy testowania gotowego rozwiązania. W tej części pracy zaprezentowano wyniki testów poprawnościowych oraz wydajnościowych kompletnego, działającego produktu. Uzupełnieniem treści rozdziału jest przedstawienie przeprowadzonych przykładowych wdrożeń mechanizmu oraz ocena realizacji poszczególnych wymagań projektowych.

1. Mechanizmy bezpieczeństwa na przykładzie systemu Google Chrome OS

W ostatnich latach, zarówno producenci sprzętu, a także oprogramowania dostrzegli, jak bardzo istotny dla ich klientów jest komfort psychiczny związany z wysokim bezpieczeństwem przetwarzanych informacji. Niestety, często zdarza się, iż próba zapewnienia odpowiedniego poziomu bezpieczeństwa danego systemu informatycznego powoduje pogorszenie ogólnej wydajności rozwiązania oraz znacząco komplikuje jego użycie. Wszystkie trzy wymienione wyżej cechy — bezpieczeństwo, wydajność oraz łatwość użycia — są istotnymi składnikami tzw. doświadczenia użytkownika (ang. *user experience*, *UX*), a umiejętne połączenie powyższych właściwości, znacząco wpływa na użyteczność (ang. *usability*) danego rozwiązania. Zaprojektowanie i wykonanie systemu informatycznego, który jest zarówno bezpieczny, jak i wygodny w użyciu, jest więc zadaniem trudnym nie tylko ze względu na skomplikowanie i istotność mechanizmów bezpieczeństwa, ale również ze względu na otoczenie, w którym owe mechanizmy działają. Stworzenie odpowiednich rozwiązań zapewniających bezpieczeństwo w tak złożonym projekcie, jakim jest system *Chrome OS* jest więc szczególnie istotne i warte dogłębnego przeanalizowania.

Treścią niniejszego rozdziału jest przegląd wyżej wspomnianych mechanizmów. W pierwszej jego części przedstawiono powody i potrzeby, które doprowadziły do zaprojektowania konkretnych rozwiązań. W drugiej części opisane zostały mechanizmy zewnętrzne — uruchamiane jeszcze przed inicjalizacją jądra systemu. Trzecia część dotyczy z kolei mechanizmów wewnętrznych — działających wewnątrz właściwego systemu operacyjnego. Rozdział kończy się krótkim podsumowaniem i oceną przeanalizowanych rozwiązań.

1.1. Cele i motywacje

Chrome OS ma być systemem używanym głównie na urządzeniach przenośnych klasy netbook, stale podłączonych do Internetu. Taki zestaw może być wykorzystywany przez jednego, wyłącznego użytkownika (np. jako dodatkowy, przenośny komputer do pracy) bądź wielu użytkowników (np. jako wypożyczany komputer w bibliotece czy kawiarni). Z takim modelem użytkowania związane jest ryzyko:

- kradzieży bądź zgubienia urządzenia,
- ataku zdalnego poprzez internet,
- ataku lokalnego poprzez podmienienie oprogramowania,
- próby nieuprawnionego dostępu do danych użytkownika.

W odpowiedzi na powyższe zagrożenia projektanci systemu *Chrome OS* postawili sobie następujące cele:

- wczesne wykrywanie naruszenia integralności systemu, co inicjuje przywrócenie oprogramowania do pierwotnego stanu,

- wprowadzenie odseparowanych kont użytkowników, których dane podlegają szyfrowaniu,
- zastosowanie wielowarstwowych, silnych, systemowych mechanizmów izolacji działających usług,
- wprowadzenie systemu automatycznych aktualizacji, w celu wyeliminowania odnalezionych błędów w oprogramowaniu.

Ponadto ze względu na docelową grupę odbiorców systemu (tzn. osoby nieposiadające wyspecjalizowanej, technicznej wiedzy), środki realizacji tych celów powinny być dla użytkownika przezroczyste oraz nie powinny widocznie zmniejszać responsywności platformy. W ramach realizacji powyższych założeń, inżynierowie systemu *Chrome OS* oprócz zaprojektowania i zaimplementowania szeregu autorskich rozwiązań, dostosowali i zintegrowali opracowane już mechanizmy bezpieczeństwa.

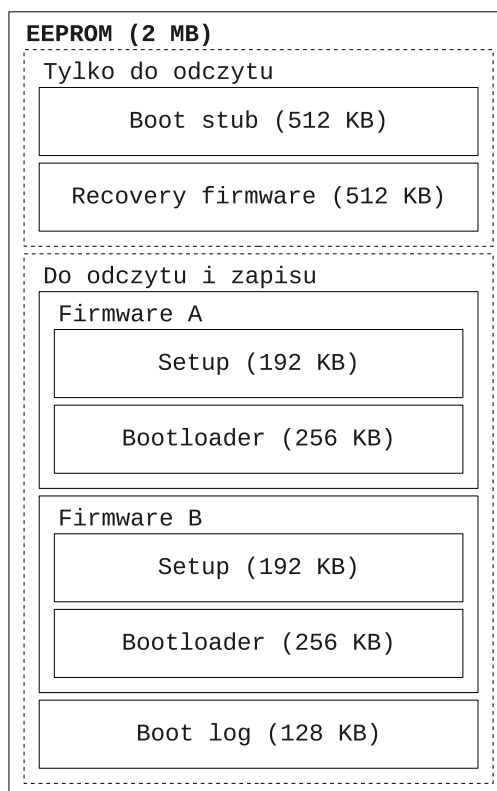
1.2. Mechanizmy zewnętrzne

Zewnętrzne mechanizmy zabezpieczeń mają na celu weryfikację autentyczności i integralności *Chrome OS*, a w razie potrzeby — przywrócenie go do pierwotnego, poprawnego stanu. Pomyślne przejście wspomnianej weryfikacji gwarantuje, iż system nie uległ uszkodzeniu (np. na skutek przerwanej aktualizacji) oraz nie został celowo zmodyfikowany bądź podmieniony (np. w celu przechwytywania danych użytkownika). Zewnętrzne mechanizmy zostały zaimplementowane pod postacią dwóch rozwiązań: *Verified Boot* — odpowiedzialnego za weryfikację autentyczności i integralności oraz *System Recovery* — odpowiedzialnego za przywrócenie systemu do poprawnego stanu.

Standardowy interfejs uruchamiania komputera PC, w postaci *x86 legacy BIOS* czy też *x86 EFI BIOS*, nie oferuje należytej weryfikacji autentyczności i integralności systemu. Wszelkie mechanizmy weryfikujące w standardowych komputerach wyposażonych w *BIOS*, można więc zaimplementować jedynie jako element rozruchowy jądra systemu. Niestety, wiąże się to z koniecznością umieszczenia kodu weryfikującego na dysku, w pamięci możliwej do modyfikacji, a więc niezaufanej. Efektywna implementacja wyżej wymienionych mechanizmów bezpieczeństwa wymaga więc modyfikacji standardu rozruchu komputera. Na potrzeby *Chrome OS* opracowano stosowne zmiany, które możliwe są do zaimplementowania przy użyciu dodatkowej pamięci typu *EEPROM* wyposażonej w wydzielony fragment tylko do odczytu. Należy dodać, iż możliwe jest uruchomienie systemu *Chromium OS* na każdym, standardowym komputerze z rodziny *x86*, jednak bez rozwiązań *Verified Boot* oraz *System Recovery*. Na rysunku 1.1 przedstawiono przykładowy układ pamięci pomocniczej. Istotnym składnikiem bezpieczeństwa *Chrome OS* jest również układ *TPM (Trusted Platform Module)*, który wspiera operacje kryptograficzne. Podobnie jednak jak dodatkowa pamięć *EEPROM* nie jest on wymagany do uruchomienia systemu.

1.2.1. Weryfikacja integralności i autentyczności — *Verified Boot*

Verified Boot gwarantuje, że system operacyjny nie uległ celowej bądź przypadkowej modyfikacji. W tym celu mechanizm *Verified Boot* złożony jest z dwóch części. Zaraz po uruchomieniu komputera rozpoczyna się pierwszy etap — wykonanie załączka programu rozruchowego (ang. *boot stub*) zawartego w obszarze pamięci chronionej przed zapisem. Jedyne zadania tego programu to inicjalizacja podstawowych



Rysunek 1.1: Przykładowy układ pomocniczej pamięci *EEPROM*. Na podstawie: [26].

podzespołów komputera, weryfikacja autentyczności i uruchomienie dalszej części firmware'u. Zrealizowanie minimalnej funkcjonalności w tej sekcji spowodowane jest tym, że program ten musi być pozbawiony błędów, ponieważ nie może być łatwo wymieniony. Autentyczność modyfikowalnego firmware'u potwierdzana jest za pomocą tzw. *root key* — oficjalnego klucza publicznego, który pozwala zweryfikować dołączony podpis cyfrowy. Klucz ten umieszczony jest w pamięci tylko do odczytu wraz z programem rozruchowy. Modyfikowalny firmware znajduje się w pamięci *EEPROM* w dwóch kopiach, na wypadek uszkodzenia, np. w procesie aktualizacji — program rozruchowy po niepomyślnej próbie uruchomienia firmware'u A podejmuje próbę uruchomienia firmware'u B.

W uproszczeniu pierwszy etap *Verified Boot* można przedstawić za pomocą następującej listy kroków (zob. rysunek 1.4):

1. Zainicjuj podstawowe peryferia komputera.
2. Jeżeli użytkownik zażądał uruchomienia *System Recovery*, pomiń ładowanie firmware'u i uruchom proces odzyskiwania systemu.
3. Jeżeli ustawiony jest znacznik odzyskiwania systemu (ang. *recovery mode cookie*), pomiń ładowanie firmware'u i uruchom proces odzyskiwania systemu.
4. Spróbuj załadować firmware A:
 - a) Załaduj firmware A do pamięci.
 - b) Zweryfikuj podpis cyfrowy firmware'u A, używając klucza głównego.
 - c) Jeśli weryfikacja zakończyła się pomyślnie, wykonaj skok do kodu firmware'u A, kończąc pierwszy etap rozruchu.
5. Ładowanie firmware'u A nie powiodło się, powtórz krok 4 dla firmware'u B.

6. Zarówno firmware A, jak i B są uszkodzone, uruchom proces odzyskiwania systemu.

Drugi etap procesu *Verified Boot* zaczyna się bezpośrednio w momencie rozpoczęcia wykonywania kodu modyfikowalnego firmware'u. Pierwsza czynność, którą trzeba wykonać w tej fazie uruchamiania systemu, to inicjalizacja i konfiguracja minimalnego zestawu peryferiów potrzebnych do załadowania systemu z urządzenia rozruchowego (ang. *boot device*) — np. kontrolerów SATA lub eMMC. Po pomyślnym zainicjowaniu urządzeń następuje skok do kodu programu rozruchowego. Początkowym zadaniem *bootloadera* jest odnalezienie pierwszego urządzenia rozruchowego, które powinno zawierać tablicę partycji zgodną z *GPT* (*GUID Partition Table*). Na rysunku 1.2 przedstawiono przykładowy układ danych i partycji na nośniku rozruchowym. Kolejny krok rozruchu systemu, to weryfikacja nagłówka *GPT*. Ze względu na dużą zmienność tej sekcji danych nie jest możliwe złożenie na niej cyfrowego podpisu, a więc program rozruchowy jedynie sprawdza sumę kontrolną *CRC32* nagłówka. Po pomyślnej weryfikacji integralności *bootloader* poszukuje partycji oznaczonych specjalnym *GUID* zawierających jądro *Chrome OS* oraz końcowy program rozruchowy w formacie *EFI*. Urządzenie rozruchowe zawiera dwie takie partycje, na wypadek gdyby jedna z nich została uszkodzona. Po wybraniu poprawnej partycji weryfikowana jest jej autentyczność poprzez sprawdzenie dołączonego podpisu cyfrowego za pomocą klucza publicznego zawartego w obszarze modyfikowalnego firmware'u. Pomyślne sprawdzanie autentyczności pozwala na załadowanie zawartości partycji do pamięci *RAM* i wykonanie końcowego programu rozruchowego, który przygotowuje podstawowe struktury danych potrzebne do inicjalizacji systemu, a następnie wykonuje skok do kodu jądra. Zaraz po tym zewnętrzne mechanizmy ochrony kończą swoje działanie, a odpowiedzialność zapewnienia bezpieczeństwa przechodzi na jądro systemu *Chrome OS*.

Schemat przebiegu drugiego etapu mechanizmu *Verified Boot* przedstawia następująca lista kroków (zob. rysunek 1.4):

1. Zainicjuj urządzenia potrzebne do wczytania jądra systemu z nośnika trwałego.
2. Wykonaj skok do kodu programu rozruchowego.
3. Sprawdź integralność tablicy partycji.
4. Odnajdź partycję z obrazem jądra.
5. Załaduj partycję do pamięci *RAM*.
6. Zweryfikuj podpis cyfrowy zawartości partycji.
7. Jeśli weryfikacja nie powiodła się:
 - a) Jeśli jest to obraz jądra A, spróbuj zweryfikować podpis partycji z obrazem jądra B.
 - b) Jeśli jest to obraz jądra B, tzn. oba jądra są niepoprawne, ustaw znacznik odzyskiwania systemu i uruchom komputer ponownie.
8. Jeśli jądro jest podpisane kluczem publicznym nieznanym *bootloaderowi*, przejdź do trybu *Developer Mode* — tryb ten pozwala na używanie innego systemu operacyjnego, niż *Chrome OS*.
9. Wykonaj skok do kodu końcowego programu rozruchowego.
10. Zainicjuj struktury niezbędne do inicjalizacji jądra systemu.
11. Wykonaj skok do kodu jądra.



Rysunek 1.2: Przykładowy układ danych na rozruchowym nośniku trwałym. Na podstawie: [26].

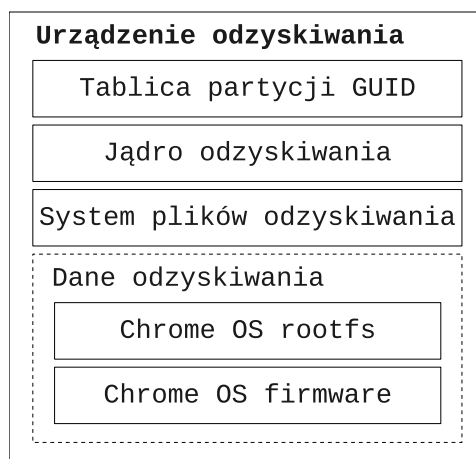
1.2.2. Odzyskiwanie systemu — System Recovery

System Recovery to mechanizm, który ma za zadanie możliwie łatwo i bezpiecznie przywrócić system do poprawnego stanu. Dodatkowo *System Recovery* umożliwia zainstalowanie alternatywnego systemu operacyjnego (na komputerze przeznaczonym dla *Chrome OS*). *System Recovery* jest uruchamiany przez załazek programu rozruchowego w dwóch sytuacjach: gdy został ustawiony znacznik odzyskiwania systemu na którymś z etapów rozruchu — została wykryta nieprawidłowość w strukturze danych lub kiedy użytkownik jawnie tego zażądał — został naciśnięty przycisk odzyskiwania umieszczony na obudowie komputera. Kod programu *System Recovery* jest zapisany w pamięci tylko do odczytu, co uniemożliwia jego podmianę bądź przypadkowe uszkodzenie. Każde uruchomienie *System Recovery* powoduje w pierwszej kolejności odnotowywanie tego faktu w dzienniku rozruchowym (ang. *boot log*), który umieszczony jest w pamięci *EEPROM* (zob. rysunek 1.1), głównie w celu dostarczenia informacji zwrotnej dla twórców systemu. Kolejny krok to inicjacja i odszukanie urządzenia odzyskiwania (ang. *recovery storage device, RSD*), które zawiera dane umożliwiające odbudowanie systemu. Proces tworzenia *RSD* polega na ściągnięciu przez użytkownika odpowiedniego obrazu systemu i umieszczeniu go na nośniku trwałym, jak np. pendrive czy też karta

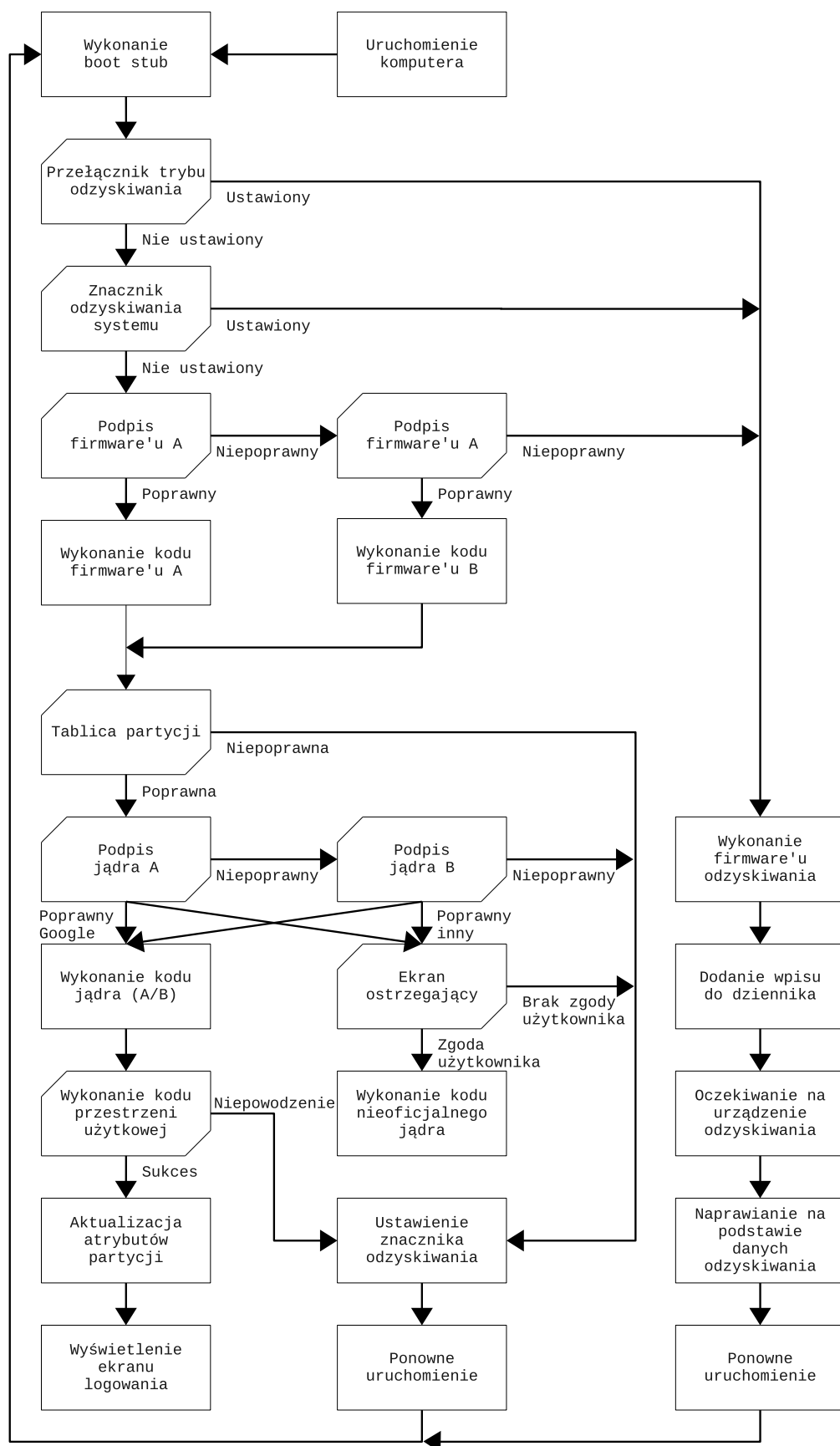
SD. Recovery firmware ze względu na swój mały rozmiar i prostotę nie pozwala na wykonanie tego procesu, a więc musi się to odbyć na innym komputerze. Gdy firmware odnajdzie odpowiedni nośnik odzyskiwania podłączony do komputera, zweryfikuje podpis cyfrowy znajdującego się na nim niewielkiego systemu operacyjnego (składającego się z jądra i systemu plików — zob. rysunek 1.3), służącego do odtworzenia zasadniczego systemu. Jeśli sprawdzenie podpisu zakończy się sukcesem, rozpocznie się rzeczywisty proces odtwarzania systemu na podstawie danych odzyskiwania (ang. *recovery data*) zawartych na *RSD*. *Recovery data* może zawierać *Chrome OS* lub alternatywny system operacyjny. Dalszy przebieg procesu odbudowy systemu jest w znacznej mierze zależny od zawartości *RSD*. W tym miejscu warto dodać, iż mechanizm *System Recovery* zapobiega atakom typu *rollback*, polegającym na zmianie oprogramowania na starsze — zawierające znane błędy. W tym celu układ *TPM* przechowuje licznik wersji, który może być modyfikowany wyłącznie przez firmware tylko do odczytu. Przy każdej próbie instalacji systemu *Chrome OS* numer aktualnie instalowanej wersji musi być większy bądź równy od wartości licznika.

Lista kroków przedstawiająca działanie mechanizmu *System Recovery* wygląda następująco (zob. rysunek 1.4):

1. Odnajdź uruchomienie trybu odzyskiwania systemu.
2. Zainicjuj urządzenie potrzebne do odbudowania systemu.
3. Odszukaj urządzenie odzyskiwania podłączone do komputera.
4. Jeśli urządzenia nie odnaleziono, wyświetl stosowny komunikat i czekaj.
5. Zweryfikuj autentyczność jądra i systemu plików znajdujących się na *RSD*.
6. Jeśli sprawdzenie podpisu powiodło się, rozpocznij odzyskiwanie systemu na podstawie *recovery data*.



Rysunek 1.3: Układ danych na przykładowym urządzeniu odzyskiwania — *RSD*. Na podstawie: [26].

Rysunek 1.4: Schemat blokowy rozruchu systemu *Chrome OS*. Na podstawie: [26].

1.3. Mechanizmy wewnętrzne

Wewnętrzne mechanizmy bezpieczeństwa służą kompleksowej ochronie systemu *Chrome OS* bezpośrednio po inicjalizacji jądra. Podstawowe cele stawiane tym mechanizmom to eliminacja „słabych punktów” systemu oraz redukcja potencjalnych zysków wynikających z udanego ataku. System *Chrome OS* został więc zaprojektowany w myśl popularnego stwierdzenia, iż nie istnieją doskonałe mechanizmy bezpieczeństwa, a więc należy być także przygotowanym na przełamanie owych zabezpieczeń.

Mechanizmy bezpieczeństwa wewnętrznego, tak jak pozostała część *Chrome OS*, zbudowane zostały na podstawie systemu *GNU/Linux*. Wybór tego systemu operacyjnego nie był oczywiście przypadkowy. *Linux* bazuje w znacznej mierze na zaufanych i sprawdzonych przez lata rozwiązaniach systemu *UNIX*, które są wzbogacane przez tysiące programistów na całym świecie. Można więc z całą pewnością stwierdzić, że system *GNU/Linux* to wypróbowana platforma, charakteryzująca się wysokim poziomem bezpieczeństwa. Typowe mechanizmy bezpieczeństwa systemu *Linux* takie jak np.: izolacja procesów, uprawnienia kont użytkowników czy ograniczanie zasobów (*rlimits*) są powszechnie znane i dokładnie opisane m.in. w [4, 16, 53]. W związku z tym, poniższy opis koncentruje się tylko na niestandardowych i nowoczesnych mechanizmach bezpieczeństwa. W tym miejscu należy dodać, iż aktualnie trwają intensywne prace nad wewnętrznymi mechanizmami ochrony, a więc poniższy opis może stać się z czasem niekompletny bądź nieaktualny.

1.3.1. Ochrona *rootfs*

Główny system plików *Chrome OS* (montowany do korzenia — /), oznaczony jako *rootfs*, zawiera wszelkie aplikacje i demony potrzebne do pracy systemu. *Rootfs* jest centralnym źródłem usług świadczonych przez system *Chrome OS*, a więc powinien być należycie chroniony przed przypadkową bądź celową modyfikacją. Środki osiągnięcia tego celu zostały przedstawione poniżej.

Rootfs podobnie jak firmware oraz jądro systemu występuje w dwóch kopiach (zob. rysunek 1.2). Główny system plików jest ściśle związany z daną kopią jądra (tzn. *rootfs* A współpracuje wyłącznie z jądrem A, a *rootfs* B wyłącznie z jądrem B), co ma na celu zachowanie spójności systemu w przypadku nieudanej aktualizacji któregoś z par.

Zbiór programów znajdujących się na głównej partycji nie jest modyfikowalny (zob. podrozdział 1.3.6). Zatem, skoro *rootfs* nie zawiera żadnych danych, które są zmieniane podczas pracy systemu, to może być montowany jako system plików tylko do odczytu. Zabieg ten jest skuteczną metodą zabezpieczenia danych przed modyfikacją przez system pracujący w trybie użytkownika.

Główny system plików jest jedynym stałym elementem systemu, który nie jest sprawdzany podczas *Verified Boot*. Weryfikacja partycji o rozmiarze ok. 75 MB podczas uruchamiania na komputerze klasy netbook trwa ok. 5 sekund [26]. Tak duże opóźnienie startu systemu, który stawia na wysoką wydajność, jest niedopuszczalne. Zaistniała więc potrzeba przeniesienia weryfikacji do przestrzeni jądra — na żądanie. W tym celu *Chrome OS* wyposażony został w wirtualne urządzenie blokowe, będące pośrednikiem pomiędzy rzeczywistym urządzeniem dyskowym, a systemem operacyjnym. Odczyt każdego bloku danych z urządzenia wirtualnego powoduje pobranie bloku z dysku, obliczenie funkcji skrótu danych i porównanie

otrzymanej wartości z zapisanym na stałe skrótem poprawnego bloku. Proces ten zapewnia weryfikację integralności danych, jednak problem ich autentyczności pozostaje wciąż nierozwiązany. W celu minimalizacji narzutu czasowego mechanizm ten wyposażony jest w wiele rozwiązań optymalizacyjnych takich jak: buforowanie, tworzenie wiązek (ang. *bundle*) skrótów oraz zapis zweryfikowanych bloków w pamięci podręcznej (ang. *caching*). Implementacja tego mechanizmu w obecnej formie nie zabezpiecza przed atakami polegającymi na podmianie *rootfs* oraz odpowiadających jej skrótów, gdyż jak wspomniano, nie jest weryfikowana autentyczność tej partycji. Możliwy jest także atak typu *rollback*, gdyż *rootfs* nie posiada żadnych znaczników wersji.

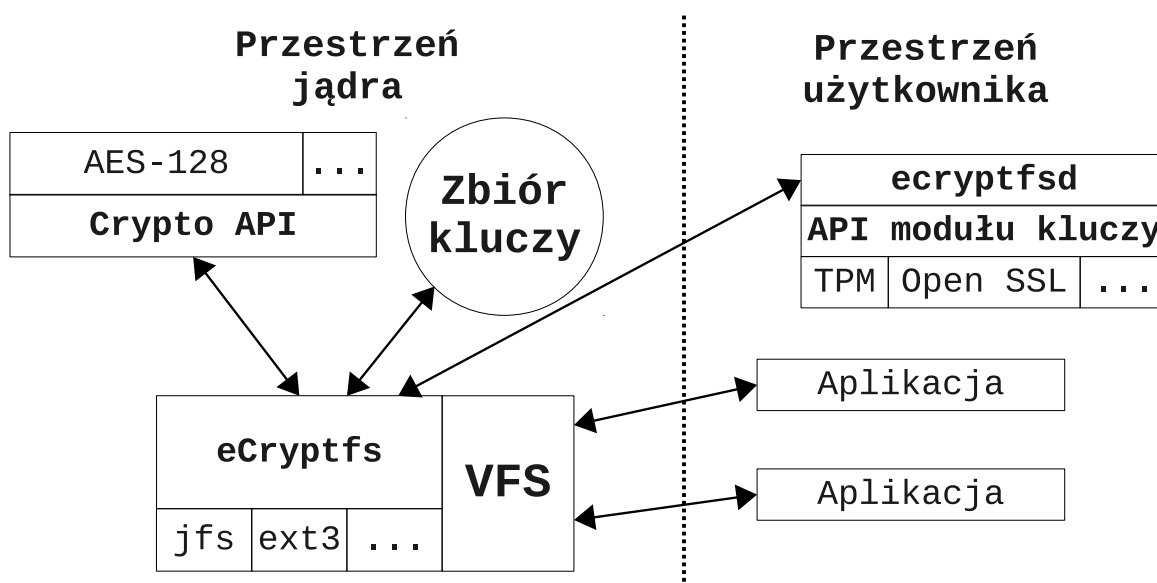
1.3.2. Ochrona danych użytkownika — eCryptfs

Zgodnie z podstawowym założeniem projektowym systemu *Chrome OS*, użytkownicy znaczną część informacji będą przechowywać w chmurze i to na niej spoczywać będzie obowiązek zabezpieczenia tych danych. Część danych użytkownika musi być jednak przechowywana na dysku lokalnym — typowo są to: pliki konfiguracyjne, zapisane hasła, pamięć *cache* przeglądarki czy po prostu dane, do których użytkownik chce mieć dostęp offline. *Chrome OS* ze względu na to, że jest systemem dla wielu użytkowników, przeznaczonym na urządzenia mobilne powinien należycie chronić te informacje.

Systemy „unikopodobne” w ochronie plików różnych użytkowników polegają na prawach własności i dostępu. Metoda ta pozwala jednak na dostęp do dowolnych danych użytkownikowi z prawami *roota*. Ponadto możliwe jest podmontowanie dysku do innego systemu w celu uzyskania dostępu do plików. Prawa dostępu i własności nie zapewniają więc wystarczającego poziomu bezpieczeństwa. Z tego powodu są one wykorzystywane w systemie *Chrome OS* w minimalnym stopniu — każdy użytkownik zalogowany do systemu widoczny jest w rzeczywistości jako ten sam *UID* (1000, o nazwie wyświetlanej *chronos*), a specyficznymi różnicami pomiędzy poszczególnymi kontami zarządzają *session manager* oraz *login manager*. Za bezpieczeństwo plików, w postaci silnego szyfrowania, odpowiada natomiast *eCryptfs*.

ECryptfs jest stosowym (ang. *stacked*), działającym w obszarze jądra, kryptograficznym systemem plików dla *Linuksa*. Stał się on częścią *Linuksa* od wersji 2.6.19 i szeroko wykorzystuje natywne, kryptograficzne *API* jądra. *ECryptfs* działa jako nakładka na niższe systemy plików (ang. *lower filesystems*) takie jak np. *ext3*, *NFS*. Gdy program działający w przestrzeni użytkownika wykonuje operacje na plikach znajdujących się w punkcie montowania *eCryptfs*, żądania trafiają przez wirtualny system plików (*Virtual Filesystem*, *VFS*) do modułu *eCryptfs*. Moduł ten, w zależności od operacji, szyfruje zapisywane dane i przekazuje je niższemu systemowi plików do zapisu bądź odszyfrowuje i zwraca dane odczytane przez niższy system plików. Klucze potrzebne do operacji kryptograficznych znajdują się w sesyjnym zbiorze kluczy (ang. *keyring*), a zarządzaniem nimi zajmuje się demon *ecryptfsd*. Schemat działania *eCryptfs* przedstawiono na rysunku 1.5.

Chrome OS wykorzystuje i rozszerza mechanizm *eCryptfs*. Przy pierwszym logowaniu dla użytkownika tworzony jest skarbiec (ang. *vault*) w postaci katalogu oraz zbiór kluczy (ang. *keyset*). Skarbiec pełni rolę magazynu danych (który można utożsamiać z katalogiem *home* użytkownika), natomiast zbiór kluczy służy zapisywaniu i odczytywaniu danych ze skarbcza. W chwili logowania skarbiec zostaje

Rysunek 1.5: Schemat poglądowy architektury *eCryptfs*. Na podstawie: [29].

(przezroczyście dla użytkownika) otwarty — odszyfrowany, a w momencie wylogowania zamknięty — zaszyfrowany. *Vault* zawiera wszelkie dane związane z danym użytkownikiem takie jak: zainstalowane wtyczki, pliki konfiguracyjne, pamięć podręczną przeglądarki, zachowane informacje. Otwarcie skarbca jest więc krokiem wymagającym do pomyślnego logowania. Zbiór kluczy jest generowany losowo (za pomocą generatora liczb pseudolosowych jądra, zasilanego entropią z układu *TPM*) i pozostaje stały na czas istnienia konta na danym komputerze. W skład zbioru kluczy wchodzi: klucz szyfrowania plików (ang. *file encryption key*, *FEEK*), klucz szyfrowania nazw plików (ang. *file name encryption key*, *FNEK*) oraz metadane. Dany *keyset* jest związany z hasłem logowania użytkownika do systemu — jest chroniony „słabym skrótem” tj. połową bitów skrótu hasła z solą (ang. *salt*). Taki zabieg ma zapewnić skuteczną ochronę zarówno zbioru kluczy, jak i hasła logowania. Zarządzaniem kluczami zajmuje się demon *cryptohomed*, który jest rozszerzeniem demona *ecryptfsd*.

Obecnie wydaje się, że nie istnieją skuteczniejsze metody ochrony danych niż silne szyfrowanie. Niestety, zaawansowana kryptografia znacząco odbija się na wydajności. Szyfrowanie zużywa, poza cyklami *CPU*, dodatkowe strony pamięci. Ponadto konieczność wykonywania złożonych obliczeń kryptograficznych może spowodować, że zapisy i odczyty z dysku staną się według planisty operacjami z dominującym *CPU* (ang. *CPU-bound*), co może zakłócić skuteczne działanie szeregowania procesów. W obliczu powyższych spostrzeżeń, dobrą decyzją jest oddzielenie danych niezmiennych (na partycji *rootfs*, zob. podrozdział 1.3.1) — nieszyfrowanych od danych modyfikowalnych — szyfrowanych, gdyż dzięki temu nie wszystkie operacje wejścia/wyjścia są tak kosztowne.

1.3.3. Ograniczanie uprawnień — *minijail*

Większość technik zabezpieczania systemu *GNU/Linux* bazuje na stosowaniu popularnej zasady najmniejszego uprzywilejowania (ang. *principle of least privilege*), która polega na nadawaniu jednostce minimalnych uprawnień potrzebnych

jej do wykonywania swoich zadań. Działanie w myśl tej dewizy zapewnia, iż udany atak na daną jednostkę spowoduje minimalne szkody (adekwatne do posiadanych uprawnień). Programiści *Chrome OS*, w celu realizacji powyższej wytycznej, postanowili wykorzystać istniejące mechanizmy *Linuksa* służące do zarządzania przywilejami — zarówno te obecne od lat, jak i te nowoczesne. Efektem ich starań jest niewielka aplikacja o nazwie *minijail*, który służy do uruchamiania innych programów w ściśle zdefiniowanym (przez listę argumentów) środowisku. Program *minijail* jest szeroko wykorzystywany w skryptach inicjalizacyjnych, w celu konfiguracji przywilejów zależnych od potrzeb. *Minijail* posiada następujące możliwości:

- stworzenie środowiska *chroot*,
- wylogowanie z konta *roota* tzn. zmiana *UID* i *GID*,
- ustawienie limitów dla zasobów za pomocą *rlimits*,
- umieszczenie procesu w przestrzeni nazw (ang. *namespace*),
- ustawienie mechanizmu *Capabilities*.

Dwa ostatnie mechanizmy, ze względu na nietypowość i złożoność, zostały omówione poniżej.

Namespaces

Namespaces jest mechanizmem jądra, który pozwala na zorganizowanie procesów w grupę — przestrzeń nazw, wewnątrz której będzie dostępny pewien logiczny zasób. Mechanizm przestrzeni nazw wraz z *Control Groups* (zob. podrozdział 1.3.4) pozwala tworzyć kontenery (ang. *containers*) — wirtualne środowiska poziomu systemu operacyjnego umożliwiające uruchamianie wielu odizolowanych instancji *Linuksa* (więcej o kontenerach można odnaleźć w [11, 45, 49]). Istnieją różne rodzaje przestrzeni nazw w zależności od izolowanego zasobu.

PID — przestrzeń nazw *PID* izoluje alokacje identyfikatorów procesów. Procesy funkcjonujące wewnątrz przestrzeni „widzą” wyłącznie inne procesy z tej przestrzeni, jednak wszystkie one są widzialne z przestrzeni globalnej (proces ma tyle numerów *PID*, do ilu przestrzeni nazw należy). Pierwszy proces stworzony w nowej przestrzeni nazw ma w niej numer *PID* = 1 i funkcjonuje analogicznie do globalnego procesu *init*. Stworzona przestrzeń istnieje tylko przez czas życia tego procesu. Więcej informacji na temat *PID namespaces* można odnaleźć w [21].

UTS — przestrzeń nazw *UTS* pozwala na zmianę nazwy hosta i domen dla procesów będących wewnątrz niej.

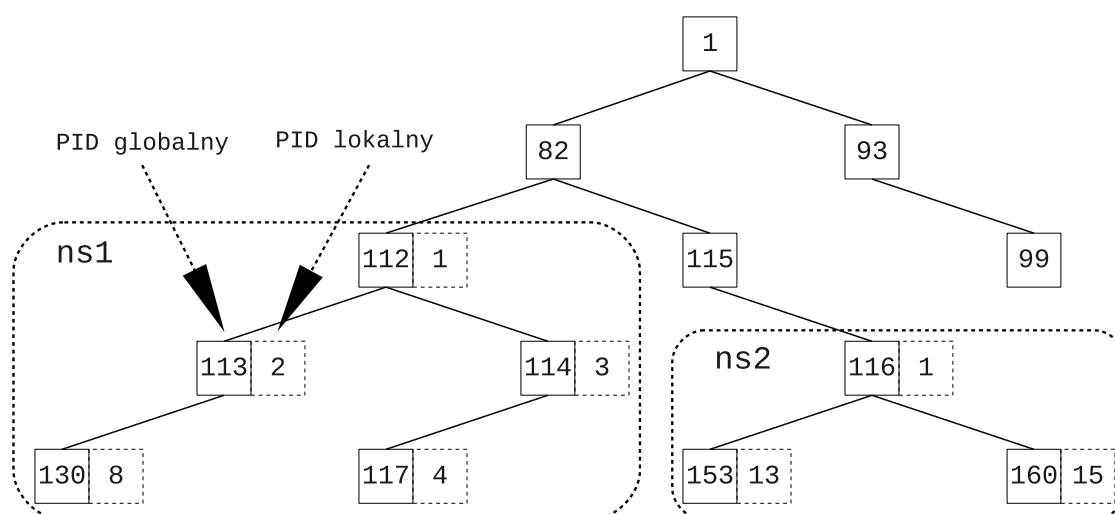
User — przestrzeń nazw *User* umożliwia zmianę mapowania numerów *UID*. Aktualnie funkcjonalność ta jest w trakcie rozwoju i nie spełnia wielu typowych wymagań.

IPC — przestrzeń nazw *IPC* izoluje identyfikatory mechanizmów *System V IPC* — struktury stworzone globalnie wewnątrz przestrzeni są „widoczne” tylko wewnątrz niej.

VFS — przestrzeń nazw *VFS* pozwala na stworzenie lokalnej perspektywy na system plików.

NET — przestrzeń nazw *NET* pozwala na izolację urządzeń sieciowych, reguł *iptables*, tablic routowania itd.

System *Chrome OS* używa przestrzeni nazw w celu tworzenia izolowanych, wirtualnych środowisk. Z tego powodu niepoprawnie działające aplikacje są w stanie zaburzyć pracę minimalnego zbioru procesów — tylko tych, które znajdują się w tej samej przestrzeni nazw. *Namespacing* jest zatem mechanizmem, który skutecznie ogranicza zakres szkód możliwych do wyrządzenia po udanym ataku.



Rysunek 1.6: Przykładowa struktura mechanizmu *PID namespace*. Poza globalną przestrzenią nazw w systemie utworzono dwie inne — *ns1* oraz *ns2*.

Capabilities

W typowym systemie „uniksopodobnym” można wyróżnić dwie kategorie użytkowników: użytkownik z uprawnieniami *roota* i użytkownik bez tych uprawnień. Taka dychotomia jest niekorzystna ze względu na bezpieczeństwo. Aplikacje, które potrzebują tylko pewnego podzbioru uprawnień *roota*, muszą być uruchamiane z pełnymi prawami, a to przeczy zasadzie minimalnego uprzywilejowania.

Problem ten rozwiązuje mechanizm *Capabilities* (skrótowo oznaczany jako *PCap*), który jest częścią standardu *POSIX*. Rozwiązanie *Capabilities* polega na podzieleniu uprawnień *roota* na wiele mniejszych (aktualnie jest ich ponad 30 i istnieje możliwość dodania kolejnych). Przykładowe uprawnienia: *CAP_NET_RAW* — pozwala na otwarcie gniazda w trybie *RAW*, *CAP_CHOWN* — pozwala wykonywać polecenie zmiany właściciela *chown*, *CAP_MKNOD* — pozwala na tworzenie plików specjalnych za pomocą *mknod*. Uprawnienia *PCap* są indywidualne dla wątku, mogą być także nadawane plikom wykonywalnym na zasadach nieco podobnych do bitu *SUID*. Należy dodać, iż mechanizm *Capabilities* jest zgodny wstecz z tradycyjnymi aplikacjami używającymi *SUID*. Interesującym dodatkiem do *PCaps* są *Securebits*, które pozwalają na wyłączenie specjalnego traktowania użytkownika *root* — przestaje on mieć jakiekolwiek dodatkowe uprawnienia i nie ma możliwości przełączenia się na „pełnoprawnego” *roota*.

Chrome OS używa mechanizmu *Capabilities*, aby w naturalny sposób ograniczyć uprawnienia aplikacji, typowo posiadających uprawnienie *roota* jak np. *XServer*, *upstream* czy *crond*. Nadrzędnym celem projektantów *Chrome OS* jest, aby żaden proces nie działał z pełnymi uprawnieniami *roota*, a jedynie z tymi, których potrzebuje. Dzięki temu, szkody spowodowane udanym atakiem na aplikację będą zminimalizowane. Ponadto system *Chrome OS* szeroko wykorzystuje wspomniane *Securebits* (w szczególności *SECURE_NOROOT*), tak aby nie była możliwa eskalacja uprawnień. Więcej informacji o *Capabilities* i *Securebits* można odnaleźć w [17, 42].

1.3.4. Ograniczanie zasobów — Control Groups

Control Groups, zwany skrótowo *cgroups*, to mechanizm *Linuksa* (obecny od wersji 2.6.24) pozwalający na kontrolowanie użycia zasobów przez procesy. Za pomocą *cgroups* możliwe jest umieszczenie procesów w grupie kontrolnej (*cgroup*), z którą związane są pewne parametry dla danego podsystemu. Grupy kontrolne mogą być zorganizowane w złożoną hierarchię, w której skonfigurowane parametry są dziedziczone. Konkretne podsystemy (zwane czasem kontrolerami zasobów (ang. *resource controllers*)) służą zarządzaniu zasobami, które może polegać na:

- księgowaniu użycia danego zasobu przez grupę,
- ograniczaniu ilości danego zasobu dla grupy,
- gwarantowaniu ilości danego zasobu dla grupy,
- izolowaniu zasobów od grupy,
- nadawaniu grupie priorytetu dostępu do zasobu.

Przykładowe podsystemy to:

blkio — pozwala skonfigurować ilościowe limity na dostęp do urządzeń blokowych,

cpu — pozwala wpływać na przydział czasu procesora,

cpuacct — generuje raporty dotyczące użycia procesora,

cpuset — pozwala przypisać konkretne procesory i linie adresowe,

devices — pozwala stworzyć listy dostępu do urządzeń,

memory — pozwala skonfigurować limity dostępnej pamięci oraz generuje raporty o jej użyciu,

net — pozwala kontrolować dostęp do pasma łącza sieciowego.

Control Groups są więc mechanizmem podobnym do *nice* czy *security limits*, jednak o daleko bogatszych możliwościach i wysokiej elastyczności.

W systemie *Chrome OS* *cgroups* są użyte do realizacji wielu mechanizmów np.:

- izolacja dostępu do zasobów pozwala na efektywną implementację piaskownicy (ang. *sandbox*) (zob. podrozdział 1.3.6),
- listy dostępu do urządzeń pozwalają na ograniczenie możliwości wtyczek,
- nadawanie priorytetów dostępu do sieci wyróżnia ważniejsze usługi (np. *Auto-update*),
- podział zadań na systemowe i użytkowe oraz odpowiedni podział czasu procesora i pamięci chroni przed atakami typu *DoS* ze strony złośliwych wtyczek i aplikacji webowych.

Systemowy zarządca zasobów w postaci *Control Groups* jest więc istotnym wsparciem istniejących, jak również źródłem nowych mechanizmów bezpieczeństwa.

1.3.5. Kontrola dostępu — Medium Access Control

Medium Access Control (MAC) to znany, ale niezbyt popularny w użyciu (nieobecny w większości znanych dystrybucji *Linuksa*) mechanizm bezpieczeństwa. Polega on na zdefiniowaniu zbioru podmiotów (ang. *subjects*) (którymi najczęściej są procesy lub wątki), zbioru obiektów (ang. *objects*) (które są np. plikami, gniazdami, segmentami pamięci) oraz polityki dostępu podmiotów do obiektów. Przestrzegania polityki dostępu pilnuje administrator bezpieczeństwa (ang. *security policy administrator*), który, aby działał skutecznie, musi być elementem jądra systemu.

Obecnie MAC nie jest jeszcze wdrożony do *Chrome OS*, jednak ma to się stać przed opublikowaniem finalnej wersji systemu. Aktualnie rozważane jest zastosowanie jednego z dwóch rozwiązań: *TOMOYO Linux*, który jest częścią jądra od

wersji 2.6.30 lub *grsecurity*, który jest rozbudowaną łątą dla *Linuksa*. Bogaty zestaw funkcji *grsecurity* (poza *MAC* wprowadza on ponad 100 modyfikacji [50]) przemawia za zastosowaniem łąty, jednak ze względu na wynikające z tego trudności w aktualizacji jądra, decyzja nie została jeszcze podjęta [26]. Niezależnie jednak od wybranego rozwiązania schemat konfiguracji *MAC* będzie taki sam. Ograniczenia dostępu zostaną nałożone na wszystkie procesy działające w systemie, a polityka zostanie ustalona na podstawie sekwencji uczących (oba rozwiązania posiadają tryb nauki polityki).

1.3.6. Systemowe API — przeglądarka Google Chrome

Większość udanych ataków na systemy komputerowe jest możliwa na skutek błędów znajdujących się w aplikacjach usługowych, a nie w kodzie samego systemu operacyjnego. Można zatem stwierdzić, iż bezpieczeństwo systemów jest w znacznej mierze zależne od poprawności działających wewnątrz nich programów. *Chrome OS* jest pod tym względem dość nietypowym systemem operacyjnym. Wszelkie aplikacje użytkowe są uruchamiane wewnątrz przeglądarki internetowej *Chrome* — pod postacią aplikacji webowych bądź wtyczek. Można więc powiedzieć, że przeglądarka *Chrome* pełni rolę maszyny wirtualnej, wewnątrz której uruchamiane są kolejne usługi. Z tego powodu, główny ciężar bezpieczeństwa zostaje przeniesiony na przeglądarkę *Chrome*, gdyż to ona musi być odporna na błędy i ataki ze strony uruchomionych aplikacji.

Przeglądarka *Chrome* ma architekturę modułową, opartą na wieloprocusowości i mechanizmach *IPC*. Centralnym modułem przeglądarki jest jej jądro (ang. *browser kernel*), które odpowiedzialne jest za komunikację z systemem operacyjnym (w tym za obsługę zdarzeń wejścia/wyjścia), udostępnianie *API* oraz zarządzanie zasobami (takimi jak np. okna, hasła, pamięci podręczne). Jądro jest jedynym zaufanym elementem przeglądarki, w związku z czym jego działania nie muszą być ograniczane. Uruchamiane aplikacje webowe oraz wtyczki traktowane są natomiast odmiennie — jako niezaufane.

Każda aplikacja internetowa uruchamiana jest za pomocą przeglądarki *Chrome* w oddzielnym procesie umieszczonym wewnątrz piaskownicy. Dzięki izolacji procesów — jądra przeglądarki i mechanizmu renderowania, ewentualne zawieszenie lub błąd drugiego nie wpływa na pracę pierwszego. Ponadto proces skojarzony z aplikacją nie może bezpośrednio wykonywać operacji i wpływać na struktury danych jądra (co byłoby możliwe w architekturze opartej na wątkach). Umieszczenie procesu aplikacji w piaskownicy ogranicza natomiast jego możliwości komunikacji z systemem operacyjnym. Niemożliwe staje się np. przechwytywanie operacji wejścia/wyjścia czy używania wywołań systemowych (ang. *system calls*, *syscalls*) do operacji na plikach bądź urządzeniach. Decyzja o opisanej wyżej separacji wydaje się być uzasadniona, gdyż zgodnie z szacunkami [2] silnik renderujący jest źródłem ok. 70% wszystkich podatności w przeglądarkach internetowych — głównie ze względu na wykonywanie kodu *JavaScript* niewiadomego pochodzenia.

Wtyczki, z tego samego powodu co aplikacje internetowe, uruchamiane są w oddzielnych procesach. Procesy wtyczek nie są jednak domyślnie uruchamiane wewnątrz piaskownicy, gdyż część z nich wymaga bezpośredniego dostępu do systemu operacyjnego (np. *Flash Plugin* może korzystać z mikrofonu czy karty dźwiękowej). Może to stwarzać bezpośrednie zagrożenie w przypadku instalowania wtyczek z niezaufanego źródła.

Na zakończenie warto dodać, iż możliwe jest zastosowanie do procesów przeglądarki mechanizmów opisanych w podrozdziałach: 1.3.3, 1.3.4, 1.3.5. Mimo wszystko, część zagrożeń pozostaje wciąż poza zasięgiem zastosowanej technologii. Dotyczy to szczególnie tych, które są związane z socjotechniką (jak np. phishing czy zamierzona instalacja maleware). Wyeliminowanie tych niebezpieczeństw jest możliwe tylko na drodze uświadamiania użytkowników.

1.3.7. Usługa aktualizacji — Autoupdate

Autoupdate jest usługą automatycznej aktualizacji składników *Chrome OS* — firmware'u, jądra oraz oprogramowania. Posiada jednak kilka cech, które odróżniają ją od tych spotykanych w innych systemach operacyjnych. Przede wszystkim aktualizacja odbywa się za pomocą bezpiecznego protokołu *HTTPS*, wymagającego od serwera certyfikatu potwierdzonego przez urząd certyfikujący (ang. *CA*). Dodatkowo podobnie jak w mechanizmie *System Recovery* (rozdział 1.2.2), *Autoupdate* uniemożliwia zastosowanie ataku typu *rollback* dzięki użyciu liczników monotonicznych układu *TPM*.

Ostatnim wartym odnotowania faktem jest to, że błędna aktualizacja nie narusza stabilności systemu ze względu na istnienie kopii zapasowych — firmware'u, jądra oraz głównego systemu plików (*rootfs*). Niestety mechanizm automatycznych aktualizacji nie jest oficjalnie wspierany w rozwijanej przez społeczność wersji systemu — *Chromium OS* [26].

1.4. Podsumowanie i ocena rozwiązań

System *Chrome OS* imponuje bogactwem i stopniem zaawansowania mechanizmów bezpieczeństwa. W dobie powszechnej informatyzacji w zasadzie każdej dziedziny życia ochrona informacji ma szczególnie dużą wartość, a więc decyzja projektowa o postawieniu na wysoki jej poziom wydaje się trafna. Specyfika dziedziny powoduje jednak, że rzeczywistą jakość zastosowanych rozwiązań będzie można rzetelnie ocenić dopiero, gdy *Chrome OS* trafi do powszechnego użytku i zyska istotną popularność. Mimo wszystko, już teraz można wskazać wiele problemów, które są poza zasięgiem zaimplementowanych mechanizmów, m.in.:

- modyfikacja sprzętu (podmiana pamięci *EEPROM*, podłączenie dodatkowego układu) — daje to możliwość pominięcia weryfikacji autentyczności i integralności,
- po udanym ataku:
 - system pozostaje w stanie naruszonej integralności aż do momentu wykrycia nieprawidłowości przez *Verified Boot* przy ponownym uruchomieniu,
 - atakujący może ustawić jako stronę startową specjalnie przygotowaną aplikację internetową, która pobiera i uruchamia odpowiedni *exploit* — zapewnia to powtarzalność ataku, pomimo istnienia *Verified Boot*,
 - atakujący może zmienić domyślny punkt dostępu sieci bezprzewodowej na *sniffer*, co umożliwia podsłuchiwanie dalszych transmisji,
 - atakujący ma dostęp do danych aktualnie zalogowanego użytkownika pomimo szyfrowania.
- ataki typu *cold boot* pozwalające na odczyt kluczy i danych w chwilę po wyłączeniu komputera,

— ataki wykorzystujące podatności w niepodpisywalnych (ze względu na dużą zmienność) danych — np. konfiguracji systemu.

Niestety, większości z tych problemów nie da się łatwo wyeliminować. Niektóre z nich występują niezależnie od użytego systemu operacyjnego, a usunięcie innych mogłoby się z kolei okazać niewspółmiernie kosztowne w stosunku do potencjalnych zysków. Niemniej jednak podejście do kwestii bezpieczeństwa zastosowane w *Chrome OS* tj. użycie wielowarstwowych, złożonych mechanizmów zapewniających tzw. głęboką obronę (ang. *defence in depth*) wydaje się istotnym krokiem naprzód w kwestii bezpieczeństwa systemów operacyjnych.

2. Kontrola dostępu do systemowego ABI

Wywołania systemowe w systemach operacyjnych z ochroną pamięci są zasadniczym sposobem interakcji procesów użytkownika z jądrem. Można zatem powiedzieć, że dla wykonującego się programu zbiór dostępnych wywołań definiuje jego możliwości. Naturalnie więc kontrola dostępu do systemowego ABI (*application binary interface*) — tzn. do wspomnianych wywołań systemowych jest istotnym zagadnieniem w tematyce zapewniania bezpieczeństwa. Zapotrzebowanie na mechanizm oferujący ochronę dostępu do ABI zostało sformułowane w dokumentacji *Chrome OS*, jednak nie odnaleziono odpowiedniego narzędzia spełniającego wszystkie potrzeby projektu [26]. W związku z powyższymi spostrzeżeniami warto dokładniej zbadać zagadnienie nadzoru dostępu do wywołań systemowych.

Treścią bieżącego rozdziału jest dogłębna analiza zagadnienia kontroli wykorzystania ABI systemu operacyjnego *GNU/Linux*. W pierwszej części przedstawione zostały korzyści płynące z użycia wyżej wspomnianego mechanizmu oraz jego konkretne zastosowania. Druga część zawiera przegląd istniejących implementacji — ich krótkie opisy, zalety oraz wady. Rozdział kończy się podsumowaniem i wnioskami płynącymi z przeprowadzonego badania.

2.1. Korzyści

Zabezpieczając system komputerowy, zakłada się, iż jądro systemu operacyjnego jest oprogramowaniem zaufanym — starannie przetestowanym, a więc pozbawionym błędów. Odmienne, jako niezaufane, traktuje się natomiast aplikacje użytkownika działające wewnątrz systemu. Procesy usługowe są często bardzo złożone, zdarza się także, że są one niedbale napisane czy też niedokładnie zweryfikowane. Te cechy sprzyjają obecności dużej liczby błędów w takim oprogramowaniu, co pociąga za sobą zagrożenie atakiem bądź destabilizacją systemu. Z tego powodu, znaczną część wysiłku wkłada się w zabezpieczenie przed błędami aplikacji, a nie przed błędami jądra systemu operacyjnego.

Zastosowanie mechanizmu kontroli dostępu do systemowego ABI przynosi wiele korzyści. Ograniczenie zbioru dostępnych wywołań systemowych daje możliwość ścisłej kontroli nad tym, co dany proces może w systemie. Jądro systemu *Linux* udostępnia dużą liczbę (w wersji 3.1, dla architektury *x86-64* jest ich ponad 300) dobrze zdefiniowanych wywołań. Możliwe jest zatem łatwe określenie minimalnego, wystarczającego zbioru usług systemowych dla wskazanego procesu. Gdyby ziarnistość wywołań systemowych *Linuksa* była mała — tzn. ABI zawierałoby niewiele funkcji, które spełniałyby wiele ról — mechanizm kontroli dostępu do usług systemowych byłby nieefektywny, gdyż zezwolenie na pewną część zbioru wywołań, dawałoby o wiele więcej możliwości, niż rzeczywiście byłoby potrzebnych. Właściwe zastosowanie opisywanego mechanizmu zapewnia więc działanie w myśl zasady

najmniejszego uprzywilejowania — mniej dostępnych funkcji pociąga za sobą mniej potencjalnych szkód po udanym ataku.

Jednym z najczęściej występujących błędów w aplikacjach użytkowych jest przepełnienie bufora (ang. *buffer overflow*). Wykorzystanie takiej podatności pozwala atakującemu zmodyfikować standardową, prawidłową ścieżkę wykonania programu — intruz może zmienić lub wstrzyknąć dodatkowy kod do wykonującego się programu. Atakujący ma więc możliwość wykonania dowolnych instrukcji z prawami użytkownika, do którego należy atakowany proces. Szkody wyrządzone na skutek istnienia błędów przepełnienia bufora mogą być więc większe, jeśli program wykonuje się z uprawnieniami *roota* lub ustawionym bitem *SUID*. Przykładami typowych aplikacji działających w wielu systemach, z wyżej wymienionymi uprawnieniami mogą być odpowiednio — *XServer* oraz *sendmail*. Zastosowanie mechanizmu kontroli dostępu do wywołań systemowych pozwala znacząco złagodzić skutki ataku tego typu. Odpowiednie wydzielenie dostępnych dla procesu usług może ograniczyć lub nawet uniemożliwić wyrządzenie jakichkolwiek szkód. Przykładowo, administrator może zabronić procesowi uruchamiania innych programów, bądź tworzenia procesów potomnych (wzbronienie dostępu do wywołań *fork()*, *vfork()*, *clone()*, *execve()*) — utrudnia to działania atakującego i uniemożliwia wykonanie ataku *DoS* (ang. *Denial of Service*) przy użyciu techniki *fork bomb*. Innym sensownym działaniem może być np. uniemożliwienie zmiany użytkownika (zabronienie *setuid()* i pokrewnych) czy powstrzymanie zmian praw dostępu i własności plików (wstrzymanie wywołań *chmod()*, *chown()* i ich odmian).

Kolejnym zastosowaniem mechanizmu kontroli dostępu do wywołań systemowych jest stworzenie dla procesu piaskownicy — tj. ograniczenia dostępnych usług do minimum (np. do *read()*, *write()* oraz *exit()*). Piaskownica przydaje się podczas uruchamiania kodu nieznanego bądź niepewnego pochodzenia i działania — np. kodu aplikacji webowych wewnątrz przeglądarki (zob. podrozdział 1.3.6). Kod taki, dopóki nie zostanie dokładnie przeanalizowany i zaakceptowany, nie powinien mieć możliwości dokonywania żadnych zmian w systemie operacyjnym.

Dzięki opisywanemu mechanizmowi administrator systemu może globalnie, dla wszystkich procesów, zabronić dostępu do pewnej grupy usług systemowych. Niektóre wywołania są używane tylko w określonych warunkach, w innych natomiast mogą być szkodliwe. Przykładem takiego wywołania systemowego może być *ptrace()* służące do debugowania oraz śledzenia procesów. Usługa ta jest szeroko stosowana jako narzędzie programistyczne w środowiskach rozwojowych. Z drugiej strony, w środowisku produkcyjnym przydatność *ptrace()* jest znikoma, a bogate możliwości tego wywołania mogą posłużyć potencjalnemu intruzowi do skutecznego zakłócenia pracy innych procesów. Zastosowanie globalnych ograniczeń usług systemowych warto także rozważyć w sytuacjach, w których dane wywołanie jest nowym, niedojrzałym jeszcze elementem systemu operacyjnego. Niestety, pomimo skrupulatnego procesu testowania i przeglądów kodu *Linuksa*, zdarzały się sytuacje, w których wywołania systemowe zawierały błędy bezpieczeństwa [12]. Administrator systemu wymagającego wysokiej stabilności i bezpieczeństwa może więc całkowicie wyłączyć możliwość używania niesprawdzonych wywołań.

Mechanizm kontroli ABI daje bogate możliwości, a konkretne jego użycie zależy wyłącznie od specyfiki procesu, na który nakładane są ograniczenia. Dla procesów o niedużym stopniu złożenia oraz o dobrze zdefiniowanej funkcjonalności łatwo skonstruować politykę dostępu do wywołań systemowych i mechanizm ten może być z łatwością zastosowany. Z drugiej strony dla aplikacji o niezbyt klarownej

strukturze i funkcjach, zastosowanie odpowiednich ograniczeń może być niewykonalne.

2.2. Zastosowania

W poprzednim podrozdziale przedstawiono rozległe korzyści płynące z zastosowania mechanizmu kontroli dostępu do systemowego ABI. W następnym kroku warto więc przeanalizować, w jaki sposób można je wykorzystać w konkretnych sytuacjach. Poniższe przypadki użycia stanowią jedynie przykład eksploatacji opisywanego mechanizmu, gdyż kompletny przegląd jest niemożliwy ze względu na mnogość potencjalnych zastosowań.

2.2.1. Kontenery procesów

Koncepcja kontenerów, opisana dokładniej w podrozdziale 1.3.3, zapewnia wzajemną izolację grup procesów. Pomimo wydzielenia wielu zasobów poszczególne kontenery, zgodnie z ideą, współdzielą to samo jądro — usługi systemowe stają się jedynym zasobem w pełni współdzielonym. Zastosowanie mechanizmu ochrony wywołań systemowych dla kontenerów pozwala zatem wprowadzić kolejny poziom izolacji — izolację usług jądra. Stosowanie kontenerów pozwala chronić wzajemnie grupy procesów, natomiast użycie opisywanej techniki umożliwia dodatkowo ochronę systemu operacyjnego przed aplikacjami działającymi wewnątrz środowisk zwirtualizowanych.

Zastosowanie rozważanego mechanizmu w połączeniu z kontenerami ma jeszcze jedną zaletę — łatwość konfiguracji. Procesy działające wewnątrz kontenera spełniają zwykle podobną funkcję — są np. jednostkami wykonawczymi usług pewnego serwera. Dla procesów o podobnym działaniu łatwo stworzyć wspólną politykę dostępu do wywołań systemowych, którą potem można zastosować w całości, dla wszystkich zadań wewnątrz kontenera.

Dla zilustrowania użycia rozważmy dwa kontenery — w pierwszym z nich znajdują się procesy serwera pewnych usług, w drugim natomiast działają aplikacje wspomagające rozwój oprogramowania używane przez programistów. W pierwszym z wymienionych kontenerów z całą pewnością warto zabronić dostępu do niepotrzebnych i potencjalnie niebezpiecznych wywołań takich, jak opisywany wyżej *ptrace()*. Jednocześnie, procesy w drugim kontenerze będą często wykorzystywać śledzenie w celu debugowania i sprawdzania oprogramowania, a więc wywołanie *ptrace()* będzie wykorzystywane — nie powinno być zabronione. Testowanie nowych wersji oprogramowania może być potencjalnie niebezpieczne ze względu na błędy i omyłki — dla drugiego kontenera warto rozważyć zakazanie wywołań typu *setuid()*, aby przypadkowe wykonanie komend z uprawnieniami *roota* nie było możliwe. Zastosowanie mechanizmu ochrony ABI w połączeniu z kontenerami pozwala więc wzmocnić ochronę przed destabilizacją systemu oraz pośrednio zabezpiecza działające wewnątrz niego usługi.

2.2.2. Serwery

Serwery usług to typ oprogramowania niezwykle powszechnego na komputerach pracujących pod kontrolą systemu *Linux*. Wśród tego typu aplikacji można wyróżnić serwery o bardzo wąskiej specjalizacji np. serwery DNS, serwery czasu NTP,

serwery FTP. Oprogramowanie tego typu jest bardzo często zaniedbywane przez administratorów maszyn serwerowych — głównie ze względu na to, iż kolejne wydania bardzo często nie dodają nowej funkcjonalności, a jedynie poprawiają wykryte błędy — nie generują więc widocznej wartości biznesowej. Z tego powodu nieaktualizowane serwery bardzo często są swego rodzaju „furtką” dla intruzów. Atakujący nie musi za bezpośredni cel obierać interesujących go elementów (często dobrze zabezpieczonych). Zamiast tego, aby dostać się do systemu może wykorzystać błędy istniejące w „pobocznych” usługach, a dopiero później wykonać zasadniczy atak, ponieważ przeprowadzony „od środka” powinien być stosunkowo łatwy.

Mechanizmy ochrony dostępu do wywołań systemowych doskonale nadają się do izolacji serwerów o wysokiej specjalizacji. Przykładowo podstawowy serwer DNS może być zaimplementowany przy użyciu jedynie trzech wywołań systemowych (*recvmsg()*, *sendmsg()*, oraz *write()*) [20] — może być więc ograniczony tylko do trzech wymaganych usług jądra, stając się mało użytecznym dla atakującego. Mechanizm kontroli dostępu do ABI pozwala skutecznie wzmocnić ochronę przed atakami na te usługi, którym często poświęca się mniej uwagi podczas zabezpieczania. Jest to szczególnie ważne, gdyż jak mówi popularne stwierdzenie: system jest tak bezpieczny, jak bezpieczny jest jego najsłabszy element.

2.2.3. Obliczenia rozproszone

Wraz z rozwojem sieci komputerowych, a w szczególności Internetu, popularne stało się rozpraszanie obliczeń, które zamiast jednego komputera, wykonuje wiele maszyn, których sumaryczna moc jest często większa od pojedynczych superkomputerów. Wiele ośrodków badawczych stworzyło projekty (np. *SETI@home* [52], *LHC@home* [22]) pozwalające ofiarować część czasu procesora na potrzeby obliczeń naukowych.

Podobny, choć komercyjny, jest projekt *CPUShare* [5], który pozwala odsprzedać część mocy obliczeniowej procesora na rzecz jednostki, która ją wykupiła w serwisie. W związku z tym, że aplikacja kliencka ściąga i wykonuje na komputerze kod nieznanego pochodzenia, istnieją uzasadnione obawy, iż uruchamiany kod może być złośliwy i może służyć np. do tworzenia sieci typu *botnet*. Na potrzeby *CPU-Share* zaprojektowano i zaimplementowano więc mechanizm ochrony usług systemowych — *seccomp* (opisany szerzej w 2.3.4), który ściśle ogranicza zbiór dostępnych wywołań. Dzięki takiemu zastosowaniu mechanizmu ochrony ABI obliczenia rozproszone mogą być w pełni bezpieczne dla komputerów klienckich.

2.2.4. Aplikacje uprzywilejowane

Wielokrotnie podkreślanym [20, 47], lecz wciąż nierozwiązanym, problemem bezpieczeństwa w systemie *Linux* są aplikacje, które muszą być uruchamiane z prawami *roota*, takie jak np. *XServer*. Wysoka podatność na błędy obszernych aplikacji, a także oczywiste korzyści związane z ich przejęciem przez intruzów czynią je częstym celem ataków. Próbą zaradzenia powyższemu problemowi jest mechanizm *Capabilities* (zob. podrozdział 1.3.3). Istnieją jednak sytuacje, w których ziarnistość ograniczeń *Capabilities* jest zbyt mała. W takich przypadkach mechanizm ochrony wywołań systemowych daje możliwość uzupełnienia brakujących reguł. Zestawienie tych dwóch mechanizmów nadal nie jest rozwiązaniem idealnym, gdyż aplikacja

taka wciąż działa z uprawnieniami o wiele wyższymi niż proces zwykłego użytkownika. Niestety obecnie, poza silną izolacją, nie istnieją alternatywne rozwiązania wyżej wymienionego problemu.

2.2.5. Sandboxing

Sandboxing jest techniką izolowania procesów, która zyskuje w ostatnim czasie coraz większą popularność. Niewielu programistów dostrzega jednak korzyści z piaskownic budowanych na bazie mechanizmów ochrony wywołań systemowych — aplikacji wykorzystujących opisywany mechanizm jest bardzo mało.

Jednym z nielicznych wyjątków jest przeglądarka Chrome (zob. podrozdział 1.3.6), która wykorzystuje autorski *sandbox*, opisany szerzej w podrozdziale 2.3.7. Część przeglądarki odpowiedzialna za renderowanie (renderer) składa się z wielu złożonych modułów takich jak np. parser czy interpreter, które wykonują duże ilości niezaufanego kodu nieznanego pochodzenia. Proces renderera jest więc wyizolowany — umieszczony w piaskownicy. Z tego powodu wykorzystanie istniejących w kodzie błędów nie umożliwia interakcji z systemem operacyjnym — wszelkie próby zostają przechwycone przez opisywany *sandbox*.

Jak widać, mechanizm kontroli dostępu do systemowego ABI może w skuteczny sposób zostać zastosowany w aplikacjach, które wykonują kod nieznanego pochodzenia. Od wielu lat, programy tego typu (m.in. *Adobe Reader*, *Adobe Flash*, *Microsoft Office*) są z powodzeniem wykorzystywane do ataków na systemy operacyjne właśnie z powodu niekontrolowanego wykonania niezaufanego kodu. Umieszczenie procesów wykonawczych w piaskownicy chroniącej usługi systemowe mogłoby znacząco utrudnić lub wręcz uniemożliwić skuteczny atak tego typu.

2.3. Istniejące rozwiązania

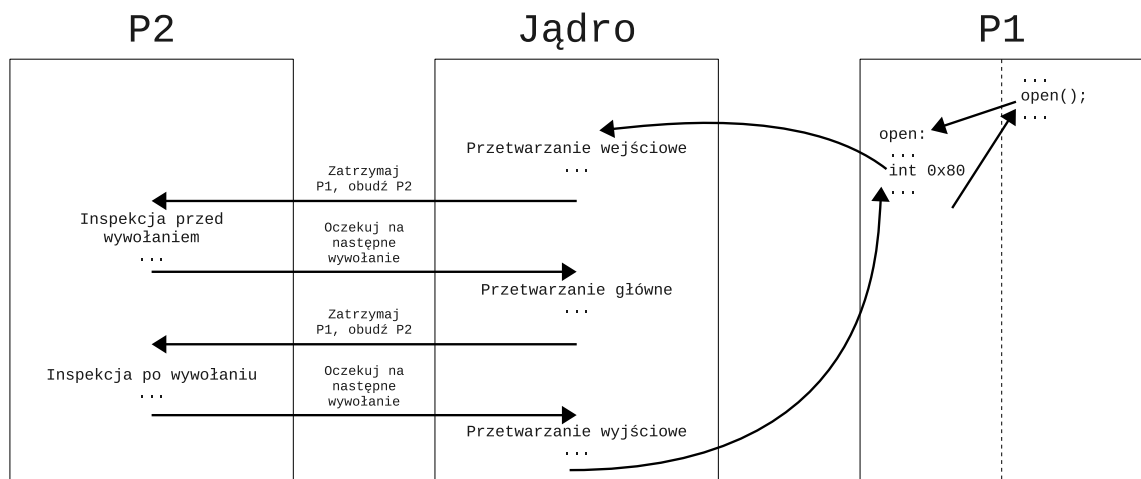
Tematyka ochrony wywołań systemowych doczekała się wielu opracowań teoretycznych oraz praktycznych implementacji. Wśród znanych inicjatyw badawczych warto wspomnieć o projektach takich jak: *Capsicum* [59], *Janus* [23], *Ostia* [24], *Paladin* [54]. Mnogość istniejących prac dotyczących opisywanego mechanizmu wskazuje, iż zagadnienie nie jest trywialne, a ponadto odgrywa istotną rolę w dziedzinie bezpieczeństwa systemów operacyjnych. Interesujące jest to, że większość implementacji stworzona została dla odmiany systemu *UNIX* — *BSD*. Należy przypuszczać, iż jest to skutek priorytetowego traktowania bezpieczeństwa w systemach tego typu.

Dla systemu *Linux* również przygotowano wiele mechanizmów, które (często pośrednio) realizują ochronę dostępu do wywołań systemowych. Ze względu na bardzo dynamiczny rozwój tego systemu operacyjnego, nierzadko cechują się one innowacyjnością i różnorodnym podejściem do implementacji. Poniżej zaprezentowano zestawienie wspomnianych mechanizmów: ich sposób działania, zalety oraz wady. Należy wspomnieć, że opis ten nie jest całościowy, a zawiera jedynie popularne oraz aktywnie rozwijane pomysły.

2.3.1. ptrace

Ptrace() to wywołanie systemowe pozwalające śledzić (ang. *trace*) wykonanie wskazanego programu. Proces śledzący może sterować wykonaniem procesu śledzonego poprzez odczyt i modyfikację jego pamięci oraz rejestrów. Jedną z możliwości *ptrace()* jest przechwytywanie wywołań systemowych wykonywanych przez proces, a zatem może ono służyć do implementacji mechanizmu kontroli dostępu do usług jądra.

Projekty (takie jak np. *SUBTERFUGUE* [7, 33]) wykorzystujące *ptrace()* do implementacji ochrony wywołań systemowych, działają w architekturze z procesem nadzorującym. W tak skonstruowanym mechanizmie, proces-nadzorca podłącza się za pomocą *ptrace()* do procesu, który ma być śledzony. Od tego momentu kontrolowany proces w chwili przejścia w tryb jądra zostaje każdorazowo wstrzymany, a sterowanie jest przekazywane do procesu nadzorującego. W tej sytuacji nadzorca może odczytać stan procesu, a następnie, zgodnie z zaprogramowaną polityką dostępu, może podjąć odpowiednie działanie — np. zezwolić na kontynuowanie wykonania, gdy wywołanie jest dozwolone albo zakończyć proces, jeśli żądanie jest zabronione.



Rysunek 2.1: Poglądowy schemat architektury mechanizmu ochrony wywołań systemowych opartej na *ptrace()*. P1 — proces nadzorowany, P2 — proces-nadzorca. Na podstawie: [33].

Zastosowanie *ptrace()* do kontroli dostępu do wywołań systemowych pozwala na dużą elastyczność — śledzone mogą być tylko wybrane procesy, a dla każdego z nich zaprogramowana może być inna polityka ochrony. Ponadto zastosowanie tej techniki nie wymaga modyfikacji kodu programów, które mają być poddane kontroli.

Największą wadą mechanizmów skonstruowanych przy pomocy *ptrace()* jest bardzo wysoki narzut wydajnościowy. Opóźnienia w wykonaniu śledzonych programów wynikają z dwóch czynników: czasu potrzebnego na transfer sterowania pomiędzy procesem-nadzorcą a procesem podlegającym kontroli oraz z czasu wymaganego do sprawdzenia polityki dostępu. Pierwszy, w zależności od tego jak często kontrolowana aplikacja wywołuje usługi jądra, może powodować nawet stukrotny narzut na czas wykonania [60]. Drugi ze składników, dla prostych reguł jest natomiast zanedbywalnie mały.

Kolejną wadą wywołania *ptrace()* jest niewygodny dla programisty interfejs oraz liczne ograniczenia z nim związane. Niektóre problemy związane z *ptrace()* mają zostać usunięte wraz z wprowadzeniem do jądra modyfikacji *utrace* [10].

2.3.2. *systrace*

Systrace to modyfikacja jądra *Linuksa* przygotowana specjalnie w celu kontroli dostępu do wywołań systemowych [46]. Mechanizm ten, poza *Linuksem*, został zaimplementowany dla systemów z rodziny *BSD* oraz *Mac OS X*. *Systrace*, podobnie jak rozwiązania oparte na wywołaniu *ptrace()*, korzysta z dodatkowego procesu — demona przechowującego politykę dostępu. Opisywany mechanizm nie wykorzystuje jednak śledzenia procesów — w niskopoziomowym kodzie przestrzeni jądra dodano sprawdzanie uprawnień procesu do wywołań. W celu minimalizacji narzutu wydajnościowego *systrace* stosuje podejście hybrydowe — proste zasady dostępu typu „zawsze zezwalaj” oraz „zawsze zabraniaj” są przechowywane w pamięci jądra i ich sprawdzenia nie muszą być konsultowane z demonem. Bardziej skomplikowane zasady są z kolei przesyłane do demona (komunikacja odbywa się za pomocą pseudourządzenia */dev/systrace*) w celu weryfikacji.

Do zalet *systrace*, oprócz eleganckiej architektury, należy zaliczyć możliwość automatycznej generacji (wyuczenia) polityki oraz dodatkowe funkcje takie jak tymczasowe nadawanie uprawnień *roota*, które jest rozszerzeniem idei mechanizmu *Capabilities* [46]. Pomimo tego *systrace* nigdy nie zyskał znacznej popularności i nie został włączony do standardowego jądra *Linuksa*. Do takiego stanu przyczyniły się odkryte w *systrace* błędy typu *time-of-check-to-time-of-use* (*TOCTOU*) [58], których naprawienie jest trudne i znacząco komplikuje architekturę mechanizmu. Wykorzystanie błędów *TOCTOU* polega na zmianie zawartości pamięci pomiędzy zdarzeniami sprawdzenia warunku i użycia rezultatu tej operacji. W przypadku mechanizmów ochrony wywołań systemowych wykorzystanie takiego błędu pozwala na ominięcie sprawdzania polityki dostępu. Kolejnym istotnym mankamentem *systrace* jest wydajność. Zgodnie z badaniami [46, 60], w zależności od specyfiki nadzorowanego programu, narzut wydajnościowy może wynosić od kilku, do nawet 30%. W zastosowaniach wymagających wysokiej responsywności jest to efektywność niezadowalająca. Na koniec należy zaznaczyć, że ostatnia wersja *systrace* pochodzi z 2009 roku i nieznane są dalsze losy projektu.

2.3.3. *Capabilities*

Capabilities opisane dokładniej w podrozdziale 1.3.3 pośrednio są mechanizmem ochrony dostępu do wywołań systemowych. Ustawiane dla procesów możliwości sprowadzają się w praktyce do zezwalania na pewne wywołania systemowe i wzbraniania ich. Specyfika tej techniki polega jednak na tym, iż są to wywołania systemowe dostępne jedynie dla *roota*. Mechanizmu tego można więc używać jedynie do ograniczania wywołań systemowych dla procesów wykonujących się z uprawnieniami superużytkownika. Kolejnym istotnym problemem bywa niewystarczająca ziarnistość — danej możliwości nie zawsze odpowiada pojedyncze wywołanie systemowe.

Capabilities są dojrzałym elementem jądra systemu *Linux*, a ich użycie jest bardzo proste i nie powoduje żadnych narzutów wydajnościowych. Pomimo tego,

mechanizm ten nie cieszy się dużą popularnością — najprawdopodobniej wynika to z ignorancji bądź przyzwyczajęń administratorów systemu *Linux*.

2.3.4. *seccomp*

Mechanizm *seccomp* to prosty *sandbox* zaprojektowany na potrzeby projektu *CPUShare* (zob. podrozdział 2.2.3). Użycie tego modułu pozwala procesowi na umieszczenie się w piaskownicy zezwalającej jedynie na wywołania *exit()*, *sigreturn()*, *read()*, *write()*. Próba wykonania jakiegokolwiek innego wywołania systemowego kończy się zakończeniem procesu za pomocą sygnału *SIGKILL*. Mechanizm aktywowany jest za pomocą wywołania *prctl()* i działanie to jest nieodwracalne — opuszczenie piaskownicy przez proces jest niemożliwe.

Seccomp jest jedynym bezpośrednim mechanizmem ochrony dostępu do wywołań systemowych obecnym standardowo w jądrze *Linuxa*. Opisywany moduł nie zyskał jednak znacznej popularności — oprócz wspomnianego projektu *CPUShare* posłużył programistom przeglądarki *Google Chrome* do stworzenia dedykowanej dla tego produktu piaskownicy (zob. podrozdział 2.3.7) — inne użycia *seccomp* nie są znane. Od momentu opublikowania modułu podjęto kilka prób modyfikacji mechanizmu, które miały na celu zwiększenie jego użyteczności. Niestety żadna z nich nie została jeszcze zaakceptowana przez społeczność programistów [13].

Niska popularność *seccomp* spowodowana jest brakiem możliwości konfiguracji modułu. Ograniczenie do czterech wyżej wymienionych wywołań jest dla wielu procesów zbyt restrykcyjne — uniemożliwia im standardowe działanie. Dodatkowo, *seccomp* zaimplementowany jest przy użyciu wolnej (śledzonej) ścieżki wywołań systemowych. Powoduje to zauważalny narzut wydajnościowy. Do zalet modułu należy zaliczyć prostotę i klarowność mechanizmu oraz wspomniane perspektywy jego rozwoju.

2.3.5. *Audit syscall*

Audit syscall to rejestrator wywołań systemowych będący częścią *Linux Auditing System* — modułu jądra rejestrującego wszystkie istotne (z punktu widzenia bezpieczeństwa systemu) zdarzenia. *Audit syscall* jest narzędziem pozwalającym jedynie odnotowywać (za pomocą demona działającego w trybie użytkownika) fakt wykonania wywołania systemowego — moduł nie umożliwia podjęcia żadnych akcji. W celu wykorzystania informacji dostarczanych przez ten system często łączy się go z implementacjami mechanizmu *MAC* (zob. podrozdział 1.3.5) — najczęściej z *SELinux*.

Połączenie *Audit syscall* z *SELinux* pozwala skonfigurować politykę reakcji systemu na wykonane wywołania. Stosowanie opisywanych narzędzi nie umożliwia zatem zabronienia czy też przerwania wywołania systemowego. W kontekście ochrony wywołań systemowych jest to istotna wada tego zestawienia — skutki działania niektórych usług jądra, takich jak np. *unlink()*, czy *truncate()* nie są łatwe do odwrócenia — administratorowi bardziej zależy na wstrzymaniu dostępu do nich niż na niwelowaniu efektów. *Audit syscall* jest wydajnym mechanizmem, który nie powoduje narzutu wydajnościowego dla procesów, dla których audyt nie został skonfigurowany. Istotną jego wadą jest jednak dosyć skomplikowana konfiguracja oparta o tworzenie reguł, która dodatkowo komplikuje się, gdy zachodzi potrzeba współpracy z *SELinux*.

Z wyżej wymienionych powodów *Audit syscall* jest nieefektywnym mechanizmem ochrony wywołań systemowych. Z drugiej strony, gdy istotne jest jedynie rejestrowanie wykonanych usług jądra, mechanizm ten jest aktualnie najwydajniejszym i najbardziej niezawodnym narzędziem dla tej klasy problemów.

2.3.6. LD_PRELOAD

LD_PRELOAD to zmienna środowiskowa pozwalająca na wymuszenie na konsolidatorze czasu uruchomienia (ang. *runtime linking process*) załadowania obiektu współdzielonego (ang. *shared object*), zanim załadowane zostaną jakiekolwiek inne biblioteki dynamiczne. Zastosowanie tej zmiennej umożliwia więc przesłonięcie dowolnych symboli bibliotek dołączanych przy starcie programu. Typowo jedną z bibliotek ładowanych przy uruchamianiu programu jest biblioteka standardowa języka C — *glibc*. Wykorzystując *LD_PRELOAD* można zatem stworzyć „opakowania” (ang. *wrapper functions*) funkcji (z *glibc*) realizujących wywołania systemowe. Funkcje opakowujące mogą zawierać kod, który pozwala na śledzenie i kontrolę dostępu do usług jądra, czyli implementację mechanizmu ochrony wywołań systemowych. Dla aplikacji z dostępnym kodem źródłowym możliwe jest również całkowite podmienienie biblioteki *glibc* już na etapie kompilacji, co pozwala na eliminację konieczności użycia zmiennej *LD_PRELOAD* przy uruchamianiu procesu. Technika ta jest wykorzystywana do implementacji mechanizmu ochrony wywołań systemowych w projektach takich jak *Plash* [48], czy *Ostia*.

Wykorzystanie *LD_PRELOAD* do implementacji mechanizmu ochrony ABI ma kilka istotnych zalet. Przede wszystkim rozwiązanie takie jest w pełni elastyczne. W zależności od konkretnych potrzeb kod funkcji opakowującej może po prostu zabraniać danego wywołania lub wręcz przeciwnie — decyzja o dopuszczeniu wykonania usług może być podjęta na podstawie złożonej polityki opartej np. na wartościach argumentów. Kolejną zaletą jest to, że narzut wydajnościowy tak skonstruowanego mechanizmu zależy wyłącznie od złożoności procesów decyzyjnych, a dla wywołań niemodyfikowanych jest on zerowy. Warto też zaznaczyć, iż zastosowanie *LD_PRELOAD* nie wymaga modyfikacji kodu aplikacji, która będzie kontrolowana.

Najpoważniejsza wada rozwiązania wynika wprost z uproszczonego podejścia do zagadnienia — zmieniane są wyłącznie wywołania biblioteczne, a nie kod jądra — mechanizm ten zatem w żaden sposób nie chroni przed niskopoziomowymi wywołaniami kodu systemowego (np. bezpośrednie użycie instrukcji *int 0x80*). Technika ta nie jest więc zabezpieczeniem przed atakami polegającymi na wstrzyknięciu kodu do działającej aplikacji, takimi jak przepełnienie bufora. Ładowanie kodu kontrolującego wyłącznie przy uruchomieniu uniemożliwia podłączenie mechanizmu do już działających aplikacji, niemożliwe jest również wyłączenie mechanizmu bez ponownego uruchamiania procesu. Ostatnią istotną wadą zastosowania zmiennej *LD_PRELOAD* jest to, że działa ona wyłącznie dla aplikacji, do których biblioteka *glibc* jest linkowana dynamicznie.

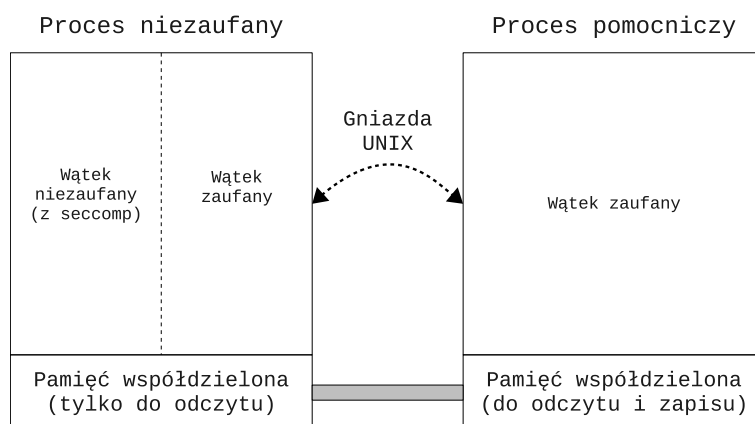
2.3.7. Chromium sandbox

Chromium sandbox to piaskownica zaprojektowana specjalnie na potrzeby przeglądarki *Google Chrome* w celu izolacji procesów renderujących. Pomimo tak wyspecjalizowanego zastosowania, budowa tego mechanizmu jest na tyle generyczna, iż może on być z powodzeniem zastosowanych dla innych procesów — na bazie

Chromium sandbox powstało kilka projektów (np. *seccomp nurse* [1]) umożliwiających zastosowanie tego narzędzia z innymi programami.

Każdy proces znajdujący się w opisywanej piaskownicy posiada dwa wątki: niezaufany oraz zaufany. Wątek niezaufany wykonuje zasadnicze obliczenia i działa w trybie *seccomp* (zob. podrozdział 2.3.4) — zbiór bezpośrednio dostępnych wywołań systemowych jest bardzo wąski. Pozostałe usługi jądra mogą być wywoływane na rzecz wątku niezaufanego jedynie za pomocą zdalnych wywołań procedur RPC skierowanych do wątku zaufanego poprzez gniazda IPC. Zadaniem wątku zaufanego jest weryfikacja żądanych usług oraz ewentualne ich wykonanie. W tym miejscu warto zauważyć, że wątek zaufany działa w potencjalnie wrogim środowisku, gdyż współdzieli pamięć z wątkiem niezaufanym — w związku z tym została podjęta decyzja, aby wątek zaufany wykorzystywał jedynie rejestry procesora (które są chronione sprzętowo) i nie wykorzystywał wspólnej pamięci.

Jednym z istotnych problemów implementacji takiego rozwiązania jest zamiana wywołań systemowych na wywołania RPC. Zastosowanie podmienionej biblioteki jest niebezpieczne ze względów opisany w podrozdziale 2.3.6. Kolejną możliwością jest modyfikacja kodu procesu renderującego, lecz takie rozwiązanie jest wyjątkowo uciążliwe w utrzymaniu i konserwacji. Ostatecznie została więc podjęta decyzja o stworzeniu prostego deasemblera, który ma odczytywać binarny kod wątku niezaufanego i zamieniać wszelkie wywołania systemowe na wywołania RPC.



Rysunek 2.2: Poglądowy schemat architektury *Chromium sandbox*. Na podstawie: [1].

Ostatnim problemem implementacyjnym jest kwestia błędów typu *TOCTOU* (zob. podrozdział 2.3.2). Argumenty wywołań systemowych przekazywane poprzez wskaźniki mogą ulec zmianie w czasie pomiędzy sprawdzeniem dostępu a rzeczywistym wykonaniem usługi. W celu rozwiązania tego problemu dla wszystkich wątków renderera tworzony jest pojedynczy, zaufany proces pomocniczy, który współdzieli kilka stron pamięci z zaufanymi wątkami. Wszelkie wywołania systemowe, które nie mogą być zweryfikowane przez wątek zaufany — albo z powodu zbyt wysokiego stopnia komplikacji, albo z powodu odwołań do pamięci — są delegowane do procesu pomocniczego. Proces pomocniczy podczas obsługi żądania kopiuje wszystkie argumenty do swojej pamięci tak, by nie mogły być one zmienione przez niezaufany wątek. Wyniki pracy procesu pomocniczego są następnie umieszczane w pamięci współdzielonej do odczytu dla wątku zaufanego.

Chromium sandbox jest mechanizmem niezwykle wyrafinowanym, ale także bardzo złożonym, przez co może być trudny w zastosowaniu i rozwoju. Oczywistą wadą tego rozwiązania jest również niska wydajność. Mnogość zastosowanych mechanizmów IPC sprawia, iż rozwiązanie to jest jednym z najwolniejszych wśród wszystkich przeanalizowanych.

2.3.8. Patch out

Patch out to prymitywna technika polegająca na usunięciu z kodu źródłowego jądra fragmentów odpowiedzialnych za niechciane w systemie wywołania. Została ona zaproponowana, lecz jeszcze niewdrożona do systemu *Chrome OS*. Wedle założeń [26] z kodu jądra powinny zostać usunięte wywołania nieprzydatne (jak np. *ptrace()*) oraz takie, które uznawane są za niedojrzałe bądź niedostatecznie przetestowane (m.in. *vmsplice()*).

Istotną zaletą tego rozwiązania jest stuprocentowa skuteczność — kod, który nie istnieje, z całą pewnością nie zostanie wykonany. Oczywistą wadą opisywanego podejścia jest natomiast całkowity brak elastyczności — usunięcie wywołania dotyczy wszystkich procesów, a ponowne udostępnienie usługi wymaga rekompilacji jądra. Technika ta jest zatem bardzo kłopotliwa w zastosowaniu.

Prezentowane rozwiązanie można wykorzystać również w inny sposób. Wielokrotnie kompilując kod jądra z różnymi podzbiorami dostępnych wywołań, można wytworzyć wiele binarnych wersji systemu, które mogą odpowiadać różnym profilom wykonywanych aplikacji. Następnie, za pomocą projektu *User Mode Linux*, możliwe jest uruchomienie wspomnianych wielu wersji *Linuksa* jako procesów w systemie pełniącym rolę nadzorcy. W ten sposób uzyskuje się wiele zwirtualizowanych środowisk uruchomieniowych dla procesów. Podejście to, kosztem większej komplikacji zastosowania i trudności w konfiguracji, daje daleko większą elastyczność niż zastosowanie pojedynczego jądra z usuniętymi wywołaniami.

2.4. Podsumowanie i wnioski

Na podstawie przeprowadzonej analizy widać, iż mechanizm kontroli dostępu do wywołań systemowych jest istotnym i uznanym środkiem bezpieczeństwa systemów operacyjnych. Mnogość potencjalnych i już istniejących zastosowań, często bardzo różnych, wskazuje, że tematyka nie jest jeszcze wyczerpana, a w zagadnieniu tym wciąż tkwi potencjał.

Przeanalizowane w rozdziale implementacje prezentują odmienne podejścia do postawionego problemu — wynika to z faktu, iż projektanci poszczególnych rozwiązań różne właściwości uznawali za priorytetowe. Wśród cech, które mogą podlegać ocenie w każdym z rozważanych przypadków, można wyróżnić: niezawodność, wydajność, konfigurowalność, łatwość użycia oraz prostotę budowy. Stworzenie mechanizmu idealnego — tj. takiego, który łączy i realizuje wszystkie powyższe postulaty — jest niemożliwe. Oczywiście istnieje zatem konieczność dokonywania wyborów i pójścia na ustępstwa: rozwiązanie o bogatych możliwościach konfiguracji traci na wydajności, łatwość użycia często wyklucza prostą budowę, a techniki niezawodne często mają ograniczony obszar zastosowania.

Wszystkie opisywane mechanizmy charakteryzuje skrajne wręcz faworyzowanie pewnych cech ponad pozostałe: *Capabilities* oraz *seccomp* są niezawodne, jednak

mogą być zastosowane w bardzo wąskim zakresie; *ptrace* i *systrace* są rozwiązaniami wygodnymi, jednak bardzo niewydajnymi; *LD_PRELOAD* oferuje szerokie możliwości i dobrą efektywność, niestety kosztem bezpieczeństwa; *Audit syscall*, *Chromium sandbox* oraz *Patch out*, pomimo swoich zalet są niezwykle niewygodne w użyciu.

Żadna z opisywanych implementacji nie zyskała szerokiego uznania i nie weszła do powszechnego użytku. Należy się więc zastanowić, czy taki stan rzeczy jest skutkiem wyżej opisanej, bezkompromisowej realizacji pomysłu ochrony systemowego ABI, czy może nie istnieje realna potrzeba wykorzystania tego typu zabezpieczeń? Analiza głosów społeczności [20] — codziennych użytkowników i programistów *Linuksa* — daje jasną odpowiedź: systemowi wciąż brakuje mechanizmu kontroli dostępu do wywołań systemowych, który byłby zarówno wydajny jak i funkcjonalny — istniejące rozwiązania są niezadowalające. Potencjalnymi głównymi odbiorcami takiego podsystemu są duże projekty jak np. *Ubuntu*, *QEMU*, *vsftpd* czy *Chrome OS*. Zapotrzebowanie jest więc rzeczywiste, a mechanizm taki może zyskać szerokie grono użytkowników.

W związku z powyższym przedmiotem następnych rozdziałów są projekt, implementacja oraz testowanie autorskiego mechanizmu kontroli dostępu do wywołań systemowych, który może sprostać wymaganiom stawianym przez społeczność *Linuksa*.

3. Projekt

W poprzednim rozdziale przedstawione zostały przesłanki z których wynika konieczność stworzenia mechanizmu kontroli dostępu do wywołań systemowych. Po przeanalizowaniu owych czynników, czyli potrzeb potencjalnych użytkowników, przypadków użycia, a także istniejących, aczkolwiek niewystarczających implementacji takiego rozwiązania, zadaniem inżyniera jest stworzenie produktu, który sprostą stawianym mu oczekiwaniom.

Proces tworzenia systemu informatycznego standardowo składa się z trzech etapów: projektowania, implementacji oraz testowania. Każdy z tych etapów jest istotnym składnikiem sukcesu całego przedsięwzięcia, jednakże to od fazy projektowania zależy najwięcej. Opisywany etap rozpoczyna się od zebrania i spisania wymagań projektowych, które następnie poddawane są wnikliwej analizie. Wnioski wynikające z przeprowadzonego badania służą w dalszej kolejności do podjęcia decyzji projektowych, które wyznaczają kierunek przebiegu kolejnego etapu, czyli implementacji. W fazie testowania efekty pracy projektanta wykorzystywane są do weryfikacji i oceny finalnego produktu. Etap projektowania systemu jest więc niezbędnym elementem przedsięwzięcia i ma niebagatelny wpływ na jego całokształt.

Bieżący rozdział w naturalny sposób odzwierciedla wyżej wymienioną kolejność działań. Pierwsza jego część składa się z przedstawienia wymagań zarówno funkcjonalnych, jak i niefunkcjonalnych. W następnym kroku wymagania te zostały dokładnie przeanalizowane, a na podstawie wykonanej pracy podjęto wstępne decyzje projektowe.

3.1. Wymagania

Zgodnie z dobrymi praktykami inżynierii oprogramowania [35], każdy projekt informatyczny należy zacząć od zebrania, zgrupowania i szczegółowego opisu wymagań — zarówno funkcjonalnych jak i niefunkcjonalnych. Rzetelne wykonanie tego etapu przedsięwzięcia znacząco ułatwia przeprowadzenie wszystkich kolejnych faz: projektowania architektury, implementacji, testowania i weryfikacji.

Wymagania projektowe w jasny i precyzyjny sposób określają, co ma być efektem pracy nad projektem. Wytyczne te sterują więc procesem projektowania, który w rezultacie ma doprowadzić do implementacji rozwiązania realizującego stawiane mu cele. Dodatkowo, w fazie wykonywania projektu, wymagania stanowią narzędzie, które w łatwy sposób pozwala ocenić stopień zaawansowania prac.

W fazie testowania wymagania stanowią podstawę do przeprowadzenia testów. Dobrze skonstruowany przypadek testowy powinien realizować część postawionych wcześniej wymogów. W idealnej sytuacji zestaw wszystkich testów powinien w pełni pokrywać zbiór wymagań projektowych. W takim wypadku, na podstawie przeprowadzonych testów, możliwa jest wymierna ocena stopnia realizacji przedsięwzięcia oraz podjęcie dalszych decyzji projektowych. Opracowane na początku projektu

wymagania mają zatem zasadniczy wpływ na ścieżkę jego realizacji i końcowy sukces.

Zdefiniowanie wymogów w oparciu o rzetelny przegląd dziedziny zwiększa szanse na powodzenie przedsięwzięcia. Z tego powodu wymagania dla rozważanego projektu mechanizmu kontroli dostępu do wywołań systemowych w zdecydowanej większości wynikają wprost z badania opisanego w rozdziale 2. Część z nich odzwierciedla konkretne potrzeby oraz przypadki użycia, inne z kolei są efektem analizy istniejących rozwiązań, a w szczególności ich wad i przeszkód w szerokim zastosowaniu. Pozostałe wymagania służą natomiast stworzeniu logicznej i spójnej wizji projektu.

3.1.1. Wymagania funkcjonalne

Lista wymagań funkcjonalnych dla rozważanego projektu nie jest obszerna — zawiera ona opis możliwości mechanizmu, założenia poczynione w kwestii zapewnienia bezpieczeństwa implementacji oraz sposób interakcji z użytkownikiem. Ograniczanie możliwości pojedynczych składowych mechanizmów bezpieczeństwa jest ich cechą charakterystyczną. Mniejsza funkcjonalność, czyli prostota, oznacza potencjalnie mniej błędów. Nacisk kładzie się więc na prawidłowe i rzetelne zrealizowanie pojedynczej funkcji, a nie na bogactwie dostępnych opcji i estetycznej atrakcyjności interfejsu użytkownika. Kompleksowość usługi bezpieczeństwa osiąga się natomiast poprzez złożenie wielu autonomicznych mechanizmów, narzędzi oraz technik (zob. rozdział 1).

1. Możliwości kontroli

Projektowany mechanizm bezpieczeństwa powinien dawać możliwość wzbraniania dostępu do konkretnego podzbioru wywołań systemowych. Podzbiór ten może składać się z dowolnych wywołań dostępnych w danej kompilacji jądra. Kontroli mogą podlegać wszystkie jednostki wykonawcze nazywane w *Linuksie* zadaniami (ang. *task*). Dla każdego zadania powinna istnieć możliwość skonfigurowania indywidualnego podzbioru zabronionych wywołań systemowych.

2. Poziom bezpieczeństwa

Aby mechanizm był skuteczny (bezpieczny) powinny być spełnione następujące warunki:

- a) zadanie nieuprzywilejowane (bez praw administratora) nie może modyfikować polityki dostępu do wywołań systemowych,
- b) zadanie nie powinno mieć możliwości pominięcia kontroli dostępu (jeśli reguły są zdefiniowane i aktywne).

3. Możliwości konfiguracji

Konstrukcja mechanizmu powinna umożliwiać modyfikację polityki dostępu w dowolnym momencie dla dowolnych zadań.

4. Interfejs użytkownika

Interfejs użytkownika projektowanego mechanizmu powinien umożliwiać podgląd aktualnych ustawień dostępowych dla zadań oraz wykonywanie czynności konfiguracyjnych opisanych w wymaganiu funkcjonalnym 3. Ponadto interfejs użytkownika powinien umożliwiać przeprowadzenie wyżej

wymienionych operacji za pomocą powłoki systemowej — konfiguracja poprzez skrypty powinna być możliwie łatwa.

5. Współpraca z innymi mechanizmami bezpieczeństwa

Zastosowanie projektowanego mechanizmu nie powinno wykluczać użycia innych mechanizmów bezpieczeństwa. W szczególności powinna być możliwa współpraca z kontenerami procesów tzn. należy zapewnić możliwość skonfigurowania jednolitej polityki dla wszystkich zadań w danym kontenerze.

3.1.2. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne zawierają charakterystykę oczekiwanej wydajności, która jest jednym z najważniejszych kryteriów oceny rozważanego projektu. Dodatkowo programowanie dla *Linuksa* — systemu wieloplatformowego — wymaga rozważenia kwestii przenoszenia kodu pomiędzy różnymi architekturami. Ostatnim opisanym w wymaganiach zagadnieniem jest wykorzystanie narzędzi wdrożeniowych, charakterystycznych dla tego systemu.

1. Efekt pracy

W zależności od sposobu realizacji projektu może on przyjąć postać modyfikacji jądra, programu działającego w przestrzeni użytkownika lub dowolnego połączenia tych technik. W przypadku kodu dla jądra systemu *Linux* powinien on przyjąć formę łaty (ang. *patch*) w formacie opisanym w dokumencie *Submitting Patches* [41]. Dodatkowo powinna istnieć możliwość zrezygnowania z kompilowania modułu poprzez ustawienie odpowiedniej wartości za pomocą narzędzia konfiguracji kompilacji jądra *menuconfig*, kod programu natomiast powinien być przygotowany w standardzie zgodnym z pakietem *Autotools*.

2. Wersja jądra

Oprogramowanie powinno działać z jądrem w wersji 3.1. W związku z dynamicznym, niezgodnym wstecz API jądra systemu *Linux* nie można poczynić żadnych założeń co do innych wersji.

3. Architektura sprzętu

Projekt powinien być przygotowany dla architektur *x86* oraz *x86-64*.

4. Przenośność kodu

Kod projektu powinien być napisany w sposób umożliwiający łatwe przeniesienie do architektur innych niż wymienione w wymaganiu niefunkcjonalnym 3. Oznacza to konieczność zminimalizowania ilości kodu pisanego w języku zależnym od platformy sprzętowej.

5. Wydajność

Zgodnie z aktualnymi trendami w rozwoju jądra *Linuksa* mechanizmy bezpieczeństwa, nawet jeśli są skompilowane, ale nieaktywne, nie powinny powodować widocznego narzutu wydajnościowego. Z tego powodu, w kwestii wydajności powinny zostać spełnione dwa warunki:

- a) zadania, dla których nie zostały zdefiniowane żadne ograniczenia wywołań systemowych, powinny być spowolnione o nie więcej niż 1%,

- b) zadania ze skonfigurowaną kontrolą dostępu do usług jądra powinny być spowolnione o nie więcej niż 5%.

6. Konwencja i styl kodu źródłowego

Kod źródłowy powinien być zgodny z wytycznymi dokumentu *Coding Style* zawartego w [41].

3.2. Decyzje projektowe

Wnikliwa analiza sformułowanych w podrozdziale 3.1 wymagań w połączeniu z wiedzą dziedzinową opisaną w rozdziale 2 umożliwiają podjęcie wstępnych decyzji projektowych. Postanowienia poczynione na tym etapie mają charakter ogólny — stanowią jedynie wskazówki dla etapu implementacji. Specyfika projektu nie pozwala na dokładne zaplanowanie rozwiązania, gdyż pewne cechy, takie jak wydajność czy właściwa synchronizacja, mogą być ocenione dopiero po przygotowaniu i przetestowaniu prototypu. Tworzenie oprogramowania systemowego wymusza minimalizację poziomów pośredniości oraz abstrakcji. Niewielka zmiana idei działania może zatem pociągać za sobą znaczące zmiany w implementacji. Niezależnie więc od doświadczenia projektanta i pomimo stabilnych wymagań, wstępne założenia projektowe mogą ulec zmianie na skutek weryfikacji w procesie implementacji.

Do podstawowych decyzji na tym etapie należy określenie ogólnej architektury rozwiązania. Wybór ten ma najistotniejszy wpływ na całokształt produktu i dalszy przebieg projektowania. Należy zatem szczegółowo rozważyć istniejące możliwości i na podstawie wiedzy dziedzinowej oraz analizy wymagań dokonać właściwego wyboru. Po określeniu architektury rozwiązania warto wykonać uproszczony model mechanizmu tak, aby zobrazować interakcje modułu z pozostałą częścią systemu operacyjnego. Ostatni etap podejmowania decyzji projektowych polega na odpowiednim uszczegółowieniu kwestii, które we wspomnianym modelu koncepcyjnym zostały pominięte.

3.2.1. Architektura mechanizmu

Mechanizm ochrony dostępu do wywołań systemowych można zaimplementować w przestrzeni jądra, w przestrzeni użytkownika albo w sposób hybrydowy. Pierwszą decyzją projektową jest wybór jednej z wyżej wspomnianych architektur. Jest to o tyle istotne, iż wybór ten ma bezpośredni wpływ na wszystkie pozostałe czynniki związane z implementacją. Dla wybrania odpowiedniej architektury kluczowe znaczenie ma analiza z rozdziału 2. Opisane tam rozwiązania prezentują różnorodne podejścia, dzięki czemu można wnioskować na temat wad i zalet każdej z dostępnych możliwości. Poniżej przedstawiono, w kontekście spełnienia wymagań projektowych, możliwe podejścia architektoniczne i wynikającą z tych rozważań decyzję.

Realizacja w przestrzeni użytkownika

Implementacja mechanizmu w całości w przestrzeni użytkownika jest zadaniem niezwykle skomplikowanym, wymagającym zastosowania zaawansowanych technik (zob. podrozdział 2.3.7). Pomimo tego, takie podejście ma pewne zalety, związane głównie ze specyfiką programowania w przestrzeni użytkownika m.in. elastyczność, dowolne wykorzystanie narzędzi i bibliotek oraz łatwe testowanie.

Niestety, narzut wydajnościowy, wynikający z przełączania kontekstu oraz komunikacji międzyprocesowej, uniemożliwia implementację mechanizmu działającego w przestrzeni użytkownika, który spełniałby restrykcyjne wymaganie WNF-5¹. Dodatkowo rozwiązania działające poza jądrem cechują się niższym bezpieczeństwem i odpornością na ataki (zob. podrozdział 2.3.6), co znacząco utrudnia rzeczywiste spełnienie wymagania WF-2.

Realizacja w przestrzeni jądra

Realizacja mechanizmu w postaci modyfikacji jądra oferuje szerokie, praktycznie nieograniczone możliwości. Programista może bowiem swobodnie wpływać na dowolne aspekty działania procesów, w tym na wywołania systemowe. Implementacja w przestrzeni jądra w szerokim zakresie umożliwia również optymalizację, co może znacząco ułatwić spełnienie wymagania WNF-5. Co więcej, jądro można uznać za stabilny element systemu operacyjnego, a więc wszelkie mechanizmy działające jako usługi w tym obszarze są niepodatne na ataki pochodzące z przestrzeni użytkownika (zob. podrozdział 2.3.4). Umożliwia to zatem spełnienie wymagania bezpieczeństwa opisanego w WF-2.

Tworzenie modułów jądra jest zadaniem trudnym, wymagającym dużej wiedzy na temat specyfiki zagadnienia oraz budowy danego systemu operacyjnego. Co więcej, środowisko jądra jest nieodporne na błędy, a więc wszelkie omyłki programistyczne mogą mieć fatalny dla systemu skutek — z tego też powodu testowanie takiego kodu jest skomplikowanym zagadnieniem. Zmiany dokonywane w jądrze, w szczególności w kodzie wywołań systemowych, mogą wymagać modyfikacji kodu zależnego od platformy sprzętowej — może to utrudniać spełnienie wymagań WNF-3 oraz WNF-4. Ostatnią wadą opisywanego podejścia jest niewygodne wdrażanie — zastosowanie tak zaimplementowanego mechanizmu wymaga podmiany jądra lub załadowania odpowiedniego modułu ładowalnego (ang. *Loadable Kernel Module*, LKM).

Realizacja hybrydowa

Realizacja hybrydowa pozwala na połączenie cech implementacji w przestrzeni jądra i użytkownika. W takim zestawieniu, kod jądra zawiera minimalną funkcjonalność polegającą na zabranianiu wykonania wywołania bądź zezwalaniu na nie. Kod w przestrzeni użytkownika zajmuje się natomiast skomplikowanymi zagadnieniami, takimi jak generowanie i sprawdzanie polityki dostępu. Elementy komunikują się za pomocą mechanizmów systemowych pozwalających na współpracę pomiędzy przestrzeniami.

Mogłoby się zatem wydawać, iż rozwiązanie kompromisowe pozwala w idealny sposób połączyć zalety obu wcześniej opisanych opcji. Niestety, jak pokazała analiza istniejących mechanizmów wykorzystujących podejście hybrydowe, architektura taka pociąga za sobą również wady wspomnianych realizacji (zob. podrozdział 2.3.1 oraz 2.3.2). Narzut wydajnościowy związany z przekazaniem kontroli z jądra do przestrzeni użytkownika uniemożliwia spełnienie wymagania WNF-5. Mechanizmy wykonane w takiej architekturze mogą również być bardziej podatne na błędy, co zagraża realizacji wymagania WF-2.

¹ WNF-X oznacza wymaganie niefunkcjonalne o numerze X, analogicznie WF-Y oznacza wymaganie funkcjonalne o numerze Y.

Decyzja

W obliczu istotnych wad, którymi obarczone są realizacje hybrydowe i w przestrzeni użytkownika, decyzja odnośnie architektury rozwiązania jest prosta. Mechanizm musi zostać zaimplementowany w całości w przestrzeni jądra. Podjęcie trudu wynikającego z charakterystyki wybranego podejścia powinno się opłacić, gdyż taki wybór daje najszersze spektrum możliwości, a więc i największą szansę na spełnienie sformułowanych wymagań.

Uzasadnieniem słuszności podjętej decyzji, poza powyższą analizą, może być również fakt, iż jedynym mechanizmem ochrony ABI, który został powszechnie uznany w środowisku programistów i użytkowników systemu *Linux* jest *seccomp* (zob. podrozdział 2.3.4), w całości zrealizowany jako moduł jądra. Wszelkie inne implementacje, właśnie ze względu na wady wynikające z przyjętej architektury, nie zyskały zaufania i nie znalazły się w kanonie mechanizmów bezpieczeństwa systemu *Linux*.

Podjęcie decyzji dotyczącej ogólnej architektury rozwiązania pozwala na prowadzenie dalszych, bardziej szczegółowych rozważań projektowych. W kolejnym punkcie przedstawiono ogólny model działania projektowanego modułu, który w naturalny sposób wynika z przyjętej metody realizacji.

3.2.2. Model modułu

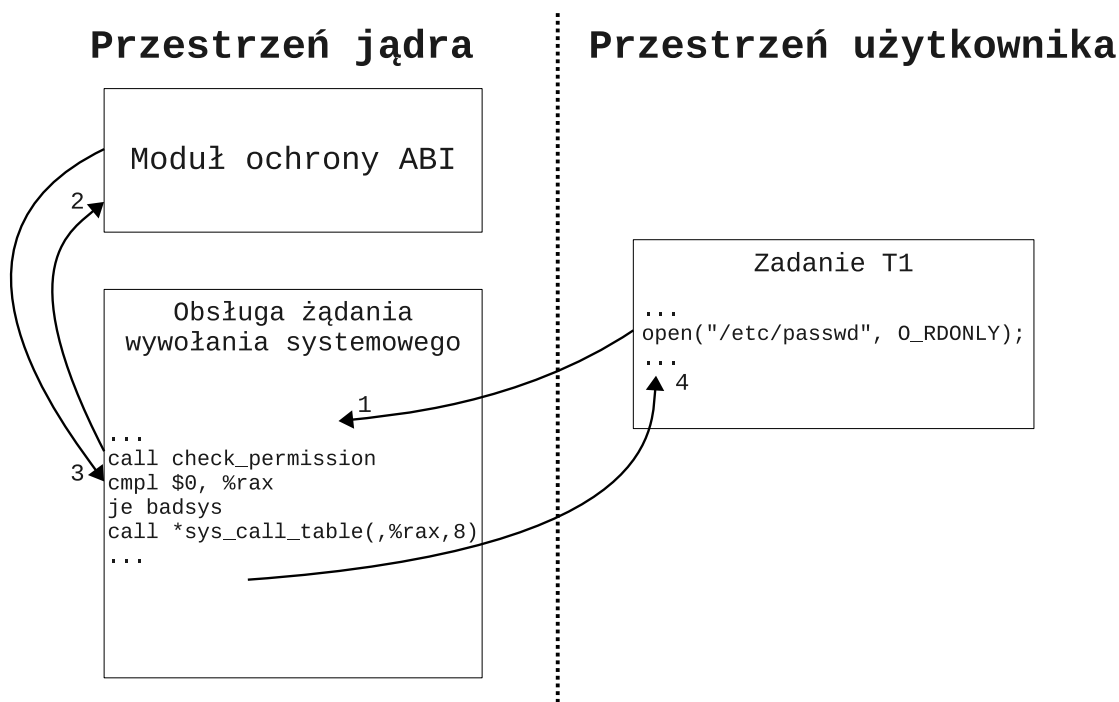
Wybór zarysu architektury rozwiązania pozwala na przejście z poziomu rozważań ogólnych, niekoniecznie związanych z konkretnym systemem operacyjnym, na poziom projektowania ściśle połączony z *Linuxem*. W tej fazie konieczna jest wnikliwa analiza kodu źródłowego *Linuksa* dotyczącego przedmiotu implementacji — wywołań systemowych. Możliwość wglądu w kod źródłowy oprogramowania jest unikalną cechą projektów *open source*. Dzięki niej programista nie musi polegać wyłącznie na dokumentacji czy opracowaniach literaturowych, często niedokładnych bądź nieaktualnych. Dokonanie przeglądu kodu źródłowego umożliwia stworzenie poglądowego modelu modułu, którego elementy można następnie kolejno rozwijać.

Wnikliwa analiza kodu obsługi żądań wywołań systemowych pokazuje, iż jest on całkowicie zależny od platformy sprzętowej. Wykonanie w przestrzeni użytkownika charakterystycznej dla platformy instrukcji przejścia w tryb jądra powoduje natychmiastowe wykonanie niskopoziomowego kodu obsługi. Nie istnieją żadne funkcje opakowujące, pośredniczące (ang. *proxy functions*) ani wywołania zwrotne (ang. *callbacks*), które pozwalałyby na podpięcie (ang. *hook*) w sposób generyczny (wspólny dla wszystkich platform) do kodu realizującego wywołania systemowe.

W związku z powyższym, aby spełnić wymaganie bezpieczeństwa rozwiązania WF-2b, należy w sposób uważny umieścić kod kontrolujący dostęp we wszystkich istniejących niskopoziomowych ścieżkach obsługi wywołań systemowych. Taka decyzja utrudnia realizację wymagań architektonicznych WNF-3 oraz WNF-4, jednak jest ona w tym wypadku uzasadniona. Wszelkie inne podejścia wymagałyby zidentyfikowania (potencjalnie bardzo wielu) wspólnych punktów w kodzie wyższych warstw realizacji wywołań systemowych, przyczyniając się tym samym do powstawania błędów bezpieczeństwa wynikających z przeoczeń.

Sformułowanie powyższego wniosku pozwala stworzyć model koncepcyjny modułu. Model taki przedstawia jedynie sposób, w jaki moduł współdziała z resztą

systemu, ukrywając szczegóły implementacyjne, które zostaną rozwinięte w dalszej fazie projektowania.



Rysunek 3.1: Model architektury i sposobu działania projektowanego mechanizmu. 1 — skok do kodu obsługi żądania wywołania systemowego, 2 — sprawdzenie uprawnień zadania do żadanego wywołania, 3 — powrót z modułu, 4 — powrót do wykonania zadania.

Ogólną ideę działania mechanizmu można opisać w kilku krokach. W momencie, gdy zadanie wykonuje pewne wywołanie systemowe, na skutek rozkazu przejścia w tryb jądra (np. *sysenter*), realizowany jest skok do kodu obsługi żądania. W tym miejscu następuje wykonanie funkcji modułu, która sprawdza, czy dana usługa jest dla aktualnego zadania dozwolona. Zwrócony wynik kontroli jest interpretowany przez kod obsługi żądania i na tej podstawie podejmowana jest decyzja — czy należy kontynuować standardowe wykonywanie wywołania, czy też należy zwrócić błąd.

Przedstawiony powyżej model ukrywa złożoność implementacyjną mechanizmu. W następnym kroku należy więc rozwinąć zagadnienie wewnętrznej budowy modułu, a w szczególności sposobu sprawdzania polityki oraz konkretnej realizacji w systemie *Linux*.

3.2.3. Sprawdzanie polityki

Zgodnie z wymaganiem WF-1 dla dowolnego zadania można ustalić dowolny podzbiór zabronionych wywołań systemowych. W związku z tym z każdym istniejącym w systemie zadaniem musi być związana pewna struktura danych przechowująca informacje o dozwolonych (ew. zabronionych) usługach jądra. Dobór odpowiedniego sposobu przechowywania tych danych ma kluczowe znaczenie zarówno dla wydajności, jak i wygody programowania mechanizmu.

Przekładając powyższy opis słowny na algorytmiczny: w mechanizmie wymagana jest konstrukcja, która na podstawie zadanej liczby naturalnej (odpowiadającej numerowi wywołania systemowego), zwróci wartość logiczną (odpowiadającą przyzwoleniu bądź wzbronieniu danego wywołania). Potrzebna jest więc prosta tablica asocjacyjna, wiążąca liczby naturalne z wartościami binarnymi (logicznymi). Tablice asocjacyjne implementuje się typowo albo przy pomocy struktur wiązanych (np. drzew), albo za pomocą struktur z dostępem losowym (np. tablic). Żeby wykonać wyбору odpowiedniej techniki realizacji należy przeanalizować cechy obu rozwiązań.

Implementacja tablicy asocjacyjnej zezwoleń

Struktury wiązane, takie jak drzewa czerwono–czarne oraz drzewa AVL, są często wykorzystywane do implementacji tablic asocjacyjnych. Wśród zalet istotnych w kontekście realizacji modułu, należy wymienić przede wszystkim wysoką elastyczność. Struktury wiązane dają większą swobodę manipulacji, która mogłaby się przydać, gdy potrzebne byłoby stworzenie dodatkowego mapowania numerów wywołań lub grupowanie ich w pewien szczególny sposób. Inną zaletą struktur wiązanych jest ich kompaktowość. Gdy zbiory zabronionych wywołań są niewielkie, drzewo przechowujące te informacje zajmuje zwykle mniej pamięci niż struktury o dostępie swobodnym.

Struktury wskaźnikowe posiadają jednak kilka istotnych wad. Najważniejszą z nich jest znacznie niższa wydajność podstawowych operacji w porównaniu ze strukturami tablicowymi. Odnalezienie wartości dla danego klucza w drzewie zbalansowanym ma złożoność czasową $O(\log n)$, co w połączeniu z koniecznością wielokrotnych odwołań do pamięci może skutkować niedopuszczalnym narzutem wydajnościowym. Kolejną wadą zaawansowanych struktur drzewiastych jest wysoka złożoność implementacyjna. Może ona prowadzić do powstawania trudnych do wykrycia błędów. Na zakończenie należy wspomnieć, iż struktury wiązane wymagają przemyślanej, często skomplikowanej synchronizacji dostępu, co dodatkowo utrudnia prawidłową implementację oraz może skutkować wysokim narzutem wydajnościowym.

Struktury o dostępie losowym nie oferują takiej elastyczności jak struktury wiązane, jednak są bardzo wydajne. Typowe operacje, które będą szeroko wykorzystywane w rozważanym module takie jak sprawdzenie bądź wstawienie wartości dla danego klucza, mają złożoność $O(1)$. Ponadto, użycie tego typu struktur jest bardzo łatwe, a zapewnienie równoległego dostępu często nie wymaga żadnych dodatkowych zabiegów.

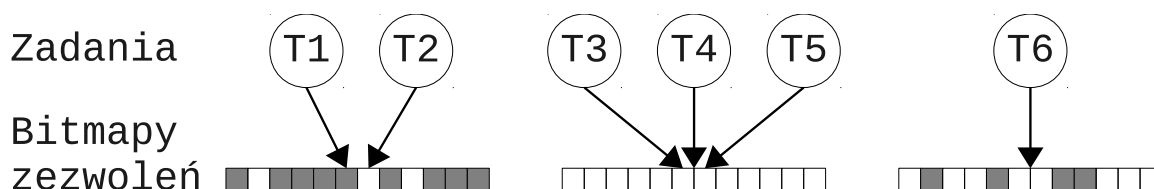
Wywołania systemowe w Linuksie są numerowane w sposób ciągły, od 0 do pewnej liczby charakterystycznej dla danej architektury sprzętowej. Liczba ta ma postać makra `NR_syscalls` o wartości znanej w momencie kompilacji. Cechy te pozwalają stwierdzić, iż w rozważanym module, najlepszą strukturą o dostępie losowym jest mapa bitowa o wielkości co najmniej `NR_syscalls`. Reprezentacja taka zapewnia oszczędność pamięci oraz łatwość użycia — bit na zadanej pozycji określa, czy wywołanie jest dozwolone, czy nie.

Powyższa analiza pozwala stwierdzić, iż lepszym wyborem jest struktura o dostępie losowym — bitmapa. Decyzja ta zapewnia maksymalnie wysoką wydajność oraz łatwość użycia. Układ tablicy wywołań systemowych *Linuksa* typowo nie ulega zmianie podczas działania systemu, nie potrzebne są więc żadne dodatkowe właściwości oferowane przez struktury wiązane.

Powiązanie zadania z bitmapą zezwoleń

Pozostała do rozważenia kwestią jest sposób powiązania zadania z odpowiadającą mu mapą bitową zezwoleń. Decyzja ta ma jednak znacznie drugorzędne. W momencie wykonywania wywołania systemowego, jądro gwarantuje, iż w globalnej dla danego procesora zmiennej *current* znajduje się wskaźnik do bloku zarządzania aktualnie wykonywanego zadania — wskaźnik do struktury *task_struct*. W najgorszym wypadku, można więc dodać do tej struktury dodatkowe pole wskazujące na bitmapę przypisaną do danego zadania.

Implementując połączenia zadań z mapami bitowymi, warto zwrócić uwagę na możliwość optymalizacji. Prawdopodobne jest, że będą istnieć pewne podgrupy wątków i procesów posiadające takie same ograniczenia wywołań, w szczególności część zadań nie będzie podlegała żadnym restrykcjom. W takim wypadku warto, aby bitmapy zezwoleń były współdzielone przez zadania o tym samym zestawie dostępnych wywołań systemowych. Takie podejście pozwoli zaoszczędzić pamięć oraz zwiększy lokalność odwołań.



Rysunek 3.2: Idea współdzielenia bitmap zezwoleń. Zadania *T1*, *T2* są uruchomione wewnątrz piaskownicy; zadania *T3*, *T4*, *T5* działają swobodnie; zadanie *T6* ma nałożone niewielkie ograniczenia.

3.2.4. Technika realizacji modułu

System *Linux* cechuje się dużą dojrzałością. Skutkiem ponad dwudziestoletniej ewolucji kodu źródłowego jest niespotykane gdzie indziej bogactwo API poszczególnych podsystemów. Zaimplementowanie nowych pomysłów może zawsze odbyć się na co najmniej kilka sposobów. Dobór najbardziej odpowiedniego z nich jest zadaniem niełatwym, wymagającym od projektanta przekrojowej wiedzy na temat jądra. Podczas rozważań należy wziąć pod uwagę udogodnienia, jakie dany podsystem udostępnia programiście i to, jak mogą one pomóc w realizacji pomysłu. Dodatkowo idea projektu powinna być zgodna z wybranym sposobem realizacji. Próby wykorzystywania niewłaściwych mechanizmów jądra w celu ułatwienia pracy prowadzą do dezorganizacji kodu źródłowego i są szczególnie niepożądane [8].

Projektowany mechanizm również może zostać zaimplementowany na wiele sposobów. Wybór jednej z możliwości przesądzi o tym, jak moduł będzie współistniał z innymi elementami jądra, jak będzie „widziany” w przestrzeni użytkownika oraz co najważniejsze — jak będzie wyglądał interfejs udostępniony użytkownikowi. Poniżej przedstawiono analizę technik implementacji, które wydają się najlepiej dopasowane do realizowanego projektu.

procfs

Jeden ze sposobów realizacji modułu polega na rozszerzeniu mechanizmu służącego do zarządzania zadaniami. Realizacja taka wymaga modyfikacji kodu odpowiedzialnego za tworzenie i usuwanie procesów — w tych miejscach należy dodać kod zarządzania bitmapami zezwoleń. Interfejs użytkownika odpowiedzialny za modyfikację uprawnień można zaimplementować przy użyciu wirtualnego systemu plików *procfs*, używanego czasem do modyfikacji stanu wykonujących się zadań.

Taka implementacja jest prosta w założeniach, jednak trudna w realizacji. Programista nie otrzymuje żadnego wsparcia od podsystemu w kwestii obsługi tworzenia i usuwania zadań. Modyfikacja tych fragmentów kodu jądra może być trudna i dodatkowo może niepotrzebnie komplikować i tak już złożoną logikę obsługi wątków i procesów.

Zaletą takiej implementacji jest to, że spełnia ona wszystkie stawiane wymagania. Mechanizm otrzymuje odpowiednie możliwości konfiguracji, interfejs użytkownika jest wygodny i powszechnie znany.

Wirtualne urządzenie

Realizacja mechanizmu w postaci urządzenia wirtualnego jest bardzo podobna do opisanej wcześniej metody z użyciem systemu plików *procfs*. W tym wypadku jednak interakcja z użytkownikiem polega na komunikacji z urządzeniem wirtualnym. Wymiana danych konfiguracyjnych może odbywać się za pomocą wywołań *read()* oraz *write()* albo za pomocą *ioctl()*.

Podejście takie, pomimo że jest bardziej eleganckie, wciąż wymaga modyfikacji kodu związanego z zarządzaniem zadaniami w jądrze. Technika ta ma więc takie same zalety i wady jak rozwiązania z użyciem *procfs*. Warto wspomnieć, że podejście z zastosowaniem wirtualnego urządzenia zostało z powodzeniem zastosowane do realizacji mechanizmu *systrace* (zob. podrozdział 2.3.2).

Przestrzeń nazw

Mechanizm przestrzeni nazw (zob. podrozdział 1.3.3) wykorzystywany jest do tworzenia warstwy pośredniości w dostępie do zasobów nazwanych. Zrealizowanie projektowanego rozwiązania w postaci przestrzeni nazw podzbioru wywołań systemowych doskonale wpisuje się zatem w ogólną ideę podsystemu *namespaces*. Przestrzenie nazw umożliwiają logiczne grupowanie zadań, a więc możliwe jest łatwe zrealizowanie dziedziczenia ograniczeń oraz zastosowanie optymalizacji opisanej w punkcie 3.2.3. Mechanizm *namespaces* używany jest do realizacji kontenerów procesów. Wybór takiego podejścia zapewnia doskonałą realizację wymagań WF-5.

Niestety implementacja nowej przestrzeni nazw jest trudna, gdyż wymaga wpięcia się w różne ścieżki wykonania kodu jądra, a dostępne API wspierające realizację jest ubogie. Obecnie mechanizm *namespaces* nie posiada żadnej natywnej metody konfiguracji — wszelkie zabiegi konfiguracyjne odbywają się podczas tworzenia nowej przestrzeni. W związku z tym, aby możliwa była realizacja wymagań WF-3 oraz WF-4, mechanizm *namespaces* musi wykorzystać zewnętrzny podsystem komunikacji użytkownika z jądrem, taki jak wspomniany wcześniej system plików *procfs* czy *debugfs*.

Podsystem grup kontrolnych

Projektowany mechanizm może również zostać zrealizowany w postaci kontrolera dostępu do zasobów *Control Groups* (zob. podrozdział 1.3.4). Wywołania systemowe zdecydowanie można zaliczyć do programowych zasobów systemu operacyjnego, a więc skorzystanie z tego właśnie mechanizmu jest uzasadnione.

Istotną cechą grup kontrolnych jest ich hierarchiczna struktura. Podobnie jak w przypadku przestrzeni nazw, jest więc możliwe grupowanie zadań, które pozwala na zastosowanie optymalnego przydziału bitmap. Bardzo ważną właściwością *Control Groups* jest łatwa konfiguracja, która odbywa się poprzez własny system plików. W celu zrealizowania wymagań WF-3 oraz WF-4 nie trzeba więc stosować żadnych dodatkowych mechanizmów. Podsystem *Control Groups* udostępnia również bogate API, co znacznie ułatwia programowanie. W celu stworzenia nowego podsystemu grup kontrolnych wystarczy zaimplementować zbiór funkcji oraz zarejestrować go w module. Nie ma więc potrzeby modyfikacji kodu jądra. Podobnie jak *namespaces*, mechanizm grup kontrolnych jako składnik kontenerów doskonale integruje się z innymi mechanizmami bezpieczeństwa.

Decyzja

Przeanalizowane techniki realizacji oczywiście nie wyczerpują listy rozwiązań. Dla przykładu warto chociażby wspomnieć o możliwości rozwinięcia mechanizmów takich jak: *LSM*, *prctl*, *Audit syscall* czy *ftrace*. Perspektywa wykorzystania powyższych modułów jest jednak nieatrakcyjna i została odrzucona na wstępnym etapie analizy ze względu na nieefektywność, trudność w zastosowaniu lub niemożność spełnienia wszystkich wymagań projektowych.

Z opisanych podsystemów najodpowiedniejsze do implementacji mechanizmu wydają się przestrzenie nazw i grupy kontrolne. Ich konstrukcja i przeznaczenie wydaje się najbardziej pasować do idei realizowanego projektu. Mechanizmy te są do siebie dosyć podobne, jednakże obecność wspieranego, wygodnego sposobu konfiguracji oraz bogate API przesądza o wyborze *Control Groups* jako narzędzia realizacji projektu.

3.2.5. Podsumowanie podjętych decyzji

Na zakończenie rozważań projektowych warto podsumować podjęte decyzje:

1. Projektowany mechanizm zostanie w całości zrealizowany w przestrzeni jądra. Takie podejście ma zapewnić bezpieczeństwo oraz szerokie możliwości optymalizacji rozwiązania.
2. Sprawdzanie polityki dostępu będzie inicjowane w każdej możliwej ścieżce żądania wywołania systemowego. W ten sposób żaden proces ani wątek nie będzie w stanie ominąć kontroli uprawnień.
3. Reguły dostępu będą przechowywane w mapie bitowej. Wartość bitu pod numerem danego wywołania określa czy żądanie jest dozwolone, czy zabronione. Zastosowanie takiej struktury danych zapewnia dowolną ziarnistość polityki dostępu, a także wysoką wydajność i łatwość w implementacji. Dana grupa procesów, o tych samych uprawnieniach, powinna posiadać współdzieloną mapę bitową — w ten sposób oszczędzana jest pamięć.
4. Mechanizm zostanie zrealizowany jako podsystem *Control Groups*. Taki wybór, ze względu na rozbudowane API grup kontrolnych, znacząco ułatwi implementację. *Control Groups* udostępniają natywny sposób konfiguracji podsystemów,

nie wymagają modyfikacji jądra i bardzo dobrze integrują się z innymi mechanizmami bezpieczeństwa.

Sformułowanie konkretnych decyzji umożliwia zakończenie projektowania. Postanowienia poczynione na tym etapie będą wytycznymi dla kolejnej fazy, czyli implementacji. Opis tego stadium projektu jest przedmiotem kolejnego rozdziału.

4. Implementacja

Implementacja to etap, w którym następuje rzeczywista realizacja projektu informatycznego. Efektami pracy w tej fazie są najczęściej kod (np. aplikacji, testów, skryptów wdrożeniowych) oraz dokumentacja projektowa (np. opis modułów, instrukcja użytkownika). Działania podejmowane podczas implementacji wynikają wprost z decyzji poczynionych w toku projektowania, a końcowe wyniki tej fazy są oceniane w trakcie testów. Bardzo często bywa jednak, że implementacja zawiera w sobie elementy ponownego planowania (np. na skutek zmieniających się wymagań bądź wcześniejszych błędów) oraz weryfikacji (np. w celu wykazania, że moduł działa poprawnie). Wykonawczy charakter tej fazy oraz wspomniana złożoność procesowa czynią implementację najbardziej czasochłonnym etapem projektu.

Implementacja modułu dla systemu operacyjnego *Linux* znacząco różni się od programowania innych projektów informatycznych — np. aplikacji webowych. Najwięcej czasu w procesie tworzenia oprogramowania działającego w przestrzeni jądra nie zajmuje faktyczne pisanie kodu, a wszelkie działania do tego prowadzące. Do takich czynności można zaliczyć: poznawanie iteracyjne i analizę otoczenia wykonawczego modułu, testowanie kilku podejść do danego problemu i wybór najlepszego, weryfikacja poprawności działania rozwiązania oraz dyskusje ze społecznością programistów i użytkowników na temat obranego sposobu realizacji. Specyfika i odmienność projektu czyni zatem proces implementacji szczególnie interesującym i wartym przeanalizowania.

Niniejszy rozdział w pierwszej części poświęcony jest budowie i programowaniu mechanizmu grup kontrolnych. W kolejnych podrozdziałach na podstawie zaprojektowanego modułu przedstawiono cechy specyficzne programowania w przestrzeni jądra oraz narzędzia, które zostały wykorzystane podczas prac. Ostatnią częścią rozdziału jest instrukcja obsługi: wdrożenia i użycia.

4.1. Control Groups

Mechanizm grup kontrolnych początkowo przyjęty został przez społeczność bardzo sceptycznie [14]. Kolejne podsystemy znacznie komplikowały kod istotnych elementów systemu takich jak m.in. planista. Dodatkowo w *Linuksie* od dawna dostępne były różne techniki ograniczania dostępu do zasobów, takie jak *security limits* oraz *rlimits*. Z drugiej strony grupy kontrolne oferowały całkowicie nowatorskie i wygodne podejście do zarządzania zasobami. Społeczność programistów *Linuksa* została więc podzielona na zwolenników i przeciwników włączenia *Control Groups* do głównej linii jądra. Finalnie, pod wpływem nacisków entuzjastów mechanizmu, został on włączony do *Linuksa* w wersji 2.6.24.

Jak okazało się później, decyzja o przyjęciu *cgroups* okazała się jak najbardziej słuszna — mechanizm został dobrze przyjęty przez użytkowników i doczekał się

wielu wdrożeń, w szczególności do zastosowań kontenerowych. Sukces grup kontrolnych spowodował, że ciągle rośnie liczba podsystemów — ochrona dostępu do zasobów oferowana przez *Control Groups* staje się coraz bardziej kompletna i precyzyjna.

Duża liczba podsystemów spowodowana jest również tym, iż programowanie nowych grup kontrolnych jest wygodne, a zarazem oferuje duże możliwości. *Linux* doczekał się więc własnego, łatwego sposobu kontrolowania dostępu do zasobów, który spełnia oczekiwania zarówno programistów jądra, jak i użytkowników. Cechy te doprowadziły do podjęcia decyzji (zob. podrozdział 3.2.4) o realizacji mechanizmu ochrony wywołań systemowych właśnie w postaci podsystemu *cgroup*.

4.1.1. Budowa

Pomimo wygodnego, dobrze udokumentowanego API, zaimplementowanie podsystemu wymaga podstawowej znajomości budowy całego mechanizmu. Grupy kontrolne zorganizowane są hierarchicznie, w sposób analogiczny do struktury plików i katalogów. Z tego powodu, naturalnym sposobem reprezentacji mechanizmu w systemie jest wirtualny system plików. Katalogom i podkatalogom w takim układzie odpowiadają grupy oraz podgrupy, natomiast zawarte w nich pliki odpowiadają za konfigurację i przyłączanie zadań.

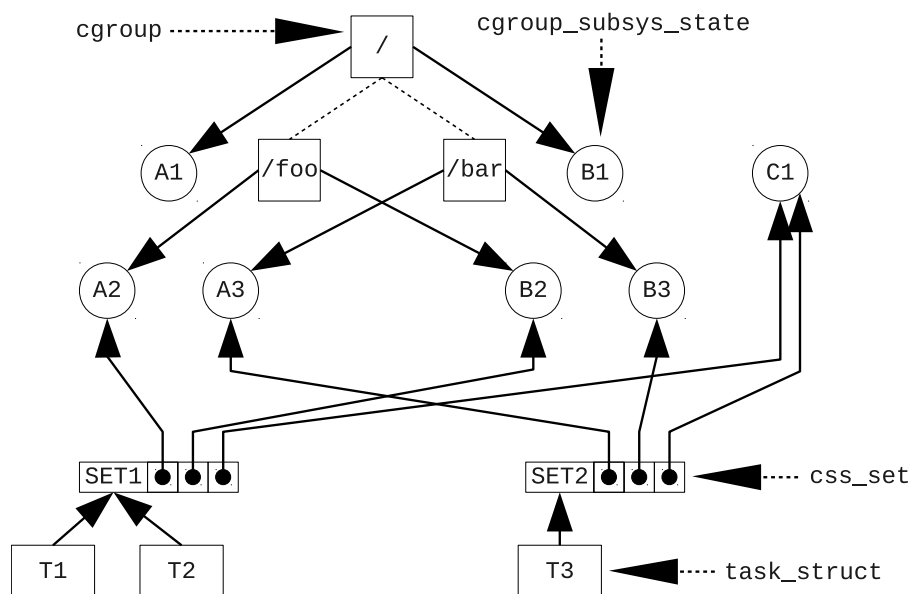
Z każdą hierarchią skojarzony jest co najmniej jeden podsystem — kontroler danego rodzaju zasobów. W danej hierarchii zamontowanych może być wiele podsystemów, można również każdy podsystem umieścić w oddzielnej strukturze. Dane zadanie może więc należeć do dowolnej liczby grup kontrolnych pod warunkiem, że grupy te dotyczą innego zasobu. W tym miejscu warto dodać, że każde zadanie należy do pewnej grupy kontrolnej danego podsystemu — domyślnie jest to grupa kontrolna najwyższego poziomu. Dzieje się tak nawet wtedy, gdy podsystem nie został jeszcze skojarzony z żadną hierarchią.

Organizacja kompozytowa ułatwia zarządzanie regułami dostępu. W typowej sytuacji podsystemy wymuszają, aby zadania znajdujące się w podgrupach były objęte większymi ograniczeniami niż zdefiniowane w grupach dla nich nadrzędnych. W ten sposób polityka dostępu do zasobów dla zadań z danej grupy kontrolnej jest zwykle określona jako iloczyn logiczny aktualnych uprawnień oraz uprawnień wszystkich grup-przodków.

Grupa kontrolna jest reprezentowana w jądrze w postaci struktury *cgroup*. Informacje w niej zawarte nie są w żaden sposób związane z zarządzanymi zasobami, a jedynie z umiejscowieniem grupy w hierarchii i systemie plików. Każdemu egzemplarzowi struktury *cgroup* przypisana jest pewna liczba obiektów definiujących stany podsystemów zamontowanych w danej hierarchii. Stan jest reprezentowany przez strukturę *cgroup_subsys_state*, która również nie przechowuje żadnych informacji o zasobach, a jest jedynie elementem pośredniczącym w dostępie do grupy kontrolnej. To na programiście podsystemu spoczywa obowiązek powiązania informacji potrzebnych do kontrolowania dostępu do zasobów z egzemplarzem struktury *cgroup_subsys_state*. Dzięki temu za każdym razem, gdy następuje odwołanie do konkretnego podsystemu, możliwe jest odzyskanie tych informacji.

Łączenie zadań z konkretnymi grupami kontrolnymi odbywa się poprzez utrzymywanie w strukturze *task_struct* listy wskazań do odpowiednich stanów podsystemów. Przydzielenie każdemu zadaniu własnej listy wskaźników do egzemplarzy *cgroup_subsys_state* mogłoby okazać się nieefektywne pamięciowo — można

bowiem wyróżnić podzbiory zadań, które należą dokładnie do tych samych grup kontrolnych, a więc są związane z identycznymi stanami podsystemów. W związku z powyższym listy wskazań są współdzielone przez zadania o takiej samej przynależności do grup. Wspólne dla zadań podzbiory są reprezentowane przez struktury `css_set`.



Rysunek 4.1: Przykładowa struktura mechanizmu *Control Groups*. Dla jądra skompilowane są trzy podsystemy — A, B, C. Podsystemy A i B zamontowane są w pojedynczej hierarchii. Główna grupa zawiera dwie podgrupy *foo* oraz *bar* ze skojarzonymi stanami podsystemów, odpowiednio A2 i B2 oraz A3 i B3. Zadania T1 oraz T2 znajdują się w podgrupie *foo*, natomiast zadanie T3 jest w podgrupie *bar*. Podsystem C nie jest zamontowany, więc wszystkie zadanie są związane ze stanem C1. Na podstawie: [45].

4.1.2. Implementacja podsystemu — API

Implementacja nowego podsystemu mechanizmu *Control Groups* standardowo składa się z trzech etapów. Są to: zdefiniowanie funkcji charakterystycznych podsystemu, stworzenie podpięć do kodu obsługi danego zasobu oraz dodanie podsystemu do tablicy mechanizmu. Wszystkie te kroki są niezbędne, aby stworzyć funkcjonalny podsystem zarządzania zasobami.

Funkcje podsystemu

Funkcje charakterystyczne są wykorzystywane do zarządzania podsystemem i jego grupami kontrolnymi. Procedury te mają postać wywołań zwrotnych wykonywanych podczas pewnego zdarzenia np. dołączenia nowego zadania do grupy kontrolnej. Zbiór tych wywołań przekazywany jest w postaci struktury `cgroup_subsys` zawierającej wskazania do zdefiniowanych przez programistę funkcji. Nie wszystkie procedury zwrotne muszą być zdefiniowane. Dotyczy to jedynie tych potrzebnych podsystemowi do działania. Poniżej zaprezentowano opis wszystkich dostępnych funkcji charakterystycznych.

create()

Funkcja *create()* wywoływana jest podczas tworzenia nowej grupy kontrolnej. Procedura ta jest więc naturalnym miejscem do umieszczenia wszelkiego kodu inicjalizującego i sprawdzającego uprawnienia. W opisywanym mechanizmie funkcja zawiera kod alokujący bitmapę wiązaną z nowo tworzoną grupą. Przygotowana mapa bitowa jest następnie wypełniana. Jeśli tworzona grupa jest grupą najwyższego poziomu to wszystkie bity są ustawiane, w przeciwnym wypadku kopiowana jest zawartość mapy bitowej grupy nadrzędnej. W ten sposób grupy podrzędne otrzymują uprawnienia równe grupie macierzystej.

destroy()

Procedura *destroy()* aktywowana jest w momencie usuwania grupy kontrolnej z systemu. Zagwarantowane jest, iż przed wykorzystaniem wywołania zwrotnego *destroy*, grupa kontrolna nie istnieje już w systemie plików i jest usunięta z listy rodziców. Standardowo w tej funkcji umieszcza się kod związany ze zwalnianiem zasobów używanych w danej grupie. Zaimplementowany mechanizm, dualnie do wywołania *create()*, w procedurze *destroy()* usuwa bitmapę grupy.

pre_destroy()

Funkcja *pre_destroy()* służy do sprawdzenia wszelkich uprawnień i warunków wymaganych do usunięcia grupy kontrolnej z systemu. Istnieje możliwość zwrócenia kodu błędu, który zakończy niepowodzeniem operację odłączenia grupy kontrolnej z systemu plików. W opisywanym mechanizmie procedura ta nie jest wykorzystywana.

can_attach(), can_attach_task(), pre_attach(), attach(), attach_task()

Funkcje z grupy *attach* wywoływane są w różnych momentach przyłączania istniejącego zadania do grupy kontrolnej. Tak duża liczba procedur tego typu wymagana jest do zagwarantowania spełnienia pewnych warunków (np. modyfikacji zasobów dla pojedynczych zadań) w określonym momencie. Przykładowo, *can_attach* wywoływane jest, zanim zadanie zostanie rzeczywiście umieszczone w grupie kontrolnej, natomiast *attach()* aktywowane jest w momencie, w którym zagwarantowane jest istnienie zadania na liście zadań grupy. Kolejność wywołań jest następująca: *can_attach()*, *can_attach_task()*, *pre_attach()*, *attach_task()*, *attach()*. W zaprojektowanym module została zaimplementowana funkcja *attach_task()* w celu zastosowania optymalizacji opisanej dokładniej w 4.2.1.

cancel_attach()

Procedura *cancel_attach()* wywoływana jest, gdy nie powiedzie się wykonanie funkcji z grupy *attach*. Typowo *cancel_attach()* zawiera kod odwracający działania poczynione w funkcjach *attach*. W opisywanym mechanizmie procedura ta nie została wykorzystana.

fork()

Funkcja *fork()* wywoływana jest, gdy zadanie jest tworzone wewnątrz grupy. Procedura ta może zostać wykorzystana w celu alokacji pewnych wymaganych przez grupę do działania zasobów powiązanych z pojedynczymi zadaniami. W zaimplementowanym module procedura ta nie jest używana.

post_fork()

Wywołanie *post_fork()* zostało dodane do rdzenia mechanizmu *cgroups* dla potrzeb modułu. Jest ono wywoływane w momencie tworzenia nowego zadania, jednak w przeciwieństwie do funkcji *fork()* istnieje gwarancja, że zadanie już

znajduje się na liście zadań grupy. Podjęcie takiego działania było niezbędne w celu uniknięcia wyścigu przy wdrażaniu optymalizacji opisanej w 4.2.1.

exit()

Funkcja *exit()* wywoływana jest w momencie zakończenia zadania. Typowe wykorzystanie tego wywołania polega na zwolnieniu zasobów alokowanych w funkcji *fork()* lub *attach()*. W opisywanym module funkcja ta nie jest używana.

populate()

Procedura *populate()* wykorzystywana jest do utworzenia plików konfiguracyjnych (o interfejsie *seq_file* [9]) dla danej grupy kontrolnej. Tak utworzone pliki można odczytywać lub zapisywać — w zależności od zdefiniowanych operacji. Zaprojektowany mechanizm, oprócz standardowych (zob. [41]), zawiera dwa dodatkowe pliki konfiguracyjne: *syscalls.allow* oraz *syscalls.deny*. Pliki te pozwalają na odczytywanie i zapisywanie odpowiednio dozwolonych i zabronionych wywołań systemowych. Dokładny sposób ich użycia został zaprezentowany w podrozdziale 4.4.

post_clone()

Funkcja *post_clone()* wywoływana jest podczas tworzenia grupy kontrolnej w celu inicjalizacji wszelkich parametrów wymaganych przed dołączeniem jakiegokolwiek zadania. W opisywanym module nie jest ona wykorzystywana.

bind()

Wywołanie *bind()* aktywowane jest, gdy grupa kontrolna przenoszona jest do innej hierarchii i innej grupy nadrzędnej. Zaimplementowany mechanizm nie wykorzystuje tej procedury.

Podpięcia

Funkcje charakterystyczne pozwalają na wygodne zarządzanie zasobami oraz tworzenie interfejsu użytkownika, ale to podpięcia są zasadniczym elementem każdego podsystemu. Podpięcia to kod związany z danym podsystemem, umieszczony w innych elementach jądra. Przykładowo dla podsystemu *device* podpięciem jest funkcja *devcgroup_inode_permission()*, która sprawdza uprawnienia aktualnego zadania do operacji na danej strukturze *i-node*. Wywołania tej procedury są zatem umieszczone w kodzie dostępu do struktur *i-node*.

W opisywanym module, zgodnie z ustaleniami poczynionymi na etapie projektowania, podpięcia znajdują się w niskopoziomowym kodzie obsługi żądania wywołania systemowego. Każda z rozważanych architektur wymaga dodania kodu do wszystkich możliwych ścieżek żądań, aby mechanizm można było uznać za bezpieczny. Dogłębna analiza kodu jądra *Linuksa* wykazała, iż architektura *x86* posiada dwie gałęzie obsługi wywołań systemowych, natomiast architektura *x86-64* ma ich pięć. Dla architektury *x86* są one związane z dwoma możliwymi sposobami wejścia w tryb jądra, natomiast dla *x86-64* istnieją oddzielne ścieżki dla wywołań śledzonych i niesledzonych oraz trzy ścieżki związane z trybem kompatybilności *x86*. Niezależnie jednak od miejsca, w którym znajduje się podpięcie oraz od tego, dla jakiej architektury podpięcie jest zaimplementowane, ma ono podobną postać. Algorytm działania tego elementu można przedstawić za pomocą poniższej listy kroków:

1. Sprawdzenie flagi *TIF_SYSCALL_USE_CGROUP* ze struktury *thread_info* wskazującej na to, czy z danym procesem związane są jakieś ograniczenia wywołań. Jeśli nie, należy ominąć całą pozostałą procedurę i wykonać wywołanie systemowe (zob. podrozdział 4.2.1).

2. Odłożenie na stos rejestrów, które mogą być modyfikowane w wyniku wywołania funkcji.
3. Przekazanie parametru — numeru wywołania systemowego.
4. Wywołanie funkcji podsystemu sprawdzającej uprawnienia zadania do danego wywołania.
 - a) Odnalezienie na podstawie struktury *task_struct* aktualnego zadania (zmienna *current*) grupy kontrolnej, do której należy.
 - b) Pobranie bitmapy zezwoleń *syscalls_bitmap* związanej z daną grupą kontrolną.
 - c) Sprawdzenie bitu o numerze przekazanym jako parametr.
 - d) Zwrócenie wyniku testu.
5. Określenie czy wynik jest pozytywny, czy negatywny.
6. Odtworzenie rejestrów na podstawie stosu.
7. Jeśli wynik z punktu 5. jest pozytywny, należy wykonać wywołanie, w przeciwnym wypadku należy zwrócić błąd *badsys*.

Funkcja sprawdzająca uprawnienia zadania do wywołania jest elementem podsystemu, a więc kod składający się na podpięcie ma niewielkie zobowiązania. Cecha ta jest zdecydowanie pożądana, gdyż dzięki temu pierwotny kod obsługi żądania nie jest dodatkowo komplikowany.

Tablica podsystemów

Ostatnim elementem tworzenia podsystemu *Control Groups* jest zarejestrowanie go w systemie. Może się to odbywać w dwojaki sposób: statycznie podczas kompilacji lub dynamicznie podczas działania systemu. W obu wypadkach należy jednak dodać wpis do tablicy podsystemów za pomocą makra *SUBSYS*, które wskazuje strukturę *cgroup_subsys* zawierającą funkcje charakterystyczne. Rejestracja statyczna nie wymaga żadnych dodatkowych kroków — skompilowany w jądrze podsystem działa od uruchomienia aż do wyłączenia systemu. Możliwości rejestracji dynamicznej (uruchamiania podsystemu) zrealizowane są za pomocą mechanizmu ładowalnych modułów *Linuksa*. Oprócz skompilowania w odpowiedniej postaci podsystem wykorzystujący ten mechanizm musi dodatkowo implementować dwie funkcje: inicjującą i usuwającą moduł.

Zaimplementowany mechanizm, podobnie jak większość innych podsystemów, funkcjonuje tylko w postaci statycznie rejestrowanej. Potencjalne korzyści z implementacji podsystemu w postaci *LKM* są niewielkie, bowiem niemożliwe jest całkowite uniknięcie narzutu wydajnościowego. Ponadto wykorzystanie funkcji rdzenia *Control Groups* związanych z tworzeniem nowych zadań całkowicie uniemożliwia implementację mechanizmu w postaci modułu ładowalnego.

4.2. Programowanie w przestrzeni jądra

Najbardziej interesującymi, a jednocześnie najtrudniejszymi elementami fazy implementacji są zagadnienia związane ze specyfiką programowania w przestrzeni jądra. Programowanie tego typu znacząco różni się od tworzenia aplikacji i bibliotek dla przestrzeni użytkownika, gdyż wymaga zwrócenia uwagi na nieco inne zagadnienia oraz zastosowania odmiennego podejścia do niektórych kwestii. Poniżej przedstawiono, na przykładzie zaimplementowanego mechanizmu, kilka istotnych

problemów, z którymi może spotkać się programista pracujący z kodem systemów operacyjnych.

4.2.1. Optymalizacja

System operacyjny jest oprogramowaniem, od którego wymaga się szczególnie wysokiej wydajności. Wszelkie problemy z responsywnością systemu powodują niezadowolenie użytkowników, a w przypadku zastosowań specjalistycznych i biznesowych mogą generować straty finansowe. Może się wydawać, że wraz z wzrastającą mocą obliczeniową komputerów, szczególne zabieganie o wysoką wydajność na poziomie programowym jest niepotrzebne, a nawet szkodliwe — optymalizacja kodu może zająć programistom bardzo dużo czasu, a przy tym nie generuje oczywistych korzyści biznesowych. Często bardziej opłaca się więc ulepszyć sprzęt, na którym działa system, nadrabiając tym samym niedostatki w optymalizacji programowej.

Wszechstronność systemu *Linux* powoduje, iż znajduje on zastosowanie w najróżniejszych dziedzinach. Część z nich wymaga wysokiej wydajności i ewentualne problemy nie zawsze mogą być rozwiązane poprzez skalowanie sprzętu. Dobrym przykładem takiej dziedziny są notowania giełdowe, w których *Linux* zdobywa coraz większą popularność [55]. W handlu akcjami, w szczególności w *HFT* (ang. *high-frequency trading*), działania podejmowane w odstępach milisekundowych mogą przynosić ogromne zyski bądź straty. Ponadto systemy giełdowe powinny obsługiwać ponad milion transakcji w ciągu sekundy. W takich wypadkach wydajność systemu operacyjnego i platformy ma zasadnicze znaczenie dla sprawnego funkcjonowania całego przedsięwzięcia. Widać zatem, iż optymalizacja programowa wciąż może mieć znaczenie, a od czasu realizacji może bardzo wiele zależeć.

Wysoka wydajność odgrywa więc znaczącą rolę w rozwoju *Linuksa*. Z tego też powodu, modułowi postawiono tak rygorystyczne wymagania co do maksymalnego dopuszczalnego narzutu. Niestety jedynym rzetelnym sposobem oceny obciążenia czasowego podsystemu jest przetestowanie działającego prototypu. Wyniki testów wydajnościowych umożliwiają następnie wskazanie elementów wymagających ewentualnej optymalizacji. W zaimplementowanym module przyjęto więc właśnie taką strategię. Kolejne prototypy były sprawdzane pod kątem generowanego narzutu wydajnościowego, a następnie optymalizowane, aż do momentu uzyskania zadowalających efektów.

Optymalizacja kontroli uprawnień

W pierwotnej wersji modułu bitmapy zezwoleń w poszczególnych grupach kontrolnych były od siebie niezależne — poszczególne bity danej grupy nie odzwierciedlały w żadnym stopniu uprawnień grup-przodków. Rzeczywiste sprawdzenie zezwoleń odbywało się więc dopiero w momencie wykonywania przez zadanie wywołania systemowego. W takim podejściu funkcja podpięcia sprawdzała bit danego wywołania systemowego w mapie bitowej, a następnie rekurencyjnie, w górę drzewa hierarchii grup kontrolnych, sprawdzani byli kolejni przodkowie. Ostateczny wynik był zatem iloczynem logicznym bitu z aktualnej bitmapy oraz z wszystkich bitmap grup-przodków.

Taki sposób implementacji jest najłatwiejszy oraz najoczywistszy i z tego powodu został on wykorzystany w prototypie. Wiąza się z nim jednak pewne poważne problemy wydajnościowe. Jak nietrudno zauważyć, narzut czasowy rośnie wraz

z „głębokością” hierarchii grup. Wstępne testy polegające na wielokrotnym wywołaniu `setuid()` (wywołanie to zostało wybrane, gdyż nie jest trywialne i nie powoduje żadnych operacji wejścia-wyjścia) i zmierzeniu czasu spędzonego w przestrzeni jądra wykazały, iż narzut wydajnościowy dla grupy pierwszego poziomu wynosił 5%, dla kolejnych natomiast 8% i 12%. Taki stan rzeczy spowodowany jest dużą ilością wywołań funkcji (skoków), które z powodu rekurencji nie mogą zostać zoptymalizowane (np. przez rozwinięcie *inline*).

Pomimo tego modyfikacja świadomie została opublikowana na listach dyskusyjnych *linux-kernel*, *containers* oraz *linux-security-module* jako wersja *RFC 1* (ang. *request for comments*), w celu przedyskutowania i wybrania najlepszego sposobu rozwiązania tego problemu [19]. Społeczność programistów *Linuksa* zgodnie wskazała, iż najlepszym sposobem optymalizacji jest utrzymywanie w danej grupie kontrolnej bitmapy, która odzwierciedla uprawnienia grup wyższego poziomu. Wprowadzenie tej poprawki wymagało zatem modyfikacji kodu odpowiedzialnego za zabranianie danego wywołania tak, aby modyfikował on bit wywołania we wszystkich potomkach. Takie rozwiązanie nie tylko poprawiło i ujednoliciło narzut wydajnościowy, ale również umożliwiło zastosowanie efektywniejszego mechanizmu synchronizacji (zob. podrozdział 4.2.4).

Optymalizacja dla zadań z wyłączoną kontrolą

Zgodnie ze stawianymi podsystemowi wymaganiami, zadania bez zdefiniowanych ograniczeń powinny działać z narzutem nie większym niż 1%. Zastosowanie optymalizacji opisanej w poprzednim podpunkcie okazało się niewystarczające. Narzut związany z koniecznością wywoływania funkcji podpięcia znacząco przewyższa 1%. Zaistniały problem również został przedyskutowany na listach dyskusyjnych. Zaproponowano wiele rozwiązań, z których wypróbowane zostały dwa najbardziej interesujące.

Pierwszy pomysł zakładał dodanie kolejnego pliku konfiguracyjnego, który umożliwiał włączenie i wyłączenie mechanizmu na poziomie danej grupy. Aby umożliwić większą elastyczność, ustawiana w ten sposób flaga nie wpływała jednak na podsystemy położone niżej w drzewie grup. Z drugiej strony takie podejście może okazać się nieintuicyjne, gdyż zaburza hierarchiczność mechanizmu. Dodatkowo wymaga ono jawnej interwencji administratora, gdy zadanie nie ma zdefiniowanych żadnych ograniczeń.

Od strony niskopoziomowej implementacji pomysł ten został zrealizowany przez dodanie dodatkowej flagi `TIF_SYSCALL_USE_CGROUP` do zbioru flag `thread_info`. Struktura `thread_info` skojarzona jest z każdym zadaniem w systemie i wspiera niskopoziomowe operacje, takie jak oznaczanie zadań śledzonych. Wykorzystanie opisanego powyżej pliku konfiguracyjnego umożliwiało ustawienie flagi `TIF_SYSCALL_USE_CGROUP` dla wszystkich zadań w grupie, co skutkowało sprawdzaniem uprawnień za każdym razem, gdy zadanie wykonywało wywołanie systemowe. Wyzerowanie tej flagi dla danej grupy zadań powodowało natomiast, iż cały kod związany z podsystemem był pomijany — „przeskakiwany”. Takie rozwiązanie okazało się bardzo efektywne. Zadania z wyłączoną kontrolą były spowalniane o zaledwie dwie instrukcje — sprawdzenie flagi i wykonanie skoku krótkiego. Zmierzony narzut wydajnościowy oscylował w okolicach błędu pomiaru — nie przekraczał wymaganego 1%.

Wspomniane wyżej problemy skłoniły jednak do rozważenia innych, wygodniejszych rozwiązań. W kolejnym podejściu plik konfiguracyjny został

usunięty i zastąpiony automatycznymi regułami ustawiania i zerowania flagi `TIF_SYSCALL_USE_CGROUP`. Z każdą bitmapą skojarzony został licznik zliczający bity wzbronień wywołań. Inkrementacja licznika z wartości 0 na 1 znaczy, że mechanizm wcześniej nie działał, a więc należy flagę ustawić; dekrementacja z wartości 1 na 0 powoduje natomiast wyłączenie mechanizmu — wyzerowanie flagi. Takie rozwiązanie pomimo tego że bardziej skomplikowane od strony implementacji, wydaje się wygodniejsze dla użytkowników. Z tego powodu zastosowano je w finalnej wersji modułu.

Przy okazji implementacji tego rozwiązania wystąpił interesujący problem synchronizacyjny, niemożliwy do usunięcia dostępnymi metodami. Brak synchronizacji pomiędzy dodawaniem nowego zadania do grupy kontrolnej przy wywołaniu `fork()` i modyfikacją uprawnień może spowodować wyciek i błędne ustawienie flagi `TIF_SYSCALL_USE_CGROUP` dla nowego procesu. Z tego powodu zaistniała konieczność dodania funkcji charakterystycznej `post_fork()` (zob. podrozdział 4.1.2), która umożliwia zagwarantowanie prawidłowej kolejności operacji.

4.2.2. Zarządzanie pamięcią

Pamięć jest niezwykle cennym zasobem przestrzeni jądra. W *Linuksie* zwykle dla systemu operacyjnego dostępna jest niewielka ilość zainstalowanej w komputerze pamięci operacyjnej. W związku z powyższym alokacja dużych obszarów pamięci jest trudna, a często niemożliwa. Restrykcyjnie kontrolowane są również obszary stosów wątków działających wewnątrz jądra. Obsługa sytuacji braku pamięci jest zatem powszechnym elementem programowania na poziomie systemu. Podczas tworzenia modułów jądra *Linuksa* należy możliwie rozsądnie zarządzać pamięcią. Przede wszystkim należy starać się alokować obszary oszczędnie i możliwie późno, a zwalniać je, gdy tylko jest to możliwe. Szczególnie niepożądane są wycieki pamięci, mogą one bowiem szybko doprowadzić do wykorzystania całej dostępnej przestrzeni, a odzyskanie tak utraconych obszarów jest właściwie niewykonalne. Jądro systemu *Linux* udostępnia kilka alokatorów pamięci, oferując przy tym różne parametry. Świadome ich wykorzystanie jest warunkiem wydajności i poprawności implementowanych rozszerzeń. Zagadnienia związane z zarządzaniem pamięcią w jądrze są więc odmienne od tych znanych z programowania w przestrzeni użytkownika. Kompleksowy opis większości problemów można odnaleźć w [27].

Przy projektowaniu podsystemu grup kontrolnych zarządzanie pamięcią jest znacznie ułatwione. Złożoność zagadnienia w znacznej mierze ukryta jest przed programistą — jest realizowana w większości przez rdzeń mechanizmu. Znaczna część kodu związanego z pamięcią polega na alokacji i dealokacji struktur przechowujących informacje powiązane z konkretnymi zasobami obsługiwanymi przez podsystem. W zaimplementowanym module jest to pojedyncza struktura o nazwie `syscalls_cgroup` tworzona dla każdej grupy kontrolnej. Informacjami przechowywanymi w tej strukturze są: bitmapa zezwoleń, licznik bitów wzbronień oraz struktura stanu podsystemu `cgroup_subsys_state`. Obiekt `syscalls_cgroup` jest usuwany w momencie niszczenia grupy kontrolnej w funkcji podsystemu `destroy()`. Istotnym elementem zarządzania pamięcią w module jest prawidłowa obsługa błędów zapobiegająca powstawaniu wycieków.

4.2.3. Programowanie defensywne

Jądro jest oprogramowaniem szczególnie narażonym na ataki, a jednocześnie kluczowym dla stabilności systemu operacyjnego. Istotnym zagadnieniem w programowaniu na poziomie systemu jest zatem uważne weryfikowanie wszelkich danych pochodzących z potencjalnie niezaufanego źródła — przestrzeni użytkownika. Błędy wynikające z niedostatecznej walidacji informacji mogą mieć najróżniejsze skutki: mogą doprowadzić do krytycznego błędu systemu, eskalacji uprawnień czy też wstrzyknięcia i wykonania kodu. Stosując technikę programowania defensywnego (ang. *defensive programming*), nie należy popadać w skrajności — weryfikacja danych, które nie mogą być niebezpieczne niepotrzebnie zaśmieca kod, a ponadto stanowi zbędny narzut wydajnościowy. Odpowiednią kontrolę przekazywanych parametrów należy więc wykonywać tam, gdzie jest to niezbędne i nigdzie więcej.

Zaimplementowany podsystem, jako że udostępnia przestrzeni użytkownika możliwości konfiguracji, wymaga podejścia defensywnego. Przekazywane przez użytkownika parametry ograniczają się do numeru wywołania, a więc z tym związana jest cała przeprowadzana walidacja. Argument liczbowy jest przekazywany do modułu najpierw pod postacią bufora znakowego, co wynika ze sposobu implementacji mechanizmu *seq_file*. Musi on być zatem w sposób bezpieczny zamieniony na postać liczbową. Do tego celu wykorzystany został parser udostępniony przez jądro. Takie podejście minimalizuje szansę popełnienia błędu. W dalszej kolejności parametr, już w postaci liczby całkowitej, sprawdzany jest pod kątem poprawności zakresu — musi odpowiadać pewnemu numerowi wywołania w systemie. Na każdym etapie weryfikacji, w razie niepowodzenia, zwracany jest odpowiedni błąd, który wstrzymuje dalsze wykonanie.

Argumentem, który nie wymaga sprawdzenia, jest numer wywołania przekazywany do funkcji podpięcia weryfikującej uprawnienia. Poprawność tego parametru jest kontrolowana w niskopoziomym kodzie wywołania, gwarantując tym samym, iż w momencie wywołania funkcji będzie on bezpieczny. Dodatkowe sprawdzenie w tym miejscu stanowiłoby niepotrzebną komplikację i niepożądany narzut wydajnościowy.

W tym miejscu można się zastanowić, czy takie podejście aby na pewno jest słuszne. Poprawność wykonania funkcji podpięcia zależy bowiem od kontekstu, w którym została ona wywołana, od tego, czy została przeprowadzona wcześniej weryfikacja argumentu. Tak, jak zaznaczono wcześniej, wydajność tej funkcji ma kluczowe znaczenie dla wydajności mechanizmu, a więc minimalizacja operacji jest w tym wypadku zdecydowanie pożądana. Pominięcie nadmiarowej, dla obecnego sposobu użycia, walidacji danych jest zatem dobrym wyborem. W celu ograniczenia błędów wynikających z ewentualnych przyszłych modyfikacji kodu, opatrzone go komentarzem wskazującym prawidłowe warunki wykonania funkcji.

4.2.4. Synchronizacja

Znaczna większość istniejących systemów operacyjnych dopuszcza współbieżne czy nawet równoległe wykonanie programów, a więc także kodu jądra. Wszelkie niespójności wynikające z jednoczesnego wykonania kodu sekcji krytycznych mogą istotnie zaburzyć lub uniemożliwić pracę systemu operacyjnego i działających wewnątrz aplikacji. Dostęp do zasobów współdzielonych takich jak np. struktury danych systemu wymaga więc właściwej, dogłębnie przemyślanej synchronizacji.

Bezwzględnym kryterium oceny zastosowanej synchronizacji jest jej poprawność. Niezależnie od rozważanego problemu należy oczekiwać, iż rozwiązanie spełnia kanoniczne wymagania sekcji krytycznej, a ponadto prowadzi do realizacji postawionego zadania. Innymi istotnymi czynnikami wpływającymi na ocenę danej metody synchronizacji są jej ogólna wydajność oraz poziom komplikacji kodu, który wprowadza. Często zdarza się, że cechy te stoją ze sobą w sprzeczności. Dla przykładu zastosowanie monitorów do synchronizacji dostępu do bufora jest eleganckie i może być dla programisty przeźroczyste. Niestety, naiwna implementacja tego mechanizmu spowoduje, że współbieżne odczyty będą wykluczane, co będzie znacząco obniżać całościową wydajność rozwiązania. Z drugiej strony zastosowanie w tym wypadku semaforów r/w będzie wymagało od programisty większej dyscypliny, ale umożliwi wzajemnie nieblokujące odczyty. Z reguły w aplikacjach użytkowych stawia się na większą czytelność i łatwość użycia kodu, natomiast w programowaniu systemowym zwykle wydajność jest ważniejsza. W obu przypadkach zdarzają się jednak wyjątki. Ostateczna decyzja zależy od specyfiki rozważanego problemu.

Najważniejszym zasobem współdzielonym w zaimplementowanym module są bitmapy zezwoleń powiązane z grupami kontrolnymi. Struktury te są odczytywane podczas sprawdzania uprawnień, modyfikacja następuje natomiast podczas zmieniania zbioru dozwolonych wywołań systemowych. Ze względu na poprawność współbieżnych odczytów sytuacjami, które wymagają wzajemnej synchronizacji są jednoczesne odczyty i zapisy oraz wiele jednoczesnych zapisów. Oba przypadki, pomimo że dotyczą tego samego zasobu współdzielonego, znacząco się różnią. Z tego powodu przyjęcie wspólnego, jednolitego sposobu rozwiązania problemu (np. zastosowanie oddzielnych semaforów binarnych dla każdej mapy bitowej) okazałoby się nieefektywne. Zaimplementowana synchronizacja została więc poprzedzona dokładną, indywidualną analizą każdego z przypadków.

Odczyt — zapis

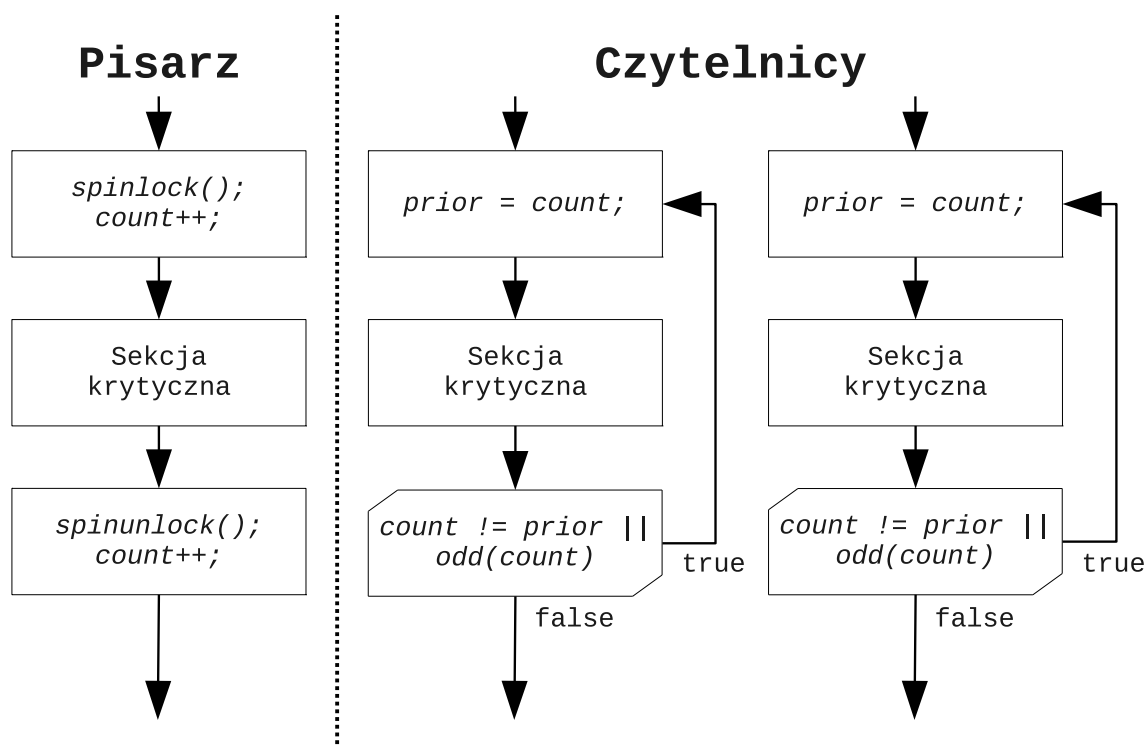
Sytuacja jednoczesnego odczytu i zapisu może wystąpić, gdy konfiguracja grupy kontrolnej jest modyfikowana, a pewne zadanie znajdujące się w niej, właśnie wykonuje wywołanie systemowe. Chociaż dochodzi wtedy do współbieżnego odczytu i modyfikacji tylko jednego bitu, nie można poczynić żadnych dodatkowych założeń, jak przebiegają te operacje na poziomie architektury sprzętowej — może zatem zdarzyć się odczyt danych w niespójnym stanie.

Sytuacja ta jest klasycznym problemem czytelników i pisarzy [51]. Narzucającym się rozwiązaniem jest zatem zastosowanie semaforów r/w zaimplementowanych właśnie do tego rodzaju zadań. Z tego powodu, mechanizm ten został użyty w pierwotnej wersji modułu. Wstępne testy wykazały jednak, że ogólna wydajność podsystemu jest niezadowalająca (narzut ok. 12% dla x86-64). W następnym kroku zbadano więc wpływ poszczególnych elementów mechanizmu na narzut. Przeprowadzona analiza wykazała, że zastosowana metoda synchronizacji jest odpowiadająca za niespełna 70% całkowitego czasu pracy mechanizmu. Tak wysoki narzut wydajnościowy dla semaforów r/w (które są zdecydowanie bardziej złożone niż zasadnicza funkcja podsystemu) wynika z wielu operacji wykonywanych w celu założenia blokady.

Zaistniała zatem konieczność odnalezienia innego, bardziej efektywnego mechanizmu synchronizacji dla rozważanego problemu. Poszukiwania ułatwiło zidentyfikowanie cech charakterystycznych zadania — odczyty obszaru pamięci bitmapy są bardzo częste, zapisy są natomiast sporadyczne. Skuteczna metoda

synchronizacji musi zatem być rozwiązaniem problemu czytelników i pisarzy i zostać zoptymalizowana dla odczytów. Mechanizmami spełniającymi te wymagania są RCU (ang. *read-copy-update*) oraz *seqlocks*. Wybór lepszego z nich okazał się łatwy — efektywniejszym rozwiązaniem dla tablic statycznych jest zastosowanie *seqlocks* [38, 41].

Seqlocks to nieskomplikowany, wydajny mechanizm, szczególnie przydatny w problemach z wieloma czytelnikami. Idea działania *seqlocks* jest bardzo prosta. Pisarze przed wejściem i po wyjściu ze swojej sekcji krytycznej wykonują taką samą operację — inkrementują licznik związany z egzemplarzem blokady. Zwiększanie o jeden jest dodatkowo powiązane z założeniem rygła pętlowego, który synchronizuje pracę pisarzy. Czytelnicy natomiast nie zajmują żadnych blokad przed wejściem do sekcji krytycznej zapisują jedynie aktualną wartość licznika w zmiennej. Jeśli po zakończeniu działania na zasobach współdzielonych aktualna wartość licznika nie jest równa wartości przechowywanej lub jest ona nieparzysta, oznacza to, że odczytane dane mogą być niespójne, gdyż operował (bądź operuje) na nich wątek pisarza. W takiej sytuacji należy powtórzyć sekcję krytyczną od nowa.



Rysunek 4.2: Zasada działania mechanizmu synchronizacji *seqlocks*. Na podstawie: [38].

Efektywność mechanizmu *seqlocks* opiera się na założeniu, iż czytelnicy nie wykonują żadnych skomplikowanych operacji. Cały narzut wydajnościowy pracy czytelników składa się z organizacji pętli oraz odczytu i porównania wartości zmiennych. Aby skutecznie wykorzystać rozwiązanie należy także znać jego wady. Użycie tego mechanizmu synchronizacji w zadaniach z częstymi bądź długimi zapisami grozi zagłodzeniem czytelników. Dodatkowo, aby w ogóle możliwe było jego zastosowanie, sekcja krytyczna czytelników musi być powtarzalna i nie powinna mieć skutków zewnętrznych.

Zastosowanie *seqlocks* w zaimplementowanym module przyniosło oczekiwane skutki — ogólny narzut wydajnościowy wynosił ok. 5%. Czas spędzony na synchronizacji zmalał do 2,5% całkowitego czasu działania podsystemu, co stanowi niebagatelną poprawę w stosunku do 70% przy użyciu semaforów r/w.

Optymalizacja sprawdzania warunków opisana w poprzednim punkcie skłoniła do ponownej weryfikacji zastosowanego mechanizmu synchronizacji. Operacje na zasobach współdzielonych, mogące występować w module współbieżnie to sprawdzanie oraz modyfikacja pojedynczego bitu. Instrukcje te wydają się bardzo proste — jeśli istniałaby możliwość niepodzielnego ich wykonania, to nie byłoby potrzeby stosowania dodatkowych metod — np. *seqlocks*. Analiza API jądra doprowadziła do odnalezienia funkcji *test_bit*, *set_bit* oraz *clear_bit*, które gwarantują atomowość. Implementacja tych procedur jest zależna od platformy sprzętowej jednak, jako że są one dostępne z poziomu języka C, programista nie musi się tym faktem przejmować.

Zastosowanie operacji atomowych w podsystemie wyeliminowało konieczność użycia *seqlocks*. Tym samym całkowicie zniknął narzut związany z synchronizacją, gdyż na platformach z rodziny x86 wyżej wymienione funkcje są kompilowane do pojedynczych instrukcji procesora.

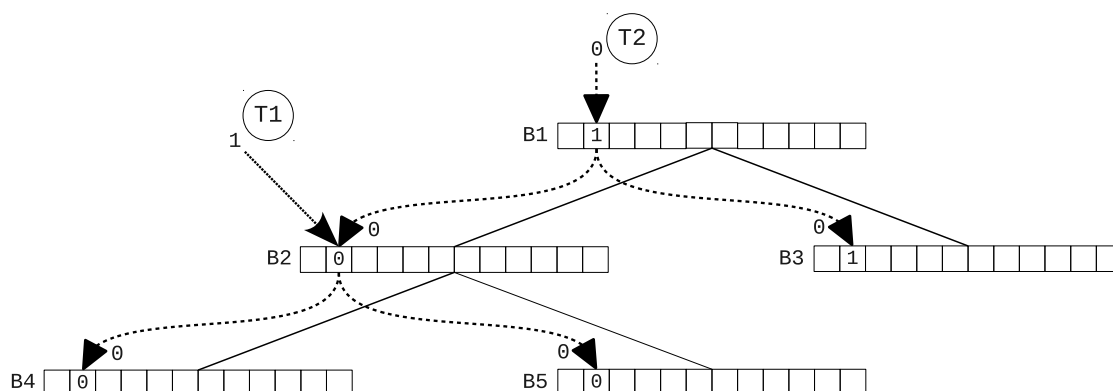
Zapis — zapis

Drugą sytuacją wymagającą synchronizacji jest próba współbieżnej modyfikacji bitmap uprawnień. Zgodnie z założeniami wywołanie systemowe w danej grupie kontrolnej może zawsze zostać zabronione. Powoduje to ograniczenie wszystkich grup potomnych. Na poziomie implementacji, w celu optymalizacji (zob. podrozdział 4.2.1) narzutu czasowego, kaskadowe wzbranianie zostało zrealizowane poprzez wyzerowanie bitów we wszystkich grupach podrzędnych. Ustawienie bitu zezwolenia na wywołanie systemowe jest natomiast możliwe tylko wtedy, gdy grupa nadrzędna również ma taki bit w stanie wysokim — bity zezwoleń nie są propagowane kaskadowo.

Brak gwarancji kolejności wykonania poszczególnych operacji stwarza zatem niebezpieczeństwo wyścigu. Niesynchronizowana, współbieżna modyfikacja bitmap uprawnień może spowodować utratę spójności danych. Przykład takiej sytuacji przedstawiono na rysunku 4.3.

Istnieje kilka możliwości rozwiązania tego problemu. Na przykład można skorzystać z każdej grupą semafor binarny, który umożliwiałby blokowanie całych poddrzew przed współbieżną modyfikacją. Rozwiązanie takie zapewniałoby blokowanie możliwie niewielkiego zbioru, jednak spowodowałoby znaczne skomplikowanie kodu. Całkowicie odmiennym podejściem jest natomiast blokowanie za pomocą pojedynczego semafora binarnego przez cały okres modyfikacji drzewa bitmap. Takie rozszerzenie sekcji krytycznej całkowicie uniemożliwi współbieżną (nawet bezkonfliktową) konfigurację uprawnień, jest jednak łatwe do zaimplementowania i oszczędne pamięciowo, gdyż nie wymaga tworzenia wielu semaforów.

Analizując problem, warto zauważyć, że polityka dostępu najprawdopodobniej nie będzie modyfikowana współbieżnie — wynik takiej operacji byłby nieoczywisty. Dodatkowo od narzędzi konfiguracyjnych zwykle nie wymaga się wysokiej wydajności, gdyż są one uruchamiane stosunkowo rzadko. W zaimplementowanym module zastosowano więc podejście z pojedynczym semaforem binarnym, wykluczającym współbieżne wykonanie całej funkcji modyfikującej uprawnienia. Dzięki temu kod



Rysunek 4.3: Sytuacja potencjalnego wyścigu. Najpierw zadanie *T1* sprawdza, czy może ustawić bit w bitmapie *B2* — odpowiadający mu bit w *B1* jest ustawiony. Następnie zadanie zostaje wyłączone i *T2* rozpoczyna pracę. *T2* kaskadowo zeruje bity danego wywołania we wszystkich mapach bitowych poczynając od *B1*, a następnie kończy działanie. Wznowione zostaje zadanie *T1*, które ustawia bit w *B2* bazując na aktualnie nieprawdziwych informacjach sprzed wyłączenia. Bitmapa *B2* pozostaje w stanie sprzecznym z założeniami.

jest prostszy i bardziej elegancki co odbywa się kosztem współbieżnej modyfikacji konfiguracji.

4.2.5. Wykorzystanie API

Jednym z kluczowych czynników sukcesu podczas budowy oprogramowania jest znajomość API platformy, dla której powstaje. Podobnie jest z programowaniem w przestrzeni jądra. Problemem w wypadku *Linuxa* jest jednak objętość i bogactwo interfejsu programistycznego, który często nie jest dostatecznie udokumentowany, zmuszając tym samym programistów do wnikliwej analizy kodu źródłowego.

Podstawową zaletą efektywnego i świadomego wykorzystania API jest możliwość łatwego tworzenia nowych modułów jądra przy pisaniu relatywnie niewielkiej ilości kodu. Oszczędzany jest więc czas programisty, a tym samym budżet projektu. Co więcej, można wierzyć, iż kod obecny w głównej linii *Linuxa* jest kodem dojrzałym i rzetelnie przetestowanym. Delegując więc pewne zadania tworzonego modułu do funkcji i mechanizmów API, minimalizowane jest prawdopodobieństwo wystąpienia błędu. Mniej kodu oznacza bowiem potencjalnie mniej błędów.

Inną korzyścią wynikającą z konsekwentnego zastosowania API jest przenośność kodu. Bardzo często funkcje obecne w interfejsie programistycznym jądra ukrywają implementację szczegółów zależnych od architektury sprzętowej. Możliwe jest zatem tworzenie modułów, które napisane są w sposób uniwersalny — współpracujący z wieloma platformami.

Ostatnią istotną kwestią związaną z wykorzystaniem API jądra jest optymalizacja. Funkcje pomocnicze, które znajdują się w interfejsie programistycznym jądra, bardzo często stosują zabiegi zwiększające wydajność działania dla konkretnej platformy sprzętowej. Co więcej, można zakładać, że algorytmy zastosowane w implementacji API *Linuxa* należą do najwydajniejszych w swojej dziedzinie, gdyż były intensywnie optymalizowane i weryfikowane przez setki programistów. Ciężko jest

zatem zaimplementować coś znacząco lepszego od efektu kolektywnej pracy całej społeczności *Linuksa*.

Rozważany mechanizm bezpieczeństwa był projektowany, a następnie implementowany z myślą o najsukuteczniejszym wykorzystaniu API jądra *Linuksa*. Poszczególne decyzje na etapie projektowania, takie jak wybór podsystemu *cgroups*, podejmowane były na podstawie dogłębnej analizy interfejsu programistycznego. W fazie implementacji miała miejsce również intensywna eksploracja API jądra, która często polegała na odkrywaniu i testowaniu różnych rozwiązań (zob. podrozdział 4.2.4).

Niewątpliwie, największym problemem napotkanym podczas poznawania interfejsu *Linuksa* była wspomniana wcześniej skromna dokumentacja tekstowa. Efektywne wykonanie niektórych zadań, w szczególności tych związanych z rdzeniem mechanizmu *cgroups*, byłoby niemożliwe bez dogłębnej analizy kodu, zajmującej znacznie więcej czasu niż lektura dokumentacji. Można jednak stwierdzić, iż wiedza zdobyta przy przeglądzie kodu źródłowego jądra jest pełniejsza i daje większe możliwości, niż zapoznanie się z nawet najobszerniejszą dokumentacją.

4.3. Wykorzystane narzędzia

Wytwarzanie oprogramowania to wieloetapowy, skomplikowany proces. W związku z tym, na każdym etapie jest on wspierany przez różne, często silnie wyspecjalizowane narzędzia. Dobór aplikacji wspierających zależy od dziedziny projektu, techniki wykonania, platformy docelowej, a także od upodobań osobistych programistów i architektów.

W rozwijaniu jądra *Linuksa* wykorzystuje się również wiele narzędzi programistycznych. Większość z nich ma bardzo długą historię, sięgającą pierwszych wersji systemu *UNIX*. W standardowym zestawie programów wykorzystywanych przy tworzeniu *Linuksa* znajduje się również wiele aplikacji napisanych specjalnie do rozwijania jądra. Baza narzędziowa programistów jądra *Linuksa* jest więc bardzo rozbudowana, dojrzała i wysoce zaawansowana.

Podczas implementacji mechanizmu zastosowanie znalazło wiele narzędzi. Poniżej znajduje się ich opis z uwzględnieniem tego, jak zostały wykorzystane.

4.3.1. Tworzenie kodu

Poza edytorem tekstu najważniejszym narzędziem wspierającym fazę tworzenia kodu jest system kontroli wersji. Oficjalnym systemem kontroli wersji głównej linii jądra *Linuksa* jest *Git* [6]. *Git* został napisany przez Linusa Torvaldsa w 2005 specjalnie na potrzeby rozwoju jądra. Wcześniejszym narzędziem zarządzającym wersjonowaniem był własnościowy *BitKeeper* [3], nie zaspokajał on jednak potrzeb społeczności. *Git* jest szybki, bezpieczny, wygodny w użyciu oraz, co bardzo ważne, posiada funkcje zaprojektowane specjalnie w celu wspierania rozwoju *Linuksa* jak np. automatyczna generacja łąt na podstawie różnic. Z powyższych powodów *Git* został wybrany do wspierania implementacji opisywanego projektu.

Podczas tworzenia podsystemu do zadań *Gita* należało:

- przechowywanie kodu źródłowego i jego kolejnych wersji,
- śledzenie różnic między główną linią *Linuksa*, a gałęzią projektu,
- uaktualnianie gałęzi do najnowszej wersji — rozwiązywanie konfliktów,
- generowanie łąt, głównie na potrzeby listy dyskusyjnej.

Powyższe problemy mogłyby oczywiście zostać rozwiązane bez wsparcia jakiegokolwiek narzędzia, jednak takie działanie byłoby niewygodne i groziłoby wystąpieniem błędów. *Git* doskonale zautomatyzował i ułatwił zarządzanie kodem źródłowym.

4.3.2. Kompilacja

Proces kompilacji jądra *Linuksa* znacząco różni się od kompilacji typowych programów. Kod jądra jest na tyle obszerny i skomplikowany, że na przestrzeni lat opracowane zostało specjalne narzędzie służące do budowania jego binarnej wersji — *kbuild* [41]. Na *kbuild* składa się wiele programów i pakietów typowych dla systemów uniksopodobnych takich jak: *gcc*, *binutils*, *libc-dev*, *make*, *fileutils* oraz duża liczba aplikacji i skryptów powłoki stworzonych specjalnie na potrzeby jądra. Centralną częścią mechanizmu *kbuild* są pliki *Makefile* oraz *Kbuild*, które definiują co i w jakiej kolejności powinno być kompilowane oraz wykonują odpowiednie operacje. Dla tych plików konfiguracją jest plik *.config*, który standardowo tworzony jest przez narzędzia typu *menuconfig*.

Podczas implementacji podsystemu zaszła konieczność modyfikacji plików *Makefile* mechanizmu *kbuild* w celu ustawienia odpowiednich parametrów kompilacji. Dodatkowo edycji wymagał plik *Kconfig*, który stanowi listę pozycji dla aplikacji *menuconfig*. *Kbuild* był intensywnie wykorzystywany przy tworzeniu modułu, gdyż proces implementacji wymagał wielokrotnej kompilacji i budowania wielu binarnych wersji jądra do testów. Niewątpliwą zaletą opisywanego mechanizmu jest łatwość jego użycia oraz skuteczność. Wykorzystywanie *kbuild* nie wymaga znacznej wiedzy, a modyfikacja parametrów kompilacji jest prosta. W zależności od konfiguracji modułów, pełna kompilacja jądra może trwać nawet kilka godzin, dlatego kompilacja przyrostowa oferowana przez *kbuild* znacząco usprawnia pracę z kodem źródłowym.

Oprócz *kbuild*, na etapie kompilacji wykorzystywane było narzędzie *objdump* [42] służące do pobierania informacji z plików obiektowych (plików pośrednich kompilacji). Odpowiednie przełączniki programu pozwalają na deasemblację binariów czyniąc *objdump* wygodnym deasemblerem. Wynikowy kod binarny modułu był analizowany w celu odnalezienia możliwości dalszych optymalizacji podsystemu, w szczególności funkcji podpięcia.

4.3.3. Uruchamianie

Po kompilacji standardowo następuje uruchamianie aktualnej wersji oprogramowania w celu wstępnego sprawdzenia i przetestowania nowych funkcjonalności. W zależności od specyfiki wytwarzanego kodu następuje to w różnoraki sposób — maszyna rozwojowa i uruchomieniowa są bardzo często rozdzielone i mają odmienne specyfikacje. W przypadku rozwijania jądra używanie tego samego środowiska do programowania i testowania kolejnych wersji jest wysoce niewskazane, a często nawet niemożliwe [15]. Z tego powodu na potrzeby tworzenia i rozwijania modułów jądra *Linuksa* warto w miarę możliwości przeznaczyć dedykowaną maszynę. Takie rozwiązanie jest wygodne i pozwala przeprowadzać rzetelne testy bezpośrednio na architekturze docelowej. Wadą takiego środowiska uruchomieniowego jest koszt związany z koniecznością zakupu dodatkowego sprzętu. W praktyce najczęściej stosuje się zatem podejście hybrydowe. Wstępne wersje kodu testowane

są w maszynach wirtualnych lub na emulatorach, a wersje kandydujące do wydania sprawdzane są na fizycznych architekturach. Wybór takiego rozwiązania obniża koszty, a jednocześnie pozostaje wygodny dla programistów.

Przy projektowaniu modułu skorzystano więc z podejścia łączonego. W początkowej fazie rozwoju podsystemu, kolejne wersje jądra uruchamiane były w maszynie wirtualnej oferowanej przez pakiet *QEMU* z rozszerzeniem *KVM*, a dopiero następnie testowane na maszynach fizycznych. Pomiedzy systemem-gospodarzem, a systemem-gościem możliwe było zestawienie sesji *SSH*. Taka organizacja środowiska programistycznego umożliwiała zbudowanie w jednym kroku (pojedynczy skrypt powłoki) nowej wersji jądra, uruchomienie jej w maszynie wirtualnej oraz przeprowadzenie podstawowych testów poprawnościowych. Podstawowy schemat pracy wyglądał więc następująco:

1. Modyfikacja kodu podsystemu.
2. Uruchomienie procesu kompilacji i testowania.
 - a) Kompilacja za pomocą mechanizmu *kbuild*.
 - b) Uruchomienie maszyny wirtualnej z nowym jądrem.
 - c) Wykonanie podstawowych testów poprawnościowych.
3. Ocena wyników.
4. Powrót do punktu 1, jeśli wyniki są niezadowalające.
5. Wykonanie testów na maszynie fizycznej.

Tak skonstruowany przebieg pracy okazał się bardzo wygodny. Kluczowym składnikiem sukcesu okazała się maksymalna automatyzacja kroków odtwórczych całego procesu, która pozwoliła zaoszczędzić dużo czasu i uniknąć błędów wynikających z omyłek.

4.3.4. Debugowanie

Każde oprogramowanie, nawet tak starannie wytwarzane jak jądro *Linuksa*, zawiera błędy [40]. Typowym postępowaniem w przypadku odkrycia błędu jest próba odtworzenia kroków potrzebnych do jego reprodukcji, stwierdzenie, w którym miejscu kodu źródłowego występuje problem oraz poprawienie fragmentu powodującego dysfunkcję. Często zdarza się jednak, iż trudno jest odnaleźć przyczynę błędu. Stosuje się wtedy narzędzia wspierające debugowanie.

Przez dwie dekady rozwoju *Linuksa* opracowano wiele narzędzi ułatwiających debugowanie błędów w jądrze. Wśród najbardziej znanych mechanizmów i aplikacji warto wymienić: *printk()*, *Oops Messages*, *kdb*, *kgdb*, *gdb*, *crash*, *kdump*, *kprobes*, *LKCD*, *diskdump*, *netdump*, *procfs*, *sysfs*, *debugfs*, *ftrace* [15, 28, 34, 41, 44, 56]. Narzędzia te znacząco różnią się przeznaczeniem, sposobem użycia oraz zastosowania, oferują jednak w połączeniu kompleksowe możliwości diagnostyki błędów.

Podczas implementacji podsystemu grup kontrolnych wykorzystano dwa z dostępnych mechanizmów — *printk()* oraz *gdb*. *Printk()* jest najstarszym i jednocześnie najbardziej prymitywnym narzędziem diagnostycznym. Jego użycie polega na umieszczeniu w kodzie modułu wywołań funkcji *printk()* (oraz pokrewnych) z odpowiednimi parametrami, analogicznymi do procedury standardowej *printf()*. Po wywołaniu funkcji *printk()* wynikowe ciągi znakowe są umieszczane w logach jądra dostępnych za pomocą programu *dmesg*. Wykorzystując ten mechanizm, można zatem uzyskać prosty, aczkolwiek skuteczny rejestrator wykonania procedur modułu, który może posłużyć np. do weryfikacji poprawności parametrów czy śledzenia obiegu struktur danych.

Drugi z wykorzystanych mechanizmów posłużył do badania błędów krytycznych tzw. *kernel panic*. Wykorzystanie *QEMU* w połączeniu z *gdb* pozwala wykonywać kod jądra w trybie pracy krokowej oraz realizować wszystkie zabiegi diagnostyczne oferowane przez debugger *gdb*. Po wykonaniu błędnej operacji istniała zatem możliwość dokładnej inspekcji elementów stanu modułu: wartości zmiennych, pozycji licznika instrukcji, zawartości poszczególnych rejestrów. Zebranie takich informacji umożliwiało w dalszej kolejności wnioskowanie na temat przyczyn obserwowanego błędu oraz sposobu jego usunięcia.

Zaimplementowany mechanizm nie wymagał stosowania bardziej wyrafinowanych metod debugowania, gdyż z góry można było się spodziewać, gdzie znajduje się obserwowany błąd i zwykle był on łatwy do wyizolowania. Podczas pracy z jądrem, które zawiera błędy niereprodukowalne, występujące sporadycznie czy też trudne do zlokalizowania, warto rozważyć możliwość użycia pozostałych, specjalnie do tego przeznaczonych narzędzi.

4.3.5. Wstępne testowanie

Z uruchamianiem skompilowanego oprogramowania wiąże się ściśle jego testowanie. W trakcie implementacji testy mają zwykle charakter wstępny — dają programiście wyobrażenie na temat skuteczności zrealizowanych pomysłów. Często zdarza się jednak, iż testy stworzone na etapie implementacji są następnie rozbudowywane i używane w fazie właściwego testowania kompletnego już rozwiązania.

Projekt *Linux* niestety nie doczekał się oficjalnej i dobrze wspieranej metody testowania kodu. Pewną próbą zmiany tej sytuacji jest *Linux Test Project* [30], który nie zyskał jeszcze popularności. Taki stan rzeczy spowodowany jest skomplikowaniem zagadnienia — testowanie jądra jest wyjątkowo trudne i trudno wskazać ogólną metodę, którą można zastosować do wszystkich jego składników. Z tego powodu testy poszczególnych modułów tworzone są zwykle przez ich autorów wedle własnej koncepcji.

Na potrzeby fazy implementacji opracowano zatem autorskie skrypty testujące, których zadaniem było wskazanie czy moduł działa poprawnie oraz czy spełnia wymagania wydajnościowe. W celu weryfikacji poprawności za pomocą powłoki systemowej wykonywano różne operacje na module (np. zabronienie wywołania systemowego, zmianę grupy kontrolnej zadania, utworzenie dodatkowej pozycji w hierarchii), a następnie badano przebieg realizacji różnych wywołań systemowych pod kątem zgodności z aktualnie ustaloną polityką. Do wszystkich tych zadań wykorzystano standardowe narzędzia powłoki takie jak *mkdir*, *echo*, *su*, które w uproszczeniu są programowym opakowaniem dla wywołań systemowych.

W celu pomiaru narzutu wydajnościowego porównywano czas programu testowego spędzony w przestrzeni systemowej dla dwóch wersji jądra — niemodyfikowanego oraz ze skompilowanym mechanizmem ochrony ABI. Program testowy wykonywał w pętli wywołanie systemowe *setuid()*. Zestawienie obu czasów pozwalało łatwo obliczyć procentowy narzut spowodowany działaniem podsystemu.

Tak skonstruowane uproszczone testy pozwalały w łatwy sposób zorientować się, czy proces implementacji przebiega właściwie, to znaczy czy istnieje szansa, aby sprostać oczekiwaniom wydajnościowym, zachowując przy tym poprawność działania. Dodatkowo stałość procedury testowania umożliwiła natychmiastową ocenę wprowadzanych zmian — dla każdej kompilacji możliwe było stwierdzenie, czy jest ona wydajniejsza od poprzedniej oraz czy nie pojawiły się błędy regresji.

4.4. Instrukcja użytkownika

Zwieńczeniem etapu implementacji jest określenie sposobu użytkowania końcowego produktu tak, aby możliwe było jego przetestowanie. Standardowo określa się metody wdrożenia stworzonego rozwiązania oraz instrukcję dla użytkownika końcowego. W przypadku zaimplementowanego podsystemu wdrożenie polega na skompilowaniu zmodyfikowanego jądra, a instrukcja użytkownika sprowadza się do opisanie interfejsu modułu w systemie.

4.4.1. Aplikacja łaty

Tradycyjnym, niezmiennym od dwudziestu lat sposobem dystrybucji modyfikacji jądra są łaty w zunifikowanym formacie *diff*. Wybór właśnie takiej formy rozpowszechniania nowych wersji okazał się trafną decyzją — również wiele projektów społecznościowych do dziś wykorzystuje ten pomysł.

Łaty są plikami tekstowymi łatwymi do odczytania i zrozumienia. Osoba analizująca poprawki jest w stanie bez problemu stwierdzić gdzie i jakie zaszyły zmiany. Ponadto, pliki te mają bardzo kompaktowy format umożliwiający ich dystrybucję poprzez listy dyskusyjne. Takie rozwiązanie daje możliwość dyskusowania nawet o pojedynczych liniach kodu, a wręcz zachęca do niego, co zapewne przyczyniło się do społecznościowego sukcesu projektu *Linux*.

Udostępnienie modułu w postaci łaty jest zatem naturalnym wyborem, zgodnym ze standardami programowania jądra *Linuksa*. Do zaaplikowania łat wymagany jest program *patch* dostępny w większości dystrybucji *Linuksa*. Sposób jego użycia oraz kroki potrzebne do skompilowania jądra z modułem zostały przedstawione w dodatku A.1.

4.4.2. Interfejs użytkownika

Interfejs użytkownika, co jest charakterystyczne dla mechanizmu grup kontrolnych, oparty jest w całości na plikach i katalogach. Podmontowanie zaimplementowanego podsystemu we wskazanym katalogu tworzy w nim grupę kontrolną najwyższego poziomu i umieszcza w folderze odpowiednie pliki konfiguracyjne. Stworzenie w katalogu reprezentującym grupę kolejnego katalogu skutkuje utworzeniem grupy podrzędnej. W taki sposób budowana jest hierarchia grup.

Z każdą grupą rozważanego podsystemu skojarzone są pliki konfiguracyjne o takich samych nazwach, ale oczywiście o odmiennej zawartości. Opis funkcjonalności oferowanej przez poszczególne pliki znajduje się poniżej, natomiast przykładowe użycie podsystemu zostało zaprezentowane w dodatku A.2.

cgroup.clone_children

Plik tworzony jest domyślnie przez mechanizm *cgroups*, nie jest wykorzystywany w module. Pozwala on sterować wywołaniami zwrótnymi *post_clone*, które nie znalazły zastosowania w podsystemie.

cgroup.event_control

Plik tworzony jest domyślnie przez mechanizm *cgroups*, nie jest wykorzystywany w module. Jest on częścią API powiadomień, które nie są zaimplementowane w podsystemie.

cgroup.procs

Plik tworzony jest domyślnie przez mechanizm *cgroups*. Odczyt z tego pliku

zwraca listę rozdzielonych znakiem nowej linii numerów *TGID* (identyfikatorów grup wątków) zadań należących do danej grupy kontrolnej. Zapis do tego pliku prawidłowego numeru *TGID* przenosi do grupy wszystkie zadania współdzielące ten identyfikator.

notify_on_release

Plik tworzony jest domyślnie przez mechanizm *cgroups* i stanowi flagę dla mechanizmu *release_agent* (zob. poniżej). Zapis wartości dodatkowo ustawia flagę, natomiast wartość zero flagę wyłącza. Odczyt z tego pliku powoduje zwrócenie wartości 0 lub 1, w zależności czy flaga jest wyłączona, czy włączona.

release_agent

Plik tworzony jest domyślnie przez mechanizm *cgroups*, ale wyłącznie w katalogu grupy najwyższego poziomu. Jeśli w pewnej grupie kontrolnej została ustawiona flaga *notify_on_release*, to w momencie gdy ostatnie zadanie odłączy się od tej grupy, zostanie wywołane polecenie umieszczone w pliku *release_agent*. Ten zabieg ma na celu ułatwienie realizacji zadań takich jak usuwanie opuszczonych grup. Odczyt z tego pliku zwraca aktualnie ustawione polecenie, natomiast zapis pozwala je zmodyfikować. Mechanizm ten może być użyty dla zaimplementowanego podsystemu.

syscalls.allow

Plik tworzony jest przez podsystem. Odczyt z tego pliku zwraca oddzielone spacjami numery dozwolonych wywołań systemowych, dla zadań z danej grupy. Zapis numeru wywołania systemowego do tego pliku próbuje dodać to wywołanie do zbioru wywołań dopuszczalnych. Aby operacja się powiodła, dodawane wywołanie musi być dozwolone w grupie nadrzędnej.

syscalls.deny

Plik tworzony jest przez podsystem. Odczyt z tego pliku zwraca oddzielone spacjami numery niedozwolonych wywołań systemowych, dla zadań z danej grupy. Zapis numeru wywołania systemowego do tego pliku dodaje to wywołanie do zbioru wywołań zabronionych.

tasks

Plik tworzony jest domyślnie przez mechanizm *cgroups*. Odczyt z tego pliku zwraca listę rozdzielonych znakiem nowej linii numerów *PID* zadań należących do danej grupy kontrolnej. Zapis do tego pliku prawidłowego numeru *PID* przenosi zadanie o tym identyfikatorze do grupy. Zapis do pliku wartości 0 powoduje dodanie zadania zapisującego do danej grupy kontrolnej.

Uwagi

Szczegółowe informacje na temat używania plików tworzonych domyślnie przez mechanizm grup kontrolnych można odnaleźć w [41] w sekcji *cgroups*. Wszelkie błędy (np. brak uprawnień, zły argument) występujące podczas zapisu do plików tworzonych przez podsystem sygnalizowane są odpowiednim błędem. Domyślnie, wszystkie pliki poza *cgroup.event_control* mają ustawiony tryb dostępu 644.

5. Testowanie

Testowanie to niezwykle ważny, często jednak zaniedbywany, etap tworzenia oprogramowania. W wielu wypadkach, znaczna część działań związanych ze sprawdzaniem zaimplementowanego produktu skupia się na końcu cyklu wytwarzania, w momencie gdy brakuje już czasu i funduszy na dalsze działania projektowe. Testowanie jest jednak fazą, w której solidnie przeprowadzone działania pozwalają stwierdzić, czy spełnione zostały wymagania, a tym samym czy oprogramowanie można uznać za ukończone. Rzetelnie przeprowadzona weryfikacja produktu oraz wyciągnięcie odpowiednich wniosków i podjęcie ewentualnych działań, mogą mieć zatem pozytywny wpływ na satysfakcję końcowego odbiorcy projektu.

Testowanie systemów operacyjnych jest zadaniem trudnym, a sama tematyka nie doczekała się tak intensywnych badań, jak pozostałe dziedziny weryfikacji oprogramowania. Z tego powodu trudno wskazać uniwersalne i skuteczne techniki testowania kodu jądra. Klasyczne podejścia oferowane przez inżynierię oprogramowania, takie jak metody strukturalne czy obiektowe [35], wydają się być w znacznej mierze nieużyteczne lub niewystarczające. W związku z powyższym, programiści systemowi najczęściej stosują autorskie sposoby testowania, dostosowane indywidualnie do natury weryfikowanych projektów.

Przykład takiego procesu testowania przedstawiono w niniejszym rozdziale, opierając się na testach rozważanego mechanizmu. Pierwsza jego część opisuje testy poprawnościowe zrealizowane w postaci testów systemowych. Kolejny fragment koncentruje się na przebiegu sprawdzania wydajności zaimplementowanego rozwiązania. W celu wykazania rzeczywistej użyteczności mechanizmu, w następnym podrozdziale przedstawiono przykładowe wdrożenia, które zostały zrealizowane w fazie testów. Rozdział kończy zestaw krytycznych komentarzy odnoszących się do realizacji poszczególnych wymagań postawionych w fazie projektowania.

5.1. Testy systemowe

Rzetelne wykazanie poprawności działania oprogramowania jest zadaniem trudnym. W zastosowaniach wymagających pewności, takich jak np. militaria, lotnictwo czy energetyka, stosuje się ustandaryzowane modele i dowody matematyczne. Ich zastosowanie wymaga jednak bardzo dużych nakładów czasowych i finansowych dlatego w praktyce, gdy absolutna poprawność nie jest konieczna wykorzystuje się pozostałe techniki. Klasyczna inżynieria oprogramowania [35] w fazie testowania proponuje zastosowanie testów różnych rodzajów (np. statystycznych, strukturalnych, modułowych), które wykazują, iż system działa poprawnie w pewnych z góry określonych przypadkach. Zastosowanie tej techniki nie daje pewności, iż oprogramowanie będzie zawsze działało właściwie — dla programów nawet niewielkich rozmiarów liczba możliwych przebiegów wykonania jest nieograniczona.

Testowanie kodu jądra znacząco różni się od testowania innych rodzajów oprogramowania. Weryfikacja pojedynczych elementów — procedur i struktur danych, a często nawet całych modułów, zwykle nie jest możliwa w odłączeniu od pozostałych elementów systemu operacyjnego. W procedurze testowania jądra nie stosuje się zatem testów jednostkowych i modułowych, a głównie testy operujące na działającym jądrze. Dla rozważanego projektu wybrana została więc metoda testowania systemowego. Testy systemowe polegają na wykonaniu w kompletnym, działającym systemie czynności odpowiadających konkretnym wymaganiom, a następnie ocenie poprawności ich rezultatów. Testowanie systemowe pozwala więc zweryfikować realizację wymagań w oderwaniu od faktycznej implementacji.

Pierwszym krokiem testowania systemowego było wyodrębnienie podstawowych operacji, które można wykonać za pomocą zaimplementowanego podsystemu. Lista tych operacji wygląda następująco:

- podmontowanie podsystemu w systemie plików,
- zabronienie wywołania systemowego,
- zezwolenie na wywołanie systemowe,
- stworzenie podgrupy,
- usunięcie podgrupy,
- przeniesienie zadania pomiędzy grupami,
- utworzenie nowego zadania z zadania już istniejącego (*fork()*),
- próba nadania uprawnień sprzecznych z założeniami,
- podanie błędnego numeru wywołania systemowego
- odmontowanie podsystemu z systemu plików.

Na podstawie powyższej listy zaprojektowano i zaimplementowano następnie piętnaście zestawów testowych (ang. *test suites*), które łącznie wykonują i sprawdzają poprawność ponad siedemdziesięciu elementarnych czynności. W celu automatyzacji procesu testowania, rozbudowano infrastrukturę stworzoną w fazie implementacji, uzyskując tym samym proste narzędzie testujące. System uruchomieniowy ma postać zbioru skryptów powłoki *sh* przez co jest łatwy w utrzymaniu i ewentualny dalszym rozwijaniu.

Tak zorganizowane testy systemowe wykazują, że moduł działa poprawnie w przypadkach przewidzianych w zestawach testowych. Nie można jednak zakładać, iż system jest bezbłędny — nie wszystkie sytuacje da się przewidzieć i zasymulować. Przykładem mogą być operacje współbieżne wykonywane na module — nie istnieje możliwość łatwego sprawdzenia wszystkich prawdopodobnych przebiegów czasowych, a tworzenie testów losowych przeczy ich powtarzalności. Rzeczywista poprawność modułu zostanie więc sprawdzona podczas faktycznej eksploatacji w środowisku produkcyjnym.

Pomimo tego, przeprowadzone testy systemowe należy zaliczyć do udanych. Sytuacje w nich sprawdzone pokrywają wymagania funkcjonalne stawiane modułowi, a więc istnieje rzeczywista podstawa do stwierdzenia, iż moduł spełnia te wymagania. Dodatkowo, dzięki testom udało się zidentyfikować, wyizolować i poprawić dwa istotne błędy. Pierwszy z nich dotyczył niewłaściwej synchronizacji dostępu do obiektów systemowych, drugi natomiast umożliwiał obejście zabezpieczeń poprzez wykorzystanie trybu kompatybilności *IA32* na architekturze *x86-64*. Pominięcie fazy testowania systemowego najprawdopodobniej doprowadziłoby więc do wdrożenia wadliwego modułu narażając tym samym potencjalnych użytkowników na straty. Na zakończenie warto zauważyć, iż stworzona infrastruktura testowa może

być użyteczna podczas dalszego rozwijania podsystemu np. do kontroli błędów regresji czy też weryfikacji nowych funkcjonalności.

5.2. Testy wydajnościowe

Kolejnym krokiem po sprawdzeniu poprawności działania mechanizmu jest zbadanie generowanego narzutu czasowego. Dokładne wymagania wydajnościowe zostały zdefiniowane na etapie projektowania, a zatem kontrola ich spełnienia jest istotnym elementem całościowej oceny modułu. Procedurę testowania efektywności czasowej mechanizmu skonstruowano na podstawie schematów testów innych rozwiązań [24, 46, 54, 59] tak, aby możliwe było ewentualne zestawienie i porównanie wszystkich implementacji. Podczas tworzenia testów starano się, aby były one możliwe do łatwego powtórzenia na innej maszynie w celach weryfikacji otrzymanych wyników.

Testy wydajnościowe składają się z dwóch etapów: mikrotestów (ang. *micro-benchmarks*) oraz makrotestów (ang. *macrobenchmarks*). Zadaniem mikrotestów jest wykazanie narzutu czasowego dla pojedynczych wywołań o odmiennej złożoności i naturze, natomiast makrotesty sprawdzają wydajność standardowych aplikacji o różnych charakterystykach. Przedmiotem każdego testu było porównanie czasów wykonania poszczególnych operacji gdy:

- mechanizm nie został skompilowany w jądro,
- mechanizm został skompilowany, ale jest nieaktywny — nie są zdefiniowane żadne ograniczenia,
- mechanizm został skompilowany i jest aktywny — zostały zdefiniowane ograniczenia, które nie wpływają na wykonywane operacje.

Dla wszystkich testów (poza testem sieciowym) mierzony był czas zegarowy (ang. *wall-clock time*), użytkowy (ang. *user CPU time*) oraz systemowy (ang. *system CPU time*) za pomocą narzędzia *time* (opakowanie usługi *getrusage()*) dostępnego w powłoce *bash*. W celu zwiększenia precyzji pomiarów każdy test został powtórzony 100 razy, a w odpowiednich tabelach zostały umieszczone wartości średnie i odchylenia standardowe¹ odczytów. Narzut wydajnościowy obliczano na podstawie typowego wzoru:

$$\frac{t_m - t_r}{t_r} \cdot 100\%$$

gdzie: t_m — czas operacji przy skompilowanym (aktywnym bądź nie) mechanizmie, t_r — czas operacji przy nieskompilowanym mechanizmie.

Testy uruchamiane były z partycji o systemie plików *ext3* na dystrybucji *Arch Linux* [57] z jądrem w wersji 3.1.6 (konfiguracja kompilacji standardowa dla dystrybucji, wzbogacona w zależności od potrzeby o opisywany moduł) przez superużytkownika. Specyfikacja maszyny testowej: *Intel Core 2 Duo P8700 @ 2.53GHz*, *4 GB RAM DDR2 (800 MHz)*, *dysk SATA 500 GB 5400 RPM*. Na maszynie testowej zostały wyłączone wszystkie niepotrzebne usługi tak, aby zewnętrzne czynniki (takie jak np. inne programy, czy bardzo częste przerwy) nie zakłócały wyników.

¹ Użyty w tabelach znak $\pm$ należy traktować wyłącznie symbolicznie — jako separator wartości średnich i odchylenia standardowego.

5.2.1. Mikrotesty

Mikrotesty pozwalają określić narzut wydajnościowy spowodowany działaniem mechanizmu dla pojedynczych wywołań systemowych. W celu przeprowadzenia testów tego rodzaju wybrano cztery wywołania o odmiennej charakterystyce:

getuid() — jedno z najprostszych wywołań, które zwraca *UID* właściciela wykonującego się zadania; należy się spodziewać, iż narzut spowodowany działaniem mechanizmu będzie relatywnie duży, gdyż operacje wykonywane przez to wywołanie są trywialne i w zasadzie ograniczają się do odczytu pola struktury *task_struct*,

setuid() — wywołanie o bardziej złożonej logice niż *getuid()*, ustawiające m.in. *UID* dla wykonującego się zadania; w teście argumentem wywołania była wartość 0, a więc żądano najwyższych uprawnień superużytkownika,

chdir() — wywołanie to ustawia aktualny katalog roboczy (ang. *current working directory, cwd*); wykonanie tego wywołania wiąże się z pewnymi operacjami wejścia/wyjścia, gdyż sprawdzana jest poprawność ścieżki oraz uprawnienia dostępu dla podanego katalogu — należy zatem spodziewać się wydłużonego czasu działania; w teście argumentem wywołania była ścieżka trójkomponentowa */usr/src/linux*,

read() — najbardziej złożone z testowanych wywołań, odczytujące dane z obiektu reprezentowanego przez zadany deskryptor; w testach odczytywano za jednym razem 32 bajty z pseudourządzenia */dev/urandom*; narzut wydajnościowy dla tego wywołania powinien być niewielki, gdyż oprócz skomplikowanej logiki operacji, w trakcie wykonania kopiowane są obszary pamięci.

Dobór tak zróżnicowanych usług systemowych powinien ukazać jak kształtuje się narzut wydajnościowy spowodowany działaniem mechanizmu w zależności od intensywności i rodzaju wywołań. Aby zminimalizować wpływ tworzenia kontekstu na wynik czasowy, pojedynczy test składał się z 10^8 wywołań. Zgodnie z założeniami każdy test wykonywany był 100 razy.

getuid()

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	7.965 ± 0.012	3.760 ± 0.081	4.199 ± 0.083	—	—	—
Nieaktywny	7.990 ± 0.005	3.742 ± 0.100	4.243 ± 0.100	0.314	-0.479	1.048
Aktywny	8.846 ± 0.055	3.731 ± 0.097	5.109 ± 0.108	11.061	-0.771	21.672

Tablica 5.1: Mikrotest *getuid()*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Mikrotest najprostszego wywołania pokazuje, iż zgodnie z przewidywaniami, cały narzut wydajnościowy widoczny w czasie zegarowym wykonania, składa się z czasu, w którym zadanie przebywa w przestrzeni jądra (czas systemowy). Ujemny narzut widoczny w czasie użytkowym jest anomalią wynikającą z niedokładności pomiarów — w idealnym przypadku należałoby się spodziewać, iż czas ten będzie identyczny — niezależny od stanu mechanizmu. Takie wyniki pomiarów wynikają wprost z założeń projektowych — cały mechanizm zaimplementowany został w przestrzeni jądra, a więc to tam następuje wydłużenie wykonania.

Narzut wydajnościowy spowodowany działaniem mechanizmu, dla tak prostego wywołania systemowego można uznać za znaczący. Gdy mechanizm jest nieaktywny, czas spędzony w przestrzeni jądra jest większy o około 1%, co przekłada się

na opóźnienie rzędu 0.3% dla całego wywołania. W przypadku, gdy zdefiniowane zostały pewne ograniczenia, czas systemowy wydłuża się o ponad 21% przekładając się na łączne 11% narzutu na wykonanie wywołania. Operacje związane ze sprawdzeniem polityki (skok do funkcji oraz odczyt bitu zezwolenia) są zdecydowanie bardziej skomplikowane niż wszystkie instrukcje usługi *getuid()*, co tłumaczy tak wysoki narzut wydajnościowy. Zgodnie z postawionymi wymaganiami, typowe zadania powinny być opóźniane o nie więcej niż 5% — nie należy jednak w tym momencie stwierdzać, iż warunek ten nie jest spełniony — trudno bowiem wskazać aplikacje, które wykonują wyłącznie najprostsze wywołania z dużą częstotliwością.

setuid()

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	52.344 ± 0.493	3.889 ± 0.114	40.693 ± 0.494	—	—	—
Nieaktywny	52.331 ± 0.540	3.864 ± 0.103	40.732 ± 0.569	-0.025	-0.643	0.096
Aktywny	53.271 ± 0.441	3.892 ± 0.106	41.610 ± 0.503	1.771	0.077	2.253

Tablica 5.2: Mikrotest *setuid()*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Setuid() jest wywołaniem o dużo bardziej skomplikowanej logice niż *getuid()* — czas wykonania wg pomiarów jest ponad sześciokrotnie większy. W związku z tym, narzut wydajnościowy spowodowany działaniem modułu jest znacznie mniejszy. W przypadku, gdy mechanizm jest nieaktywny, nakład czasowy oscyluje w granicach błędu pomiarowego, natomiast podczas regularnego funkcjonowania podsystemu, czas spędzony w przestrzeni jądra jest o nieco ponad 2% większy, co przekłada się na ogólny narzut w wysokości 1.7%. Mikrotest ten potwierdza również, iż cały narzut zawiera się w czasie systemowym — czas spędzony w przestrzeni użytkownika nie wydłuża się, gdy mechanizm jest skompilowany w jądrze. Dobry wynik wywołania o bardziej skomplikowanej logice przetwarzania niż *getuid()* pozwala przypuszczać, iż narzut dla kompletnych aplikacji będzie niewielki — mieszczący się w wyznaczonych granicach.

chdir()

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	98.341 ± 0.869	3.576 ± 0.115	94.710 ± 0.879	—	—	—
Nieaktywny	100.067 ± 0.749	3.741 ± 0.145	96.271 ± 0.741	1.755	4.614	1.648
Aktywny	101.102 ± 0.751	3.726 ± 0.118	97.319 ± 0.765	2.808	4.195	2.755

Tablica 5.3: Mikrotest *chdir()*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Wyniki czasowe dla wywołania *chdir()* znacząco odstają od pozostałych testów i są trudne do zinterpretowania. Najbardziej zaskakującą obserwacją jest narzut wydajnościowy (ok. 4%) w czasie użytkowym, który ciężko związać z jakimś faktem. Można się spodziewać, iż wpływ na wyniki doświadczenia ma fakt, iż wywołanie *chdir()* wykonuje operacje dyskowe, których przebieg zależy od wielu losowych czynników. Hipotezę tę potwierdzają miary niepewności (odchylenie standardowe), które dla czasów *real* oraz *sys* są o połowę większe niż w pozostałych mikrotestach.

Na uwagę zasługuje jednak fakt, iż mimo wszystko ogólny narzut związany z mechanizmem jest niewielki i wynosi niespełna 2% i 3% odpowiednio gdy mechanizm jest nieaktywny i aktywny.

read()

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	342.565 ± 0.381	4.536 ± 0.167	337.848 ± 0.434	—	—	—
Nieaktywny	342.761 ± 0.356	4.410 ± 0.158	338.166 ± 0.387	0.057	−2.778	0.094
Aktywny	343.494 ± 0.262	4.479 ± 0.142	338.834 ± 0.281	0.271	−1.257	0.292

Tablica 5.4: Mikrotest *read()*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Read() jest najwolniejszym (ponad czterdziestokrotnie dłuższe działanie niż *getuid()*) wywołaniem spośród wszystkich rozważanych. Z tego też powodu zaobserwowany narzut wydajnościowy jest znikomy i wynosi znacznie poniżej 1%. Można zatem spodziewać się, iż zastosowanie mechanizmu w aplikacjach odczytujących duże ilości danych nie wykaże zauważalnego spowolnienia działania.

Wnioski

Mikrotesty pojedynczych wywołań pokazały, że spodziewany narzut wydajnościowy dla typowych aplikacji najprawdopodobniej będzie mieścił się w określonych limitach. Jedynym niepokojącym wynikiem były czasy dla wywołania *getuid()* — trudno jednak zoptymalizować funkcję kontrolującą politykę w taki sposób, aby była prostsza niż ta usługa jądra. Z tego powodu trudno spodziewać się, aby było możliwe osiągnięcie w pełni zadowalających wyników dla tego właśnie wywołania.

Analizując wyniki wszystkich testów można zauważyć, że dla liczby wywołań 10^8 narzut czasu systemowego wyrażony w sekundach jest zbliżony i wynosi ok. 0.045 s oraz 0.930 s odpowiednio gdy mechanizm jest nieaktywny i aktywny. Jest to zgodne z oczekiwaniami, gdyż według założeń projektowych, niezależnie od wywołania operacje wykonywane przez moduł są takie same. Na tej podstawie można także szacować, iż na maszynie testowej, skompilowanie mechanizmu powoduje dla pojedynczego wywołania narzut czasowy ok. 0.5 ns, natomiast sprawdzenie polityki trwa ok. 10 ns. Liczby te obarczone są jednak dużym błędem pomiarowym i należy traktować je wyłącznie orientacyjnie.

5.2.2. Makrotesty

Zadaniem makrotestów jest sprawdzenie najważniejszego wskaźnika kwestii wydajności — narzutu dla typowych aplikacji. Oczywistym jest, iż niemożliwe jest zbadanie każdego istniejącego programu, a ponadto określenie, które aplikacje są „typowe” jest zadaniem trudnym ze względu na brak obiektywnych źródeł oceny. Z powyższych powodów, do makrotestów wybrano aplikacje, które zyskały uznanie i popularność w różnych testach wydajnościowych, a przy tym reprezentują odmienną naturę wykonywanych operacji. Lista stworzonych makrotestów wygląda następująco:

gzip — pierwszy test polegał na kompresji pliku przy pomocy narzędzia *gzip*, które jest aplikacją intensywną obliczeniowo (*CPU bound process*); w teście kompresowany był plik o rozmiarze 1 GB zawierający losowe dane,

find + cat — kolejny test składał się z rekursywnego przejścia po wszystkich katalogach kodu źródłowego *Linuksa* (wersja 3.1.6) za pomocą programu *find* i odczytaniu zawartości poleceniem *cat* bez wyprowadzania wyników na standardowe wyjście; takie zestawienie powoduje wykonanie wielu wywołań systemowych związanych z dyskowymi operacjami wejścia/wyjścia (*I/O bound process*),

kompilacja jądra — kolejny test polegał na kompilacji jądra *Linuksa*, która jest najbardziej popularnym testem wśród programistów tego systemu; operacje wykonywane podczas tego procesu są skomplikowane, bogate w wywołania systemowe i składają się z wielu obliczeń oraz operacji wejścia/wyjścia; można zatem powiedzieć, iż kompilacja w uniwersalny sposób testuje wydajność i ogólną odporność systemu na błędy; podczas testu jądro kompilowane było z domyślną konfiguracją oraz w czterech równoległych zadaniach (opcja *-j4*) tak, aby przy tej sposobności przetestować jakość synchronizacji zastosowanej w mechanizmie,

apache httpd — ostatni test polegał na przetestowaniu wydajności demona http uruchomionego na maszynie testowej; narzędziem użytym w tym celu było *ab* — popularny tester serwerów HTTP, przeznaczony specjalnie dla *apache httpd*; test miał za zadanie sprawdzić wydajność mechanizmu w zastosowaniach sieciowych, w których występuje wiele wywołań systemowych; demon serwował stronę o rozmiarze 150 KB, a narzędzie *ab* wykorzystywało 100 wątków (*-c100*) wykonujących łącznie 10000 zapytań (*-n10000*), komputery połączone były w sieć *Ethernet 100Base-TX*.

Każdy z powyższych testów został powtórzony 100 razy.

gzip

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	47.919 ± 0.379	45.349 ± 0.068	2.278 ± 0.058	—	—	—
Nieaktywny	48.060 ± 0.479	45.388 ± 0.278	2.294 ± 0.088	0.294	0.086	0.702
Aktywny	48.113 ± 0.876	45.429 ± 0.834	2.359 ± 0.064	0.405	0.176	3.556

Tablica 5.5: Makrotest *gzip*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Wedle oczekiwań, proces *gzip* większość czasu spędza w przestrzeni użytkownika ze względu na intensywność obliczeniową. Zgodnie z obserwacjami przeprowadzonymi w fazie mikrotestów, narzut wydajnościowy nie ujawnia się w czasie użytkownika, a jedynie w czasie spędzonym w przestrzeni jądra i z tego powodu wnikliwej analizie należy poddać czas systemowy i jego wpływ na postrzegany czas pracy całego zadania.

Liczba wykonanych wywołań systemowych jest relatywnie niewielka, stąd niski czas systemowy procesu. Pomimo tego, zaobserwowany narzut wydajnościowy czasu systemowego jest nieznaczny — ok. 0.7%, gdy żadna polityka nie jest zdefiniowana oraz ok. 3.5% gdy mechanizm dokonuje faktycznej kontroli. Ze względu na znacznie dłuższy czas działania całego procesu, całkowity narzut jest w każdym z przypadków znacznie mniejszy niż 1%.

Ze względu na bardzo dobre wyniki czasu systemowego dla pliku o wielkości 1 GB, można się spodziewać, że dla plików o niewielkich rozmiarach opóźnienie czasu działania *gzip* również będzie się mieściło w określonych limitach. Jest to jednak trudne do zbadania ze względu na niedokładność pomiarów przy krótkich czasach działania.

find + cat

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	57.920 ± 4.300	7.288 ± 0.642	12.997 ± 1.096	—	—	—
Nieaktywny	57.491 ± 4.374	7.186 ± 0.595	12.986 ± 1.133	−0.741	−1.400	−0.085
Aktywny	58.427 ± 4.068	7.347 ± 0.600	13.147 ± 1.016	0.875	0.810	1.154

Tablica 5.6: Makrotest *find + cat*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Kolejny makrotest, podobnie jak mikrotest *chdir()*, obarczony jest dużymi niepewnościami pomiarowymi, które również należy łączyć z losowym przebiegiem operacji wejścia/wyjścia. Działania związane z odwiedzaniem katalogów i odczytem plików są złożone i wykonują wiele żądań usług w przestrzeni jądra, stąd czas systemowy znacznie przewyższający czas użytkowy procesu. Pomimo tego, narzut wydajnościowy spowodowany działaniem mechanizmu jest znikomy. W przypadku gdy mechanizm jest nieaktywny różnice w czasach oscylują w granicy błędów pomiarowych, natomiast podczas faktycznej pracy, czas systemowy jest wydłużony o nieco ponad 1%. Czas zegarowy działania całego procesu znacznie przewyższa sumę czasu użytkownika i czasu systemowego ze względu na oczekiwanie na operacje wejścia/wyjścia. Z tego powodu ogólny, postrzegany narzut wydajnościowy jest jeszcze mniejszy — poniżej 1%.

Kompilacja jądra

Stan mechanizmu	Czasy [s]			Narzut [%]		
	real	user	sys	real	user	sys
Nieskompilowany	345.024 ± 1.645	588.541 ± 0.509	66.895 ± 0.328	—	—	—
Nieaktywny	344.277 ± 1.412	587.687 ± 0.518	66.614 ± 0.275	−0.217	−0.145	−0.420
Aktywny	345.066 ± 0.698	589.251 ± 0.747	67.456 ± 0.419	0.012	0.121	0.839

Tablica 5.7: Makrotest *kompilacja jądra*: pomiary czasowe — wartości średnie oraz odchylenia standardowe i obliczone na tej podstawie narzuty wydajnościowe.

Kompilacja to najobszerniejszy ze wszystkich makrotestów, który dodatkowo łączy operacje wejścia/wyjścia z intensywnymi obliczeniami. W wynikach testu może dziwić fakt, iż czas zegarowy jest mniejszy od sumy czasu użytkownika i czasu systemowego — dzieje się tak, ponieważ kompilacja prowadzona była w czterech równoległych zadaniach wykonywanych na dwóch rdzeniach, dla których czas obliczany jest niezależnie.

Najważniejszą obserwacją testu jest to, że wysoka intensywność wywołań systemowych testu nie przekłada się na wysoki narzut wydajnościowy, który dla mechanizmu nieaktywnego jest zanedbywalnie mały, a w przypadku aktywnej pracy dla czasu systemowego wynosi mniej niż 1%. Biorąc pod uwagę długi łączny czas wykonania testu, narzut staje się niezauważalny. Należy pamiętać, iż kompilacja wykonywana była współbieżnie — w przypadku braku zrównoleglenia, wydłużenie czasu systemowego przez mechanizm powodowałoby jeszcze mniejszy wpływ na ogólny czas pracy. Dobre wyniki czasu systemowego pokazują również dobrą jakość zastosowanej synchronizacji — zadania znajdujące się w tym samym kontenerze nie blokują się wzajemnie.

apache httpd

Stan mechanizmu	Zapytania na sekundę [1/s]	Czas zapytania [ms]	Transfer [KB/s]
Nieskompilowany	76.04 ± 0.004	13.152 ± 0.001	11470.25 ± 0.509
Nieaktywny	76.04 ± 0.000	13.152 ± 0.001	11470.37 ± 0.581
Aktywny	76.04 ± 0.000	13.152 ± 0.000	11470.46 ± 0.070

Tablica 5.8: Makrotest *apache httpd*: najważniejsze statystyki zwrócone przez narzędzie *ab* — wartości średnie i odchylenia standardowe.

Wyniki ostatniego z makrotestów nie wykazały żadnego narzutu wydajnościowego dla aplikacji sieciowej. Takie rezultaty można uzasadnić tym, iż opóźnienia wprowadzane przez łącze danych są nieporównywalnie większe od opóźnień mechanizmu. Rzetelne przeprowadzenie testu z demonem *httpd* wymagałoby zbierania długoterminowych osiągnięć rzeczywistego wdrożenia mechanizmu na serwer i szczególnej analizy okresów maksymalnego obciążenia. Tylko w takich warunkach możliwe byłoby zaobserwowanie spadków wydajności, których oczekiwana wartość wynosi maksymalnie kilka procent. Należy jednak pamiętać, że wyniki takiego testu obciążone byłyby dużymi niepewnościami pomiarowymi ze względu na niedeterminizm większości sieci komputerowych.

Wnioski

Makrotesty pokazały, iż zastosowanie opisywanego mechanizmu nie powoduje znaczących narzutów wydajnościowych w aplikacjach różnego typu. Osiągnięte przez moduł wyniki mieszczą się w określonych rygorystycznych limitach pozostawiając przy tym istotny „zapas” czasowy. Narzut wydajnościowy w aplikacjach intensywnych obliczeniowo pozostaje niewielki, gdyż znaczna część czasu pracy aplikacji to czas spędzony w przestrzeni użytkownika, na którą moduł nie ma wpływu. Dla aplikacji wykonujących wiele operacji wejścia/wyjścia opóźnienie jest natomiast zanedbywalnie małe ze względu na długi czas oczekiwania na gotowość urządzeń. Na koniec warto zaznaczyć, iż wobec powyższych spostrzeżeń, całkowity narzut spowodowany działaniem mechanizmu maleje wraz z całkowitym czasem działania aplikacji. Jest to cecha jak najbardziej pożądana, gdyż wydajność jest zwykle ważniejsza w programach działających relatywnie długo, niż w aplikacjach wykonujących proste i krótkie operacje.

5.2.3. Podsumowanie

Wykonane testy wydajnościowe pokazały, iż efektywność czasowa mechanizmu jest zadowalająca i spełnia postawione wymagania. Warto także zaznaczyć, iż osiągi modułu są widocznie lepsze od wspomnianych wcześniej implementacji referencyjnych. Należy jednak pamiętać, iż podsystem od początku projektowany był z zamiarem osiągnięcia minimalnego narzutu, mniejszego od innych dostępnych projektów — z tego też powodu jego funkcjonalność jest często uboższa. Można się spodziewać, iż wzbogacenie projektu o nowe możliwości zmniejszyłoby jego wydajność. Nie można zatem jednoznacznie wskazać, które rozwiązanie jest najlepsze — wszystko zależy od konkretnych potrzeb końcowego użytkownika.

5.3. Przykładowe wdrożenia

Wyżej przedstawione testy poprawnościowe i wydajnościowe mają charakter symulacyjny, a więc są w pewien sposób sztuczne i nie ukazują rzeczywistego zastosowania mechanizmu. W celu wykazania realnej przydatności zaimplementowanego modułu wdrożono go w dwóch systemach, w których użycie opisywanego podsystemu ma faktyczne uzasadnienie. Testowanie w środowisku produkcyjnym jest, w przeciwieństwie do testowania systemowego, długotrwałe i pozwala zidentyfikować inną klasę błędów np. wycieki pamięci czy zakleszczenia. Dodatkowo realne wdrożenie pozwala poznać odczucia użytkowników, na podstawie których można wnioskować o faktycznej przydatności mechanizmu. Tego typu opinie mogą okazać się szczególnie przydatne przed fazą powszechnego zastosowania, która ma zwykle miejsce po zakończeniu etapu testowania.

5.3.1. Serwer DNS

Pierwsze wdrożenie polegało na zastosowaniu mechanizmu do kontrolowania serwera DNS (*nsd* [36]) niewielkiej sieci lokalnej. W początkowej fazie użyto narzędzie *strace* do określenia, które wywołania systemowe są potrzebne do prawidłowego działania usługi, a na tej podstawie skonstruowano zbiór wywołań zabronionych. Po skompilowaniu jądra w wersji z mechanizmem ochrony, dla oprogramowania serwera stworzono specjalną grupę kontrolną, w której wprowadzono wspomniany zbiór niepożądanych wywołań. Podczas kilku kolejnych dni normalnej pracy demona *nsd* nie zanotowano żadnych dysfunkcji czy zauważalnych spadków wydajności.

Aby zaprezentować przydatność mechanizmu, w kolejnym kroku wprowadzono do kodu źródłowego *nsd* sztuczny błąd (gdyż aktualnie nie są znane błędy bezpieczeństwa w oprogramowaniu *nsd*) pozwalający wykonać w zasadzie dowolny kod. Uruchomienie tak spreparowanego demona *nsd* w systemie z nieskompilowanym podsystemem ochrony wywołań systemowych umożliwiło przejęcie kontroli nad maszyną i np. usunięcie bądź odczytanie istotnych plików, utworzenie sesji zdalnego logowania czy też dalsze penetrowanie zasobów sieci. Szkody wyrządzone przez atakującego mogły być wyjątkowo dotkliwe, gdyż demon ten działa z uprawnieniami superużytkownika. Próba wykonania tych samych operacji, gdy serwer został uruchomiony wewnątrz restrykcyjnej grupy kontrolnej zakończyła się niepowodzeniem. Możliwości atakującego były bardzo niewielkie i w zasadzie ograniczały się do wpływania na sposób działania samego *nsd*, a nie całego systemu.

Wdrożenie mechanizmu do takiego zastosowania można zatem uznać za sukces, gdyż istotnie zwiększa bezpieczeństwo maszyny serwerowej. Należy jednak zauważyć, iż znaczne ograniczenie możliwości atakującego było możliwe, ponieważ *nsd* używa wąskiego, dobrze zdefiniowanego zbioru wywołań systemowych. Warto dodać, iż wyznaczenie odpowiedniego zbioru wywołań zabronionych jest zadaniem trudnym, co stanowi istotną wadę tego mechanizmu bezpieczeństwa.

5.3.2. Serwer zdalnej kompilacji i uruchamiania

Po zakończeniu etapu implementacji, za pośrednictwem serwisu *Unix & Linux Stack Exchange* [31] skupiającego użytkowników tych systemów, nawiązano kontakt z osobą potrzebującą rozwiązania typu *sandbox*. Mechanizm tego typu potrzebny był do zabezpieczenia serwera zdalnej kompilacji i uruchamiania, który

służy prowadzeniu zawodów algorytmicznych podobnych do SPOJ [37] czy TopCoder [32]. Uczestnicy konkursów tego typu nadsyłają rozwiązania pewnych zadań algorytmicznych w postaci kodów źródłowych, które są następnie kompilowane i uruchamiane. Rozwiązania otrzymane od zawodników muszą być testowane w izolowanym środowisku tak, aby nie mogły one w żaden sposób wpłynąć na maszynę oceniającą. Administrator serwera zgodził się na testowe zastosowanie podsystemu do stworzenia piaskownicy dla zadań zawodników.

W przypadku tego wdrożenia wyznaczenie listy wywołań dozwolonych było łatwe, gdyż wynikają one wprost z regulaminu zawodów. Proces konkursowy może odczytywać standardowe wejście (*read()*), zapisywać dane do standardowego wyjścia (*write()*), alokować pamięć (*brk()*) oraz zakończyć działanie (*exit()*, *exit_group()*). Dla zadań zawodników stworzono więc odpowiednią grupę i skonfigurowano wyżej wymienioną politykę. Tak skonstruowany *sandbox* zapewnił dokładnie taki poziom ochrony, jaki był wymagany. Administrator serwera szczególnie zadowolony był z łatwości obsługi modułu oraz z jego wysokiej wydajności. Zaimplementowany podsystem zastąpił starszy, niewydajny mechanizm oparty na wywołaniu *ptrace()*, który przez swoją nieefektywność wpływał ważąco na wyniki czasowe rozwiązań. Ostatecznie moduł uzyskał pozytywną ocenę w testach i bardzo prawdopodobne, że zostanie zastosowany w produkcyjnej wersji serwera zawodów.

5.4. Ocena realizacji wymagań

Po zakończeniu fazy testów należy rzetelnie odnieść się do wymagań postawionych na etapie projektowania. Bezpośrednim wskaźnikiem realizacji oczekiwań są tu wyniki wszystkich przeprowadzonych testów i wdrożeń. Poniżej opatrzone zatem komentarzem dotyczącym jakości realizacji każde ze sformułowanych wymagań.

5.4.1. Wymagania funkcjonalne

1. Możliwości kontroli

Zgodnie z oczekiwaniami, moduł daje możliwość zdefiniowania dowolnego zbioru wywołań zabronionych poprzez określenie numerów niepożądanych usług. Kontroli podlegają pojedyncze zadania *Linuksa*, co wynika z konstrukcji mechanizmu grup kontrolnych. Istnieje możliwość skonfigurowania indywidualnego zbioru ograniczeń dla wskazanego zadania, jednak może to okazać się niewygodne, gdyż wymaga stworzenia oddzielnej grupy. Przy dużej ilości zadań z odmiennymi uprawnieniami takie rozwiązanie może okazać się nienaturalne, trudne w zarządzaniu i nieefektywne pamięciowo. Należy jednak zaznaczyć, że zwykle niewielka liczba zadań powinna podlegać jakimkolwiek ograniczeniom, a dodatkowo często są one podobne bądź jednakowe (zob. podrozdział 2.2.1) — w takim wypadku mechanizm będzie przyjazny w użytkowaniu.

2. Poziom bezpieczeństwa

Pierwsze z wymagań poziomu bezpieczeństwa jest zapewnione przez sam rdzeń mechanizmu grup kontrolnych. Po podmontowaniu systemu plików modułu, który jest jedynym sposobem konfiguracji, domyślnymi uprawnieniami dostępu jest zestaw bitów *-rw-r--*, a więc w istocie jedynie administrator może wprowadzać zmiany w działaniu podsystemu. Dodatkowo, co należy uznać jako

zaletę, istnieje naturalna możliwość modyfikacji uprawnień pliku oraz zastosowanie dodatkowego mechanizmu ACL uzyskując tym samym bardziej złożoną i dostosowaną do potrzeb politykę dostępu do konfiguracji podsystemu.

W kwestii realizacji kolejnego podpunktu należy zauważyć, iż zadanie nie ma możliwości pominięcia ochrony dostępu, gdyż funkcja podpięcia wywoływana jest w każdej ścieżce, przez którą może przechodzić zadanie wykonujące wywołanie systemowe. Aby wykazać słuszność tego stwierdzenia, testy systemowe używają wszystkich istniejących ścieżek żądań wywołań w celu weryfikacji wymuszania polityki dostępu. Jedynym niebezpieczeństwem pozwalającym potencjalnie ominąć mechanizm, mogą być błędy w innych modułach jądra, które umożliwiają wykonanie skoku pod dowolny adres — w tym bezpośrednio pod adres wywołania systemowego. Wpłynięcie na tę kwestię jest jednak w ramach zaimplementowanego modułu niemożliwe.

3. **Możliwości konfiguracji**

Mechanizm zezwala na modyfikację polityki w dowolnym momencie — podsystem może być modyfikowany już od momentu inicjalizacji grup kontrolnych aż do wyłączenie systemu.

4. **Interfejs użytkownika**

Wygodna konfiguracja podsystemu jest wykonywana standardowo przy pomocy plików *tasks*, *syscalls.allow* oraz *syscalls.deny*. Interfejs plikowy jest wyjątkowo łatwy w użytkowaniu przez skrypty powłoki, gdyż podstawowe czynności wymienione w wymaganiu można wykonać za pomocą poleceń *cat*, *echo*, *ls* oraz *cd*. Jedyną niedogodnością związaną z konfigurowaniem podsystemu jest brak możliwości współbieżnego wykorzystywania interfejsu użytkownika (zob. podrozdział 4.2.4), co jednak w typowych zastosowaniach nie powinno utrudniać pracy.

5. **Współpraca z innymi mechanizmami bezpieczeństwa**

Podsystem może być bez problemów zestawiony z innymi mechanizmami bezpieczeństwa. Dodatkowo moduł jako element systemu grup kontrolnych jest idealnym narzędziem do wsparcia budowy kontenerów procesów.

5.4.2. **Wymagania нефunkcjonalne**

1. **Efekt pracy**

Modyfikacja jądra dostarczona jest w postaci łaty w zunifikowanym formacie *diff* zgodnym z dokumentem *Submitting Patches*. Na potrzeby projektu zmodyfikowany został również plik *Kconfig* tak, aby za pomocą narzędzia *menuconfig* istniała możliwość włączania i wyłączania kompilacji modułu. Efekt pracy jest więc w całości zgodny ze standardami przyjętymi dla rozwoju kodu *Linuksa*.

2. **Wersja jądra**

Modyfikacja testowana była dla jądra w wersji 3.1.6 dostępnej na oficjalnej stronie archiwów kodu jądra [39]. Warto dodać, iż modyfikacja sprawdzana była również z wersjami kandydującymi do wydania (ang. *release candidate*)

jądra 3.2, a więc istnieje duża szansa, iż będzie ona zgodna z wersją finalną.

3. Architektura sprzętu

Moduł oferuje identyczną funkcjonalność dla architektur *x86* oraz *x86-64*, co potwierdzają testy systemowe przeprowadzone na obydwu platformach. Kod maszynowy związany z architekturą *x86-64* jest jednak nieco bardziej skomplikowany, gdyż musi obsługiwać tryb zgodności z architekturą *x86*. Nie wpływa to jednak w żaden sposób na efekt końcowy widziany przez użytkownika mechanizmu.

4. Przenośność kodu

Zdecydowana większość mechanizmu została napisana w sposób przenośny w języku C przy wykorzystaniu wielu funkcji API jądra zapewniających efektywne działanie na różnych platformach. Podczas przenoszenia modułu na inną platformę można więc zakładać, iż część napisana w języku C będzie działać możliwie wydajnie dla danej architektury. Umieszczenie wywołania funkcji podpięcia w odpowiednich ścieżkach możliwe było jednak wyłącznie za pomocą języka specyficznego dla platformy. Przeniesienie mechanizmu na inną architekturę będzie więc wymagało napisania odpowiednich fragmentów w asemblerze. Nie powinno to jednak stanowić doraźnego problemu, gdyż operacje wykonywane w niskopoziomowym kodzie platformy sprowadzają się do wywołania funkcji i sprawdzenia wartości zwracanej.

W przyszłości warto mimo wszystko rozważyć przetłumaczenie fragmentów kodu w języku C na asembler w celu dalszej optymalizacji działania mechanizmu.

5. Wydajność

Wykonane testy wydajnościowe pokazują, iż dla pojedynczych, bardzo prostych wywołań systemowych narzut wydajnościowy przekracza 5%. Trudno sobie jednak wyobrazić programy wykonujące wyłącznie wywołania typu *getuid()* czy *getpid()*, do tego z dużą częstotliwością. Makrotesty pokazują natomiast, że dla standardowych programów, niezależnie od tego czy są związane z CPU czy z operacjami wejścia/wyjścia, mechanizm nie powoduje istotnego narzutu wydajnościowego — zarówno gdy jest włączony jak i wyłączony. Można zatem stwierdzić, że wymaganie to zostało spełnione, gdyż ciężko jest wskazać rzeczywiste aplikacje, dla których opóźnienia spowodowane przez mechanizm byłyby znaczące i przekraczałyby określone rygory.

6. Konwencja i styl kodu źródłowego

Kod mechanizmu pisany był z dbałością o styl i konwencję. Finalna jego wersja sprawdzona była pod tym kątem przez narzędzie *checkpatch.pl* dostarczane z jądrem, które za pomocą statycznej analizy ocenia jakość kodu.

Na podstawie powyższej analizy można stwierdzić, iż implementacja spełnia postawione wymagania. Na etapie testowania można zatem uznać końcowy produkt za sukces. Najważniejszym i najbardziej miarodajnym wskaźnikiem jakości pracy będzie jednak zainteresowanie użytkowników, a w dalszej fazie — ich satysfakcja. Dopiero na podstawie informacji zwrotnej można będzie bowiem ocenić, czy mechanizm spełnia swoje najważniejsze zadanie — zwiększa bezpieczeństwo dystrybucji *Linuksa*.

Podsumowanie

Ponieważ realizacja założeń została zweryfikowana pozytywnie, cel działań można uznać za osiągnięty. W pracy zaprezentowano kompletny proces powstawania oprogramowania dla systemu *Linux*, czego efektem końcowym jest moduł bezpieczeństwa. Pierwszym krokiem na drodze do stworzenia produktu była analiza rzeczywistych potrzeb użytkowników, dzięki której zidentyfikowano pewną klasę nierozwiązanych dotychczas problemów. Następnie, poza zbadaniem istoty samego zagadnienia, rozważono z czego wynika wspomniana nisza — dlaczego istniejące rozwiązania są niedostateczne? Na podstawie przeglądu dziedziny stworzono charakterystykę produktu, który może sprostać oczekiwaniom społeczności. Wspomniana charakterystyka została sformalizowana w postaci wymagań, które później kierowały przebiegiem prac w fazie projektowania i implementacji. Oba wymienione etapy okazały się szczególnie interesujące ze względu na specyfikę programowania w przestrzeni jądra *Linuksa*. Doświadczenie tego typu jest całkowicie odmienne od pozostałych rodzajów wytwarzanego oprogramowania — wymaga zastosowania odrębnych technik i zwrócenia uwagi na zupełnie inne zagadnienia powiązane głównie z systemami operacyjnymi i architekturą komputera.

Efektom prac przeprowadzonych na etapie implementacji jest funkcjonalny moduł bezpieczeństwa, który w następnym kroku został dokładnie przetestowany zarówno pod względem poprawności, jak i wydajności. Uzupełnieniem fazy testów były przeprowadzone wdrożenia, których zadaniem było pokazanie rzeczywistych przypadków użycia dla stworzonego mechanizmu. Cały proces opisany w pracy miał więc przebieg klasycznego projektu informatycznego, w którym na początku zebrane i spisane zostały wymagania, następnie na tej podstawie stworzono projekt, który został zaimplementowany, przetestowany i w końcu wdrożony.

Gotowy moduł wraz z przygotowanym zestawem testów będzie dalej rozwijany poza ramami pracy dyplomowej. Aktualnie, nadrzędnym celem projektu jest włączenie go do głównej linii *Linuksa*, co zostało zasygnalizowane w rozdziale dotyczącym implementacji. Decyzja taka wydaje się prawidłowa z kilku względów. Przede wszystkim dalszy rozwój projektu pozwala poszerzyć horyzonty wiedzy. Ponadto obecność kodu mechanizmu w *Linuksie* może spowodować, iż zostanie on powszechnie wykorzystany i okaże się przydatny dla wielu użytkowników. Szerokie wdrożenie może przyczynić się natomiast do odnalezienia niezidentyfikowanych wcześniej błędów, a tym samym do polepszenia jakości modułu. Istnieje również wiele pomysłów na dalsze wzbogacenie funkcjonalności i jakości działania mechanizmu, m.in.:

- wzbogacenie możliwości kontroli o sprawdzanie wartości konkretnych argumentów przekazanych do wywołania systemowego,
- stworzenie programu wspomagającego tworzenie polityk na podstawie automatycznej analizy wykonania programów,
- dalsze optymalizacje wydajności,
- przeniesienie na inne architektury sprzętowe, w szczególności *ARM*.

Niezależnie jednak od dalszych prac nad mechanizmem, w obliczu powyższych spostrzeżeń, pracę można uznać za wykonaną zgodnie z planem — zakończoną sukcesem.

A. Instrukcja modułu

A.1. Aplikacja łaty i kompilacja jądra

```
1 $ ls
2 syscalls_cgroup.patch
3 $ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.1.5.tar.bz2
4 $ tar -xjf linux-3.1.5.tar.bz2
5 $ cd linux-3.1.5
6 $ patch -p1 -s < ../syscalls_cgroup.patch
7 $ make menuconfig
8 ...
9 General setup -->
10     [*] Control Group support -->
11         [*] Syscalls controller for cgroups
12 ...
13 $ make
14 ...
15 Kernel: arch/x86/boot/bzImage is ready (#1)
16 ...
```

Wydruk A.1: Sekwencja poleceń wymagana do zaaplikowania łaty i skompilowania jądra.

Komentarz

Jak widać na podstawie linii 5, 6, narzędzie *patch* musi być uruchamiane z katalogu zawierającego pliki jądra, natomiast sama łata powinna znajdować się w katalogu nadrzędnym. Jest to związane z konwencją używania aplikacji *diff* i tym, jak opisuje ona ścieżki. Dodatkowe wyjaśnienia tych kwestii można odnaleźć w dokumentach *applying-patches.txt* oraz *SubmittingPatches* z [41].

Narzędzie *menuconfig* służy do tworzenia pożądanej konfiguracji jądra, w tym do wybrania modułów do skompilowania. Aby mechanizm ochrony wywołań systemowych był dostępny, należy oznaczyć pozycję *Syscalls controller for cgroups* dostępną w menu *Control Group support*.

A.2. Przykładowe użycie podsystemu

```
1 # mkdir /cgroup
2 # mount -t tmpfs cgroup_root /cgroup
3 # cd /cgroup
4 # mkdir syscalls
5 # mount -t cgroup -o syscalls syscalls_root syscalls
6 # ls
```

```
7 cgroup.clone_children cgroup.procs      release_agent  syscalls.deny
8 cgroup.event_control  notify_on_release syscalls.allow  tasks
9 # cat syscalls.allow
10 0 1 ... 311
11 # cat syscalls.deny
12
13 # cat tasks
14 1
15 2
16 ...
17 # mkdir test1
18 # cd test1
19 # ls
20 cgroup.clone_children cgroup.procs      syscalls.allow  tasks
21 cgroup.event_control  notify_on_release syscalls.deny
22 # echo 0 > tasks
23 # cat tasks
24 2357
25 2374
26 # echo 83 > syscalls.deny
27 # cat syscalls.deny
28 83
29 # mkdir test2
30 mkdir: cannot create directory 'test2': Function not implemented
31 # echo 83 > syscalls.allow
32 # mkdir test2
33 # cd ..
34 # echo 83 > syscalls.deny
35 # cd test1
36 # mkdir test3
37 mkdir: cannot create directory 'test3': Function not implemented
38 # echo 83 > syscalls.allow
39 -bash: echo: write error: Operation not permitted
```

Wydruk A.2: Przykład poleceń montujących i wykorzystujących podsystem.

Komentarz

Zakłada się, że wszystkie powyższe polecenia są wykonywane przez super-użytkownika. Na początku w katalogu */cgroup* tworzony jest tymczasowy system plików (1, 2), aby utworzona hierarchia nie zachowała się po ponownym uruchomieniu komputera (nie jest to potrzebne). Następny krok powoduje utworzenie katalogu *syscalls* (4) oraz zamontowanie w nim systemu plików *cgroup* dla podsystemu o nazwie *syscalls* (5). Po pomyślnym zamontowaniu hierarchii dostępne są pliki konfiguracyjne podsystemu (6). Domyślnie dla grupy kontrolnej najwyższego poziomu wszystkie wywołania są dozwolone (10, 11). Do grupy tej należą też wszystkie zadania systemu (13). W kolejnym etapie tworzona jest podgrupa o nazwie *test1* (17) oraz dodane zostaje do niej zadanie aktualnie działającej powłoki systemu (22). Co ciekawe, lista zadań rozważanej grupy zawiera dwa zadania (23) — dzieje się tak ponieważ wywołanie programu *cat* powoduje utworzenie zadania potomnego, które trafia do tej samej grupy kontrolnej co zadanie nadrzędne. Wpisanie wartości 83 do pliku *syscalls.deny* (26) powoduje dodanie wywołania o numerze 83 do listy wywołań zabronionych (27). Numerowi 83 w systemie demonstracyjnym odpowiada wywołanie *mkdir* — z tego powodu wydanie polecenia *mkdir test2* (29) nie udaje się — zostaje zwrócony błąd (30). Po dodaniu wywołania 83 do listy wywołań dozwolonych (31), próba stworzenia katalogu *test2* kończy się

sukcesem (32). Na zakończenie demonstracji wywołanie *mkdir* zostaje zabronione w grupie najwyższego poziomu (33, 34), powodując tym samym wzbronienie we wszystkich grupach podrzędnych (37). Próba zezwolenia na wywołanie 83 w grupie *test1* kończy się zatem niepowodzeniem (39), gdyż jest ono zabronione w grupie wyższego poziomu.

Bibliografia

- [1] Nicolas Bareil. *seccomp nurse homepage*. Dostęp internetowy: <http://www.chdir.org/~nico/seccomp-nurse/> [10.01.2012].
- [2] Adam Barth, Collin Jackson, Charles Reis, The Google Chrome Team. The Security Architecture of the Chromium Browser. 2007. Dostęp internetowy: <http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf> [10.01.2012].
- [3] BitMover. *BitKeeper homepage*. Dostęp internetowy: <http://www.bitkeeper.com/> [10.01.2012].
- [4] Daniel Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2000.
- [5] Caca Labs. *CPUShare homepage*. Dostęp internetowy: <http://caca.zoy.org/wiki/CPUShare> [10.01.2012].
- [6] Scott Chacon. *Git project homepage*. Dostęp internetowy: <http://git-scm.com/> [10.01.2012].
- [7] Mike Coleman. *SUBTERFUGUE homepage*. Dostęp internetowy: <http://subterfuge.org/> [10.01.2012].
- [8] Johnatan Corbet. How to Participate in the Linux Community: A Guide to The Kernel Development Process, August 2008. Dostęp internetowy: http://www.linuxfoundation.org/sites/main/files/How-Participate-Linux-Community_0.pdf [10.01.2012].
- [9] Jonathan Corbet. Driver porting: The seq_file interface. *LWN.net*, February 2003. Dostęp internetowy: <http://lwn.net/Articles/22355/> [10.01.2012].
- [10] Jonathan Corbet. Introducing utrace. *LWN.net*, March 2007. Dostęp internetowy: <http://lwn.net/Articles/224772/> [10.01.2012].
- [11] Jonathan Corbet. Notes from a container. *LWN.net*, October 2007. Dostęp internetowy: <http://lwn.net/Articles/256389/> [10.01.2012].
- [12] Jonathan Corbet. vmsplice(): the making of a local root exploit. *LWN.net*, February 2008. Dostęp internetowy: <http://lwn.net/Articles/268783/> [10.01.2012].
- [13] Jonathan Corbet. Seccomp and sandboxing. *LWN.net*, May 2009. Dostęp internetowy: <http://lwn.net/Articles/332974/> [10.01.2012].
- [14] Jonathan Corbet. KS2011: Coming to love control groups. *LWN.net*, October 2011. Dostęp internetowy: <http://lwn.net/Articles/464288/> [10.01.2012].
- [15] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, 2005. Dostęp internetowy: <http://lwn.net/Kernel/LDD3/> [10.01.2012].
- [16] James Cox, Berny Goodheart. *Sekrety magicznego ogrodu. UNIX System V Wersja 4 od środka*. Wydawnictwa Naukowo-Techniczne, Warszawa, 2001.
- [17] Jake Edge. Restricting root with per-process securebits. *LWN.net*, April 2008. Dostęp internetowy: <http://lwn.net/Articles/280279/> [10.01.2012].
- [18] Jake Edge. Google's Chromium sandbox. *LWN.net*, August 2009. Dostęp internetowy: <http://lwn.net/Articles/347547/> [10.01.2012].
- [19] Jake Edge. Limiting system calls via control groups? *LWN.net*, October 2011. Dostęp internetowy: <http://lwn.net/Articles/463683/> [10.01.2012].
- [20] Jake Edge. LSS: The kernel hardening roundtable. *LWN.net*, September 2011. Dostęp internetowy: <http://lwn.net/Articles/458805/> [10.01.2012].

- [21] Pavel Emelyanov, Kir Kolyshkin. PID namespaces in the 2.6.24 kernel. *LWN.net*, November 2007. Dostęp internetowy: <http://lwn.net/Articles/259217/> [10.01.2012].
- [22] European Organization for Nuclear Research. *LHC@home homepage*. Dostęp internetowy: <http://lhcathe.web.cern.ch/LHCathome/> [10.01.2012].
- [23] Tal Garfinkel. Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools. *Proc. Network and Distributed Systems Security Symposium*, February 2003. Dostęp internetowy: <http://www.stanford.edu/~talg/papers/traps/traps-ndss03.pdf> [10.01.2012].
- [24] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum. Ostia: A Delegating architecture for Secure System Call Interposition. *Proc. Network and Distributed Systems Security Symposium*, February 2004. Dostęp internetowy: <http://benpfaff.org/papers/ostia.pdf> [10.01.2012].
- [25] Google Inc. *Chromium Browser Design Docs*, 2011. Dostęp internetowy: <http://dev.chromium.org/developers/design-documents> [10.01.2012].
- [26] Google Inc. *Chromium OS Design Docs*, 2011. Dostęp internetowy: <http://dev.chromium.org/chromium-os/chromiumos-design-docs> [10.01.2012].
- [27] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.
- [28] Vivek Goyal, Haren Myneni, Dave Anderson. Debugging Linux Kernel Using Kdump. Dostęp internetowy: http://www.linuxsymposium.org/2006/kdump_slides.pdf [10.01.2012].
- [29] Mike Halcrow. eCryptfs: a Stacked Cryptographic Filesystem. *Linux Journal*, April 2007. Dostęp internetowy: <http://www.linuxjournal.com/article/9400> [10.01.2012].
- [30] IBM, SGI. Linux Test Project homepage. Dostęp internetowy: <http://ltp.sourceforge.net/> [10.01.2012].
- [31] Stack Exchange Inc. *UNIX & Linux Stack Exchange*. Dostęp internetowy: <http://unix.stackexchange.com/> [10.01.2012].
- [32] TopCoder Inc. *TopCoder homepage*. Dostęp internetowy: <http://www.topcoder.com/> [10.01.2012].
- [33] K. Jain, R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proc. Network and Distributed Systems Security Symposium*, 1999. Dostęp internetowy: <http://seclab.cs.sunysb.edu/seclab/pubs/ndss00.pdf> [10.01.2012].
- [34] Sebastien Jan, Michael Opdenacker, Thomas Petazzoni. Embedded Linux kernel and driver development. March 2011. Dostęp internetowy: <http://free-electrons.com/docs/kernel/> [10.01.2012].
- [35] Andrzej Jaskiewicz. *Inżynieria oprogramowania*. Helion, Gliwice, 1997.
- [36] NLnet Labs. *nlnetlabs.nl: Name Server Daemon homepage*. Dostęp internetowy: <http://www.nlnetlabs.nl/projects/nsd/> [10.01.2012].
- [37] Sphere Research Labs. *Sphere Online Judge*. Dostęp internetowy: <http://www.spoj.pl/> [10.01.2012].
- [38] Christoph Lameter. Effective Synchronization on Linux/NUMA Systems. May 2005. Dostęp internetowy: <http://www.lameter.com/gelato2005.pdf> [10.01.2012].
- [39] The Linux Foundation. *Linux Kernel Archives*. Dostęp internetowy: <http://kernel.org/> [10.01.2012].
- [40] The Linux Foundation. *Linux Kernel Bug Tracker*. Dostęp internetowy: <http://bugzilla.kernel.org/> [10.01.2012].
- [41] The Linux Foundation. *Linux Kernel Documentation*. Dostęp internetowy: <http://lxr.linux.no/linux/Documentation/> [10.01.2012].
- [42] The Linux Foundation. *The Linux man-pages*. Dostęp internetowy: <http://www.kernel.org/doc/man-pages/> [10.01.2012].
- [43] Robert Love. *Linux Kernel Development, Third Edition*. Pearson Education Inc., 2010.
- [44] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox, 2008.

- [45] Paul B. Menage. Adding Generic Process Containers to the Linux Kernel. Dostęp internetowy: <http://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf> [10.01.2012].
- [46] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th Usenix Security Symposium*, strony 257–272, 2002. Dostęp internetowy: <http://research.google.com/pubs/archive/33459.pdf> [10.01.2012].
- [47] Joanna Rutkowska. The Linux Security Circus: On GUI isolation. *The Invisible Things Lab's blog*, April 2011. Dostęp internetowy: <http://theinvisiblethings.blogspot.com/2011/04/linux-security-circus-on-gui-isolation.html> [10.01.2012].
- [48] Mark Seaborn. *Plash Wiki*. Dostęp internetowy: <http://plash.beasts.org/wiki/> [10.01.2012].
- [49] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, Larry Peterson. Container-based Operating System Virtualization: a Scalable, High-performance Alternative to Hypervisors. In: *Proceedings of EuroSys 2007*, strony 275–288, 2007. Dostęp internetowy: <http://www.cs.princeton.edu/~mef/research/vserver/paper.pdf> [10.01.2012].
- [50] Brad Spengler. *Grsecurity Documentation*. Dostęp internetowy: <http://en.wikibooks.org/wiki/Grsecurity> [10.01.2012].
- [51] Andrew S. Tanenbaum. *Systemy operacyjne. Wydanie III*. Helion, Gliwice, 2010.
- [52] University of California, Berkeley. *SETI@home homepage*. Dostęp internetowy: <http://setiathome.berkeley.edu/> [10.01.2012].
- [53] Uresh Vahalia. *Jądro systemu UNIX. Nowe horyzonty*. Wydawnictwa Naukowo-Techniczne, Warszawa, 2001.
- [54] Jeffrey A. Vaughan, Andrew D. Hilton. *Paladin: Helping Programs Help Themselves with Internal System Call Interposition*, 2010. Dostęp internetowy: <http://www.cs.ucla.edu/~jeff/docs/paladin.pdf> [10.01.2012].
- [55] Steven J. Vaughan-Nichols, Paula Rooney. The London Stock Exchange moves to Novell Linux. *ZDNet: Linux and Open Source*, February 2011. Dostęp internetowy: <http://www.zdnet.com/blog/open-source/the-london-stock-exchange-moves-to-novell-linux/8285> [10.01.2012].
- [56] Sreekrishnan Venkateswaran. *Essential Linux Device Drivers*. Prentice Hall, 2008.
- [57] Judd Vinet, Aaron Griffin. *Arch Linux Distribution homepage*. Dostęp internetowy: <http://www.archlinux.org/> [10.01.2012].
- [58] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. Dostęp internetowy: <http://www.watson.org/~robert/2007woot/2007usenixwoot-exploitingconcurrency.pdf> [10.01.2012].
- [59] Robert N. M. Watson, Jonathan Anderson, Ben Laurie. Capsicum: practical capabilities for UNIX. August 2010. Dostęp internetowy: <http://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf> [10.01.2012].
- [60] Jörg Zinke. System call tracing overhead, October 2009. Dostęp internetowy: http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf [10.01.2012].