Bruce Dang, Alexandre Gazet, Elias Bachaalany
Sébastien Josse

INŻYNIERIA ODWROTNA W PRAKTYCE

NARZĘDZIA I TECHNIKI

Helion

Tytuł oryginału: Practical Reverse Engineering x86, x64, ARM, Windows® Kernel, Reversing Tools, and Obfuscation

Tłumaczenie: Konrad Matuk

ISBN: 978-83-283-0678-3

Copyright © 2014 by Bruce Dang.
Published by John Wiley & Sons, Inc., Indianapolis, Indiana.

All rights reserved. This translation published under license with the original publisher John Wiley & Sons, Inc.

Translation copyright © 2015 by Helion S.A.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without either the prior written permission
of the Publisher

Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/inodpr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorach 9
O korektorze merytorycznym 11
Podziękowania 13
Wstęp 17

Rozdział 1. Architektura x86 i x64 21
Rejestry i typy danych 22
Zbiór instrukcji 23
Składnia 24
Przenoszenie danych 25
Ćwiczenie 30
Operacje arytmetyczne 31
Operacje stosu i wywoływanie funkcji 32
Ćwiczenia 36
Sterowanie wykonywanym programem 37
Mechanizm systemowy 44
Translacja adresów 45
Przerwania i wyjątki 47
Analiza krok po kroku 47
Ćwiczenia 54
x64 55
Rejestry i typy danych 55
Przenoszenie danych 56
Adresowanie kanoniczne 56
Wywołanie funkcji 57
Ćwiczenia 57

Kup ksi k Pole ksi k

Rozdział 2. Architektura ARM 59
Podstawowe funkcje 60
Typy danych i rejestry 63
Opcje systemu i ustawienia 65
Instrukcje — wprowadzenie 66
Ładowanie i zapisywanie danych 67
Instrukcje LDR i STR 67
Inne zastosowania instrukcji LDR 71
Instrukcje LDM i STM 72
Instrukcje PUSH i POP 76
Funkcje i wywoływanie funkcji 77
Operacje arytmetyczne 80
Rozgałęzianie i wykonywanie warunkowe 81
Tryb Thumb 85
Polecenia switch-case 85
Rozmaitości 87
Kompilacja just-in-time i samomodyfikujący się kod 87
Podstawy synchronizacji 88
Mechanizmy i usługi systemowe 89
Instrukcje 91
Analiza krok po kroku 91
Co dalej? 98
Ćwiczenia 98

Rozdział 3. Jądro systemu Windows 107
Podstawy systemu Windows 108
Rozkład pamięci 108
Inicjalizacja procesora 109
Wywołania systemowe 111
Poziom żądań przerwania urządzenia 123
Pule pamięci 125
Listy deskryptorów pamięci 126
Procesy i wątki 126
Kontekst wykonywania 128
Podstawy synchronizacji jądra 129
Listy 130
Szczegóły implementacji 131
Analiza krok po kroku 138
Ćwiczenia 142

Kup ksi k Pole ksi k

Wykonywanie asynchroniczne i ad hoc 146
Wątki systemowe 147
Elementy robocze 148
Asynchroniczne wywoływanie procedur 150
Opóźnione wywoływanie procedur 154
Timery 158
Wywołania zwrotne procesów i wątków 159
Procedury zakończenia 161
Pakiety żądań wejścia-wyjścia 162
Struktura sterownika 164
Punkty rozpoczęcia 165
Obiekty sterownika i urządzenia 166
Obsługa pakietów IRP 167
Popularne mechanizmy zapewniające komunikację pomiędzy kodem użytkownika a kodem jądra 168
Inne mechanizmy systemowe 170
Analiza krok po kroku 173
Rootkit w architekturze x86 173
Rootkit w architekturze x64 188
Dalszy rozwój 195
Ćwiczenia 196
Rozwijanie pewności siebie i utrwalanie wiadomości 197
Poszerzanie horyzontów 198
Analiza prawdziwych sterowników 201

Rozdział 4. Debugowanie i automatyzacja 203
Narzędzia i podstawowe polecenia służące do debugowania 204
Określanie ścieżki plików symboli 205
Okna debugera 205
Obliczanie wartości wyrażenia 206
Zarządzanie procesami i debugowanie zdarzeń 210
Rejestry, pamięć i symbole 214
Punkty wstrzymania 223
Kontrolowanie procesów i modułów 226
Inne polecenia 229
Skrypty i debugowanie 230
Pseudorejestry 231
Aliasy 233
Język 240
Pliki skryptów 251
Skrypty jako funkcje 255
Przykładowe skrypty przydatne podczas debugowania 260

Korzystanie z narzędzi SDK 267
Pojęcia 268
Tworzenie rozszerzeń narzędzi debugujących 272
Praktyczne rozszerzenia, narzędzia i źródła wiedzy 274

Rozdział 5. Zaciemnianie kodu 277
Techniki zaciemniania kodu 279
Po co zaciemniać kod? 279
Zaciemnianie oparte na danych 282
Zaciemnianie oparte na sterowaniu 287
Jednoczesne zaciemnianie przepływu sterowania i przepływu danych 293
Zabezpieczanie przez zaciemnianie 297
Techniki rozjaśniania kodu 297
Natura rozjaśniania kodu: odwracanie przekształceń 298
Narzędzia przeznaczone do rozjaśniania kodu 303
Rozjaśnianie kodu w praktyce 319
Studium przypadku 335
Pierwsze wrażenie 335
Analiza semantyki procedury 337
Obliczanie symboliczne 339
Rozwiązanie zadania 340
Podsumowanie 342
Ćwiczenia 343
Bibliografia 343

Dodatek A Sumy kontrolne SHA1 347

Skorowidz 349

Kup ksi k Pole ksi k

Rozdział

2

Architektura ARM

Firma Acron Computers pod koniec lat 80. stworzyła 32-bitową architekturę RISC, która początkowo nosiła nazwę Acron RISC Machine (później przemianowano ją na Advanced RISC Machine). Architektura ta okazała się na tyle dobra, że zaczęto ją stosować również w produktach innych firm. Z tego powodu powstała spółka o nazwie ARM Holdings. Sprzedawała ona licencje pozwalające na zastosowanie architektury ARM w wielu różnych urządzeniach. Architektury tej używa się w licznych systemach wbudowanych, takich jak telefony komórkowe, elektronika samochodowa, odtwarzacze MP3, telewizory. Pierwsza wersja tej architektury została wprowadzona na rynek w roku 1985. Obecnie powstaje jej siódma generacja (ARMv7). Istnieje wiele serii procesorów ARM (np. ARM7, ARM7TDMI, ARM926EJ-S, Cortex). Nie należy ich mylić z różnymi wersjami specyfikacji architektury, oznaczanymi jako ARMv1 – ARMv7. Jest dostępnych kilka wersji wspomnianej architektury, przy czym większość urządzeń korzysta z wersji ARMv4, 5, 6 lub 7. Wersje oznaczone numerami 4 i 5 są już relatywnie dość „stare", niemniej jednak procesory oparte na tych wersjach architektury są najczęściej spotykane (według danych przedstawianych przez firmę ARM „powstało ponad 10 miliardów" tego typu procesorów). W popularnych urządzeniach elektronicznych spotyka się nowsze wersje tej architektury. Na przykład produkowany przez firmę Apple odtwarzacz IPod Touch trzeciej generacji wyposażono w procesor o architekturze ARMv6, a wszystkie późniejsze urządzenia, takie jak iPhone, iPad i Windows Phone 7, produkowano na bazie architektury ARMv7.

Firmy Intel i AMD same projektują i produkują swoje procesory. Natomiast ARM podchodzi do tego nieco inaczej. Firma ta projektuje architekturę, a następnie sprzedaje innym firmom licencje na produkcję układów. Firmy te wytwarzają procesory i montują je w swoich urządzeniach. Na przykład firmy Apple, NVIDIA, Qualcomm i Texas Instruments produkują własne procesory, takie jak A, Tegra, Snapdragon i OMAP, lecz ich architektura jest licencjonowana przez ARM. We wszystkich

Kup ksi k

Pole ksi k

tych układach zastosowano ten sam model zarządzania pamięcią, który zdefiniowano w dokumentacji architektury ARM. Do procesorów mogą być dodane oczywiście pewne funkcje zwiększające ich możliwości; przykładowo rozszerzenie instrukcji sprzętowych Jazelle pozwala procesorowi na bezpośrednie przetwarzanie kodu Java, a rozszerzenie Thumb zapewnia obsługę 16- i 32-bitowych instrukcji, co umożliwia tworzenie gęstszego kodu (standardowe instrukcje ARM są 32-bitowe). Rozszerzenie Debug pozwala inżynierom analizować pracę procesora za pomocą specjalnego urządzenia. Rozszerzenia są oznaczane za pomocą liter (np. J, T, D). Producenci, w zależności od swoich wymagań, mogą kupić odpowiednią licencję. To właśnie dlatego w oznaczeniach procesorów zgodnych z architekturą ARMv6 (a także ze starszymi jej wersjami) zastosowano litery — np. ARM1156T2 oznacza procesor zgodny z architekturą ARMv6 z rozszerzeniem Thumb-2. Ta konwencja nazewnictwa procesorów nie jest stosowana w przypadku architektury ARMv7, gdzie podaje się nazwę modelu oraz jeden z trzech sufiksów: A (aplikacje), R (czas rzeczywisty) i M (mikrokontroler). Na przykład seria procesorów ARMv7 Cortex-A jest zoptymalizowana do wykonywania aplikacji, a układy z rodziny Cortex-M są przeznaczone do pracy w mikrokontrolerach i obsługują tylko wykonywanie instrukcji w trybie Thumb.

W niniejszym rozdziale opisano architekturę ARMv7 zdefiniowaną w *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition* (ARM DDI 0406B).

Podstawowe funkcje

Architektura ARM jest oparta na architekturze RISC, a więc istnieją pewne różnice pomiędzy architekturą ARM a architekturą CISC (x86, x64). W nowych procesorach firmy Intel dodano pewne funkcje znane wcześniej z architektury RISC (procesory te nie charakteryzują się „czystą" architekturą CISC). Po pierwsze, zestaw instrukcji ARM jest o wiele mniejszy od zestawu instrukcji obsługiwanych przez procesory x86, z tym że w tej architekturze znalazło się więcej rejestrów ogólnego przeznaczenia. Po drugie, w przypadku procesorów ARM instrukcje charakteryzują się stałą długością (w zależności od trybu pracy są one 16- lub 32-bitowe). Po trzecie, w architekturze ARM dostęp do pamięci uzyskuje się za pomocą modelu *load-store* (pol. „załaduj z pamięci — zapisz w pamięci"). Dane przed przetworzeniem muszą być przeniesione z pamięci do rejestrów. Dostęp do pamięci odbywa się tylko za pomocą instrukcji załaduj (`LDR`) i zapisz (`STR`). Jeżeli chcesz dokonać inkrementacji 32-bitowej wartości zapisanej pod danym adresem pamięci, to musisz najpierw załadować wartość spod tego adresu do rejestru, wykonać operację inkrementowania, a następnie zapisać wartość z powrotem w pamięci. W architekturze x86 większość instrukcji mogła dokonywać bezpośrednich operacji na danych zapisanych w pamięci. To, co w przypadku architektury x86 wymaga wykonania jednej instrukcji, w przypadku architektury ARM wymaga wykonania trzech instrukcji (jednej instrukcji ładowania, jednej inkrementowania i jednej zapisu). Osobie zajmującej się inżynierią odwrotną może się wydawać, że odczytanie takiego kodu będzie trwało dłużej, ale w praktyce nie ma to znaczenia, gdy przyzwyczaisz się do tej architektury.

W architekturze ARM występuje także kilka różnych poziomów uprawnień przypisywanych procesom. W przypadku architektury x86 przywileje były określane za pomocą czterech ringów.

Kup ksi k Pole ksi k

Ring 0 charakteryzował się najwyższymi przywilejami, a ring 3 — najniższymi. W architekturze ARM przywileje są określone przez osiem trybów pracy:

- USR — ang. *user* — tryb użytkownika,
- FIQ — ang. *fast interrupt request* — tryb obsługujący przerwania o wysokich priorytetach,
- IRQ — ang. *interrupt request* — tryb obsługi przerwań o niskim priorytecie,
- SVC — ang. *supervisor* — tryb superużytkownika,
- MON — ang. *monitor* — tryb monitorowania,
- ABT — ang. *abort* — tryb obsługi wyjątków związanych z pamięcią,
- UND — ang. *undefined* — tryb obsługi nieznanych wyjątków,
- SYS — ang. *system* — tryb wykorzystywany przez system operacyjny.

Kod uruchomiony w danym trybie charakteryzuje się pewnymi uprawnieniami i dostępem do rejestrów, do których aplikacje działające w innych trybach nie mają dostępu. Na przykład kod uruchomiony w trybie USR nie może modyfikować rejestrów systemowych, które zwykle mogą być modyfikowane tylko w trybie SVC. Tryb USR charakteryzuje się najmniejszą liczbą uprawnień. Istnieje wiele różnic natury technicznej, jednak w dużym uproszczeniu można powiedzieć, że tryb USR przypomina ring 3 znany z architektury x86, a tryb SVC jest odpowiednikiem ringu 0. Większość systemów operacyjnych implementuje instrukcje jądra w trybie SVC, a aplikacje użytkownika — w trybie USR. Dzieje się tak w przypadku systemów Windows i Linux.

W rozdziale 1. pisaliśmy, że procesory x64 mogą obsługiwać aplikacje 32-bitowe, 64-bitowe lub oba rodzaje aplikacji. Podobnie procesory ARM mogą pracować w dwóch stanach: ARM i Thumb. Te dwa stany obsługują różny zestaw instrukcji i nie mają nic wspólnego z poziomami uprawnień. Na przykład kod uruchomiony w trybie SVC może być wykonywany w trybie ARM lub Thumb. W trybie ARM instrukcje są zawsze 32-bitowe, a w trybie Thumb mogą być 16- lub 32-bitowe. Tryb pracy procesora zależy od dwóch rzeczy:

- Podczas rozgałęziania instrukcji `BX` i `BLX` najmniej znaczący bit docelowego rejestru może przybrać wartość 1 — wtedy procesor jest przełączany w tryb Thumb. (Instrukcje mogą być 2- lub 4-bajtowe. Procesor będzie ignorował najmniej znaczące bity, a więc nie zaistnieją problemy z wyrównywaniem).
- Jeżeli bit `T` znajdujący się w rejestrze stanu aktualnie wykonywanego programu (`CPSR`) przyjmuje wartość 1, oznacza to, że procesor działa w trybie Thumb. Składnia rejestru `CPSR` zostanie wyjaśniona w kolejnym podrozdziale, ale na razie możesz rozumieć ten rejestr jako nieco bardziej rozbudowaną wersję rejestru `EFLAGS`, znanego Ci z architektury x86.

Procesor ARM jest uruchamiany w trybie ARM i pozostaje w tym trybie, dopóki nie zostanie jawnie lub niejawnie przełączony w tryb Thumb. W praktyce wiele współczesnych systemów operacyjnych korzysta głównie z kodu wykonywanego w trybie Thumb, ponieważ zapewnia on większą gęstość kodu (mieszanka instrukcji 16- i 32-bitowych może zajmować mniej pamięci od ciągu instrukcji 32-bitowych). Aplikacje mogą działać w dowolnym trybie. Większość instrukcji Thumb i ARM charakteryzuje się identycznymi mnemonikami, 32-bitowe instrukcje Thumb oznaczono sufiksem `.w`.

Kup ksi k Pole ksi k

Uwaga **Dość często tryb Thumb mylnie porównuje się do rzeczywistego trybu pracy procesora o architekturze x68 lub x64, a tryb ARM — do trybu chronionego wymienionych architektur. Jest to błędne rozumowanie. Większość systemów operacyjnych uruchomionych na platformach x86 i x64 działa w trybie chronionym i bardzo rzadko, o ile w ogóle tak się dzieje, przełącza procesor w tryb rzeczywisty. W przypadku platformy ARM instrukcje mogą być wykonywane naprzemiennie w trybach ARM i Thumb. Zwróć uwagę również na to, że te tryby pracy nie mają nic wspólnego z trybami określającymi uprawnienia, które opisaliśmy we wcześniejszym akapicie (USR, SVC itd.).**

Istnieją dwie wersje trybu Thumb: Thumb-1 i Thumb-2. Thumb-1 występował w architekturze ARMv6 i jej starszych wersjach. Instrukcje w tym trybie były zawsze 1-bitowe. W trybie Thumb-2 dodano obsługę większej liczby instrukcji, które mogą być 16- lub 32-bitowe. Architektura ARMv7 wymaga Thumb-2, a więc ilekroć piszemy o Thumb, mamy tak naprawdę na myśli Thumb-2.

Istnieją również inne różnice między pracą w trybie ARM a pracą w trybie Thumb, lecz nie możemy tu omawiać ich wszystkich. Niektóre instrukcje mogą być na przykład dostępne w trybie ARM, a niedostępne w trybie Thumb (lub na odwrót). Takie szczegółowe informacje znajdziesz w oficjalnej dokumentacji architektury ARM.

Poza różnymi stanami pracy procesory ARM charakteryzują się również obsługą wykonywania warunkowego. Oznacza to, że instrukcje mogą zawierać pewne warunki arytmetyczne, które muszą zostać spełnione przed uruchomieniem instrukcji. Na przykład instrukcja może zawierać warunek określający to, że może zostać wykonana tylko w sytuacji, gdy w efekcie działania poprzedniej instrukcji uzyskano 0. W przypadku procesorów x86 niemalże żadne instrukcje nie były obwarowane warunkami. (Procesory firmy Intel posiadają kilka instrukcji, które bezpośrednio obsługują wykonywanie warunkowe: `CMOV` i `SETNE`). Wykonywanie warunkowe jest przydatną funkcją, ponieważ pozwala skrócić rozgałęzienia instrukcji (których wykonywanie zajmuje wiele cykli pracy procesora) i zmniejszyć liczbę wykonywanych instrukcji (co umożliwia zwiększenie gęstości kodu). Wszystkie instrukcje występujące w trybie ARM obsługują wykonywanie warunkowe, ale domyślnie są wykonywane bezwarunkowo. W trybie Thumb wykonywanie warunkowe należy umożliwić za pomocą specjalnej instrukcji `IT`.

Kolejną wyjątkową cechą architektury ARM jest obsługa przesunięcia bitowego. Niektóre instrukcje mogą „zawierać” inne instrukcje arytmetyczne, które przestawiają lub obracają zawartość rejestru. Jest to przydatne rozwiązanie, ponieważ umożliwia wykonanie niektórych operacji za pomocą jednej instrukcji (pozwala uniknąć stosowania całego szeregu instrukcji). Możesz chcieć pomnożyć przez 2 zawartość jakiegoś rejestru, a następnie zapisać wynik operacji w innym rejestrze. Normalnie wymagałoby to wykonania dwóch instrukcji (mnożenia, a następnie zapisania wartości w innym rejestrze), lecz dzięki przesunięciu bitowemu możesz zawrzeć działanie mnożenia (przesunięcie bitów o jedną pozycję w lewo) w instrukcji `MOV`. Taka instrukcja wyglądałaby następująco:

```
MOV R1, R0, LSL #1    ; R1 = R0·2
```

Typy danych i rejestry

Podobnie jak języki wysokiego poziomu architektura ARM obsługuje operacje na różnych typach danych. Obsługiwane są następujące typy danych: *byte* (8 bitów), *half-word* (16 bitów), *word* (32 bity) i *double word* (64 bity).

W architekturze ARM zdefiniowano 16 32-bitowych rejestrów ogólnego przeznaczenia, które oznaczono R0, R1, R2, …, R15. Programiści tworzący aplikacje mogą korzystać z tych wszystkich rejestrów, ale w praktyce tylko 12 pierwszych może być naprawdę wykorzystywanych w roli rejestrów ogólnego przeznaczenia (w przypadku architektury x86 wspomnianą rolę odgrywały rejestry takie jak EAX i EBX). Trzy ostatnie rejestry architektury ARM pełnią pewne określone funkcje:

- Rejestr R13 definiuje wskaźnik stosu (SP). W architekturach x86 i x64 taką samą rolę odgrywały rejestry ESP i RSP. Wskaźnik stosu wskazuje wierzchołek stosu programu.
- R14 można określić mianem rejestru łączącego (LR). Standardowo podczas uruchamiania funkcji zapisywany jest w nim adres zwrotny. Niektóre instrukcje bezwarunkowo korzystają z tego rejestru. Na przykład instrukcja BL zawsze zapisuje adres zwrotny w rejestrze LR przed wykonaniem kolejnego rozgałęzienia. Rejestru o identycznej funkcji nie przewidziano w architekturach x86 i x64, ponieważ w przypadku tych architektur adres zwrotny jest zawsze przechowywany na stosie. Jeżeli jakiś kod nie zapisuje adresu zwrotnego w rejestrze LR, to może korzystać z tego rejestru jak ze zwyczajnego rejestru ogólnego przeznaczenia.
- Rejestr R15 można określić mianem licznika rozkazów (PC). Podczas pracy w trybie ARM w rejestrze tym znajduje się adres obecnie przetwarzanej instrukcji plus 8 (2 instrukcje ARM do przodu). W czasie pracy w trybie Thumb w rejestrze tym znajduje się adres obecnie przetwarzanej instrukcji plus 4 (2 16-bitowe instrukcje Thumb do przodu). Rejestr ten działa podobnie jak rejestry EIP i RIP , występujące w architekturach x86 i x64, z tym wyjątkiem, że przytoczone rejestry zawsze zawierają adres kolejnej instrukcji, która ma zostać wykonana. Kolejna ważna różnica jest taka, że kod może bezpośrednio zapisywać dane w rejestrze PC i bezpośrednio je z niego odczytywać. Zapisanie adresu w rejestrze PC spowoduje natychmiastowe uruchomienie instrukcji znajdującej się pod tym adresem. Warto przeanalizować ten szczegół. Przyjrzyj się fragmentowi kodu, który ma zostać wykonany w trybie Thumb:

```
1: 0x00008344   push   {lr}
2: 0x00008346   mov    r0, pc
3: 0x00008348   mov.w  r2, r1, lsl #31
4: 0x0000834c   pop    {pc}
```

Po wykonaniu 2. linii kodu w rejestrze R0 będzie się znajdowała wartość 0x0000834a (=0x00008346+4):

```
(gdb) br main
rozkaz przerwania 1 pod adresem 0x8348
...
rozkaz przerwania 1, 0x00008348 w funkcji main ()
(gdb) disas main
zrzut kodu asemblera funkcji main:
   0x00008344 <+0>:   push   {lr}
   0x00008346 <+2>:   mov    r0, pc
```

Kup ksi k Pole ksi k

```
=> 0x00008348 <+4>:   mov.w  r2, r1, lsl #31
   0x0000834c <+8>:   pop    {pc}
   0x0000834e <+10>:  lsls   r0, r0, #0
```
koniec zrzutu kodu asemblera
```
(gdb) info register pc
pc   0x8348   0x8348 <main+4>
(gdb) info register r0
r0   0x834a   33610
```

Pod adresem `0x00008348` ustawiono rozkaz przerwania. Przyjrzyjmy się zawartości rejestrów `PC` i `R0`. Jak widać, zawartość rejestru `PC` wskazuje na 3. instrukcję (znajdującą się pod adresem `0x00008348`), która ma za chwilę zostać wykonana — w rejestrze `R0` znajduje się wartość wcześniej wczytana z rejestru `PC`. Przykład ten ilustruje, że podczas bezpośredniego odczytu rejestru `PC` zachowuje się on zgodnie z definicją, przy czym w czasie debugowania rejestr `PC` zawiera wskaźnik instrukcji, która ma zostać wykonana jako kolejna.

Dzieje się tak z powodu zachowania przez architekturę ARM wstecznej zgodności z wykonywaniem potokowym w starszych procesorach, które pobierały 2 instrukcje do przodu względem aktualnie przetwarzanej instrukcji. W obecnie produkowanych procesorach ARM przetwarzanie potokowe jest o wiele bardziej skomplikowanym procesem, niemniej jednak zachowano również wspomnianą definicję przetwarzania potokowego w celu zapewnienia zgodności z produkowanymi wcześniej procesorami.

Podobnie jak ma to miejsce w przypadku innych architektur, procesory ARM przechowują informacje o obecnie przetwarzanym procesie w rejestrze stanu aktualnie wykonywanego programu (`CPSR`). Z punktu widzenia programisty rejestr `CPSR` działa podobnie jak rejestry `EFLAGS` i `RFLAGS` w architekturach x86 i x64. W niektórych dokumentacjach znajdziesz również informacje na temat rejestru statusu aktualnie wykonywanej aplikacji (`APSR`), który jest nazwą umowną zbioru niektórych pól znajdujących się w rejestrze `CPSR`. Rejestr `CPSR` zawiera wiele flag. Niektóre z nich przedstawiono na rysunku 2.1 (pozostałe flagi zostaną omówione w dalszej części tego rozdziału).

- `E` (bit określający porządek bajtów) — Procesory ARM mogą obsługiwać dane zapisane w formacie *big-endian* lub *little-endian*. Jeżeli temu bitowi przypisze się wartość `0`, to procesor będzie obsługiwał dane w formacie *little-endian*, a jeżeli przypisze mu się wartość `1`, to procesor będzie obsługiwał dane w formacie *big-endian*. Przez większość czasu korzysta się z danych zapisanych w formacie *little-endian*, a więc bit ten będzie przyjmował zwykle wartość `0`.
- `T` (bit trybu Thumb) — Po przypisaniu temu bitowi wartości `1` procesor przełączany jest w tryb Thumb. W przeciwnym wypadku będzie on pracował w trybie ARM. Jednym ze sposobów na jawne przełączanie między trybami Thumb i ARM jest modyfikacja tego bitu.
- `M` (bity trybów) — Te bity określają aktualne uprawnienia (USR, SVC itd.).

Rysunek 2.1.

Opcje systemu i ustawienia

W architekturze ARM można spotkać się z koprocesorami obsługującymi dodatkowe instrukcje i funkcje na poziomie systemu. Na przykład w systemie obsługującym jednostkę zarządzania pamięcią (MMU) dostęp do jej ustawień musi uzyskać kod rozruchowy i kod jądra systemu operacyjnego. W przypadku architektur x86 i x64 ustawienia te są zapisywane w rejestrze CR0 lub CR4. W architekturze ARM ustawienia te są przechowywane przez koprocesor numer 15. W architekturze ARM przewidziano 16 koprocesorów. Są one identyfikowane za pomocą numerów: CP0, CP1, …, CP15. (W kodzie oznacza się je nazwami: P0, …, P15). Pierwszych 13 koprocesorów jest opcjonalnych albo zarezerwowanych przez architekturę ARM. Opcjonalne koprocesory mogą zostać użyte przez producentów w celu implementacji przewidzianych przez nich specjalnych instrukcji lub funkcji. Na przykład koprocesory CP10 i CP11 są zwykle używane podczas wykonywania operacji na liczbach zmiennoprzecinkowych. Obsługują one również technologię NEON. Każdy koprocesor ma swoje „kody operacyjne” i rejestry, które mogą być obsługiwane przez specjalne instrukcje ARM. Koprocesory CP14 i CP15 są używane podczas debugowania, a także przechowują ustawienia systemowe. Koprocesor CP15 można określić mianem **koprocesora sterującego pracą systemu** — przechowywane są w nim prawie wszystkie ustawienia systemu (ustawienia zapisywania w pamięci podręcznej, stronicowania, obsługi wyjątków itp.).

Uwaga **Technologia NEON zapewnia obsługę instrukcji SIMD (pojedynczych instrukcji przetwarzających wiele danych). Rozwiązanie takie przydaje się w aplikacjach multimedialnych. W architekturze x86 zestawy wspomnianych instrukcji są zapewniane przez technologie SSE i MMX.**

Każdy koprocesor posiada 16 rejestrów i 8 kodów operacyjnych. Składnia tych rejestrów i kodów operacyjnych jest charakterystyczna dla danego koprocesora. Dostęp do koprocesora może być uzyskany tylko za pomocą instrukcji MRC (odczyt) i MCR (zapis). Instrukcje te w roli argumentów przyjmują numer koprocesora, numer rejestru i kod operacji. Na przykład w celu odczytania bazy rejestru translacji (w architekturach x86 i x64 podobną funkcję pełnił rejestr CR3) i zapisania odczytanych danych w rejestrze R0 należy posłużyć się następującym kodem:

```
MRC p15, 0, r0, c2, c0, 0 ; zapisz TTBR w r0
```

Kod ten spowoduje „odczytanie rejestru C2/C0 koprocesora numer 15 za pomocą kodu operacji 0/0 i zapisanie odczytanej wartości w rejestrze ogólnego przeznaczenia R0”. W każdym koprocesorze znajduje się wiele rejestrów, a dodatkowo każdy koprocesor obsługuje wiele kodów operacji, tak więc musisz zapoznać się z dokumentacją. Tylko to pozwoli Ci dokładnie zrozumieć znaczenie każdej instrukcji. Niektóre rejestry (C13/C0) są zarezerwowane dla systemu operacyjnego — umieszczane są tam dane dotyczące danego procesu lub wątku.

Instrukcje MRC i MCR nie wymagają wysokiego poziomu uprawnień (można je uruchomić w trybie USR), przy czym niektóre rejestry koprocesorów i kody operacji mogą być obsługiwane tylko w trybie SVC. Próba odczytania pewnych rejestrów bez odpowiednich uprawnień zakończy się wyjątkiem.

Kup ksi k Pole ksi k

W praktyce bardzo rzadko spotyka się instrukcje tego typu zastosowane w kodzie uruchamianym przez użytkownika. Znacznie częściej można je znaleźć w programach rozruchowych, firmwarze, niskopoziomowym kodzie zapisanym w pamięciach ROM lub kodzie jądra systemu.

Instrukcje — wprowadzenie

Teraz możesz już przystąpić do analizy ważnych instrukcji występujących w architekturze ARM. Poza obsługą wykonywania warunkowego i przesuwania bitów architektura ARM obsługuje również pewne instrukcje, które nie mają swoich odpowiedników w architekturze x86. Po pierwsze, niektóre instrukcje mogą wykonać sekwencję operacji na podanym zakresie rejestrów. Przykładowo, aby zapisać zawartość 5 rejestrów (np. `R6 – R10`) pod określonym adresem, zapisanym w rejestrze `R1`, możliwe jest zastosowanie instrukcji `STM R1, {R6-R10}`. Zawartość rejestru `R6` zostanie zapisana pod adresem `R1`, zawartość rejestru `R7` — pod adresem `R1+4`, a zawartość adresu `R8` — pod adresem `R1+8`. Rejestry, które nie mają kolejnych numerów, należy oddzielać od siebie przecinkami (np. `{R1,R5,R8}`). W składni języka asemblera ARM zakresy rejestrów podaje się zwykle w nawiasach klamrowych. Po drugie, niektóre instrukcje mogą opcjonalnie uaktualniać rejestr bazowy po wykonaniu operacji zapisu lub odczytu. Aby skorzystać z tej funkcji, należy po nazwie rejestru dodać wykrzyknik (`!`). Na przykład gdybyś podaną wcześniej instrukcję zapisał w następujący sposób: `STM R1!,{R6-R10}` i uruchomił ją, to zawartość rejestru `R1` zostałaby uaktualniona — umieszczono by w nim kolejny adres po adresie, pod którym zapisano zawartość rejestru `R10`. Poniższy przykład wyjaśnia tę zasadę. Przeanalizuj go.

```
01: (gdb) disas main
02: zrzut kodu asemblera funkcji main:
03: => 0x00008344 <+0>:   mov   r6,  #10
04:    0x00008348 <+4>:   mov   r7,  #11
05:    0x0000834c <+8>:   mov   r8,  #12
06:    0x00008350 <+12>:  mov   r9,  #13
07:    0x00008354 <+16>:  mov   r10, #14
08:    0x00008358 <+20>:  stmia sp!, {r6, r7, r8, r9, r10}
09:    0x0000835c <+24>:  bx    lr
10: koniec zrzutu asemblera
11: (gdb) si
12: 0x00008348 w funkcji main ()
13: ...
14: 0x00008358 w funkcji main ()
15: (gdb) info reg sp
16: sp            0xbedf5848   0xbedf5848
17: (gdb) si
18: 0x0000835c w funkcji main ()
19: (gdb) info regsp
20: sp            0xbedf585c   0xbedf585c
21: (gdb) x/6x 0xbedf5848
22: 0xbedf5848:  0x0000000a   0x0000000b   0x0000000c
0x0000000d
23: 0xbedf5858:  0x0000000e   0x00000000
```

Kup ksi k Pole ksi k

W 15. linii wyświetlono zawartość rejestru SP (0xbedf5848) przed uruchomieniem instrukcji STM. Instrukcję tę uruchomiono w wierszach 17. i 19. — w wierszu 19. znajduje się uaktualniona zawartość rejestru SP. W 21. linii kodu wyświetlono sześć wartości typu *word*, zaczynając od starej zawartości rejestru SP. Zwróć uwagę na to, że zawartość R6 była zapisana pod starym adresem rejestru SP, zawartość R7 pod adresem SP+0x4, R8 pod adresem SP+0x8, R9 pod SP+0xc, a R10 pod SP+0x10. Nowy adres przypisany rejestrowi SP (0xbedf585c) jest kolejnym adresem po adresie, pod którym zapisano R10.

Uwaga **STMIA i STMEA są pseudoinstrukcjami instrukcji STM (dają one ten sam efekt). Dezasemblery wyświetlają jedną z nich. Niektóre będą wyświetlać STMEA, jeżeli bazowym rejestrem jest SP, a w kontekście innych rejestrów będzie wyświetlana pseudoinstrukcja STMIA. Inne dezasemblery będą posługiwać się instrukcją STM, a jeszcze inne będą zawsze wyświetlać pseudoinstrukcję STMIA. Nie ma jednej, ogólnie przyjętej konwencji. Jeżeli używasz wielu dezasemblerów, to musisz się do tego przyzwyczaić.**

Ładowanie i zapisywanie danych

W jednym z poprzednich podrozdziałów stwierdziliśmy, że architektura ARM posługuje się modelem *load-store* (pol. „załaduj z pamięci — zapisz w pamięci") — przed wykonaniem operacji na danych muszą one zostać umieszczone w rejestrze. Dostęp do pamięci mają tylko instrukcje odczytujące dane z pamięci i zapisujące w niej dane. Wszystkie pozostałe instrukcje mogą przetwarzać wyłącznie zawartość rejestrów. Ładowanie danych z pamięci polega na odczytaniu ich i zapisaniu w rejestrze. Natomiast zapisywanie danych w pamięci polega na odczytaniu ich z rejestru i umieszczeniu pod określonym adresem. Pary instrukcji LDR-STR, LDM-STM i PUSH-POP służą do odczytywania i zapisywania danych.

Instrukcje LDR i STR

Instrukcje te mogą odczytać z pamięci bądź zapisać w niej 1 bajt, 2 lub 4 bajty danych. Ich pełna składnia jest dość złożona — istnieje kilka różnych sposobów określania przesunięcia, a także uaktualniania rejestru bazowego. Przeanalizuj najprostszy przypadek:

```
01: 03 68   LDR   R3,[R0] ;R3 = *R0
02: 23 60   STR   R3,[R4] ;*R4 = R3;
```

W instrukcji widocznej w 1. linii R0 jest rejestrem bazowym, a R3 — docelowym. Instrukcja ta ładuje wartość typu *word* z adresu R0 do R3. W instrukcji widocznej w 2. linii R4 jest rejestrem bazowym, a R3 — docelowym. Instrukcja ta zapisuje wartość przypisaną rejestrowi R3 pod adresem wskazywanym przez rejestr R4. Jest to prosty przykład, ponieważ adres pamięci jest określany przez rejestr bazowy.

Podstawowymi argumentami przyjmowanymi przez instrukcje LDR i STR są rejestr bazowy i przesunięcie. Przesunięcie może być podane w trzech formach, a każda z tych form może być wyrażona

Kup ksi k Pole ksi k

w trzech trybach. Zaczniemy od omówienia trzech form przesunięcia, jakimi są: bezpośredni adres, rejestr i rejestr skalowany.

W pierwszej formie przesunięcia w roli bezpośredniego adresu podawana jest po prostu wartość typu *integer*. Jest to wartość uzyskana w wyniku operacji dodawania wartości przesunięć do rejestru bazowego lub odejmowania wartości przesunięć od tego rejestru — w ten sposób możliwe jest uzyskanie dostępu do danych, których przesunięcie jest znane w momencie kompilacji programu. Technika ta jest najczęściej stosowana w celu uzyskania dostępu do określonego pola struktury lub tablicy metod wirtualnych. Ogólnie przyjęto następujący sposób jej zapisu:

- `STR Ra, [Rb, imm]`
- `LDR Ra, [Rc, imm]`

`Rb` jest adresem rejestru bazowego, a `imm` jest przesunięciem dodawanym do `Rb`.

Załóżmy, że na przykład w rejestrze `R0` zapisano wskaźnik struktury `KDPC`. Przyjrzyj się przedstawionemu kodowi:

Definicja struktury

```
0:000> dt ntkrnlmp!_KDPC
   +0x000 Type            : UChar
   +0x001 Importance      : UChar
   +0x002 Number          : Uint2B
   +0x004 DpcListEntry    : _LIST_ENTRY
   +0x00c DeferredRoutine : Ptr32 void
   +0x010 DeferredContext : Ptr32 Void
   +0x014 SystemArgument1 : Ptr32 Void
   +0x018 SystemArgument2 : Ptr32 Void
   +0x01c DpcData         : Ptr32 Void
```

Kod

```
01: 13 23   MOVS   R3, #0x13
02: 03 70   STRB   R3, [R0]
03: 01 23   MOVS   R3, #1
04: 43 70   STRB   R3, [R0,#1]
05: 00 23   MOVS   R3, #0
06: 43 80   STRH   R3, [R0,#2]
07: C3 61   STR    R3, [R0,#0x1C]
08: C1 60   STR    R1, [R0,#0xC]
09: 02 61   STR    R2, [R0,#0x10]
```

Tym razem `R0` jest rejestrem bazowym, a wartościami informującymi o przesunięciu są: `0x1`, `0x2`, `0xC`, `0x10` i `0x1C`. Zaprezentowany fragment kodu można przedstawić za pomocą języka C w taki sposób:

```
KDPC *obj = ...;             /* R0 jest obiektem obj.*/
obj->Type = 0x13;
obj->Importance = 0x1;
obj->Number = 0x0;
obj->DpcData = NULL;
obj->DeferredRoutine = R1;   /* Nie znamy R1 .*/
obj->DeferredContext = R2;   /* Nie znamy R2. */
```

Taka forma zapisu przesunięcia przypomina instrukcję `MOV Reg, [Reg+Imm]`, znaną z architektur x86 i x64.

Przesunięcie można również wyrazić za pomocą rejestru. Technikę tę stosuje się często w przypadku kodu, który musi uzyskać dostęp do tablicy o indeksie określanym w trakcie działania programu. Zapis tego typu przesunięcia ma ogólny format:

- `STR Ra, [Rb, Rc]`
- `LDR Ra, [Rb, Rc]`

W zależności od kontekstu `Rb` i `Rc` mogą pełnić funkcję adresu bazowego lub przesunięcia. Przyjrzyj się tym dwóm przykładom:

Przykład 1.

```
01: 03 F0 F2 FA   BL   strlen
02: 06 46         MOV  R6,R0
; R0 jest wartością zwracaną przez funkcję strlen.
03: ...
04: BB 57 LDRSB R3, [R7,R6]
; W tym przypadku R6 definiuje przesunięcie.
```

Przykład 2.

```
01: B3 EB 05 08   SUBS.W   R8, R3, R5
02: 2F 78         LDRB     R7, [R5]
03: 18 F8 05 30   LDRB.W   R3, [R8,R5]
; W tym przykładzie R5 jest adresem bazowym, a R8 przesunięciem.
04: 9F 42         CMP      R7, R3
```

Taka forma zapisu przesunięcia przypomina instrukcję `MOV Reg, [Reg+Reg]`, znaną z architektur x86 i x64.

Trzecia forma określająca przesunięcie korzysta z rejestru skalowanego. Technika ta jest stosowana często w kontekście pętli iterującej tablicę. Przesunięcie jest skalowane za pomocą operacji przesuwania bitów. Ogólnie technikę tę można zapisać za pomocą następującego kodu:

- `LDR Ra, [Rb, Rc, <element określający przesunięcie>]`
- `STR Ra, [Rb, Rc, <element określający przesunięcie>]`

`Rb` jest rejestrem bazowym, `Rc` adresem bezpośrednim, a `<element określający przesunięcie>` definiuje operację wykonywaną na `Rc` — zwykle są to operacje przesunięcia w lewo lub w prawo, które mają na celu przeskalowanie przesunięcia. Na przykład:

```
01: 0E 4B         LDR      R3, =KeNumberNodes
02: ...
03: 00 24         MOVS     R4, #0
04: 19 88         LDRH     R1, [R3]
05: 09 48         LDR      R0, =KeNodeBlock
06: 00 23         MOVS     R3, #0
07:   loop_start
08: 50 F8 23 20   LDR.W    R2, [R0,R3,LSL#2]
09: 00 23         MOVS     R3, #0
10: A2 F8 90 30   STRH.W   R3, [R2,#0x90]
```

Kup książkę Poleć książkę

```
11: 92 F8 89 30   LDRB.W   R3, [R2,#0x89]
12: 53 F0 02 03   ORRS.W   R3, R3,#2
13: 82 F8 89 30   STRB.W   R3, [R2,#0x89]
14: 63 1C         ADDS     R3, R4, #1
15: 9C B2         UXTH     R4, R3
16: 23 46         MOV      R3, R4
17: 8C 42         CMP      R4, R1
18: EF DB         BLT      loop_start
```

KeNumberNodes jest globalną zmienną typu *integer*, a KeNodeBlock — globalną tablicą wskaźników KNODE.

Zmienne te są ładowane do rejestrów przez kod zapisany w liniach numer 1 i 5 (składnię tych linii wyjaśnimy później). Kod umieszczony w 8. wierszu iteruje tablicę KeNodeBlock (R0 jest bazą, R3 jest indeksem mnożonym przez 2, ponieważ mamy do czynienia z tablicą wskaźników, a na tej platformie wskaźniki są 4-bajtowe). W wierszach oznaczonych numerami 10 – 13 inicjowane są niektóre pola elementu KNODE. W 14. linii inkrementowany jest indeks. W 17. linii indeks jest porównywany z rozmiarem tablicy (R1 określa rozmiar tablicy — zobacz linię numer 4). Jeżeli indeks jest mniejszy od rozmiaru tablicy, to pętla jest nadal wykonywana.

Kod ten w języku C można ogólnie wyrazić w następujący sposób:

```
int KeNumberNodes = …;
KNODE *KeNodeBlock[KeNumberNodes] = …;
for (int i=0; i < KeNumberNodes; i++) {
   KeNodeBlock[i].x = …;
   KeNodeBlock[i].y= …;
   …
}
```

Taka forma zapisu przesunięcia przypomina instrukcję MOV Reg, [reg+idx*scale], znaną z architektur x86 i x64.

Omówiliśmy trzy formy przedstawiania przesunięcia. Teraz czas przyjrzeć się trybom adresowania: bezpośredniemu, przedindeksowemu i poindeksowemu. (W niektórych publikacjach wspomniane wcześniej tryby określane są mianem trybu przedindeksowego, przedindeksowego z buforowaniem i postindeksowego. Zastosowana przez nas terminologia jest zgodna z terminologią użytą w oficjalnej dokumentacji architektury ARM). Wymienione tryby adresowania różnią się jedynie modyfikacją rejestru bazowego. We wszystkich wcześniejszych przykładach modyfikowano rejestr bazowy, działając w trybie adresowania bezpośredniego. Tryb ten jest najczęściej stosowany. Łatwo rozpoznać go po tym, że w kodzie asemblera nie ma nigdzie wykrzyknika (!), a bezpośredni adres bazowy podany jest w nawiasach kwadratowych. Ogólna składnia tego trybu adresowania jest taka: LDR Rd, [Rn, offset].

W trybie adresowania przedindeksowego rejestr bazowy będzie uaktualniony przed operacją, w której zostanie użyty. Semantyka takiego wyrażenia przypomina stosowanie w jednoskładnikowych operacjach prefiksów ++ i --, znanych z języka C. Omawiany tryb adresowania można przedstawić za pomocą ogólnej składni: LDR Rd, [Rn, offset]!. Na przykład:

```
12 F9 01 3D   LDRSB.W R3,   [R2 ,#-1]!    ; R3 = *(R2-1)
                                          ; R2 = R2-1
```

W trybie adresowania poindeksowego rejestr bazowy jest używany w roli ostatecznego adresu, a następnie jest uaktualniany — dodaje się do niego wartość przesunięcia. Przypomina to notację przyrostkową języka C (++ i --), stosowaną w jednoskładnikowych operacjach. Omawiany tryb adresowania można przedstawić za pomocą ogólnej składni: `LDR Rd,[Rn], offset`. Na przykład:

```
10 F9 01 6B   LDRSB.W R6,[R0],#1    ; R6 = *R0
                                    ; R0 = R0+1
```

Formy adresowania przedindeksowego i poindeksowego są zwykle spotykane w kodzie, który uzyskuje wielokrotnie dostęp do danych znajdujących się w tym samym buforze. Na przykład taki kod może zawierać pętlę sprawdzającą, czy dany znak łańcucha jest jednym z pięciu poszukiwanych znaków. Kompilator może wtedy zastosować technikę adresowania rejestru bazowego odpowiednią dla instrukcji inkrementacji.

Uwaga Oto wskazówka, która ułatwi Ci rozpoznawanie różnych trybów adresowania stosowanych w instrukcjach `LDR` i `STR`. Jeżeli widzisz znak `!`, to znaczy, że jest to tryb przedindeksowy. Natomiast jeżeli w nawiasach kwadratowych ujęto wyłącznie rejestr bazowy, to znaczy, że jest to tryb poindeksowy. We wszystkich pozostałych przypadkach będziesz mieć do czynienia z trybem bezpośredniego określania przesunięcia.

Inne zastosowania instrukcji LDR

Wcześniej pisaliśmy, że instrukcja `LDR` służy do wczytywania danych z pamięci do rejestru — czasem jednak można ją spotkać w następujących formach:

```
01: DF F8 50 82   LDR.W   R8, =0x2932E00 ; LDR R8, [PC, x]
02: 80 4A         LDR     R2, =a04d ; "%04d" ; LDR R2, [PC, y]
03: 0E 4B         LDR     R3, =__imp_realloc ; LDR R3, [PC, z]
```

Zgodnie z uwagami zamieszczonymi wcześniej w tym podrozdziale taka składnia nie jest poprawna. Technicznie rzecz biorąc, są to **pseudoinstrukcje** — instrukcje tego typu są stosowane przez dezasemblery w celu ułatwienia użytkownikowi przeglądania kodu. Wewnętrznie korzystają one z formy bezpośredniej instrukcji LDR — w roli rejestru bazowego zastosowano rejestr PC. Rozwiązanie takie można określić mianem **adresowania *PC-relative*** (jest to odpowiednik **adresowania *RIP-relative*** w architekturze x64). W architekturze ARM zwykle spotyka się literał zlokalizowany w obszarze pamięci przeznaczonym do zapisu stałych, łańcuchów i informacji o przesunięciach. Dostęp do tego obszaru pamięci można uzyskać w sposób niezależny od jego pozycji. (Literał jest częścią kodu, a więc będzie znajdował się w tej samej sekcji). W podanym wcześniej fragmencie kod odwołuje się do 32-bitowej stałej, łańcucha i danych dotyczących przesunięcia importowanej funkcji zapisanej w literale. Zaprezentowana pseudoinstrukcja jest przydatna, ponieważ pozwala na przeniesienie 32-bitowej stałej do rejestru za pomocą pojedynczej instrukcji. W zrozumieniu tego może pomóc Ci kolejny fragment kodu:

Kup ksi k Pole ksi k

```
01: .text:0100B134 35 4B   LDR   R3, =0x68DB8BAD
     ; Jest to tak naprawdę instrukcja LDR R3, [PC, #0xD4],
     ; teraz PC = 0x0100B138.
02: ...
03: .text:0100B20C AD 8B DB 68 dword_100B20C DCD 0x68DB8BA
```

Jak dezasembler skrócił pierwszą instrukcję z `LDR R3, [PC, #0xD4]` i przedstawił ją w alternatywnej formie? Taka operacja mogła być wykonana, ponieważ kod działa w trybie Thumb, a w rejestrze `PC` zapisano adres obecnie wykonywanej instrukcji plus 4, czyli `0x0100B138`. Kod korzysta z bezpośredniej formy adresowania — odczytywane są dane typu *word* z adresu `0x0100B20C` (`=0x100B138+0xD4`), a akurat tam znajduje się stała, którą chcemy załadować.

Inną podobną instrukcją jest `ADR`, która uzyskuje adres etykiety lub funkcji i umieszcza go w rejestrze. Na przykład:

```
01: 00009390 65 A5         ADR      R5, dword_9528
02: 00009392 D5 E9 00 45   LDRD.W   R4, R5, [R5]
03: ...
04: 00009528 00 CE 22 A9+dword_9528 DCD 0xA922CE00 , 0xC0A4
```

Ta instrukcja jest zwykle stosowana w celu implementacji tablic skoków lub wywołań zwrotnych — tam, gdzie niezbędne jest przekazanie adresu funkcji do innej funkcji. Procesor, wykonując tę instrukcję, oblicza przesunięcie względem rejestru `PC` i zapisuje je w rejestrze docelowym.

Instrukcje LDM i STM

Instrukcje `LDM` i `STM` działają podobnie do instrukcji `LDR` i `STR`, ale mogą odczytywać lub zapisywać wiele danych typu *word* znajdujących się pod adresem bazowym. Przydają się podczas przenoszenia wielu bloków danych do i z pamięci. Mają one ogólną składnię:

- `LDM<mode> Rn[!], {Rm}`
- `STM<mode> Rn[!], {Rm}`

`Rn` jest rejestrem bazowym przechowującym adres, z którego dane będą odczytywane lub pod którym będą zapisywane. Dodatkowy wykrzyknik (`!`) informuje o tym, że rejestr bazowy powinien zostać uaktualniony przez nowy (zwrócony) adres. `Rm` jest zakresem rejestrów, do których dane zostaną wczytane lub z których zostaną zapisane. Opisywane instrukcje mogą działać w czterech trybach:

- `IA` (inkrementuj po) — Zapis danych rozpoczyna się od komórki pamięci wskazywanej przez adres bazowy. Zwracany jest adres umiejscowiony 4 bajty nad ostatnio zwróconym adresem. Jest to tryb domyślny, używany, gdy nie określono żadnego innego trybu.
- `IB` (inkrementuj przed) — Zapis danych rozpoczyna się od komórki pamięci umiejscowionej 4 bajty nad adresem bazowym. Zwracany jest adres komórki pamięci, w której zapisano dane.
- `DA` (dekrementuj po) — Zapis danych kończy się w komórce pamięci wskazywanej przez adres bazowy. Zwracany jest adres 4 bajty poniżej najniższego adresu, pod którym zapisano dane.
- `DB` (dekrementuj przed) — Zapis danych kończy się w komórce pamięci położonej 4 bajty poniżej adresu bazowego. Zwracany jest adres pierwszej komórki pamięci.

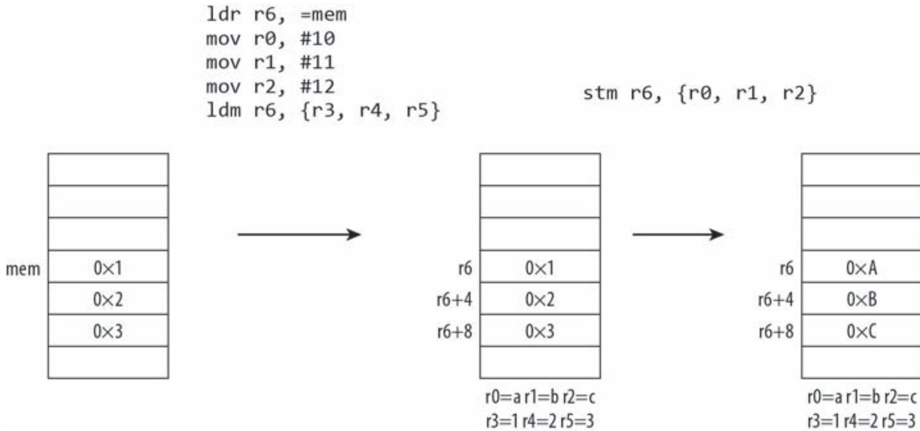
Na początku może się to wydawać dość skomplikowane, a więc przeanalizujmy następujący przykład na podstawie debugera:

```
01: (gdb) br main
02: punkt wstrzymania 1: 0x8344
03: (gdb) disas main
04: zrzut kodu asemblera funkcji main:
05:   0x00008344 <+0>:   ldr   r6, =mem ; Nieco zmodyfikowano.
06:   0x00008348 <+4>:   mov   r0, #10
07:   0x0000834c <+8>:   mov   r1, #11
08:   0x00008350 <+12>:  mov   r2, #12
09:   0x00008354 <+16>:  ldm   r6, {r3, r4, r5}; tryb IA
10:   0x00008358 <+20>:  stm   r6, {r0, r1, r2}; tryb IA
11: ...
12: (gdb) r
13: punkt wstrzymania 1, 0x00008344 w funkcji main ()
14: (gdb) si
15: 0x00008348 w funkcji main ()
16: (gdb) x/3x $r6
17: 0x1050c <mem>: 0x00000001  0x00000002  0x00000003
18: (gdb) si
19: 0x0000834c w funkcji main ()
20: ...
21: (gdb)
22: 0x00008358 w funkcji main ()
23: (gdb) info reg r3 r4 r5
24: r3   0x1   1
25: r4   0x2   2
26: r5   0x3   3
27: (gdb) si
28: 0x0000835c w funkcji main ()
29: (gdb) x/3x $r6
30: 0x1050c <mem>: 0x0000000a  0x0000000b  0x0000000
```

W 5. linii kodu adres pamięci zapisano w rejestrze R6. Pod tym adresem (0x1050c) znajdują się trzy elementy typu *word* (zobacz linia oznaczona numerem 17). W liniach o numerach 6 – 8 rejestrom R0 – R2 przypisano pewne stałe. W 9. wierszu kodu trzy elementy typu *word* są ładowane do rejestrów R3 – R5, zaczynając od komórki pamięci określonej przez zawartość rejestru R6. W 29. wierszu kodu widzimy, że właściwe wartości zostały zapisane. Operacje te zilustrowano na rysunku 2.2.

Oto ten sam eksperyment, ale tym razem skorzystano z możliwości zwracania adresu:

```
01: (gdb) br main
02: punkt wstrzymania 1: 0x8344
03: (gdb) disas main
04: zrzut kodu asemblera funkcji main:
05:   0x00008344 <+0>:   ldr   r6, =mem ; Nieco zmodyfikowano.
06:   0x00008348 <+4>:   mov   r0, #10
07:   0x0000834c <+8>:   mov   r1, #11
08:   0x00008350 <+12>:  mov   r2, #12
09:   0x00008354 <+16>:  ldm   r6!, {r3, r4, r5}; tryb IA ze zwracaniem adresu
10:   0x00008358 <+20>:  stmia r6!, {r0, r1, r2}; tryb IA ze zwracaniem adresu
11: ...
12: (gdb) r
```



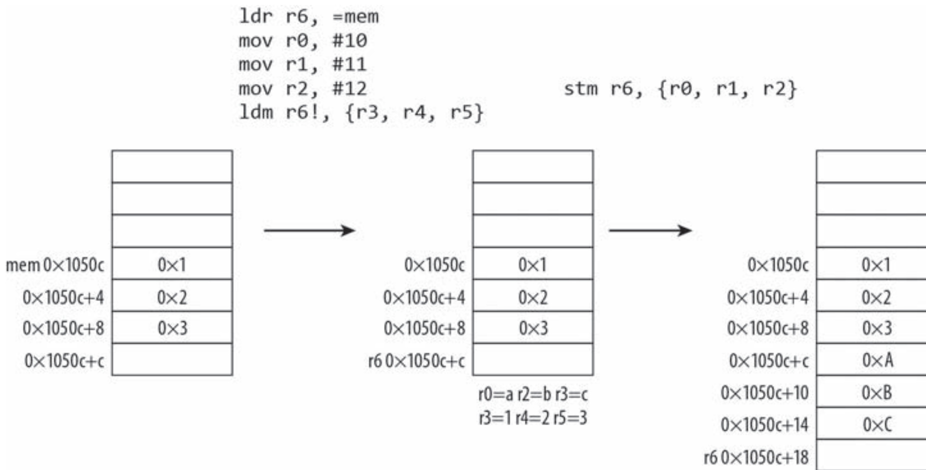
Rysunek 2.2.

```
13: punkt wstrzymania 1, 0x00008344 w funkcji main ()
14: (gdb) si
15: 0x00008348 w funkcji main ()
16: ...
17: (gdb)
18: 0x00008354 w funkcji main ()
19: (gdb) x/3x $r6
20: 0x1050c <mem>: 0x00000001 0x00000002 0x00000003
21: (gdb) si
22: 0x00008358 w funkcji main ()
23: (gdb) info reg r6
24: r6 0x10518 66840
25: (gdb) si
26: 0x0000835c w funkcji main ()
27: (gdb) info reg $r6
28: r6 0x10524 66852
29: (gdb) x/4x $r6-12
30: 0x10518 :      0x0000000a 0x0000000b 0x0000000c
0x00000000
```

W 9. linii zastosowano tryb IA ze zwracaniem adresu, a więc zawartość rejestru r6 jest nadpisywana przez adres komórki pamięci leżącej 4 bajty nad ostatnio użytą komórką (zobacz linia oznaczona numerem 23). Tę samą technikę zastosowano w wierszach kodu oznaczonych numerami 10, 27 i 30. Efekt działania zaprezentowanego fragmentu kodu pokazano na rysunku 2.3.

Instrukcje LDM i STM mogą podczas jednego wywołania przenosić wiele elementów, a więc są często używane w blokowych operacjach kopiowania lub przenoszenia. Mogą one być użyte na przykład w celu implementacji funkcji memcpy, gdy ilość danych przeznaczonych do skopiowania jest znana w momencie kompilacji programu. Wspomniane instrukcje działają podobnie do znanej z architektury x86 instrukcji MOVS, poprzedzonej prefiksem REP. Przyjrzyj się dwóm fragmentom kodu wygenerowanym przez dwa różne kompilatory na podstawie tego samego kodu źródłowego:

Kup ksi k Pole ksi k



Rysunek 2.3.

Kompilator A

```
01: A4 46        MOV       R12, R4
02: 35 46        MOV       R5, R6
03: BC E8 OF 00  LDMIA.W   R12!, {R0-R3}
04: OF C5        STMIA     R5!, {R0-R3}
05: BC E8 OF 00  LDMIA.W   R12!, {R0-R3}
06: OF C5        STMIA     R5!, {R0-R3}
07: 9C E8 OF 00  LDMIA.W   R12, {R0-R3}
08: 85 E8 OF 00  STMIA.W   R5, {R0-R3}
```

Kompilator B

```
01: 30 22        MOVS R2, #0x30
02: 21 46        MOV  R1, R4
03: 30 46        MOV  R0, R6
04: 23 F0 17 FA  BL   memcpy
```

Kod ten służy jedynie do skopiowania 48 bajtów z jednego bufora do drugiego. Pierwszy kompilator posługiwał się instrukcjami LDM i STM oraz zwracaniem adresu. Odczytywał i zapisywał dane w porcjach po 16 bajtów. Drugi kompilator po prostu wywoływał swoją implementację funkcji memcpy. Osoba zajmująca się inżynierią odwrotną takiego kodu może rozpoznać zastosowanie funkcji memcpy po tym, że niektóre wskaźniki źródeł i celów są używane przez instrukcje LDM i STM wraz z pewnym zestawem rejestrów. Warto o tym pamiętać, ponieważ często stosuje się tego typu rozwiązania.

Instrukcje LDM i STM spotyka się również często na początku i na końcu funkcji wykonywanych w trybie ARM. W tym kontekście pełnią one funkcję prologu i epilogu. Na przykład:

```
01: F0 4F 2D E9   STMFD   SP!, {R4-R11,LR}  ; Zapisuje rejestry i adres zwrotny.
02: ...
03: F0 8F BD E8   LDMFD   SP!, {R4-R11,PC}  ; Przywraca rejestry i zwraca dane.
```

STMFD jest pseudoinstrukcją STMDB, a LDMFD jest pseudoinstrukcją LDMIA i LDM.

Uwaga **Do instrukcji STM i LDM często dodaje się sufiksy FD, FA, ED lub EA. Tworzy się w ten sposób po prostu pseudoinstrukcje instrukcji LDM i STM, które działają w różnych trybach (IA, IB itd.). Skojarzone funkcje to: STMFD i STMDB, STMFA i STMIB, STMED i STMDA, STMEA i STMIA, LDMFD i LDMIA, LDMFA i LDMDA, a także LDMEA i LDMDB. Zapamiętanie tego wszystkiego może być dość trudne. Najszybciej zrozumiesz to, tworząc rysunki ilustrujące działanie każdej instrukcji.**

Instrukcje PUSH i POP

Ostatnimi instrukcjami należącymi do instrukcji ładujących dane do pamięci oraz odczytujących dane z pamięci są instrukcje PUSH i POP. Działają one podobnie do instrukcji LDM i STM, z tym że:

- stosują rejestr SP w roli adresu bazowego,
- SP jest automatycznie uaktualniany.

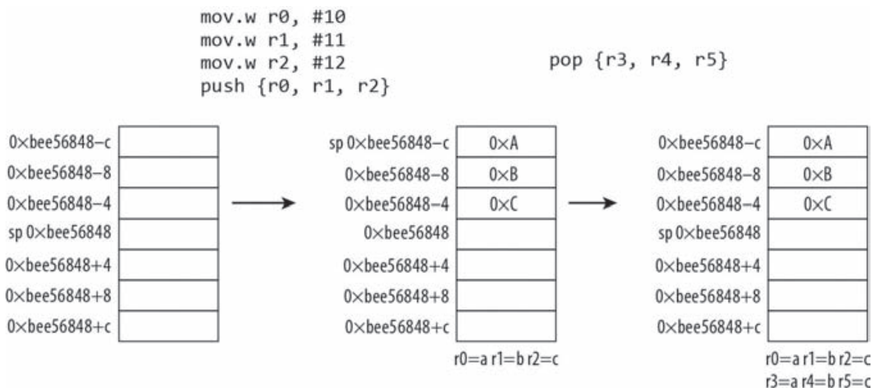
Stos rośnie w dół tak, jak miało to miejsce w przypadku architektur x86 i x64. Ogólna składnia tych instrukcji ma postać: PUSH/POP {Rn}, gdzie Rn może być zakresem rejestrów.

Instrukcja PUSH odkłada rejestry na stos (tak, że ostatnia lokalizacja znajduje się 4 bajty poniżej obecnego wskaźnika stosu), a następnie aktualizuje rejestr SP — zapisuje w nim adres pierwszej lokalizacji. Instrukcja POP ładuje do rejestru dane, zaczynając od pozycji wskazywanej przez bieżący wskaźnik stosu, po czym wpisuje do rejestru SP adres pamięci znajdujący się 4 bajty nad ostatnią lokalizacją. Instrukcje PUSH i POP funkcjonują tak samo jak działające w trybie zwracania adresu instrukcje STMDB i LDMIA, korzystające z rejestru SP jako wskaźnika bazowego. Oto krótki przykład ilustrujący działanie tych instrukcji:

```
01: (gdb) disas main
02: zrzut kodu asemblera funkcji main:
03:   0x00008344 <+0>:   mov.w   r0, #10
04:   0x00008348 <+4>:   mov.w   r1, #11
05:   0x0000834c <+8>:   mov.w   r2, #12
06:   0x00008350 <+12>:  push    {r0, r1, r2}
07:   0x00008352 <+14>:  pop     {r3, r4, r5}
08: ...
09: (gdb) br main
10: punkt wstrzymania 1: 0x8344
11: (gdb) r
12: punkt wstrzymania 1, 0x00008344 w funkcji main ()
13: (gdb) si
14: 0x00008348 w funkcji main ()
15: ...
16: (gdb)
17: 0x00008350 w funkcji main ()
18: (gdb) info reg sp          ; bieżący wskaźnik stosu
19: sp   0xbee56848   0xbee56848
20: (gdb) si
21: 0x00008352 w funkcji main ()
22: (gdb) x/3x $sp             ; spis uaktualniony po wykonaniu instrukcji push
23: 0xbee5683c:   0x0000000a   0x0000000   b0x0000000c
24: (gdb) si                   ; zdjęcie danych ze stosu do rejestru
25: 0x00008354 w funkcji main ()
26: (gdb) info reg r3 r4 r5    ; nowe rejestry
```

```
27: r3   0xa   10
28: r4   0xb   11
29: r5   0xc   12
30: (gdb) info regsp          ; nowa wartość sp (4 bajty nad ostatnią lokalizacją)
31: sp   0xbee56848   0xbee56848
32: (gdb) x/3x $sp-12
33: 0xbee5683c:   0x0000000a   0x0000000b   0x0000000c
```

Działanie tego kodu pokazano na rysunku 2.4.



Rysunek 2.4.

Instrukcje PUSH i POP spotyka się najczęściej na początku i na końcu funkcji. W tym kontekście odgrywają one rolę prologu i epilogu (tak jak instrukcje STMFD i LDMFD w trybie ARM). Na przykład:

```
01: 2D E9 F0 4F   PUSH.W   {R4-R11,LR} ; Zapisuje rejestry i adres zwrotny.
02: ...
03: BD E8 F0 8F   POP.W    {R4-R11,PC} ; Przywraca rejestry i zwraca dane.
```

Niektóre dezasemblery wykorzystują tę technikę w celu określenia granic funkcji.

Funkcje i wywoływanie funkcji

W przypadku architektur x86 i x64 funkcje były wywoływane tylko za pomocą instrukcji CALL i rozgałęziane wyłącznie przy użyciu instrukcji JMP. W architekturze ARM do wywoływania funkcji korzysta sie z kilku różnych instrukcji, zależnie od sposobu kodowania wywoływanych funkcji. Podczas wywołania funkcji procesor musi wiedzieć, jaki kod ma być dalej przetwarzany po zakończeniu jej działania. Kod ten jest określany przez **adres zwrotny**. W przypadku architektury x86 instrukcja CALL bezwarunkowo odkłada adres zwrotny na stos przed przejściem do funkcji docelowej. Gdy funkcja docelowa zostanie wykonana, jej kod wznawia wykonywanie funkcji wywołującej, ściągając odłożony wcześniej adres ze stosu i ładując go do rejestru EIP.

Ogólnie rzecz biorąc, mechanizm ten działa podobnie w architekturze ARM, przy czym charakteryzują go pewne różnice względem mechanizmu znanego z architektury x86. Po pierwsze, adres zwrotny może być przechowywany na stosie albo w rejestrze powiązań (LR). Po zakończeniu

działania wywołanej funkcji adres zwrotny jest jawnie ściągany ze stosu i ładowany do rejestru PC, o ile nie istnieje bezwarunkowe rozgałęzienie kierujące do rejestru LR. Po drugie, rozgałęzienie może przełączać między trybami ARM i Thumb w zależności od najmniej znaczącego bitu adresowanego elementu. Po trzecie, w przypadku architektury ARM przyjęto standardową konwencję wywoływania, która mówi, że 4 pierwsze 32-bitowe parametry są przekazywane za pośrednictwem rejestrów (R0 – R3), a pozostałe są odkładane na stos. Zwracana wartość jest przechowywana w rejestrze R0.

Podczas wywoływania funkcji stosowane są instrukcje B, BX, BL i BLX.

Co prawda instrukcja B jest rzadko używana w kontekście wywoływania funkcji, ale może być zastosowana do przeniesienia kontroli. Jest to po prostu rozgałęzienie bezwarunkowe działające jak znana z architektury x86 instrukcja JMP. Instrukcja B jest zwykle używana w pętlach i konstrukcjach warunkowych w celu powrotu do początku danej konstrukcji lub przerwania jej. Może być użyta również po to, aby wywołać funkcję, która niczego nie zwraca. Instrukcja B może określać cel tylko na podstawie etykiety przesunięcia — nie może do realizacji tego zadania korzystać z rejestrów. W takim kontekście instrukcja B ma następującą składnię: B imm, gdzie imm jest przesunięciem względem obecnie wykonywanej instrukcji. (Nie są tu brane pod uwagę flagi wykonywania warunkowego, które zostaną omówione w podrozdziale „Rozgałęzianie i wykonywanie warunkowe”). Warto tu zauważyć, że instrukcje ARM i Thumb charakteryzują się 4- i 2-bajtowym wyrównaniem, a więc docelowa wartość przesunięcia musi być liczbą parzystą. Oto przykładowy fragment kodu ilustrujący zastosowanie instrukcji B:

```
01: 0001C788   B      loc_1C7A8
02: 0001C78A
03: 0001C78A loc_1C78A
04: 0001C78A   LDRB   R7, [R6,R2]
05: ...
06: 0001C7A4   STRB.W R7, [R3,#-1]
07: 0001C7A8
08: 0001C7A8 loc_1C7A8
09: 0001C7A8   MOV    R7, R3
10: 0001C7AA   ADDS   R3, #2
11: 0001C7AC   CMP    R2, R4
12: 0001C7AE   BLT    loc_1C78A
```

W 1. linii kodu instrukcję B zastosowano w celu wykonania bezwarunkowego skoku uruchamiającego pętlę. Na razie możesz zignorować pozostałe instrukcje.

Instrukcja BX jest instrukcją „skoku i zmiany” — podobnie jak instrukcja B przenosi sterowanie wykonywanym kodem do docelowej funkcji, ale może również przełączać procesor między trybami ARM i Thumb. Adres docelowej instrukcji jest przechowywany w rejestrze. Instrukcje skoku, których nazwy kończą się na literę „X”, mogą przełączać tryb pracy procesora. Jeżeli najmniej znaczący bit adresu docelowego przyjmuje wartość 1, to procesor jest automatycznie przełączany w tryb Thumb. W przeciwnym wypadku będzie pracował w trybie ARM. Instrukcja ta ma składnię BX <rejestr>, gdzie rejestr przechowuje adres docelowy. Instrukcja ta jest najczęściej używana w kontekście kończenia wykonywania jakiejś funkcji i skoku do LR (to jest BX LR) w przypadku przejęcia kontroli nad procesorem przez kod wymagający zmiany trybu pracy procesora (a więc przejścia

Kup książkę Poleć książkę

z trybu ARM w tryb Thumb lub na odwrót). W skompilowanym kodzie prawie zawsze zobaczysz zapis `BX LR`, znajdujący się na końcu funkcji. Podobnie w architekturze x86 funkcje kończy instrukcja `RET`.

Instrukcja `BL` jest instrukcją „skoku i powiązania". Działa ona podobnie jak instrukcja `B`, ale zapisuje adres zwrotny w rejestrze `LR` przed przeniesieniem kontroli nad procesorem do docelowego kodu. Instrukcja ta jest najbliższym odpowiednikiem instrukcji `CALL`, znanej z architektury x86. Jest ona często używana podczas wywoływania funkcji. Charakteryzuje się taką samą składnią jak instrukcja `B` (w roli argumentu przyjmuje tylko przesunięcie). Oto krótki fragment kodu pokazujący wywołanie funkcji, a także jej zakończenie.

```
01: 00014350   BL     foo ; LR = 0x00014354
02: 00014354   MOVS   R4, #0x15
03: ...
04: 0001B224 foo
05: 0001B224   PUSH   {R1-R3}
06: 0001B226   MOV    R3, 0x61240
07: ...
08: 0001B24C   BX     LR ; Wraca do 0x00014354.
```

W 1. linii kodu funkcja `foo` jest wywoływana za pomocą instrukcji `BL`. Przed przeniesieniem kontroli do docelowej funkcji instrukcja `BL` zapisuje adres zwrotny (`0x00014354`) w rejestrze `LR`. Funkcja `foo` wykonuje pewne zadania i wraca do funkcji, która ją wywołała (`BX LR`).

Instrukcję `BLX` można nazwać instrukcją „skoku, powiązania i zmiany". Instrukcja ta jest podobna do instrukcji `BL`, z tym że pozwala również zmienić tryb pracy procesora. Największa różnica między tymi instrukcjami jest taka, że `BLX` może w charakterze rozgałęzienia przyjmować rejestr lub przesunięcie. Gdy instrukcja `BLX` przyjmuje przesunięcie, wtedy zawsze dochodzi do przełączenia trybu pracy procesora (z ARM na Thumb lub odwrotnie). Instrukcja ta ma podobną charakterystykę do instrukcji `BL`, a więc można traktować ją jako odpowiednik instrukcji `CALL`, znanej z architektury x86. W praktyce instrukcje `BL` i `BLX` są używane podczas wywoływania funkcji. Instrukcja `BL` jest zazwyczaj używana, gdy funkcja mieści się w zakresie 32 MB, a instrukcja `BLX` jest stosowana, gdy zakres wywoływanego elementu jest nieznany (do takiej sytuacji dochodzi na przykład w przypadku wskaźnika funkcji). W trybie Thumb instrukcja `BLX` jest zwykle stosowana do wywoływania procedur biblioteki. W trybie ARM w tym celu używana jest zwykle instrukcja `BL`.

Po przeanalizowaniu wszystkich instrukcji służących do obsługi bezwarunkowych rozgałęzień i bezpośredniego wywoływania funkcji, a także zwracania danych przez funkcję (`BX LR`) jesteś gotowy do tego, aby przeanalizować całą funkcję i utrwalić zdobytą wiedzę.

```
01: 0100C388             ; void *__cdecl mystery(int)
02: 0100C388             mystery
03: 0100C388 2D E9 30 48   PUSH.W   {R4,R5,R11,LR}
04: 0100C38C 0D F2 08 0B   ADDW     R11, SP, #8
05: 0100C390 0C 4B         LDR      R3, =__imp_malloc
06: 0100C392 C5 1D         ADDS     R5, R0, #7
07: 0100C394 6F F3 02 05   BFC.W    R5, #0, #3
08: 0100C398 1B 68         LDR      R3, [R3]
09: 0100C39A 15 F1 08 00   ADDS.W   R0, R5, #8
10: 0100C39E 98 47         BLX      R3
11: 0100C3A0 04 46         MOV      R4, R0
```

```
12: 0100C3A2 24 B1         CBZ      R4, loc_100C3AE
13: 0100C3A4 EB 17         ASRS     R3,R5,#0x1F
14: 0100C3A6 63 60         STR      R3, [R4,#4]
15: 0100C3A8 25 60         STR      R5,[R4]
16: 0100C3AA 08 34         ADDS     R4,#8
17: 0100C3AC 04 E0         B        loc_100C3B8
18: 0100C3AE              loc_100C3AE
19: 0100C3AE 04 49         LDR      R1,=aFailed ; „niepowodzenie...”
20: 0100C3B0 2A 46         MOV      R2, R5
21: 0100C3B2 07 20         MOVS     R0,#7
22: 0100C3B4 01 F0 14      FC       BL foo
23: 0100C3B8
24: 0100C3B8              loc_100C3B8
25: 0100C3B8 20 46         MOV      R0, R4
26: 0100C3BA BD E8 30 88   POP.W    {R4,R5,R11,PC}
27: 0100C3BA ; koniec funkcji mystery
```

W funkcji tej znajdują się pewne elementy, które zostały omówione wcześniej (na razie nie zwracaj uwagi na pozostałe instrukcje):

- Prolog funkcji znajduje się w linii oznaczonej numerem 3 (zastosowano w nim instrukcję `PUSH {…, LR}`), a epilog funkcji — w linii oznaczonej numerem 26.
- W 10. linii kodu funkcja `malloc` jest wywoływana za pomocą instrukcji `BLX`.
- W 22. linii kodu wywoływana jest funkcja `foo` za pomocą instrukcji `BL`.
- W 26. linii kodu funkcja wraca za pomocą sekwencji `POP […, PC}` do uprzednio wykonywanego kodu.

Operacje arytmetyczne

Po wczytaniu pewnej wartości z pamięci do rejestru kod może wykonywać na niej różne operacje. Najprostszą operacją jest przeniesienie wartości do innego rejestru za pomocą instrukcji `MOV`. Źródłem tej instrukcji może być stała, rejestr lub element poddany operacji przesuwania bitów. Oto przykłady użycia instrukcji `MOV`:

```
01: 4F F0 0A 00   MOV.W   R0, #0xA        ; r0 = 0xa
02: 38 46         MOV     R0, R7          ; r0 = r7
03: A4 4A A0 E1   MOV     R4, R4, LSR #21 ; r4 = (r4 >> 21)
```

W 3. linii kodu widoczny jest argument, który przed przeniesieniem jest poddawany operacji przesuwania bitów. Do operacji przesuwania bitów można zaliczyć operację przesunięcia w lewo (`LSL`), przesunięcia w prawo (`LSR`, `ASR`) i obrotu (`ROR`, `RRX`). Operacje te są przydatne, ponieważ pozwalają pracować na stałych, które normalnie nie mogłyby być przetworzone w bezpośredniej formie. Instrukcje ARM i Thumb mogą być 16- lub 32-bitowe, a więc nie mogą bezpośrednio przyjmować 32-bitowych stałych w roli argumentów. Operacje przesuwania bitów pozwalają na przekształcanie wartości i przenoszenie ich do innych rejestrów. Sposobem na przeniesienie 32-bitowej stałej do rejestru jest także podzielenie jej na dwie 16-bitowe połówki, które mogą być przeniesione jedna po drugiej. Zwykle czynności te wykonuje się za pomocą instrukcji `MOVW` i `MOVT`. Instrukcja `MOVT` zapisuje dane w 16 wyższych bitach rejestru, a instrukcja `MOVW` — w 16 niższych bitach rejestru.

Kup ksi k Pole ksi k

Podstawowymi operacjami arytmetycznymi i logicznymi są operacje ADD, SUB, MUL, AND, ORR i EOR. Oto przykłady ich użycia:

```
01: 4B 44         ADD    R3, R9             ; r3 = r3+r9
02: 0D F2 08 0B   ADDW   R11, SP, #8        ; r11 = sp+8
03: 04 EB 80 00   ADD.W  R0, R4, R0,LSL#2   ; r0 = r4+(r0 << 2)
04: EA B0         SUB    SP, SP, #0x1A8     ; sp = sp–0x1a8
05: 03 FB 05 F2   MUL.W  R2, R3, R5         ; r2 = r3·r5 (32-bitowy wynik)
06: 14 F0 07 02   ANDS.W R2, R4, #7         ; r2 = r4&7 (flaga)
07: 83 EA C1 03   EOR.W  R3, R3, R1, LSL#3  ; r3 = r3^(r1 << 3)
08: 53 40         EORS   R3, R2             ; r3 = r3^r2 (flaga)
09: 43 EA 02 23   ORR.W  R3, R3, R2, LSL#8  ; r3 = r3|(r2 << 8)
10: 53 F0 02 03   ORRS.W R3, R3, #2         ; r3 = r3|2 (flaga)
11: 13 43         ORRS R3, R2               ; r3 = r3|r2 (flaga)
```

Zwróć uwagę na to, że na końcu nazw niektórych instrukcji znajduje się litera „S”. W przeciwieństwie do architektury x86 instrukcje arytmetyczne w architekturze ARM nie mają określonych domyślnych flag warunkowych. Sufiks „S” informuje o tym, że instrukcja powinna określić arytmetyczną flagę warunkową (zero, wartość ujemna itp.) w zależności od rezultatu wykonania danej instrukcji. Zauważ, że instrukcja MUL obcina wynik — w rejestrze docelowym zapisuje tylko 32 dolne bity. Pełny wynik 64-bitowego mnożenia można uzyskać za pomocą instrukcji SMULL i UMULL (więcej informacji na ten temat znajdziesz w dokumentacji architektury ARM).

Gdzie jest instrukcja wykonująca dzielenie? W architekturze ARM nie przewidziano instrukcji wykonującej bezpośrednio operację dzielenia. (Układy ARMv7-R i ARMv7-M obsługują instrukcje SDIV i UDIV, ale nie są one tutaj omawiane). W praktyce każde środowisko posiada programową implementację operacji dzielenia, a kod może ją po prostu wywołać w razie potrzeby. Oto przykład kodu środowiska Windows C:

```
01: 41 46         MOV   R1, R8
02: 30 46         MOV   R0, R6
03: 35 F0 9E FF   BL    __rt_udiv ; programowa implementacja instrukcji udiv.
```

Rozgałęzianie i wykonywanie warunkowe

Każdy omówiony dotychczas przykład był wykonywany liniowo. W większości programów spotkasz polecenia warunkowe i pętle. Na poziomie asemblera konstrukcje te są implementowane za pomocą flag warunkowych przechowywanych w rejestrze stanu aplikacji (APSR). Rejestr ten jest aliasem rejestru CPSR i działa podobnie jak znany z architektury x86 rejestr EFLAG. Na rysunku 2.5 przedstawiono ułożenie w rejestrach następujących flag:

- N (flaga wartości ujemnej) — Przyjmuje wartość 1, jeżeli wynik operacji jest ujemny (jej najbardziej znaczący bit ma wartość 1).
- Z (flaga zera) — Przyjmuje wartość 1, jeżeli wynik operacji jest zerem.
- C (flaga przeniesienia) — Przyjmuje wartość 1, jeżeli wynik operacji przeprowadzonej na dwóch liczbach bez znaków przekroczy dopuszczalny zakres.

Rysunek 2.5.

- V (flaga przekroczenia zakresu) — Przyjmuje wartość 1, jeżeli wynik operacji przeprowadzonej na dwóch liczbach ze znakami przekroczy dopuszczalny zakres.
- IT (bity „jeżeli-wtedy") — Mogą tu być zapisane różne warunki instrukcji IT wykonywanej w trybie Thumb; zostaną one opisane w dalszej części niniejszego rozdziału.

Flagi N, Z, C i V odpowiadają flagom SF, ZF, CF i OF, znajdującym się w znanym z architektury x86 rejestrze EFLAG. Są one używane podczas implementacji poleceń warunkowych i pętli w językach programowania wysokiego poziomu. Używa się ich także do obsługi wykonywania warunkowego na poziomie instrukcji. Za pomocą tych flag można wyrazić równość. W tabeli 2.1 przedstawiono często spotykane zależności i odpowiadające im flagi.

Tabela 2.1. Kody warunkowe i ich znaczenia

SUFIKS (KOD)	ZNACZENIE	FLAGI
EQ	Równy	Z==1
NE	Różny	Z==0
MI	Minus, wartość ujemna	N==1
PL	Plus, wartość dodatnia lub 0	N==0
HI	Bez znaku, większy	C==1 i Z==0
LS	Bez znaku, mniejszy	C==0 lub z==1
GE	Ze znakiem, większy od lub równy	N==V
LT	Ze znakiem, mniejszy	N!=V
GT	Ze znakiem, większy	Z==0 i N==V
LE	Ze znakiem, mniejszy od lub równy	Z==1 lub N!=V

Instrukcje mogą być wykonywane warunkowo po dodaniu do ich nazw sufiksów wymienionych w tabeli. Na przykład BLT oznacza wykonanie skoku, jeśli warunek LT jest spełniony. (W podobny sposób w architekturze x86 działała instrukcja JL). Instrukcje domyślnie nie uaktualniają flag warunkowych. Są one aktualizowane dopiero po dodaniu sufiksu „S". Instrukcje porównujące (CBZ, CMP, TST, CMN i TEQ) automatycznie aktualizują flagi, ponieważ zwykle są one używane przed instrukcjami rozgałęzienia.

Kup ksi k Pole ksi k

Prawdopodobnie najczęściej spotykaną instrukcją porównania jest `CMP`. Korzysta się z niej za pomocą następującej składni: `CMP Rn, X`, gdzie `Rn` jest rejestrem, a `X` może być adresem komórki pamięci, adresem rejestru lub operacją przesunięcia bitów. Przedstawiona składnia jest podobna do składni używanej w architekturze x86. Instrukcja `CMP` wykonuje operację `Rn-X`, ustawia odpowiednie flagi, a następnie ignoruje wynik. Zwykle umieszcza się ją przed rozgałęzieniem warunkowym. Oto przykład użycia tej instrukcji w formie kodu asemblera i pseudokodu języka C:

ARM

```
01: B3 EB E7 7F   CMP.W   R3,R7, ASR #31
02: 05 DB         BLT     loc_less
03: 01 DC         BGT     loc_greater
04: BD 42         CMP     R5, R7
05: 02 D9         BLS     loc_less
06:             loc_greater
07: 07 3D         SUBS     R5, #7
08: 6E F1 00 0E   SBC.W    LR, LR, #0
09:             loc_less
10: A5 FB 08 12   UMULL.W  R1, R2, R5, R8
11: 87 FB 08 04   SMULL.W  R0, R4, R7, R8
12: 0E FB 08 23   MLA.W    R3, LR, R8, R2
```

Pseudo-C

```
if (r3 < r7) { goto loc_less; }
   else if ( r3 > r7) { goto loc_greater; }
   else if ( r5 < r7) {goto loc_less; }
```

Następną często spotykaną instrukcją porównania jest `TST`. Charakteryzuje się ona taką samą składnią jak instrukcja `CMP`. Jej semantyka jest identyczna z semantyką znanej z architektury x86 instrukcji `TEST`. Instrukcja `TST` wykonuje działanie `Rn&X` i ustawia odpowiednio flagi, a następnie odrzuca wynik działania. Jest zwykle używana do sprawdzenia tego, czy jakieś dwie wartości są równe, lub do sprawdzenia flag. Po tej instrukcji, tak jak po większości instrukcji porównujących, zwykle umieszcza się rozgałęzienie warunkowe. Oto przykład jej zastosowania:

```
01: AB 8A         LDRH    R3, [R5,#0x14]
02: 13 F0 02 0F   TST.W   R3, #2
03: 09 D0         BEQ     loc_10179DA
04: ...
05:             loc_10179BE
06: AA 8A         LDRH    R2, [R5,#0x14]
07: 12 F0 04 0F   TST.W   R2, #4
08: 02 D0         BEQ     loc_10179E8
```

W trybie Thumb-2 często używa się instrukcji porównania `CBZ` i `CBNZ`. Ich składnia jest prosta: `CBZ/CBNZ Rn, `*`etykieta`*, gdzie `Rn` jest rejestrem, a *`etykieta`* — przesunięciem do rozgałęzienia, które ma zostać wykonane, jeżeli warunek jest spełniony. Instrukcja `CBZ` wykonuje skok do etykiety, jeżeli w rejestrze zapisano 0. `CBNZ` działa tak samo, ale sprawdza warunek niezerowy. Instrukcji tych zwykle używa się w celu określenia tego, czy liczba jest zerem lub wskaźnikiem `NULL`. Oto typowy przykład ich zastosowania:

Kup ksi k Pole ksi k

ARM

```
01: 10 F0 48 FF   BL    foo
    ; foo zwraca wskaźnik w r0.
02: 28 B1         CBZ   R0, loc_100BC8E
03: ...
04:              loc_100BC8E
05: 01 20         MOVS  R0, #1
06: 28 E0         B     locret_100BCE4
07: ...
08:              locret_100BCE4
09: BD E8 F8 89   POP.W  {R3-R8,R11,PC}
```

Pseudo-C

```
type *a;
a = foo(...);
if (a == NULL) { return 1; }
```

CMN i TEQ są kolejnymi instrukcjami porównującymi, które wykonują na swoich argumentach operacje dodawania i alternatywy rozłącznej. Instrukcje te są rzadko spotykane, a więc nie będziemy ich szczegółowo opisywać.

Już wiesz, że instrukcja rozgałęzienia (B) po dodaniu odpowiednich sufiksów (BEQ, BLE, BLT, BLS itd.) działa jako instrukcja rozgałęzienia warunkowego. Tak naprawdę większość instrukcji ARM może być uruchamiana warunkowo w taki sam sposób. Jeżeli warunek nie będzie spełniony, to procesor nie wykona takiej instrukcji. Przetwarzanie warunkowe zaimplementowane na poziomie instrukcji może zmniejszyć liczbę rozgałęzień, a więc przyspieszyć działanie programu. Przyjrzyj się temu przykładowi:

ARM

```
01: 00 00 50 E3   CMP      R0, #0
02: 01 00 A0 03   MOVEQ    R0, #1
03: 68 00 D0 15   LDRNEB   R0, [R0,#0x68]
04: 1E FF 2F E1   BX       LR
```

Pseudo-C

```
unk_type *a = ...;
if (a == NULL) { return 1; }
else { return a->off_48; }
```

Od razu możesz zauważyć, że R0 jest wskaźnikiem, ponieważ w 3. linii kodu znajduje się instrukcja LDR. Kod umieszczony w 1. linii sprawdza, czy R0 jest wskaźnikiem NULL. Jeżeli warunek ten jest spełniony (EQ), to kod znajdujący się w 2. linii przypisuje rejestrowi R0 wartość 1. W przeciwnym wypadku NEQ ładuje do rejestru R0 wartość spod adresu R0+0x68 (zobacz 3. linia kodu), a następnie kończy swoje działanie. Warunki EQ i NEQ nie mogą zostać jednocześnie spełnione, a więc tylko jedna z tych instrukcji zostanie wykonana. Zwróć uwagę na to, że w kodzie nie umieszczono żadnych instrukcji rozgałęzienia.

Kup ksi k Pole ksi k

Tryb Thumb

Instrukcje Thumb, w przeciwieństwie do większości instrukcji ARM, nie mogą być wykonywane warunkowo bez instrukcji warunkowej `IT` (wyjątek stanowi instrukcja `B`). Instrukcja ta występuje tylko w trybie Thumb-2. Pozwala na warunkowe wykonanie maksymalnie czterech instrukcji umieszczonych po niej. Jej ogólna składnia ma postać: `IT`*`xyz cc`*, gdzie *`cc`* jest kodem warunkowym dla 1. instrukcji, a *`x`*, *`y`* i *`z`* opisują warunki dla 2., 3. i 4. instrukcji. Warunki dla 2., 3. i 4. instrukcji są definiowane za pomocą litery `T` lub `E`. `T` oznacza, że instrukcja zostanie wykonana, jeżeli warunek *`cc`* zostanie spełniony, a `E` oznacza, że instrukcja zostanie wykonana tylko wtedy, gdy spełniony zostanie warunek przeciwny do *`cc`*. Przeanalizuj poniższy przykład:

ARM

```
01: 00 2B         CMP       R3, #0
    ; Sprawdza i określa warunek.
02:12 BF          ITEE      NE
    ; Rozpoczyna blok IT.
03: BC FA 8C F0   CLZNE.W   R0, R12
    ; pierwsza instrukcja
04: B6 FA 86 F0   CLZEQ.W   R0,R6
    ; druga instrukcja
05: 20 30         ADDEQ     R0, #0x20
    ; trzecia instrukcja
```

Pseudo-C

```
if (R3 != 0) {
   R0 = countleadzeros(R12);
}else {
   R0 = countleadzeros(R6);
   R0+= 0x20
}
```

Kod znajdujący się w 1. linii wykonuje porównanie i ustawia flagę warunkową. W 2. linii kodu zostają określone warunki i rozpoczyna się blok polecenia `if-then`. `NE` jest warunkiem wykonania 1. instrukcji. Pierwsza litera `E` (po `IT`) informuje, że warunek wykonania 2. instrukcji jest odwrotnością warunku wykonania 1. instrukcji (`EQ` jest odwrotnością `NE`). Druga litera `E` informuje, że wykonanie 3. instrukcji obłożone jest takim samym warunkiem. W liniach oznaczonych numerami 3 – 5 umieszczone są instrukcje znajdujące się wewnątrz bloku `IT`.

Instrukcja `IT` z powodu swojej elastyczności może być użyta do zminimalizowania liczby instrukcji wymaganych do implementacji krótkich poleceń warunkowych wykonywanych w trybie Thumb.

Polecenia switch-case

Polecenia `switch-case` można rozumieć jako ciąg wielu poleceń `if-else`. Wyrażenia sprawdzające warunki oraz etykiety docelowe są znane w momencie kompilacji programu, a więc kompilatory zwykle tworzą tablicę skoku, w której umieszczają adresy (tryb ARM) lub przesunięcia (tryb Thumb)

Kup ksi k Pole ksi k

procedury obsługującej każdy przypadek. Po określeniu indeksu tabeli skoku kompilator pośrednio dokonuje rozgałęzienia do właściwej procedury, ładując jej adres do rejestru PC. W trybie ARM zwykle jest to implementowane za pomocą instrukcji LDR oraz rejestru PC, pełniącego funkcję rejestru docelowego i bazowego. Przeanalizuj następujący przykład:

```
01:  ; R1 jest przypadkiem.
02: 0B 00 51 E3    CMP   R1, #0xB    ; Czy mieści się w zakresie?
03: 01 F1 9F 97    LDRLS PC, [PC,R1,LSL#2]    ; Tak, przełącz za pomocą
                                              ; indeksu tabeli.
04: 14 00 00 EA    B     loc_DD10    ; brak przerwania
05: 3C DD 00 00+   DCD   loc_DD3C    ; początek tablicy skoku
06: 4C DD 00 00+   DCD   loc_DD4C
07: 68 DD 00 00+   DCD   loc_DD68
08: 8C DD 00 00+   DCD   loc_DD8C
09: BC DD 00 00+   DCD   loc_DDBC
10: F0 DD 00 00+   DCD   loc_DDF0
11: 38 DE 00 00+   DCD   loc_DE38
12: 38 DE 00 00+   DCD   loc_DE38
13: EC DC 00 00+   DCD   loc_DCEC    ; przypadek-indeks 8
14: EC DC 00 00+   DCD   loc_DCEC    ; przypadek-indeks 9
15: 3C DD 00 00+   DCD   loc_DD3C
16: 3C DD 00 00    DCD   loc_DD3C
17:              loc_DCEC    ; procedura obsługująca przypadki 8, 9
18: 00 00 A0 E3    MOV   R0, #0
19: 08 10 41 E2    SUB   R1, R1, #8
20: 04 30 A0 E3    MOV   R3, #4
21: 14 00 82 E5    STR   R0, [R2,#0x14]
22: BC 31 C2 E1    STRH  R3, [R2,#0x1C]
23: 10 10 82 E5    STR   R1, [R2,#0x10]
```

Kod znajdujący się w 2. linii sprawdza, czy przypadek mieści się w zakresie. Jeżeli nie mieści się, to uruchamiana jest domyślna procedura obsługi (zobacz 4. wiersz). Kod w 3. linii jest wykonywany warunkowo (gdy R1 znajduje się w zakresie). Program dokonuje skoku do procedury obsługi wskazywanej przez tablicę skoku. Adres z tej tablicy jest ładowany do rejestru PC. Jak zapewne sobie przypominasz, rejestr PC (w trybie ARM) jest umiejscowiony 8 bajtów za obecnie wykonywaną instrukcją, a więc tablica skoku jest zwykle przechowywana 8 bajtów od instrukcji LDR.

Technika ta działa podobnie w trybie Thumb, przy czym tablica skoku zamiast adresów zawiera przesunięcia. W architekturze ARM obsługiwane są dwie nowe instrukcje: TBB i TBH, przydatne podczas implementacji tablicy rozgałęzień za pomocą przesunięć o pół słowa. W przypadku instrukcji TBB elementy tablicy są 1-bajtowymi wartościami, a w przypadku instrukcji TBH są elementami typu *half-word*. Aby można było określić cel rozgałęzienia, elementy tablicy muszą zostać pomnożone przez 2 i dodane do adresu znajdującego się w rejestrze PC. Gdybyśmy w poprzednim przykładzie zastosowali instrukcję TBB, to otrzymalibyśmy taki oto kod:

```
01: 0101E600 0B 29         CMP    R1, #0xB      ; Czy mieści się w zakresie?
02: 0101E602 76 D8         BHI    loc_101E6F2   ; Nie, przerwij.
03: 0101E604 04 26         MOVS   R6, #4
04: 0101E606 DF E8 01 F0   TBB.W  [PC,R1]       ; Rozgałęź za pomocą przesunięcia.
05: 0101E60A 06          jpt_101E606 DCB 6      ; początek tablicy skoku
06: 0101E60B 09            DCB    9
07: 0101E60C 0F            DCB    0xF
```

```
08: 0101E60D 18           DCB     0x18
09: 0101E60E 24           DCB     0x24
10: 0101E60F 32           DCB     0x32
11: 0101E610 45           DCB     0x45
12: 0101E611 45           DCB     0x45
13: 0101E612 6D           DCB     0x6D   ; przesunięcie dla 8
14: 0101E613 6D           DCB     0x6D   ; przesunięcie dla 9
15: 0101E614 06           DCB     6
16: 0101E615 06           DCB     6
17: ...
18: 0101E6E4             loc_101E6E4   ; procedura obsługująca przypadki 8, 9
19: 0101E6E4 B1 F1 08 03  SUBS.W  R3, R1, #8
20: 0101E6E8 00 20        MOVS    R0,#0
21: 0101E6EA 60 61        STR     R0,[R4,#0x14]
```

W trybie Thumb rejestr `PC` jest położony 4 bajty po obecnie wykonywanej instrukcji, a więc element tablicy obsługujący przypadek numer 8 (`0x6d`) będzie znajdował się pod adresem `0x0101E612` (`=0x0101E60A+8`), a procedura obsługi znajdzie się pod adresem `0x101E6E4 (=PC+(0x6d·2))`. Podobnie jak to miało miejsce w poprzednim przykładzie, tablica skoku jest zwykle umiejscawiana po instrukcjach `TBB` i `TBH`. Zapamiętaj, że instrukcje `TBB` i `TBH` są używane tylko w trybie Thumb.

Rozmaitości

W tym podrozdziale ogólnie opiszemy zagadnienia, które nie są bezpośrednio związane z inżynierią odwrotną, ale warto o nich wiedzieć. Im więcej wiedzy zdobędziesz, tym lepiej dla Ciebie. Jeżeli czytasz tę książkę pierwszy raz, możesz pominąć ten podrozdział. Gdy wrócisz do niego później, z pewnością łatwiej będzie Ci zrozumieć jego zawartość.

Kompilacja just-in-time i samomodyfikujący się kod

Architektura ARM obsługuje kompilację *just-in-time* (JIT) oraz samomodyfikujący się kod (SMC). Kod JIT jest natywnym kodem dynamicznie generowanym przez kompilator JIT. Na przykład kod zapisany w językach Microsoft .NET jest kompilowany do kodu zapisanego w języku pośrednim (MSIL), który jest następnie przetwarzany do postaci natywnego kodu maszynowego (x86, x64, ARM itd.) — kod ten może zostać wykonany przez procesor. SMC jest kodem generowanym lub modyfikowanym przez bieżący strumień instrukcji. Kod SMC występuje często w powłoce systemowej. Jest on dekodowany i wykonywany w trakcie działania programu. Kody JIT i SMC wymagają umieszczenia nowych danych w pamięci, pobrania ich i przetworzenia przez procesor.

Procesory ARM posiadają dwie oddzielne pamięci podręczne: przeznaczoną dla instrukcji (*i-cache*) i danych (*d-cache*). Instrukcje są przekazywane do procesora za pośrednictwem *i-cache*, a dane przetwarzane przez procesor są przekazywane do niego za pośrednictwem *d-cache*. Nie ma gwarancji, że te dwa rodzaje pamięci są spójne — dane umieszczone w jednej pamięci podręcznej nie muszą być od razu widoczne dla danych umieszczonych w drugiej pamięci. Na przykład wyobraź sobie sytuację, w której w pamięci *i-cache* umieszczono cztery instrukcje wchodzące w skład strumienia instrukcji, a w tym samym czasie użytkownik generuje nowe instrukcje lub dokonuje modyfikacji

instrukcji (co prowadzi do zmiany danych zapisanych w pamięci *d-cache*). Pamięci te nie są spójne, a więc zawartość pamięci *i-cache* może nie wiedzieć o tej modyfikacji. Uruchomienie nieaktualnej wersji instrukcji może spowodować zawieszenie programu lub wygenerowanie nieprawidłowych danych. Podczas tworzenia kodu JIT lub powłoki systemu sytuacja taka jest wysoce niepożądana. Rozwiązaniem jest jawne wymuszenie odświeżania pamięci *i-cache* (można to określić mianem **płukania pamięci podręcznej**). W architekturze ARM dokonuje się tego, uaktualniając rejestr w koprocesorze sterującym pracą systemu (`CP15`):

```
01: 4F F0 00 00   MOV.W   R0,#0
02: 07 EE 15 0F   MCR     p15, 0, R0,c7,c5, 0
```

W większości systemów operacyjnych znajduje się interfejs dokonujący tej operacji, a więc nie będziesz musiał tworzyć go samodzielnie. W systemie Linux możesz skorzystać z polecenia `__clear_cache`, a w systemie Windows — z `FlushInstructionCache`.

Podstawy synchronizacji

W architekturze ARM nie występuje odpowiednik instrukcji `cmpxchg` (porównaj i wymień), znanej z architektury x86. Zamiast niej mamy do dyspozycji dwie instrukcje: `LDREX` i `STREX`. Instrukcje te działają podobnie do instrukcji `LDR` i `STR`, ale uzyskują wyłączny dostęp do pamięci przed operacjami odczytu i zapisu. Są one używane razem w celu implementacji operacji porównania i wymiany. Na przykład:

ARM

```
01: 01 21         MOVS      R1, #1
02:              loc_100C4B0
03: 54 E8 00 2F   LDREX.W   R2, [R4]
04: 1A B9         CBNZ      R2, loc_100C4BE
05: 44 E8 00 13   STREX.W   R3, R1, [R4]    ; R3 jest rezultatem.
06: 00 2B         CMP       R3, #0
07: F8 D1         BNE       loc_100C4B0
```

Pseudo-C

```
if (InterlockedCompareExchange(&r4, 1, 0) == 0) {wykonaj operację; }
```

Kod znajdujący się w 3. linii ładuje dane do rejestru `R2` i porównuje je z 0. Jeżeli dane te okażą się inne, to są wymieniane na 0, a rezultat jest zapisywany w rejestrze `R3`. Jest to implementacja funkcji `InterlockedCompareExchange` systemu Windows.

Od czasu do czasu trafisz na kod zawierający instrukcje `DMB`, `DSB` i `ISB`. Są to bariery, które przed uruchomieniem kolejnej instrukcji mają zapewniać, że zawartość pamięci i załadowane instrukcje będą ze sobą zsynchronizowane. Stosowanie ich jest czasami konieczne w celu zapewnienia właściwej kolejności wykonywanych instrukcji. (Bez tych instrukcji, pełniących funkcję barier, procesor mógłby wykonać polecenia w innej kolejności niż ta wskazywana przez kod asemblera, a inne wykonywane wątki mogłyby nie wiedzieć o modyfikacji niektórych danych). Dlatego instrukcje te spotkasz często w kodzie implementującym blokady.

Kup ksi k Pole ksi k

Mechanizmy i usługi systemowe

Procesor ARM po uruchomieniu rozpoczyna działanie w trybie ARM od przetworzenia danych zapisanych pod adresem 0x00000000 lub 0xFFFF0000 (w zależności od ustawienia koprocesora numer 15). Decyduje o tym bit wektorowy (V), znajdujący się w rejestrze sterowania systemem (CP15, C1/C0). Jeżeli bit ten ma wartość 0, to przyjmowany jest wektor wyjątku 0x00000000, w przeciwnym wypadku przyjmowany jest wektor 0xFFFF0000. Adres ten zwykle odnosi się do pamięci Flash (pamięć RAM nie została jeszcze zainicjowana, a więc nie można z niej korzystać). Dane znajdujące się pod tym adresem nazywane są **wektorami wyjątku**. W architekturze ARM przewidziano listę zdefiniowanych pierwotnie wektorów, która rozpoczyna się pod adresem bazowym. Procedura obsługi wyjątku RESET znajduje się na początku tej listy, a więc jest wykonywana po każdym kolejnym uruchomieniu procesora. Jest to pierwszy kod uruchamiany podczas rozruchu systemu, a więc zwykle rozpoczyna się od sprawdzenia podstawowej konfiguracji sprzętowej — następnie wykonywane są kolejne czynności mające na celu rozruch systemu. Oto wektor wyjątku zapisany w pamięci ROM pewnego urządzenia sprzętowego:

```
01: 00000000 1A 00 00 EA   B   vect_RESET
02: 00000004 12 00 00 EA   B   vect_UNDEFINED_INSTRUCTION
03: 00000008 12 00 00 EA   B   vect_SUPERVISOR_CALL ; (for SWI/SVC)
04: 0000000C 12 00 00 EA   B   vect_PREFETCHABORT
05: ...
06: 00000054              vect_UNDEFINED_INSTRUCTION
07: 00000054 FE FF FF EA   B   vect_UNDEFINED_INSTRUCTION
08: 00000058              vect_SUPERVISOR_CALL
09: 00000058 FE FF FF EA   B   vect_SUPERVISOR_CALL
10: 0000005C              vect_PREFETCHABORT
11: 0000005C FE FF FF EA   B   vect_PREFETCHABORT
12: ...
13: 00000070              vect_RESET
14: 00000070 1C F1 9F E5   LDR   PC, =0x10000078
15:    ; Mapuje kod pod adresem 0x10000078.
16:    ; Rozpoczęcie wykonywania tego kodu.
17: ...
18: 10000078 18 01 9F E5   LDR   R0, =0x2001
19: 1000007C 11 0F 0F EE   MCR   p15, 0, R0,c15,c1, 0
20:    ; Inicjalizuje rejestr określony przez producenta.
21: 10000080 00 00 A0 E1   NOP
22: 10000084 00 00 A0 E1   NOP
23: 10000088 00 00 A0 E1   NOP
24: 1000008C 78 00 A0 E3   MOV   R0, #0x78
25: 10000090 10 0F 01 EE   MCR   p15, 0, R0,c1,c0, 0
26:    ; Inicjalizuje kontrolny rejestr systemu.
```

Po inicjalizacji sprzętu kod wyjątku RESET powoduje wykonanie przez procesor skoku do programu rozruchowego, który jest zwykle zapisany w pamięci Flash lub na wymiennym nośniku, takim jak karty pamięci MMC lub SD. Niektóre urządzenia korzystają z popularnego, otwartego programu rozruchowego U-Boot. Program rozruchowy inicjalizuje kolejne sprzętowe składniki systemu, a następnie wczytuje obraz systemu operacyjnego, mapuje go w pamięci głównej, po czym przenosi do niego kontrolę nad wykonywanymi instrukcjami. Dopiero teraz system operacyjny może zostać uruchomiony, a użytkownik może rozpocząć pracę z urządzeniem.

Kup ksi k Pole ksi k

System operacyjny zarządza zasobami sprzętowymi i zapewnia użytkownikom usługi. Kod użytkownika (zwykle wykonywany w trybie USR) ma niższe uprawnienia od kodu jądra systemu operacyjnego (zwykle wykonywanego w trybie SVC), a więc musi istnieć jakiś interfejs pozwalający mu na generowanie żądań realizowanych przez system operacyjny. W praktyce interfejs ten jest implementowany za pomocą przerwania programowego lub instrukcji pułapki obsługiwanej przez procesor. Usługi są często implementowane jako wywołania systemowe. Na przykład w celu wygenerowania wywołania systemowego w systemie Linux, uruchomionym na komputerze wyposażonym w procesor o architekturze x86, można skorzystać z przerwania `0x80` lub specjalnej instrukcji `SYSENTER`. W architekturze x64 funkcję tę pełni instrukcja `SYSCALL`. W architekturze ARM nie przewidziano specjalnej instrukcji obsługującej wywołanie systemowe, a więc zamiast instrukcji używa się przerwania programowego. Przerwanie sprzętowe powoduje przełączenie procesora w tryb nadzorczy, który pozwala na obsługę przerwania. Tego typu przerwanie może zostać wywołane za pomocą instrukcji `SWI` i `SVC`. Instrukcje te działają identycznie, ale mają inne nazwy.

Obie instrukcje w roli parametru przyjmują bezpośredni adres — w niektórych systemach operacyjnych funkcję parametru pełni indeks tabeli wywołań systemowych. W innych systemach operacyjnych numer wywołania musi być umieszczony w rejestrze (np. w systemie Windows używa się w tym celu rejestru `R12`). W niektórych wersjach systemu Linux numer wywołania systemowego jest umieszczany w rejestrze `R7`, a argumenty są przekazywane za pośrednictwem rejestrów `R0` – `R2`. Na przykład:

Linux (Ubuntu)

```
01: 05 20 A0 E1    MOV    R2, R5    ; trzeci argument
02: 06 10 A0 E1    MOV    R1, R6    ; drugi argument
03: 09 00 A0 E1    MOV    R0, R9    ; pierwszy argument
04: 92 70 A0 E3    MOV    R7, #0x92
; numer wywołania systemowego
05: 00 00 00 EF    SVC    0    ; Wykonaj wywołanie systemowe.
06: 04 00 70 E3    CMN    R0, #4
; Sprawdź zwróconą wartość.
07: 00 30 A0 13    MOVNE R3, #0
; przeniesienie warunkowe uzależnione od zwróconej wartości
```

Windows RT

```
funkcja ZwCreateFile znajdująca się w bibliotece ntdll
4F F0 53 0C    MOV.W    R12, #0x53
01 DF          SVC      1
70 47          BX       LR
; koniec funkcji ZwCreateFile
```

Instrukcja SVC przełącza procesor w tryb nadzorczy, kopiuje odpowiednie rejestry użytkownika do własnej przestrzeni adresowej, wykonuje zadania żądane przez funkcje i zwraca dane po wykonaniu operacji. Skąd instrukcja SVC wie, gdzie ma zwrócić wynik przeprowadzonych operacji? Zwykle jest on zwracany do instrukcji znajdującej się po SVC. Przed przetworzeniem wyjątku adres zwrotny jest kopiowany w trybie SVC do rejestru sprzężonego `R14_svc`. Rejestry sprzężone są aktywne tylko wtedy, gdy procesor działa w pewnym trybie pracy. Na przykład w trybie SVC rejestrami sprzężonymi są `R13_svc` i `R14_svc` — przechowywane są w nich inne wartości niż wartości przechowywane w rejestrach `R13` i `R14` w trybie USR.

Kup ksi k Pole ksi k

Instrukcja BKPT jest przeznaczona do wywoływania programowych punktów przerwań. Punkt przerwania jednak może być zaimplementowany na kilka sposobów. W tym celu można użyć instrukcji BKPT, która uruchamia procedurę obsługującą wyjątek anulowania pobierania wstępnego. Procedura ta może przekazać kontrolę debugerowi. Inną popularną metodą generowania punktu przerwania jest uruchomienie procedury obsługi nieznanej instrukcji w wyniku próby uruchomienia nieznanej instrukcji. W architekturze ARM przewidziano wiele instrukcji, ale bardzo łatwo stworzyć instrukcję o takiej nazwie, która nie występuje w tej architekturze.

Instrukcje

Każda instrukcja wykonywana w trybie ARM ma zakodowany warunek arytmetyczny jej uruchomienia. Standardowym warunkiem jest AL (zawsze uruchamiaj). Ten warunek jest kodowany przez 4 najbardziej znaczące bity kodu operacji (bity 28 – 31). Warunek AL jest definiowany jako 0b1110, czyli 0xE. Jeżeli przyjrzysz się dokładnie fragmentom kodu asemblera (w trybie ARM), to zauważysz, że często kod ten kończy się na 0xE*. Jeżeli otworzysz te instrukcje w edytorze pozwalającym na ich podgląd w trybie szesnastkowym, to zauważysz, że zapis 0xE* pojawia się co 4 bajty. Na przykład:

```
FE FF FF EA FE FF FF EA FE FF FF EA FE FF FF EA
FE FF FF EA 1C F1 9F E5 00 00 A0 E1 18 01 9F E5
11 0F 0F EE 00 00 A0 E1 00 00 A 0E1 00 00 A0 E1
78 00 A0 E3 10 0F 01 EE 00 00 A0 E1 00 00 A0 E1
00 00 A0 E1 00 00 A0 E3 17 0F 08 EE 17 0F 07 EE
```

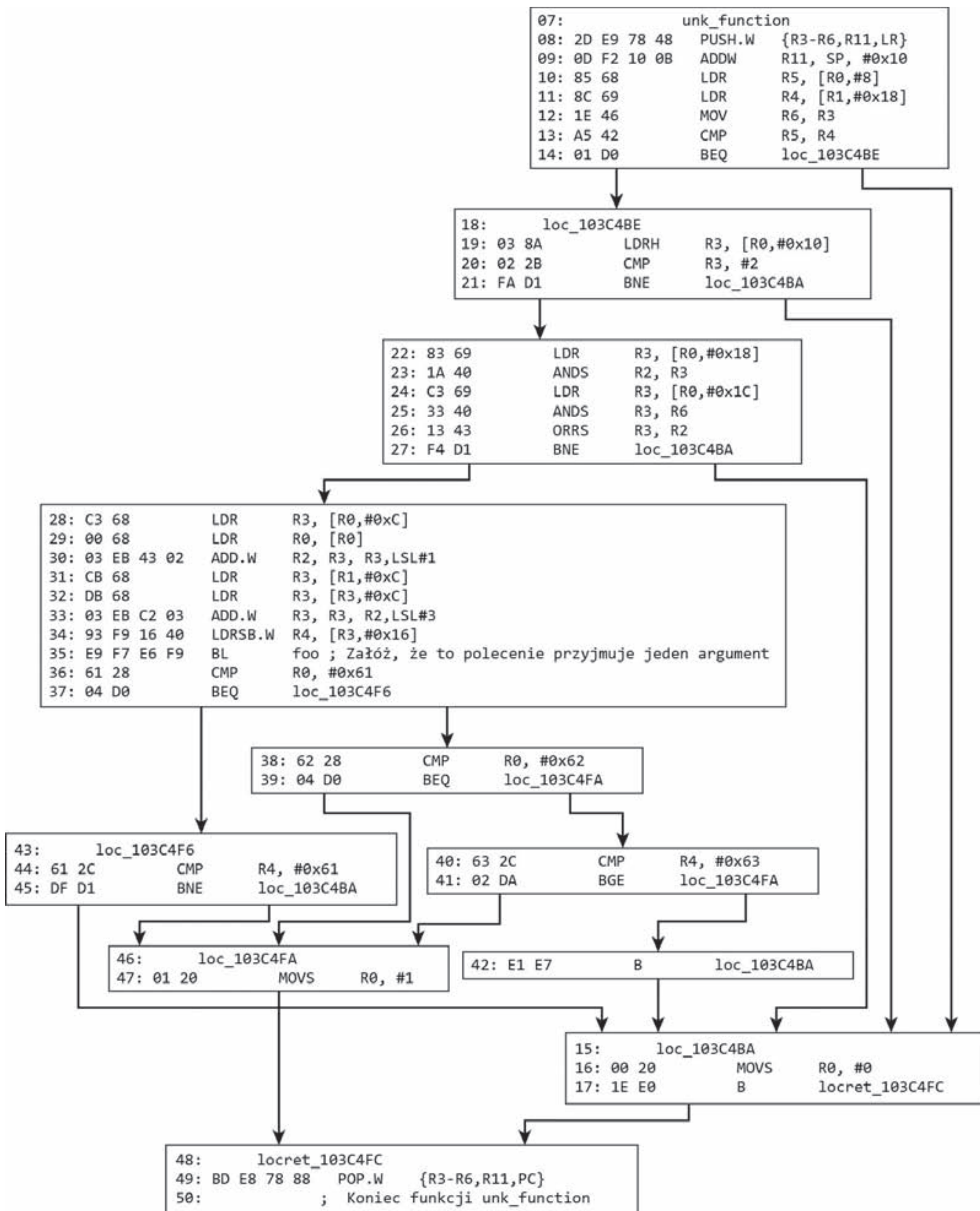
Do czego wiedza ta może Ci się przydać? Kod w architekturze ARM jest często zapisywany w pamięci ROM lub Flash i może nie być zapisany w pliku o określonym formacie. W inżynierii odwrotnej często pracuje się nad surowym zrzutem pamięci, nie dysponując żadnym kontekstem. Warto umieć określić architekturę, patrząc na sam kod operacji. Kolejnym powodem są exploity. Kod powłoki może być wbudowany w exploit dostarczony drogą sieciową lub będący częścią dokumentu. Aby przeanalizować taki kod, musisz oddzielić go od pozostałych danych przesyłanych za pośrednictwem sieci. Nie zawsze granice tego kodu są oczywiste. Jeśli jednak będziesz potrafił dostrzec pewne prawidłowości w analizowanym ciągu danych, dość szybko zauważysz miejsca, w których zaczyna się i kończy poszukiwany przez Ciebie kod. Umiejętność rozpoznawania granic instrukcji w na pozór losowym fragmencie danych jest czymś bardzo ważnym. Być może docenisz ją później.

Analiza krok po kroku

Po opanowaniu teoretycznych podstaw możesz teraz zastosować je w praktyce i dokonać pełnej dekompilacji nieznanej funkcji. W tym celu przyda Ci się wiedza na temat wielu pojęć i technik opisanych w tym rozdziale, a więc będzie to dla Ciebie doskonała okazja do sprawdzenia swoich umiejętności. Próbując zdekompilować wspomnianą funkcję, opanujesz pewne umiejętności, o których tylko napomknęliśmy we wcześniejszych fragmentach niniejszego rozdziału. Funkcja jest dość

Kup ksi k Pole ksi k

długa, a więc postanowiliśmy zaprezentować ją za pomocą grafu (zobacz rysunek 2.6). W dalszej części tego podrozdziału będziemy się odnosić do ponumerowanych linii kodu widocznych na tym rysunku.



Rysunek 2.6.

Oto kontekst, w jakim wywołana zostaje funkcja `unk_function`:

```
01: 17 9B        LDR  R3, [SP,#0x5c]
02: 16 9A        LDR  R2,[SP,#0x58]
03: 51 46        MOV  R1,R10
04: 20 46        MOV  R0, R4
05: FF F7 98 FF  BL   unk_function
```

Rozpoczynając analizę nieznanej funkcji (lub dowolnego fragmentu kodu), należy określić to, co na pewno wiemy o takiej funkcji. Poniżej wymieniono wnioski, które można wysnuć, przyglądając się zaprezentowanemu kodowi. Wnioski zostały opatrzone uwagami informującymi Cię o tym, który wiersz kodu dostarczył danej informacji:

- Kod jest wykonywany w trybie Thumb, a zestaw instrukcji to Thumb-2. Wiemy to, ponieważ: 1) w prologu i epilogu — w wierszach oznaczonych numerami, odpowiednio, 1 i 49 — zastosowano instrukcje `PUSH` i `POP`; 2) instrukcje są 16- lub 32-bitowe; 3) niektóre instrukcje widoczne w kodzie przetworzonym przez dezasembler posiadają sufiks `.W`, który świadczy o tym, że są 32-bitowe.
- Funkcja zachowuje zawartość rejestrów `R3` – `R6` i `R11`. Wiemy to dlatego, że umieszczone tam dane są zachowywane w prologu funkcji (linia 1.) i przywracane w jej epilogu (linia 49.).
- Funkcja przyjmuje maksymalnie cztery argumenty (`R0` – `R3`) i zwraca jedno wyrażenie logiczne (`R0`). Wiemy to, ponieważ zgodnie z binarnym interfejsem aplikacji ARM (ARM ABI) cztery pierwsze parametry są przekazywane za pośrednictwem rejestrów `R0` – `R3` (pozostałe są odkładane na stos), a wartość zwracana przez funkcję jest umieszczana w rejestrze `R0`. Wiemy, że przekazywane są „maksymalnie cztery argumenty", ponieważ rejestry `R0` – `R3` są inicjowane przez pewne wartości przed wywołaniem funkcji (zobacz linia numer 5). W kodzie nie widać żadnych innych instrukcji odkładających dane (kolejne argumenty) na stosie. W tej chwili prototyp funkcji można przedstawić następująco:

  ```
  BOOL unk_function(int, int, int, int)
  ```

- Pierwsze dwa argumenty są „wskaźnikami obiektów". Wiemy to, gdyż `R0` i `R1` stanowią adres bazowy instrukcji ładującej dane (linie o numerach 10 – 11). Dane będą najprawdopodobniej strukturami, gdyż dostęp do nich jest uzyskiwany za pomocą przesunięcia `0x10`, `0x18`, `0x1c` itd. (linie oznaczone numerami 10, 11, 19, 22, 24, 28 itd.). Możesz być niemalże pewien, że nie są to tablice, ponieważ sposób, w jaki uzyskuje się dostęp do tych danych, nie jest sekwencyjny. W tej chwili nie wiadomo, czy `R0` i `R1` zawierają wskaźniki do struktury jednego typu, czy do dwóch różnych struktur. Na razie możemy założyć, że są to dwie struktury różnego typu, a więc prototyp funkcji możemy przedstawić w taki sposób:

  ```
  BOOL unk_function(struct1 *, struct2 *, int, int)
  ```

- `loc_103C4BA` jest adresem, pod którym dane zwróci polecenie `return 0`; `loc_103C4FA` jest adresem, pod którym dane zwróci polecenie `return 1`; z kolei `locret_103C4FA` jest adresem, pod którym znajdą się dane przetwarzane po zakończeniu funkcji. Rozgałęzienia te wskazują na granice funkcji.

- Trzeci i czwarty argument to wartości typu *integer*. Wiemy to, ponieważ R2 i R3 są poddawane operacjom AND i ORR (zobacz linie o numerach 23, 25 i 26). Co prawda istnieje niewielkie prawdopodobieństwo, że mogą być wskaźnikami, ale musiałyby być wskaźnikami kodującymi i dekodującymi. Gdyby parametry te były wskaźnikami, to widzielibyśmy, że są używane w operacjach ładowania danych i odczytywania danych, a takich operacji nie widzimy.
- Zawartość rejestru R11 jest lokowana 0x10 bajtów nad wskaźnikiem stosu, ale nie jest nigdy używana po tej instrukcji, a więc możemy ją zignorować.
- Funkcja foo (zobacz linia oznaczona numerem 35) przyjmuje jeden argument. Z powodu ograniczonego miejsca nie przedstawiliśmy tutaj całego kodu tej funkcji. Załóż, że informacja ta ma po prostu ułatwić Ci pracę.

Po wyliczeniu rzeczy, których możemy być pewni, czas skorzystać z nich, aby na podstawie dedukcji uzyskać inne przydatne informacje. Kolejnym ważnym zadaniem jest zagłębienie się w dwóch zidentyfikowanych, tajemniczych strukturach. Oczywiście nie będziesz w stanie ustalić całego układu wspomnianych struktur, ponieważ analizowana funkcja odwołuje się tylko do pewnych ich elementów, ale możesz określić typy pól tych struktur.

R0 jest typu struct1 *. Kod znajdujący się w linii oznaczonej numerem 10 ładuje pole składowe z przesunięciem 0x8, a następnie porównuje je z R4 (zobacz 13. wiersz kodu). R4 jest polem składowym o przesunięciu 0x18 w strukturze struct2 (R1). Składowe są ze sobą porównywane, a więc muszą być tego samego typu. Kod znajdujący się w 13. linii porównuje te dwa pola. Jeżeli są sobie równe, to wykonywane są dane zapisane w pamięci pod adresem loc_103C4BE. W przeciwnym wypadku zwracane jest 0 (zobacz linia oznaczona numerem 15). Sprawdzana jest równość, a więc możemy domyślić się, że te dwa pola zawierają wartości typu *integer*.

Kod znajdujący się w 19. linii ładuje kolejne pole składowe ze struktury struct1 i porównuje załadowaną wartość z liczbą 2. Jeżeli porównywane liczby nie są równe, to zwracane jest 0 (zobacz linia oznaczona numerem 21). Można więc wywnioskować, że pole to zawiera zmienną typu *short*, ponieważ instrukcja LDRH wczytuje dane typu *half-word*.

W wierszach oznaczonych numerami 22 – 23 ładowane jest kolejne pole składowe struktury struct1. Jest ono następnie poddawane, wraz z trzecim argumentem (podejrzewamy, że jest to liczba typu *integer*), operacji AND. Kod znajdujący się w liniach o numerach 25 – 27 wykonuje podobną operację na czwartym argumencie. Po tych operacjach możesz wnioskować, że pola składowe o przesunięciach 0x18 i 0x1c są liczbami typu *integer*.

Na razie możemy przedstawić następującą definicję struktury:

```
struct1
...
   +0x008 field08_i    ; takiego samego typu jak struct2.field18_i
...
   +0x010 field10_s    ; short
...
   +0x018 field18_i    ; integer
   +0x01c field1c_i    ; integer

struct2
...
   +0x018 field18_i    ; takiego samego typu jak struct1.field08_i
```

Uwaga Możesz przyjąć konwencję polegającą na umieszczaniu w nazwach pól informacji o ich typie i przesunięciu. Na przykład, dodając do nazwy pola sufiks „i”, oznaczymy, że zawiera ono dane typu *integer* (lub dowolny inny 32-bitowy typ danych). Za pomocą sufiksu „s” możesz oznaczyć zmienne typu *short* (16-bitowe), sufiksu „c” — zmienne typu *char* (1-bajtowe), a sufiksu „p” — dowolny wskaźnik. Pozwoli Ci to w łatwy sposób przypomnieć sobie typy pól. Gdy odkryjesz, do czego tak naprawdę stosowane jest dane pole, możesz nadać mu jakąś bardziej wymowną nazwę.

Po określeniu typów tych pól możesz utworzyć pseudokod ilustrujący kod asemblera od linii 1. do 27.:

```
struct1 *arg1 = ...;
struct1 *arg2 = ...;
int arg3 = ...;
int arg4 = ...;

BOOL result = unk_function(arg1, arg2, arg3, arg4);
if (arg1->field08_i == arg2->field18_i) {
   if (arg1->field10_s != 2) return 0;
   if ( ((arg1->field18_i & arg3) |
      (arg1->field1c_i & arg4)
      ) != 0
   ) return 0;
...
}else {
   return 0;
}
```

Wskazówka To dość podejrzane, że operacja AND została przeprowadzona na dwóch sąsiadujących ze sobą polach typu *integer*. Może to być tak naprawdę jedna 64-bitowa wartość typu *integer*, która została podzielona na dwa rejestry lub dwie lokalizacje pamięci. Zabieg taki często stosuje się podczas pracy z 64-bitowymi stałymi w 32-bitowej architekturze.

Bystrzy czytelnicy mogą zauważyć, że kod znajdujący się w liniach o numerach 25 – 27 jest poniekąd zbędny. Instrukcja ANDS inicjuje flagę warunkową, a instrukcja ORRS natychmiast ją nadpisuje. Instrukcja BNE odczytuje stan flagi zmodyfikowany przez instrukcję ORRS, a więc tak naprawdę warunki określane przez instrukcję ANDS są zbędne. Kompilator generuje ten na pozór zbędny kod, ponieważ optymalizuje gęstość kodu: instrukcja AND zajmuje 4 bajty, a instrukcja ANDS — tylko 2 bajty. W ten sam sposób optymalizowane są instrukcje MOV i MOVS. Taką technikę optymalizacji bardzo często spotyka się w kodzie uruchamianym w trybie Thumb.

Kod znajdujący się w 28. linii ładuje kolejne pole struktury struct1 do rejestru R3. W 29. wierszu ta sama struktura z zerowym przesunięciem jest ładowana do rejestru R0. W 30. linii kodu rejestrowi R2 przypisywana jest wartość R3·3 (=R3+(R3<<1)). Kod znajdujący się w 31. linii ładuje do rejestru R3 pole struktury struct2, a następnie uzyskuje dostęp do kolejnego pola, używając tego adresu jako wskaźnika bazowego. Może to oznaczać, że mamy do czynienia ze wskaźnikiem kolejnej struktury o przesunięciu 0xC, znajdującej się wewnątrz struktury struct2. W 32. wierszu kodu do rejestru R3 ładowane jest pole z tej nowej struktury. W 33. linii wartość ta jest modyfikowana

(R3+R2·8), a w 34. linii kodu jest ona używana w charakterze adresu bazowego podczas operacji wczytywania z kolejnej struktury do rejestru R4 zmiennej typu *short* (bez znaku) o przesunięciu 0x16.

Uaktualnijmy definicję struktury przed zagłębieniem się w dalszej analizie funkcji.

```
struct1
   +0x000 field00_i    ; integer
...
   +0x008 field08_i    ; takiego samego typu jak struct2.field18_i
   +0x00c field0c_i    ; integer
...
   +0x010 field10_s    ; short
...
   +0x018 field18_i    ; integer
   +0x01c field1c_i    ; integer
...

struct2
...
   +0x00c field0c_p    ; struct3*
...
   +0x018 field18_i    ; takiego samego typu jak struct1.field08_i
...

struct3
...
   +0x00c field0c_p    ; struct4*
...
struct4 (size=0x18=24)    // dlaczego?
...
   +0x016 field16_c    ; char
   +0x017 end
```

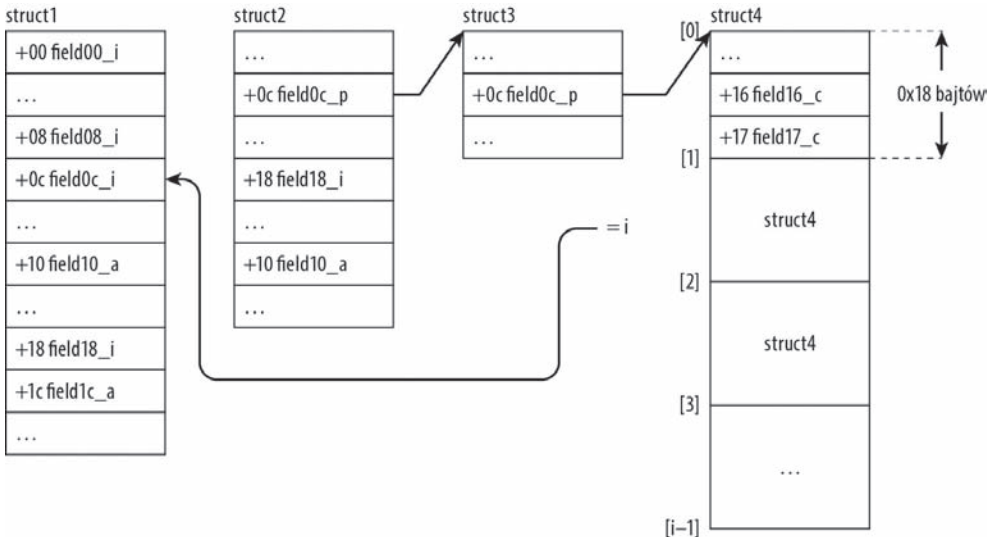
Mogłeś domyślić się, że masz do czynienia z tabelą, patrząc na współczynnik skalowania-mnożenia (zobacz linie oznaczone numerami 30 i 33). Nie były to dwie tablice, ponieważ rejestry R2 – R3 w 30. wierszu kodu nie zawierały adresu bazowego, lecz indeks. Ponadto operacja mnożenia adresu przez liczbę 3 nie miałaby sensu. Bazowy adres tablicy znajduje się w rejestrze R3 w 33. linii kodu — jest on indeksowany za pomocą rejestru R2. Możesz domyślić się, że długość elementów tablicy wynosi 0x18 (24) — w uproszczeniu wykonywana była operacja R2·3·8, gdzie R2 jest indeksem, a 24 jest skalą.

Zależności między czterema strukturami przedstawiono na rysunku 2.7.

Oto pseudokod odzwierciedlający zawartość linii o numerach 28 – 35:

```
r3 = arg1->field0c_i;
r2 = r3 + r3<<1
   = arg1->field0c_i*3;
r3 = arg2->field0c_p;
r3 = arg2->field0c_p->field0c_p;
r3 = arg2->field0c_p->field0c_p + r2*8
   = arg2->field0c_p->field0c_p + arg1->field0c_i*24;
   = arg2->field0c_p->field0c_p[arg1->field0c_i];
r4 = arg2->field0c_p->field0c_p[arg1->field0c_i].field16_c;
r0 = foo(arg1->field00_i);
```

Kup ksi k Pole ksi k



Rysunek 2.7.

Badana funkcja ma po prostu porównać wartość zwróconą przez funkcję `foo` z `r4`. Pełny pseudokod tej funkcji wygląda następująco:

```
struct1 *arg1 = ...;
struct2 *arg2 = ...;
int arg3 = ...;
int arg4 = ...;

BOOL result = unk_function(arg1, arg2, arg3, arg4);
BOOL unk_function(struct1 *arg1, struct2 *arg2, int arg3, int arg4)
{
   char a;
   int b;
   if (arg1->field08_i == arg2->field18_i) {
      if (arg1->field10_s != 2) return 0;
      if ( ((arg1->field18_i & arg3) |
          (arg1->field1c_i & arg4)
         ) != 0
      )return 0;
   b = foo(arg1->field00_i);
   a = arg2->field0c_p->field0c_p[arg1->field0c_i].field16_c;
   if (b == 0x61 && a != 0x61) {
      return 0;
   }else {return 1;}
   if (b == 0x62 && a >= 0x63) {
      return 1;
   }else {return 0;}
   } else {
   return 0;
   }
}
```

Jak widać, pomimo tego, że w funkcji zastosowano wiele niepołączonych ze sobą struktur danych, których struktura nie jest do końca jasna, byłeś w stanie określić typy pewnych pól i zależności między niektórymi elementami funkcji. Dowiedziałeś się również, że analizując instrukcje i ich kody warunkowe, można rozpoznać długość zmiennej, a także to, czy jest ona ze znakiem, czy bez znaku.

Co dalej?

Po przeczytaniu tego rozdziału dysponujesz już podstawowymi umiejętnościami wymaganymi podczas programowania zwrotnego kodu ARM. Celowo ominęliśmy pewne szczegóły — staraliśmy się, żeby rozdział ten nie przypominał instrukcji obsługi. Aby rozwijać swoje umiejętności, musisz wykonać trochę praktycznych ćwiczeń i zapoznać się bliżej z dokumentacją architektury ARM (czynności te najlepiej połączyć). Dokumentacja techniczna to dość trudna lektura, ale dzięki wiedzy wyniesionej z tego rozdziału będzie dla Ciebie bardziej zrozumiała.

Kolejnym krokiem powinien być zakup urządzenia wyposażonego w procesor ARM i eksperymentowanie z nim. Istnieje wiele urządzeń wyposażonych w układy ARM. W celach naukowych najlepiej zaopatrzyć się w płytkę BeagleBoard lub PandaBoard. Wymienione płytki rozwojowe stworzono po to, aby wprowadzić ich użytkowników w świat pracy nad systemami wbudowanymi, opartymi na architekturze ARM. Płytki te mają wiele zastosowań, są względnie tanie (kosztują od około 800 zł do 1000 zł), posiadają obszerną dokumentację i rzeszę użytkowników. (Mogłeś jeszcze nie spotkać osoby znającej się na architekturze ARM, ale to nic nie szkodzi. Niezbędne wiadomości nabyłeś już podczas lektury tego rozdziału. Inni użytkownicy wymienionych wcześniej płytek mogą pomóc Ci w rozwiązywaniu problemów z peryferiami oraz ich programowaniem). Możesz na wspomnianych płytkach uruchomić Linuksa wraz z pełnym środowiskiem programistycznym, a więc praca z nimi stanowi doskonały sposób na zweryfikowanie wiedzy na temat architektury ARM.

Ćwiczenia

Dołączone ćwiczenia mają na celu ugruntowanie Twojej wiedzy, a także zwiększenie motywacji. W niektórych ćwiczeniach celowo użyto instrukcji, które nie zostały omówione w tym rozdziale. Podczas pracy nad takimi ćwiczeniami będziesz musiał wyrobić w sobie bardzo ważny nawyk zaglądania do dokumentacji technicznej. Aby zmusić Cię do myślenia, ćwiczenia nie zostały opatrzone kontekstami wywołania funkcji. Każda z przedstawionych funkcji jest samodzielna, a więc można ją w pełni zdekompilować. Niektóre funkcje wybrano tak, abyś mógł zweryfikować efekty swojej pracy. Warto, żebyś podczas samodzielnej pracy tworzył komentarze, notatki, jak również rozrysowywał zależności między rozgałęzieniami i etykietami.

Pracując nad kodami znajdującymi się w ćwiczeniach, wykonaj kolejno (o ile jest to możliwe) następujące czynności:

- Określ, czy kod jest uruchamiany w trybie Thumb, czy ARM.
- Określ składnię każdej instrukcji. W przypadku instrukcji `LDR` i `STR` określ również tryb adresowania.

Kup ksi k Pole ksi k

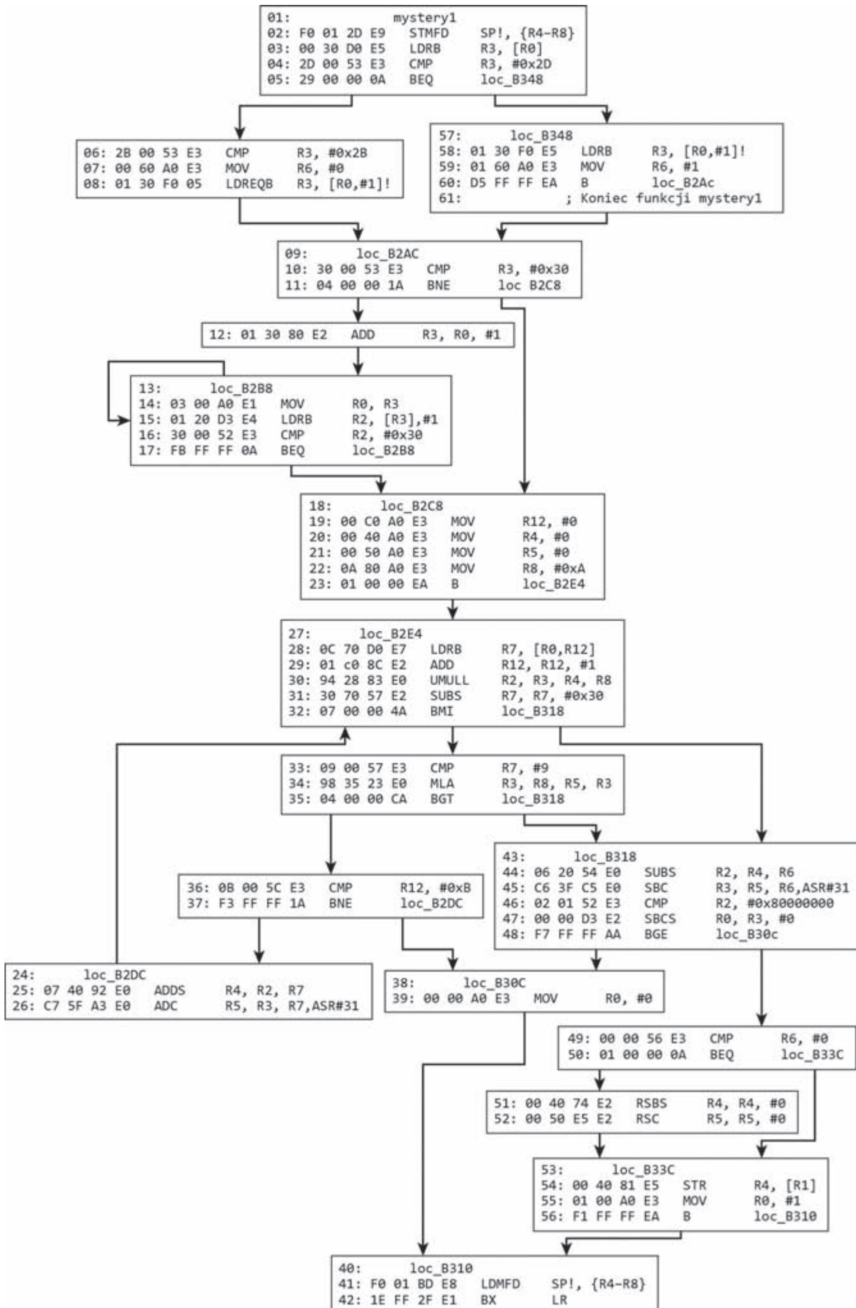
- Określ typ każdego obiektu (liczbę bitów użytych do jego reprezentacji, a także to, czy jest to wartość ze znakiem, czy bez znaku). W przypadku struktur określ rozmiary pól, ich typy, a następnie nazwij je w logiczny sposób. Czasami, gdy funkcja odwołuje się tylko do kilku pól struktury, nie da się ustalić typów wszystkich jej pól. Po określeniu typu każdej zmiennej wyjaśnij sobie (lub komuś innemu), jak do tego doszedłeś.
- Utwórz prototyp funkcji.
- Zidentyfikuj prolog i epilog funkcji.
- Wyjaśnij działanie funkcji, a następnie zapisz ją za pomocą pseudokodu.
- Dokonaj dekompilacji funkcji — zapisz ją w języku C i nadaj jej logiczną nazwę.

1. Na rysunku 2.8 przedstawiono funkcję, która przyjmuje dwa argumenty. Na początku funkcja może wydawać się skomplikowana, ale tak naprawdę jej działanie jest dość pospolite. Uzbrój się w cierpliwość.
2. Na rysunku 2.9 zaprezentowano funkcję, która znalazła się w eksportowanej tabeli.
3. Oto prosta funkcja:

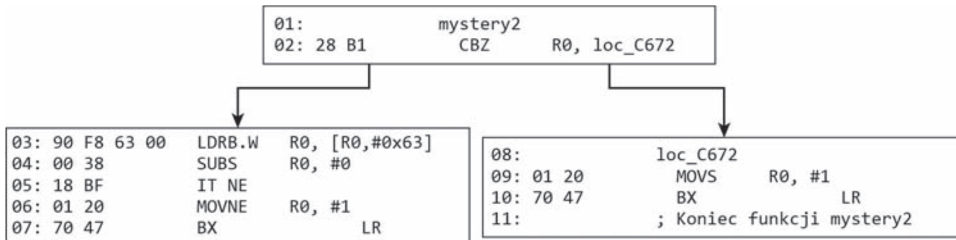
```
1:   mystery3
02: 83 68   LDR   R3, [R0,#8]
03: 0B 60   STR   R3, [R1]
04: C3 68   LDR   R3, [R0,#0xC]
05: 00 20   MOVS  R0, #0
06: 4B 60   STR   R3, [R1,#4]
07:70 47    BX    LR
08:    ; koniec funkcji mystery3
```

4. Na rysunku 2.10 znajduje się kolejna prosta funkcja.
5. Na rysunku 2.11 przedstawiono równie prostą funkcję. Nazwy łańcuchów zostały zmienione, abyś nie oszukiwał i nie starał się znaleźć tej funkcji w internecie.
6. Nad funkcją pokazaną na rysunku 2.12 będziesz musiał się nieco zastanowić.
7. Na rysunku 2.13 zaprezentowano pewien standardowy program zaimplementowany w sposób, którego możesz jeszcze nie znać.
8. Widoczny na rysunku 2.14 element `byteArray` jest tablicą 256 znaków: `byteArray[] = {0, 1, …, 0xff}`.
9. Do czego służy funkcja pokazana na rysunku 2.15?
10. Na rysunku 2.16 przedstawiono jedną z funkcji Windows RT. W razie potrzeby możesz korzystać z portalu MSDN. Zignoruj procedury ochronne elementów `cookie` (instrukcje `PUSH` i `POP`).
11. Widoczna na rysunku 2.17 funkcja `sub_101651C` przyjmuje trzy argumenty i niczego nie zwraca. Jeżeli uda Ci się wykonać to ćwiczenie, możesz się położyć i odpocząć.

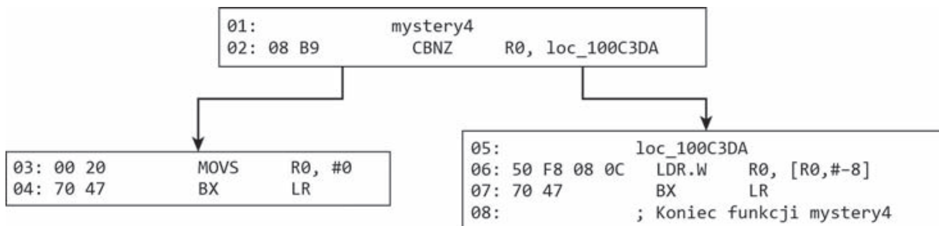
Kup ksi k Pole ksi k



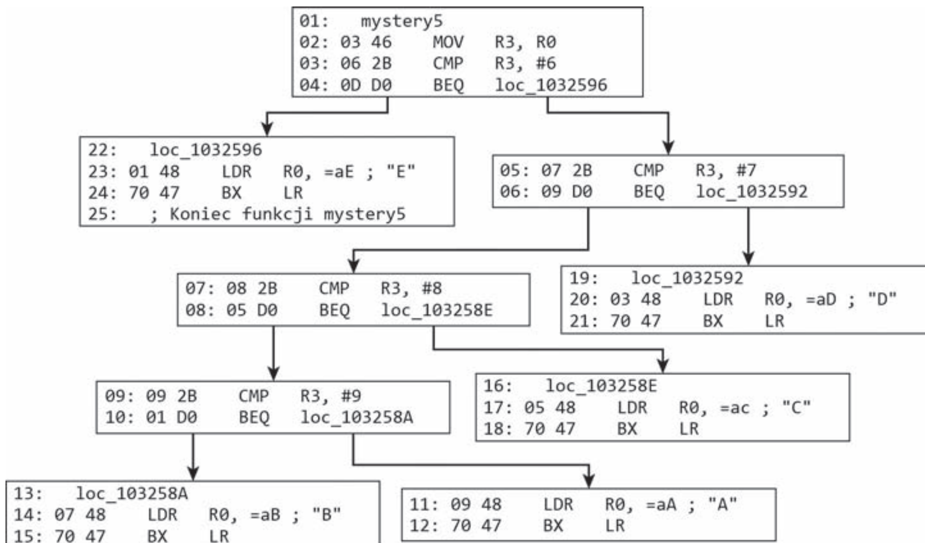
Rysunek 2.8.



Rysunek 2.9.

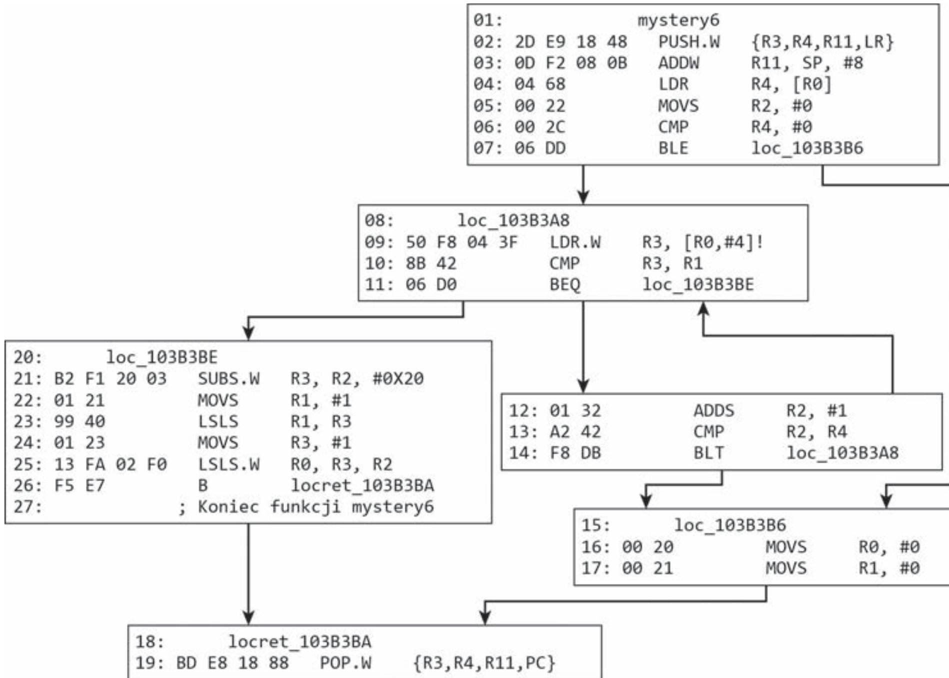


Rysunek 2.10.

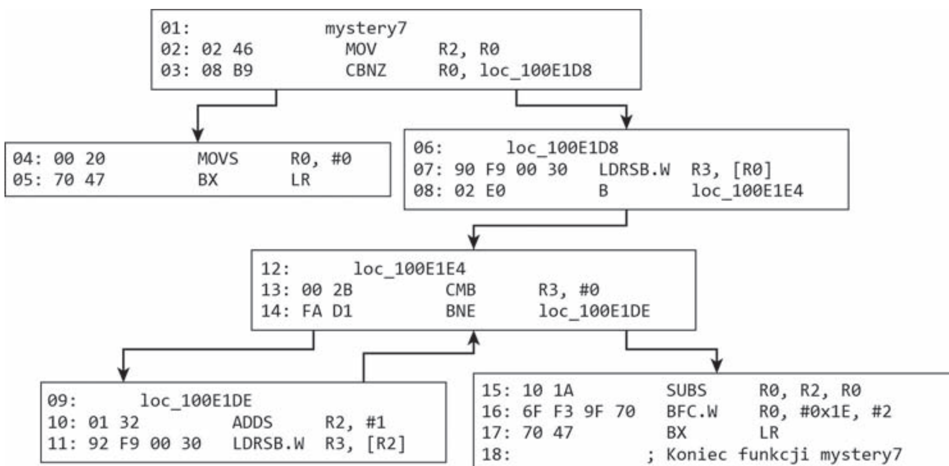


Rysunek 2.11.

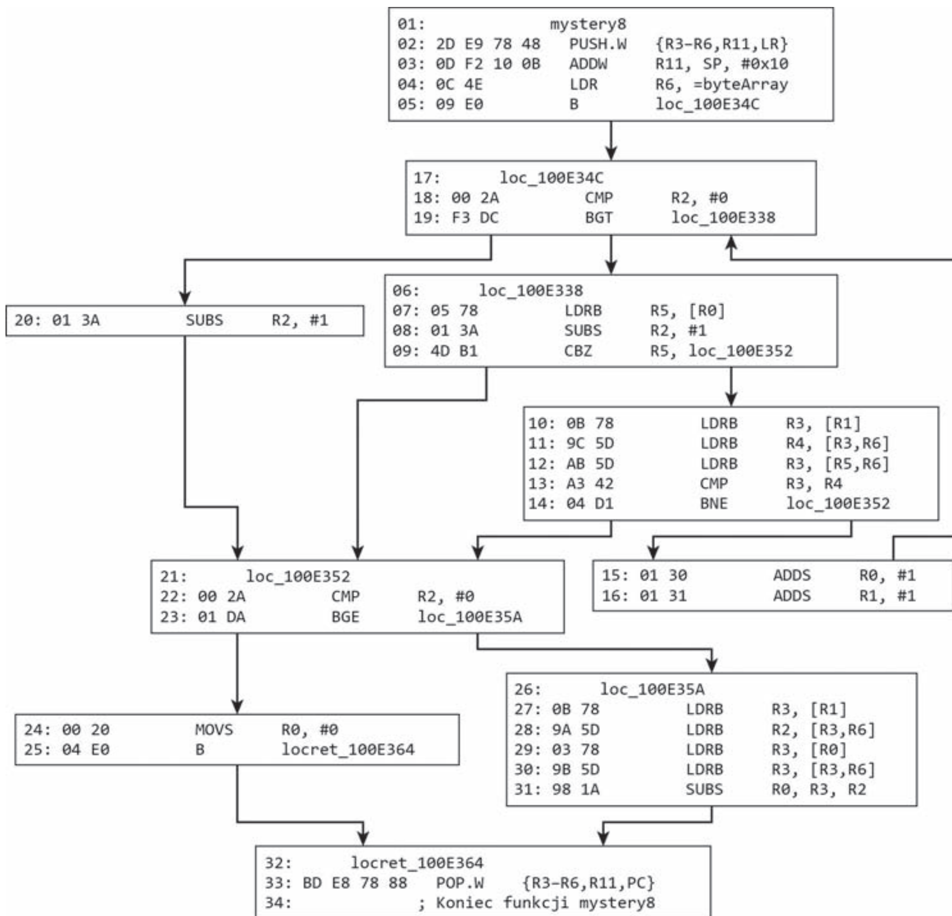
Kup ksi k Pole ksi k



Rysunek 2.12.

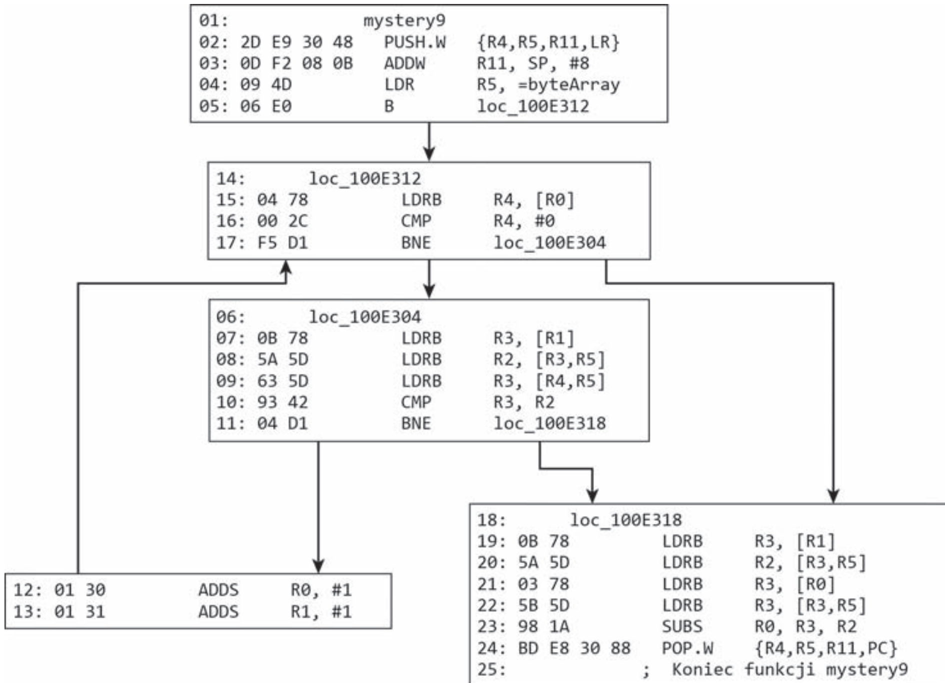


Rysunek 2.13.

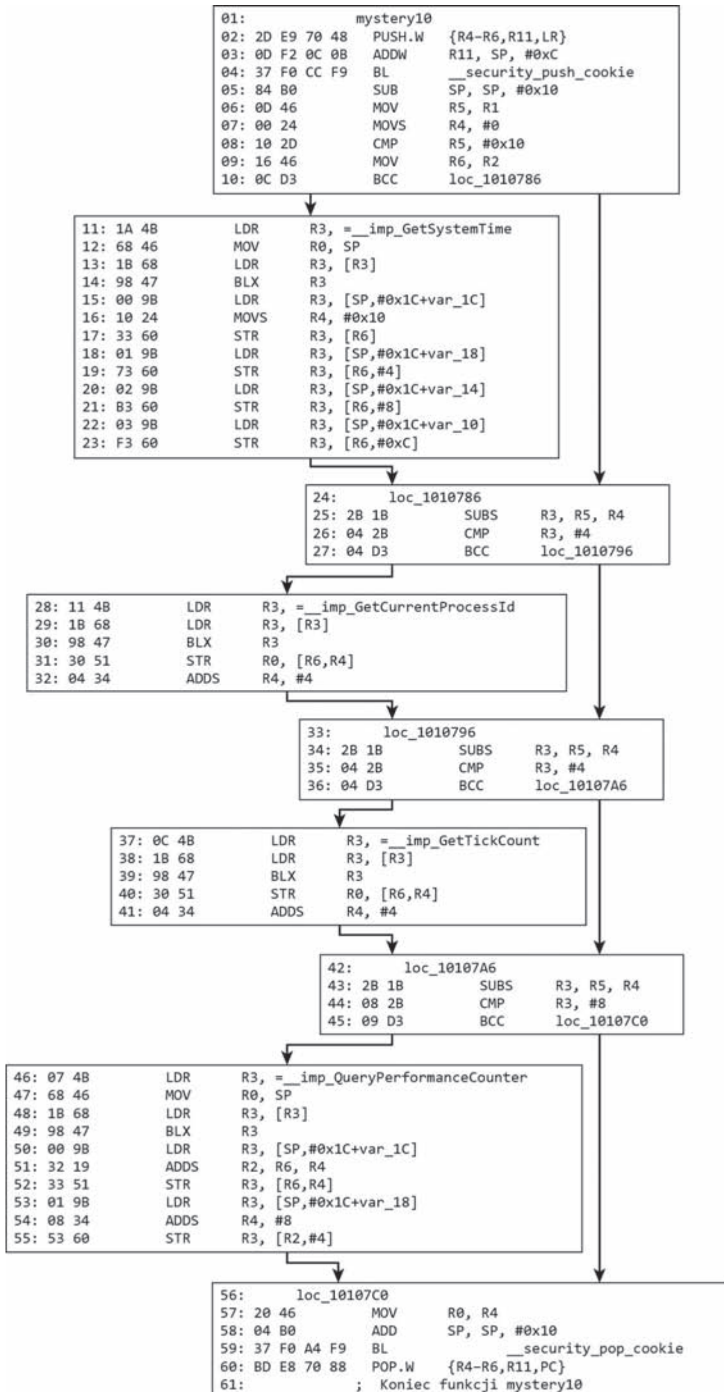


Rysunek 2.14.

Kup ksi k Pole ksi k

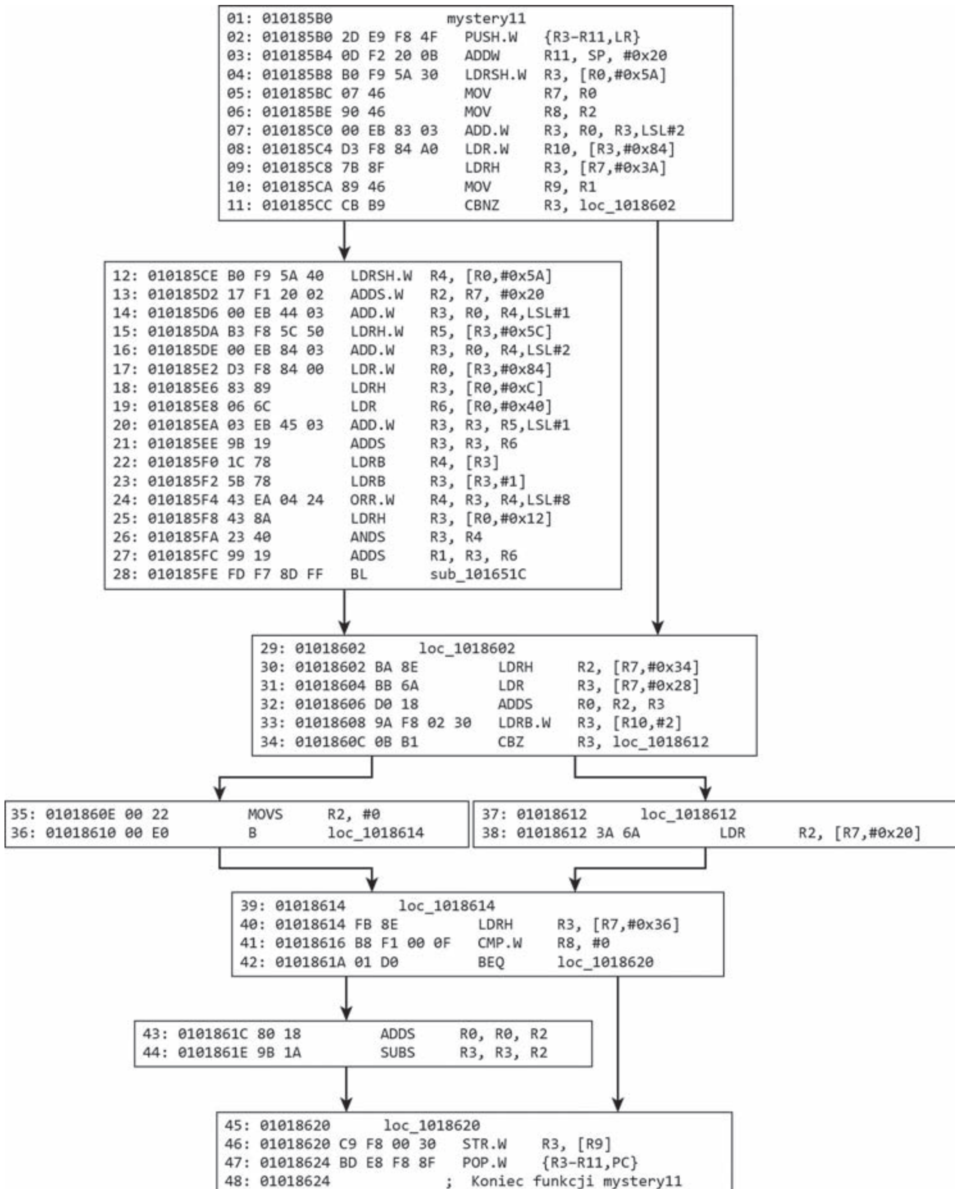


Rysunek 2.15.

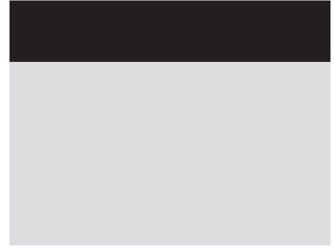


Rysunek 2.16.

Kup ksi k Pole ksi k



Rysunek 2.17.



Skorowidz

A

ABI, Application Binary Interface, 35
adres
  bazowy IDT, 50
  zwrotny, 77
adresowanie
  kanoniczne, 56
  PC-relative, 71
  poindeksowe, 71
  przedindeksowe, 71
  RIP-relative, 56, 71
adresy
  fizyczne, 45
  wirtualne, 45
agresywne optymalizacje, 195
algorytm
  AES, 297
  DES, 296
  zaciemniający, 295
alias skryptu @call, 255
aliasy
  automatyczne, 239
  definiowane przez użytkownika, 233, 236
  o ustalonej nazwie, 238
analiza
  dynamiczna, 300
  działającego programu, 210
  kompletności, 302
  niezawodności, 301
  nieznanej funkcji, 93
  pracy procedur DPC, 157
  prawdziwych sterowników, 201
  procedury IOCTL, 181
  próbki B, 188
  próbki C, 138
  rootkitów, 316
  semantyki procedury, 337
  statyczna, 299, 319
  śledcza, 316
  złośliwego oprogramowania, 334
  zrzutów awaryjnych, 204
APC, Asynchronous Procedure Call, 150
architektura
  ARM
    funkcje, 60, 77
    instrukcje, 66, 91
    kompilacja just-in-time, 87
    ładowanie danych, 67
    opcje systemu, 65
    operacje arytmetyczne, 80
    rejestry, 63
    rozgałęzianie, 81
    samomodyfikujący się kod, 87
    synchronizacja, 88
    typy danych, 63
    usługi systemowe, 89
    ustawienia, 65
    wykonywanie warunkowe, 81
    zapisywanie danych, 67

architektura
CISC, 60
RISC, 60
VxStripper, 315
x64, 55
adresowanie kanoniczne, 56
przenoszenie danych, 56
rejestry, 55
typy danych, 55
wywołanie funkcji, 57
x86, 21
instrukcje, 23
kod asemblera, 24
operacje arytmetyczne, 31
operacje stosu, 32
przenoszenie danych, 25
przerwania, 47
rejestry, 22
sterowanie programem, 37
translacja adresów, 45
typy danych, 22
wyjątki, 47
wywoływanie funkcji, 32
archiwa UPX, 261
ASCII, 248
asembler, 24, 37
asynchroniczne wywołania procedur, APC, 129, 150
atak white box, 278
automatyzacja, 203

B

baza obrazu modułu, 260
bazowy
rejestr tablicy translacji, 108
wskaźnika ramki, 36
biblioteka BeaEngine, 55
biblioteki
dla WOA, 267
DLL, 272
bieżący poziom uprzywilejowania, CPL, 21
blok
.catch, 244, 245
sterowania procesorem, PRCB, 109
blokady pętlowe, 130, 155
bloki, 241
błąd, 114
błąd strony, 114
błędy skryptów, 244
buforowanie, 169

C

CFG, Control Flow Graph, 292
CPL, Current Privilege Level, 21
czas pracy procedury DPC, 157
czasomierze, 130

D

DBT, Dynamic Binary Translator, 312
d-cache, 87
debuger
CDB, 204
jądra, 50
JIT, 205
KD, 204
NTSD, 204
WinDbg, 204
Debugging Tools, 204
debugowanie, 203
kontrolowanie procesów i modułów, 226
narzędzia, 204
obliczanie wartości wyrażenia, 206
polecenia, 204, 229, 230
punkty wstrzymania, 223
zarządzanie procesami, 210
zdarzeń, 210
defekt, 47
definicja struktury, 68
dekompilacja elementu DriverEntry, 176
DEP, 55, 199
deskryptor tabeli serwisowej, 112
deszyfrator łańcuchów, 266
dezasembler, 24, 67, 72
DML, Debugger Markup Language, 229, 251
dostęp do
argumentów, 253
bufora, 169
elementu listy, 137
interfejsów programistycznych, 268
LAPIC TPR, 124
pakietu IRP, 181
pamięci, 25, 26, 27, 60

pola SystemBuffer, 181
struktury PCR, 111
struktury TEB, 128
tabeli wywołań systemowych, 171
tablicy IO_STACK_LOCATION, 178
urządzeń, 174
wskaźnika CurrentStackLocation, 192
DPC, Deferred Procedure Call, 154
DRM, 277
dynamiczna analiza programu, 288
dynamiczne testowanie, 300
dynamiczny translator binarny, DBT, 312
dyspozytor, 294
KiSystemCall64, 119, 120
przerwań systemowych, 120
wywołań systemowych, 118
działanie
procedury DriverUnload, 176
procedury IOCTL, 181
dziedzina abstrakcji, 299
dzielenie liczb, 32

E

edytowanie zawartości pamięci, 221
ekwiwalencja
obliczeniowa, 278
składniowa, 277
element
DriverEntry, 173
EPROCESS, 227
KDPC, 133
LIST_ENTRY, 131
PTE, 45
typu half-word, 86
elementy
robocze, 148
kolejka, 148
parametry, 148
procedura, 148
struktura, 148
WDF
rama projektowa KMDF, 164
rama szkieletowa UMDF, 164
epilog funkcji, 36
exploit, 91

F

flaga
przeniesienia, 37
przepełnienia, 37
zera, 37
znaku, 37
flagi warunkowe, 81
format
big-endian, 64
little-endian, 64
funkcja, 77
0x402CEC, 201
addme, 35
CDECL, 191
CreateThread, 53
CreateToolhelp32Snapshot, 51
DebugCreate(), 274
DeferredRoutine, 154
DriverEntry, 166
DriverInit, 165
ExAllocatePool, 152
foo, 94
func1, 337
func2, 336
GetKernelName, 140
GetLastError, 244
GetLoadedModuleList, 140
InitializeListHead, 131, 132, 142
InsertHeadList, 143
InsertTailList, 143
InterlockedCompareExchange, 88
IoAllocateIrp, 164
IoAllocateWorkItem, 148
IoCallDriver, 164
IoCompleteRequest, 152, 168
IoCreateDevice, 167
IoGetCurrentIrpStackLocation, 192
IoGetCurrentProcess, 189
IoGetRelatedDeviceObject, 191
IopfCompleteRequest, 162
IoQueueWorkItem, 148
IRP_MJ_READ, 179
IsListEmpty, 135
KeGetCurrentThread, 192
KeInitializeDpc, 155
KeInitializeTimer, 159

Kup ksi k Pole ksi k

funkcja
- KeInsertQueueDpc, 134, 155, 156
- KeRaiseIrql, 187
- KeResumeThread, 153
- KernelRoutine, 152
- KeSetTimerEx, 159
- KeSuspendThread, 153
- KiRetireDpcList, 156
- KiStartDpcThread, 157
- KiSystemCall64, 122
- MapMdl, 187
- memset(), 30
- MessageBoxA, 317
- NormalRoutine, 152
- ntdll!NtQueryInformationProcess, 122
- printf, 42
- PsCreateSystemThread, 147
- PsGetCurrentProcessId, 188
- PspInitializeCallbacks, 160
- QueryInterface, 270
- RemoveEntryList, 145
- RemoveHeadList, 144
- RundownRoutine, 152
- strcmp(), 209
- stricmp(), 209
- strlen(), 29
- sub_1000AE3B, 42
- sub_1000CEA0, 55
- sub_10300, 175, 176
- sub_10460, 183
- sub_11553, 140
- sub_115DA, 138–140
- sub_13846, 54
- SYSCALL, 118, 119
- unk_function, 93

funkcje
- FORCEINLINE, 179
- inline, 288
- interfejsu Win32, 51
- IRP, 166
- outline, 288
- rejestrów GPR, 22
- rozszerzenia, 273

G

GPR, General Purpose Registers, 22
graf, 92
graf przepływu sterowania, 292, 294, 309

H

hakowanie interfejsów programistycznych, 316
homomorfizm, 283

I

i-cache, 87
identyfikator SystemProcessId.SystemThreadId, 211
implementacja
- list, 131
- maszyny wirtualnej, 322, 323, 341
- zawieszenia wątku, 153

informacje o
- błędzie, 166
- interakcjach procesu, 314
- procedurach DPC, 155
- statusie, 23
- wywołaniach systemowych, 171
- zmiennych lokalnych, 223

inkrementacja priorytetu wykonywania, 152
instrukcja
- ADD, 24
- ANDS, 95
- B, 78
- BKPT, 91
- BL, 79
- BLX, 79
- branch, 131
- BX, 78
- call, 131
- CALL, 34, 287, 290
- CBNZ, 83
- CBZ, 83
- CMN, 84
- CMOV, 62, 294
- CMOVZ, 291
- CMP, 37, 83
- DIV, 32
- DMB, 88
- DSB, 88

IDIV, 32
IMUL, 31, 32, 294
INT, 168
INT 3, 200
ISB, 88
IT, 85
Jcc, 37
JMP, 37
LDM, 72, 74, 75
LDMIA, 76
LDR, 24, 67–71, 86
LDREX, 88
LEA, 25, 29
LODS, 30
LOOP, 44
MCR, 65
MOV, 25, 69, 70, 80
MOVS, 24, 29, 74
MOVSB, 28, 29
MOVSD, 28
MOVSW, 28
MOVZX, 179
MRC, 65
MUL, 31
ORRS, 95
POP, 33, 76
PUSH, 33, 76
PUSH-RET, 289, 290
RDMSR, 23
RET, 34
SCAS, 29
SETNE, 62
SMULL, 81
STM, 67, 72–75
STMDB, 76
STOS, 29, 30
STR, 24, 67, 69
STREX, 88
SVC, 90, 121, 122
SYSENTER, 120, 121
SYSEXIT, 121
SWI, 90
SYSCALL, 90, 115
SYSENTER, 23, 90, 120
SYSRET, 119
TBB, 86
TBH, 86
TEQ, 84
TEST, 37
TST, 83
UMULL, 81
WRMSR, 23

instrukcje
- arytmetyczne, 37
- porównujące, 82, 84
- procesora ARM, 60
- procesora x86, 23
- przesunięcia, 31
- SIMD, 65
- Thumb, 85
- warunkowe, 242

interfejs
- AuxKlibQueryModuleInformation, 146
- binarny aplikacji, ABI, 35
- DbgEng, 204
- DeviceIoControl, 168
- IDebugAdvanced4, 270
- IDebugClient5, 269
- IDebugControl4, 269
- IDebugDataSpaces4, 269
- IDebugRegisters2, 269
- IDebugSymbols3, 269
- IDebugSystemObejcts4, 270
- IoAttachDevice, 167
- IoSetCompletionRoutine, 161
- KeCancelTimer, 159
- KeInitializeApc, 151
- KeInitializeEvent, 129
- KeInsertQueueApc, 152
- KeStackAttachProcess, 129
- MmGetPhysicalAddress, 198
- PsCreateSystemThread, 147
- WDM, 164
- x64 ABI, 57
- ZwCreateSection, 172

interfejsy
- debugera DbgEng, 268
- programistyczne rozszerzenia WinDbg, 271

interpretacja
- abstrakcyjna, 299
- aliasów, 236

IRP, I/O Request Packet, 162
IRQL, Interrupt Request Level, 109, 123

J

jądro systemu Windows, 107
jednostka MMU, 45
język
  DML, 229
  MSIL, 87
  skryptowy, 240
języki pośrednie, 87
JIT, just-in-time, 87, 205

K

kasowanie plików, 195
katalog
  \??, 174
  stron, PD, 45
KMDF, 164
kod
  IOCTL, 168–170, 182
  jądra, 168
  RESET, 89
  SMC, 87
  użytkownika, 168
  zdekompilowany, 50
kodowanie, 323
kodowanie danych, 283
kody warunkowe, 37, 82
kolejka
  elementów roboczych, 148
  procedur APC, 152
  procedur DPC, 155, 156, 157
komentarze, 240
kompilator, 27, 41
kompilator JIT, 87
kompletność, 301
komunikacja pomiędzy kodami, 168
komunikat o błędzie, 244
konfigurowanie zdarzeń, 213
kontekst
  arbitralny, 129
  ataku białej skrzynki, 295
  maszyny wirtualnej, 327
  systemowy, 129
  wątku, 129
konteksty wykonywania, 128
kontrolowanie procesów i modułów, 226
konwencja wywoływania
  CDECL, 35
  FASTCALL, 35
  STDCALL, 35
  THISCALL, 35
konwersja wartości, 27
koprocesor, 65
kryptografia białej skrzynki, 295

L

LAPIC, 124
lista, 130
  cykliczna dwukierunkowa, 131
  deskryptorów pamięci, MDL, 126
  jednokierunkowa, 131
  jednokierunkowa sekwencyjna, 131
  zdarzeń, 213
listy
  dodawanie elementy, 134
  głowa, 140, 142
  MDL, 183, 185
  ogon, 155
  usuwanie elementu, 135
LLVM, 312
lokalność
  sekwencyjna, 288
  tymczasowa, 288
lokalny kontroler przerwań, 124

Ł

ładowanie
  danych, 67
  sterownika, 165
łańcuchy, 240
łączenie wyrażeń, 207

M

magazyn symboliczny, 300
makroinstrukcja
  #FIELD_OFFSET(), 208
  col, 229
  CONTAINING_RECORD, 137
  CTL_CODE, 169
  NT_SUCCESS(), 175

mapowanie pliku, 204
martwy kod, 284, 293
maska domyślna, 216
maski rejestrów, 215
maszyna wirtualna, 295, 327
maszynowa kontrola błędów, 157
MDL, Memory Descriptor List, 126
mechanizm
  DPC, 149
  Kernel Patch Protection, 171
  KiIdleLoop, 156
  systemowy, 44
  zapobiegania wykonywania danych, 55
menedżer
  modułu, 316
  obiektów, 173
  wejścia-wyjścia, 162, 163
metoda
  code_binding, 326
  METHOD_BUFFERED, 169, 170
  METHOD_IN_DIRECT, 169
  METHOD_NEITHER, 169
  METHOD_OUT_DIRECT, 169
metody buforowania, 169
  bez sprawdzania, 169
  bezpośrednie wejście-wyjście, 169
  buforowane wejście-wyjście, 169
mikrooperacje, 312
MMU, Memory Management Unit, 45
mnożenie liczb, 31
model load-store, 60, 67
moduł
  LLVM, 319
  rozpakowujący, 317
modyfikacje
  aliasów, 214
  emulatora QEMU, 314
monitor plików, 264
monitorowanie
  wyjątków, 212
  zdarzeń, 212
MSR, Model Specific Registers, 23
muteks
  strzeżony, 130
  szybki, 130

N

nagłówki WDK, 162
narzędzia
  do debugowania, 204
  do rozjaśniania kodu, 303
  SDK, 267
narzędzie
  IDA, 303
  Metasm, 304, 324, 330
  Miasm, 310
  Vellvm, 313
  VirtualKD, 274
  VxStripper, 312, 332
nawias
  klamrowy, 242, 252
  kwadratowy, 25
nazwa
  aliasu, 234
  domyślna urządzenia, 174
  kernel32, 219
  pola, 95
nieudokumentowana struktura, 138
niezawodność, 301
niskopoziomowa maszyna wirtualna, 312
normalizacja, 314, 318
notacja
  AT&T, 24
  Intel, 24
numer wywołania systemowego, 120
numery IRQL, 124

O

obiekt
  DRIVER_OBJECT, 166, 174
  EPROCESS, 189
obiekty
  COM, 204
  sterownika, 166
  typu dyspozytor, 157
  typu Timer, 130
  urządzeń, 167, 174
  zdarzeń, 129
obliczanie
  skoków, 306
  symboliczne, 300, 329, 339
  wartości wyrażenia, 206

obsługa
kolejki procedur DPC
mechanizm KiIdleLoop, 156
wątek KiExecuteDpc, 157
zmiana IRQL, 157
pakietów IRP, 167
pakietu IRP, 178
pakietu żądań IRP, 186
pamięci, 219
przerwania, 90
przerwań, 47
tablic, 27
urządzeń wejścia-wyjścia, 161
wyjątków, 47, 115, 122, 213
wyjątku, 89
zdarzeń, 213
ochrona
modyfikacji jądra systemu, 171
własności intelektualnej, 277
odczyt wirtualnego adresu, 45
odwołania symboliczne, 174
odwracanie przekształceń, 298
OEP, Original Entry Point, 317
okna debugera, 205
określanie ścieżki plików symboli, 205
ominięcie wskaźnika ramki, 36
operacja
IRP_MJ_DEVICE_CONTROL, 168, 175
pop, 32
push, 32
operacje
arytmetyczne, 31, 80
na bitach, 31
na listach, 131
na stosie, 33
stosu, 32
wejścia-wyjścia, 169
operator, 207
$iment(), 209, 251
$scmp(), 209
$sicmp(), 209
$spat(), 210
$vvalid(), 210
by(), 209
dwo(), 209
indeksowania tablic, 209poi(), 208
OR, 215
poi(), 209
sizeof(), 208
strzałki, 208
trójargumentowy, 208
wo(), 209
wyłuskiwania, 208
opis pamięci, 172
opóźnione wywołania procedur, DPC, 129
oprogramowanie Windows Driver Kit, 130
optymalizacja
kodu, 284
przez dziurkę, 286
oryginalny punkt rozpoczęcia, OEP, 263, 317
orzeczenia nieprzejrzyste, 292

P

PAE, Physical Address Extension, 45
pakiet Debugging Tools, 204
pakiety żądań wejścia-wyjścia, IRP, 162, 167, 178
pamięć, 219
edytowanie zawartości, 221
podręczna
d-cache, 87
i-cache, 87
parametr określający adres, 219
parametr zakresu, 219
polecenia, 221, 222
zrzut, 220
parametr ProcessHandle, 129
parametry
elementu roboczego, 148
IRP, 168
żądania wejścia- -wyjścia, 180
PD, 45
PDPT, 45
PE, Portable Executable, 317
pętla
do-while, 247
for, 42, 245
foreach, 248, 250
while, 247
platforma MMU, 171
plik
calc.exe, 204
dml.doc, 230
kernel32.dll, 116

kernelbase.dll, 116
ntdll.dll, 116
ntoskrnl.exe, 140
sum-mem.wds, 254
pliki
*.cab, 204
*.dmp, 204
.def, 274
.log, 239
.wds, 255
DLL, 165
PE, 204
skryptów, 251
ze skryptami, 253
płukanie pamięci podręcznej, 88
podstawa systemu liczbowego, 207
pola struktury KDPC, 154
pole
BaseDllName, 140
CompletionRoutine, 161
CurrentStackLocation, 179
DriverUnload, 166
FileInformationClass, 193
FileObject, 193
FullDllName, 140
ImageSectionObject, 193
IoStatusBlock, 182
KeServiceDescriptorTable, 187
LIST_ENTRY, 139
MajorFunction, 179
SystemBuffer, 181
polecenia
debugera, 210
otwierające plik, 251
rozszerzenia, 273
switch-case, 85
polecenie
!for_each_process, 227
!process, 226
$$, 240
$$><, 253
$$>a<, 253
$><, 253
.continue, 245
.dvalloc, 256
.dvfree, 250
.elsif, 242
.expr, 206
.foreach, 260
.if, 242
.leave, 245
.printf, 229
.sympath, 205
?, 206
??, 206, 207
@call, 258
|Ns, 212
~Ns, 211
aS, 234, 237
bc, 224
bd, 224
d, 220
dt, 223
dps, 221
e, 222
f, 222
foreach, 250
gc, 228
goto, 41
if-else, 38
j, 243
lm, 226, 260
n, 206
r, 214, 215
r?, 246
REPL, 320
rF, 216
rm, 216
rX, 216
switch-case, 39
sx, 213
połączenie procesora, 47
porównywanie łańcuchów, 209
powtarzanie operacji, 28
poziom
APC, 124
bierny, 124
DISPATCH_LEVEL, 125, 130, 155–157
dyspozytora, 124
IPI_LEVEL, 124
LocalSystem, 200
uprawnień, 55
żądań przerwania urządzenia, IRQL, 123

prefiks
#, 256
@, 54, 231, 256
@@, 207
~N, 212
REP, 29, 74
preprocesor, 109
procedura, 142–146
0x4038F0, 190
DllMain, 47
DriverUnload, 166, 175, 176
InsertHeadList, 133, 134
InsertTailList, 133, 134
IoBuildDeviceIoControlRequest, 201
IoCompleteRequest, 161
IoCreateDevice, 167
IOCTL, 147, 181
IOCTL_1_handler, 183
IoGetCurrentIrpStackLocation, 178
IoSetCompletion, 161
IoSetCompletionRoutine, 163
ISR, 154
KeInitializeTimer/Ex, 130
kernelbase!CreateFileW, 116
KeWaitForSingleObject, 193
KiSystemServiceExit, 123
MmProbeAndLockPages, 185
obsługi wyjątku, 89
PsGetCurrentProcess, 111
PsGetCurrentThread, 111
RemoveEntryList, 137
RemoveHeadList, 135
RemoveTailList, 136
StartRoutine, 147
sub_10300, 175
procedury
APC, 150
bez rozgałęzień, 183
DPC, 154
czas pracy, 157
wątkowe, 156
przydzielające, 167
typu forceinline, 179
wywoływane z opóźnieniem, DPC, 150, 154
zakończenia, 161
związane z timerami, 158
proces IOCTL_PROCESS, 170
procesory ARM, 64
procesy, 127
program
AuxKlibQueryModuleInformation, 141
rozpakowujący archiwa UPX, 261
prolog funkcji, 36, 49, 80
propagacja stałych, 299
prototyp
funkcji, 51
procedur przydzielających, 167
zwrotnego wywołania, 151, 189
próbka
B, 188
C, 138
D, 201
E, 201
F, 202
G, 171
H, 54
J, 40, 47
L, 55
przeglądanie z powrotami, 305, 306
przekazywanie argumentów, 253
przekształcenia
homomorficzne, 283
matematyczne, 284
przenoszenie danych, 25, 56
przepływ
danych, 281
sterowania, 281
przerwania, 47, 114
przerwania sprzętowe, 47, 90, 114
przestrzeń pamięci jądra, 108
przesunięcie, 69, 70, 138
przetwarzanie
kodu w trybie concolic, 340
kolejki procedur DPC, 156
pakietu IRP, 182
przybliżenie
dokładne, 281
zgrubne, 281
przybliżony częściowy porządek, 299
przywileje, 61
pseudoalgorytm, 325
pseudoinstrukcje, 67, 71
pseudokod, 95–97
pseudorejestr, 231, 246
definiowany przez użytkownika, 232
zdefiniowany pierwotnie, 231

PT, 45
PTE, Page Table Entry, 45
pule pamięci, 125
pułapka, 47, 114, 118, 123
punkt
  rozpoczęcia DriverEntry, 165
  wstrzymania
    nieustalony, 224
    programowy, 223, 225
    sprzętowy, 224, 225
    warunkowy, 225, 243

Q

QEMU, Quick EMUlator, 312

R

rama projektowa, 304, 310
  rozszerzeń DbgEng, 268
  rozszerzeń WdbgExts, 268
rejestr, 55, 63, 214
  CPSR, 64
  CR3, 108
  EBP, 22
  ECX, 22
  EDI, 22
  EFLAG, 81
  EFLAGS, 23, 37
  ESI, 22
  ESP, 22, 33
  IDTR, 50, 115
  R13, 63
  R14, 63
  R15, 63
  Rm, 72
  Rn, 72
  TTBR, 108
rejestry
  do debugowania, 23
  ogólnego przeznaczenia, GPR, 22
  specyficzne dla modelu, MSR, 23
  sterowania, 170
  UDPR, 255
relewantna struktura definicji, 51
ring, 21
rootkit, 171, 173, 176
  w architekturze x64, 188
  w architekturze x86, 173
rozdzielanie, 300, 306
rozgałęzianie instrukcji, 61
rozjaśnianie kodu, 297, 298, 303
  na bazie wzorca, 319
  oparte na analizie programów, 320
  spłaszczonego, 332
rozkład pamięci, 108
rozmiar
  instrukcji, 24
  pakietu IRP, 192
rozszerzenia
  adresów fizycznych, PAE, 45
  debugera, 271, 272
  EngExtCpp, 268
  pętli foreach, 250
  praktyczne, 274
  testujące, 274
rozszerzenie
  !analyze, 274
  !exploitable, 275
  DBT QEMU, 313
  narly, 274
  PyKd, 275
  qb-sync, 275
  SOS, 274
rozwijanie
  pętli, 323
  stałych, 282, 283
rzutowanie typu, 208

S

samomodyfikujący się kod, SMC, 87
schematy kodowania danych, 283
sekcja, 172
semantyka
  abstrakcyjna, 299
  programu, 299
  śladów, 299
separacje przywilejów, 21
składnia
  C++, 206
  MASM, 206

Kup książkę Poleć książkę

skrypt, 230
bp_displayfn.wds, 264
call.wds, 256
dvalloc.wds, 259
init.wds, 256, 258
pi.wds, 260
sigma.wds, 259
test.wds, 260
skrypty jako funkcje, 255
SMC, 87
solver ograniczenia, 300
spaghetti code, 288
specyfikacja NDIS, 197
specyfikator $$, 240
specyfikatory formatu danych, 222, 229
spłaszczanie
grafu przepływu sterowania, 294
kodu, 322, 323
stałe FILE_DEVICE_*, 170
stan aplikacji, 81
standardowa implementacja, 341
status
pakietu IRP, 178, 193
STATUS_PENDING, 182
sterowanie
pośrednioścą
za pomocą procesora, 289
za pomocą systemu operacyjnego, 291
procesami, 210
skokiem warunkowym, 309
urządzeniem wejścia-wyjścia, IOCTL, 168
wątkami, 210
wykonywanym programem, 37
sterownik, 126, 129, 141, 146
analiza budowy, 195
dostęp do bufora, 169
dostęp do tablicy IO_STACK_LOCATION, 178
kasowanie plików, 195
KSECDD, 200
ładowanie, 165
model WDM, 164
obiekty, 166
oprogramowania spadkowego, 164
procesora, PCR, 109, 192
przyłączanie urządzeń, 167
punkt rozpoczęcia pracy, 165
spadkowy typu filtr, 164
ścieżka rejestracji, 166
tworzenie sekcji, 172
typu filtr, 167, 168
typu minifiltr systemu plików, 164
stos, 32
strona, 45
strona zawierająca kod, 46
struktura
DEVICE_OBJECT, 166
DRIVER_OBJECT, 165
EPROCESS, 126
ETHREAD, 126
FAST_MUTEX, 130
GUARDED_MUTEX, 130
IO_STACK_LOCATION, 161–163, 169, 180
IO_STATUS_BLOCK, 178
IRP, 162, 163, 179, 192
KAPC, 150
KDPC, 68, 132, 154
KdVersionBlock, 140
KLDR_DATA_TABLE_ENTRY, 140, 142
KNODE, 148
KPRCB, 148, 155
KPROCESS, 126
KSERVICE_TABLE_ DESCRIPTOR, 171
KSPIN_LOCK, 130
KTHREAD, 126, 152, 153
KTIMER, 130, 158
KUSER_SHARED_DATA, 117
LARGE_INTEGER, 188
LDR_DATA_TABLE_ENTRY, 140
LIST_ENTRY, 131, 132, 137
MDL, 126
nt!_KIDTENTRY64, 115
PCR, 109
PE, 317
PRCB, 109, 130
PROCESSENTRY32, 52, 53
PsLoadedModuleList, 140
sterownika, 164
TEB, 128, 218
typu KDPC, 26
UNICODE_STRING, 141, 173
WINDBG_EXTENSION_APIS, 271, 272
struktury
niejawne, 128
nieprzezroczyste, 198

powtórzenia, 245
związane z timerami, 158
studium przypadku, 335
analiza semantyki procedury, 337
obliczanie symboliczne, 339
rozwiązanie zadania, 340
sumy kontrolne SHA1, 347
symbol KeServiceDescriptorTable, 171
symbole, 223
polecenia, 223
symboliczny stan programu, 300
synchronizacja, 88
synchronizacja jądra, 129
system
binarny, 27
liczbowy, 207
systemowe rejestry sterowania, 170
szkodliwe oprogramowanie, 277
szyfrowanie, 323

Ś

ścieżka
plików symboli, 205
RegistryPath, 166
śledzenie kodu, 211

T

tablica, 27
IDT, 50, 115, 116, 168, 246
IDTR, 50
IO_STACK_LOCATION, 178
KeServiceDescriptorTable, 112
KeServiceDescriptorTableShadow, 112
KiServiceTable, 112, 114, 171, 172
MajorFunction, 166, 175
PspCreateProcessNotifyRoutine, 160
PspCreateThreadNotifyRoutine, 160
PspLoadImageNotifyRoutine, 160
struktur KDPC_DATA, 156
W32pServiceTable, 112, 114
wektorów wyjątków, 121
wskaźników i przesunięć funkcji, 112
wskaźników KNODE, 70
tablice
globalne, 112
deskryptorów przerwań, 115, 245
skoku, 40, 85
stron, PT, 45
struktur, 192
TCG, Tiny Code Generator, 312
technika
DEP, 199
DRM, 277
rozjaśniania kodu, 297
zaciemniania kodu, 279, 322
technologia
MMX, 65
NEON, 65
SSE, 65
timer okresowy, 159
timery, 158
procedury, 158
struktury, 158
tokenizacja
danych, 249
łańcucha, 248
pliku, 250
transformacja programu, 277
translacja adresów wirtualnych, 45
tryb
ABT, 61
ARM, 62
chroniony, 21
concolic, 340
FIQ, 61
IRQ, 61
MON, 61
rzeczywisty, 21
SVC, 61, 90
SYS, 61
Thumb, 61, 62, 85
UND, 61
USR, 61, 90
tryby pracy, 61
tworzenie
aliasów, 233
deszyfratora łańcuchów, 266
monitora plików, 264
rozszerzeń narzędzi debugujących, 272
sekcji, 172
wątków, 147

Kup ksi k Pole ksi k

typ danych, 22, 55, 63
byte, 22, 27
double word, 22, 27, 28
integer, 31
quad word, 22
word, 22, 27
typy wyrażeń, 310

U

ukrycie kodu w algorytmie, 295
UMDF, 164
unia, 163
AssociatedIrp, 181
Overlay, 192
Parameters, 193
SetFile, 192
Unicode, 264
uprawnienia procesów, 200
usługi systemowe, 89
usuwanie martwego kodu, 284

V

Vellvm, Verified LLVM, 334

W

warstwa abstrakcji sprzętowej, 124
wartości przesunięć, 198
wartościowanie częściowe, 300
wątek, 127
ExpWorkerThread, 149
KiExecuteDpc, 157
wątki systemowe, 129, 147
wątkowe procedury DPC, 156
WBAC, White-Box Attack Context, 295
WDF, Windows Driver Foundation, 164
WDM, Windows Driver Model, 164
wektor wyjątku, 89
wersje trybu Thumb, 62
wiązanie kodu, 310
Windows
inicjalizacja procesora, 109
IRQL, 124
konteksty wykonywania, 128
listy deskryptorów pamięci, 126
procesy, 126
pule pamięci, 125
rozkład pamięci, 108
synchronizacja jądra, 129
wątki, 126
wywołania systemowe, 111
wirtualizacja kodu, 322
właściwości wirtualnej czarnej skrzynki, 295
wskaźnik
CurrentStackLocation, 192
do katalogu tabel stron, PDPT, 45
DeviceExtension, 167
DpcStack, 156
DRIVER_OBJECT, 139
IO_STACK_LOCATION, 179
IO_WORKITEM, 148
KdVersionBlock, 139, 140, 142
lpExtensionApis, 273
PsLoadedModuleList, 140
ramki, 36
stosu, 36
wskaźniki obiektów, 93
wstawianie zbędnego kodu, 293
wstrzymanie wykonywania programu, 212
wtyczka Flash, 294
wyjątek, 47, 114, 212
błędu strony, 114
KiSWIException, 122
wykonywanie kodu
ad hoc, 147
asynchroniczne, 147
w trybie concolic, 300, 339
wyłuskiwanie wskaźników, 208
wyświetlanie
selektora, 218
zawartości rejestrów, 217
wywołania
systemowe, 90, 111, 120, 171
zwrotne, 151, 190
dezasemblera, 304
procesów, 159
wątków, 159
wywoływanie funkcji, 32, 57, 77
wzorzec dynamicznego obliczania skoków, 306

Kup książkę Poleć książkę

Z

zabezpieczanie przez zaciemnianie, 297
zaciemnianie
kodu, 277, 279, 322, 324
oparte na danych, 282
oparte na sterowaniu, 287
oparte na wzorcu, 285
przepływu danych, 293
przepływu sterowania, 293
zależności między strukturami, 96
zapisywanie danych, 67
zapobieganie wykonywaniu danych, DEP, 199
zarządzanie
pamięcią, 45
prawami cyfrowymi, DRM, 277
procesami, 210
rejestrami, 214
zastosowanie
APC, 153
DPC, 155
instrukcji LDR, 71
list, 138
muteksu, 130
narzędzia Metasm, 324
procedur zakończenia, 162
programu VxStripper, 332
timerów, 159
zaślepka wywołań systemowych, 171
zawieszenie wątku, 153
zdarzenia, 129, 213, 214
zdarzenie SuspendEvent, 153
zestawianie wyrażeń, 207
zmienna środowiskowa _NT_SYMBOL_PATH, 205
znak, 240
$, 25, 239, 253
@, 239, 266
zrzut zawartości pamięci, 220
zrzuty awaryjne, 204
zwalnianie pamięci, 250
zwijanie stałych, 282

Ż

żądanie do sterownika, 129

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

DOWIEDZ SIĘ,
JAK DZIAŁAJĄ PROGRAMY!

Na użytkowników globalnej sieci czekają coraz wymyślniejsze pułapki. Każdego dnia grozi im zarażenie oprogramowaniem szpiegującym, rozsyłającym niechciane wiadomości lub wykorzystującym moc obliczeniową procesora do nieznanych celów. Wykrywanie tego typu zagrożeń i przeciwdziałanie im wymaga dogłębnej analizy niechcianego oprogramowania. Jak to zrobić? Na te i wiele innych pytań odpowiedzi dostarczy ta wspaniała książka!

Dzięki niej zrozumiesz, jak działają procesory x86, x64 oraz ARM, zgłębisz tajniki jądra systemu Windows oraz poznasz najlepsze narzędzia, które wspomogą Cię w Twoich działaniach. W trakcie lektury kolejnych stron dowiesz się, jak korzystać z debuggera, jaką strukturę mają sterowniki oraz czym są pakiety żądań wejścia-wyjścia, a następnie — po co zaciemnia się kod oraz jakie narzędzia są do tego potrzebne. Techniką odwrotną do zaciemniania jest rozjaśnianie kodu. Zastanawiasz się, które narzędzia są skuteczniejsze? Przekonaj się sam! Ta pasjonująca lektura dostarczy Ci mnóstwo wiedzy na temat działania oprogramowania.

Dzięki tej książce:

- **poznasz architekturę procesorów x86, x64 oraz ARM**
- **zaznajomisz się z działaniem jądra systemu Windows**
- **poznasz najlepsze praktyki debugowania programów**
- **nauczysz się zaciemniać i rozjaśniać kod**
- **prześledzisz sposób działania aplikacji**

Bruce Dang — starszy inżynier ds. bezpieczeństwa oprogramowania w firmie Microsoft. Pracuje dla giganta z Redmond od ponad 9 lat, dba o bezpieczeństwo użytkowników najpopularniejszego systemu operacyjnego. Swój czas wolny poświęca zgłębianiu zagadnień związanych z bezpieczeństwem systemów.

Alexandre Gazet — starszy specjalista ds. bezpieczeństwa w QuarksLab. W kręgu jego zainteresowań znajdują się inżynieria odwrotna, bezpieczeństwo systemów oraz dynamiczna ingerencja w wykonywany kod binarny. Prowadzi badania związane z rozjaśnianiem kodu.

Elias Bachaalany — inżynier ds. bezpieczeństwa oprogramowania w firmie Microsoft. Rozwija wewnętrzne narzędzia, bada błędy związane z bezpieczeństwem oraz opracowuje poprawki. Wniósł duży wkład w rozwój EMET 3.0 i EMET 3.5. Jest autorem narzędzia PyHiew oraz artykułów związanych z branżą IT.

Helion
33596 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

WILEY

ISBN 978-83-283-0678-3

cena: 69,00 zł