مهندسی معکوس

برای مبتدیان

دنیس یوریچو

برگردان: مهندس محسن مصطفی جوکار

انتشارات پندار پارس

سرشناسه	:	یوریچو، دنیس Yurichev, Dennis
عنوان و نام پدیدآور	:	مهندسی معکوس برای مبتدیان/ دنیس یوریچو ؛ برگردان محسن مصطفی‌جوکار.
مشخصات نشر	:	تهران : پندار پارس ، 1395.
مشخصات ظاهری	:	1266 ص.: مصور، جدول.
شابک	:	978-600-8201-24-3: 850000 ریال
وضعیت فهرست نویسی	:	فیپا
یادداشت	:	عنوان اصلی:Reverse engineering for beginners
موضوع	:	مهندسی معکوس
موضوع	:	Reverse engineering
شناسه افزوده	:	جوکار، محسن‌مصطفی، 1365 - ، مترجم
رده بندی کنگره	:	1395TA168/5 9م9ی/
رده بندی دیویی	:	620/00420285
شماره کتابشناسی ملی	:	4473995

انتشارات پندارپارس

دفتر فروش: انقلاب، ابتدای کارگرجنوبی، کوی رشتچی، شماره 14، واحد 16 **www.pendarepars.com**

تلفن: 66572335 - تلفکس: 66926578همراه: 09122452348 **info@pendarepars.com**

نام کتاب	:	**مهندسی معکوس برای مبتدیان**
ناشر	:	**انتشارات پندار پارس**
تالیف	:	**دنیس یوریچو**
برگردان	:	**محسن مصطفی جوکار**
چاپ نخست	:	**آذر ماه 95**
شمارگان	:	**500 نسخه**
طرح جلد	:	**رامین شکرالهی**
چاپ، صحافی	:	**روز**
قیمت	:	**85000 تومان** **شابک** : 978-600-6529-24-3

سخن اختصاصی آقای دنیس یوریچو (نویسنده کتاب) درباره ترجمه فارسی کتابش توسط مهندس جوکار در ایران و انتشار آن به وسیله انتشارات پندار پارس:

" I was very happy to work with Iranian publisher **PendarePars**, because of so smooth process. I would recommend them to other technical writers.
I also obligor of **Mohsen Mostafa Jokar**, who was translator to **Persian language**.
Reverse engineering may be critical skill not in very near future, but right now, if we would keep focus on notorious attacks on nuclear facilities and other well-known organizations. As it seems, cyberwar is real and we are fighting right now. It's not possible to analyze malware, backdoors, vulnerabilities and so on without Reverse engineering knowledge. Reverse engineering is also may be applied to more peaceful activities like learning how things are done and how to hack and improve them. I personally started as curios hacker and I got to know a huge part of my knowledge in this way. I hope my humble introduction will help you to start learning it. Like any other skill, one should learn whole lifetime, so this book is only start of never ending journey. Hope, you'll like it!" --
Dennis Yurichev

سخن مترجم

به جرأت می توان گفت که مهندسی معکوس، مهم‌ترین و سخت‌ترین فیلد در امنیت است. به دلیل دانش بالایی که برای یادگیری این تخصص نیاز است، متاسفانه نمی‌توان کتابی از پایه نوشت که تمام مفاهیم را در بر گیرید؛ اما نویسنده اصلی این کتاب تلاش کرده است که از ابتدا این تکنیک را برای شما شرح دهد. نکته مهم در مورد این کتاب این است که رایگان بوده و روز به روز به مفاهیم آن افزوده می‌شود و با کمی جست‌وجو در اینترنت متوجه خواهید شد که نخستین کتابی است که تمام متخصصان امنیت به شما پیشنهاد می‌کنند. نویسنده در این کتاب با انجام آزمایش‌های عملی مانند شکستن قفل دانگل یا دست‌کاری برنامه‌ها و بازی‌ها، این تکنیک را برای شما جالب و یا شاید هم اشتیاق شما را برای یادگیری آن بیشتر کرده است. در این کتاب مثال‌های زیادی را خواهید دید و حتی یک وب‌سایت مجزا هم به تمرین‌های کتاب اختصاص داده شده است. گفتنی است که این کتاب با مجوز رسمی از سوی خود نویسنده ترجمه شده است.

محسن مصطفی جوکار

mohsen1365b@yahoo.com

پاییز 95

فهرست

تقدیم به:

```
            .Ltext0:
                        .section    .rodata
            .LC0:
0000 546F206D               .string     "To my Papa and Mama"
   79205061
   70612061
   6E64204D
   616D6100
                        .text
                        .globl      main
            main:
            .LFB0:
                        .cfi_startproc
0000 55                         pushq       %rbp
                        .cfi_def_cfa_offset 16
                        .cfi_offset 6, -16
0001 4889E5                     movq        %rsp, %rbp
                        .cfi_def_cfa_register 6
0004 BF000000                   movl        $.LC0, %edi
   00
0009 B8000000                   movl        $0, %eax
   00
000e E8000000                   call        printf
   00
0013 90                         nop
0014 5D                         popq        %rbp
                        .cfi_def_cfa 7, 8
0015 C3                         ret
                        .cfi_endproc
            .LFE0:
            .Letext0:
```

پیش‌گفتار

چندین معنی از «مهندسی معکوس» وجود دارد:

1) مهندسی معکوس نرم‌افزار: کاوش کردن برنامه‌های کامپایل شده؛
2) اسکن کردن ساختارهای 3D و دستکاری محصولات دیجیتالی پیشرو برای تکثیر و تولید آنها؛
3) ایجاد دوباره‌ی ساختار (DBMS سیستم مدیریت پایگاه داده).

این کتاب درباره‌ی معنی نخست است. مباحثی که به صورت عمیق بحث خواهد شد:

x86/x64, ARM/ARM64, MIPS, Java/JVM.

مباحثی که به صورت گذرا بررسی خواهد شد:

Oracle RDBM، Itanium، دانگل‌های محافظت از کپی، LD_PRELOAD، سرریز پشته، ELF 10، قالب فایل‌های قابل حمل و اجرایی win32، ساختار x86-64، بخش‌های مهم فراخوان‌های سیستمی، TLS 11، کد مستقل از موقعیت (PIC 12)، بهینه‌سازی پروفایل هدایت شده C++ STL، OpenMP، SEH.

تمرین‌ها

همه‌ی تمرین‌ها در وب‌سایت جداگانه‌ای قرار گرفته است: http://challenges.re

درباره‌ی نویسنده

دنیس پوریچو (Dennis Yurichev) یک برنامه‌نویس و مهندس معکوس باتجربه است. با آدرس ایمیل dennis@yurichev.com یا اسکایپ dennis.yurichev می‌توان با او ارتباط برقرار کرد.

قدردانی نویسنده

برای صبر و حوصله در پاسخ دادن به تمام پرسش‌های من: آندری "herm1t" برانوویچ و اسلاوا "Avid" کازیکو.

برای ارسال یادداشت‌هایی درباره‌ی اشتباه‌ها و بی‌دقتی‌های من: استانیسلاو "Beaver" بابریسکای، الکساندر لیسنکو، شل راکت، ژو رویجین و چنگمین هور.

برای کمک‌های گوناگون دیگر به من: اندرو زوبینسکی، آرنود پاتاد (با نام مستعار rtp در کانال #debian-arm در شبکه) IRC، الیاکساندر اوتای.

برای برگردان کتاب به زبان چینی ساده: شی آن کای.

برای برگردان کتاب به زبان کره‌ای: بون‌های مین.

برای ویراستاری: الکساندر "Lstar" چرننکی، ولادیمیر باتاو، آندری بریزهیک، مارک "Logxen" کوپر، یوآن جوکن کانگ، مال مالاکاو، لوئیس پورتر، جارله تورسن.

"وسیل کویلو" کار بسیاری در زمینه‌ی غلط‌گیری و تصحیح اشتباه‌های فراوان من انجام داد.

برای تصاویر و طرح جلد: اندی نیچایوسکی.

همچنین با سپاس از تمامی کاربران در github.com که در یادداشت‌ها و اصلاحات به من کمک کردند.

از بسته‌های LATEX بسیار استفاده کردم، از سازندگان آن سپاس‌گزارم.

کمک مالی

همان‌گونه که می‌دانید، نگارش به صورت فنی، زمان و تلاش بسیاری نیاز دارد. این کتاب رایگان است و به صورت آزادانه و به شکل کد منبع (LaTeX) در دسترس است و باید برای همیشه به همین صورت باقی بماند. همچنین این کتاب هیچ تبلیغی ندارد. طرح کنونی من برای این کتاب، افزودن اطلاعات بسیاری در موارد زیر است:

- Objective-C
- Visual Basic
- ترفندهای ضد اشکال‌زدایی
- Windows NT اشکال‌زدایی هسته
- NET
- Oracle RDBMS

اگر می‌خواهید درباره‌ی تمام این موضوع‌ها به نوشتن ادامه دهم، کمک مالی شما می‌تواند یاری‌گر من باشد.

راه‌های اهدای کمک، در آدرس beginners.re آمده است. نام همه‌ی کمک‌کننده‌ها در این کتاب گنجانده خواهد شد. واقعیت این است که کمک‌کنندگان مالی، حق دارند از من بخواهند موارد موجود در طرحم را برای نوشتن دوباره بررسی کنم.

نام کمک‌کنندگان مالی

25 * anonymous, 2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($127), Richard S Shultz ($20), Jang Minchang ($4.5), Luis Rocha ($63), Joris van de Vis ($20), ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (e10+50), Sergey Lukianov (300 RUR),Ludvig Gislason (200 SEK), Gn Ahn ($50), Triop AB (100 SEK), ($4), Martin Haeberli ($10), Ange Albertini (e10+50), Sergey Lukianov (), Philippe Teuwen Victor Cazacov (e5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), B.V. (e25), Joona Oskari Heikkilzacov (e5), Tobias Sturzenegger (10 CHF), Sonny Redfive ($25), Vladimir Dikovski (e50), Jiarui Hong Thai ($15), Bayna AlZaabi ($exandre Borges Pillay (100.00 SEK), Jim_Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (e10), Salikov Alexander (500 (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian RUR), Oliver Whitehouse NOK), Vitaly Osipov ($100), Aguilera (e20), Hans-Martin Malikov Alexander (500 RUR), Ol .Yuri Romanov (1000 RUR), Aliaksandr Autayeu (e10), Tudor Azoitei ($40), Z0vsky (e10)

پرسش و پاسخ کوتاه

پرسش: چرا باید امروزه زبان اسمبلی را فرا بگیریم؟

پاسخ: تنها شاید در صورتی نیاز به کدنویسی به زبان اسمبلی نداشته باشید که توسعه‌دهنده‌ی سیستم عامل باشید. کامپایلرهای مدرن، بهینه‌سازی را نسبت به انسان، بسیار بهتر انجام می‌دهند. همچنین پردازنده‌های امروزی، دستگاه‌های بسیار پیچیده‌ای هستند و دانش اسمبلی، برای درک اجزای داخلی آنها، کمکی نمی‌کند. می‌توان گفت دست‌کم دو حوزه وجود دارد که درک خوب از زبان اسمبلی می‌تواند در آن مفید باشد: نخستین و مهم‌ترین آن، پژوهش در زمینه‌ی امنیت/نرم‌افزارهای مخرب است. همچنین یک راه خوب در هنگام اشکال‌زدایی، برای به دست آوردن درک بهتر از کدهای کامپایل شده نیز همین است. **در نتیجه، این کتاب برای کسانی است که می‌خواهند زبان اسمبلی را درک کنند، نه کدنویسی در آن را، و به همین دلیل، نمونه‌های بسیاری از خروجی کامپایلر، در این کتاب آمده است.**

پرسش: بر روی یک لینک در سند PDF کلیک کردم، چگونه می‌توانم به عقب برگردم؟

پاسخ : در Adobe Acrobat Reader از کلیدهای Alt+Left Arrow استفاده کنید.

پرسش: کتاب شما حجیم است! آیا نمونه‌ی مختصری از آن وجود دارد؟

پاسخ: نسخه‌ی سبک‌تر آن (کتاب زبان اصلی)، در آدرس http://beginners.re/#lite وجود دارد.

پرسش: اگر برای یادگیری مهندسی معکوس تلاش کنم، مطمئن نیستم بتوانم آن را یاد بگیرم؟

پاسخ: شاید زمان متوسط برای آشنا شدن با محتویات نسخه‌ی کوتاه، یک یا دو ماه باشد.

پرسش: آیا می‌توانم این کتاب را چاپ کنم؟ آیا می‌توانم از آن برای آموزش استفاده کنم؟

پاسخ: البته! (منظور، نسخه زبان اصلی کتاب است نه ترجمه فارسی). به همین دلیل است که این کتاب با مجوز Creative Commons است. همچنین در صورت تمایل، می‌توانید نسخه‌ی مربوط به خود را بسازید. اطلاعات بیشتر، در این آدرس است:

https://github.com/dennis714/RE-for-beginners/blob/master/HACKING.md

پرسش: آیا می‌توانم کتاب شما را به زبان‌های دیگر برگردان کنم؟

پاسخ: به آدرس https://github.com/dennis714/RE-for-beginners/blob/master/Translation.md بروید.

پرسش: چگونه می‌توانم یک شغل مرتبط با مهندسی معکوس پیدا کنم؟

پاسخ: گاهی در reddit موضوعاتی در رابطه با استخدام، به مهندسی معکوس اختصاص داده شده است (آدرس‌های http://go.yurichev.com/17333 و http://go.yurichev.com/17334 را ببینید) تلاش کنید نگاهی به آنها بیاندازید. موضوعات تا حدودی مرتبط با استخدام را می‌توان در netsec مربوط به reddit یافت:

http://go.yurichev.com/17335

پرسش: من یک پرسش دارم؟

پاسخ: آن را به‌وسیله‌ی آدرس ایمیل dennis@yurichev.com برایم بفرستید یا پرسش خود را در فرم وب سایت من مطرح کنید:

forum.yurichev.com

تعریف و تمجید از کتاب مهندسی معکوس برای مبتدیان

- "این کتاب بسیار خوب نوشته شده و رایگان است... شگفت‌انگیز است" . دنیل بیلار از Siege Technologies, LLC.
- "عالی و رایگان" . پیت فینیگن متخصص امنیت Oracle RDBMS.

- ''این کتاب جالب، کار بزرگی است''. مایکل سیکورسکی نویسنده‌ی کتاب زیر:

Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.

- ''آفرین برای آموزش بسیار خوب...''. هربرت باس، استاد تمام در دانشگاه Vrije آمستردام و یکی از نویسندگان کتاب Modern Operating Systems (4th Edition) .
- ''این کتاب، شگفت‌انگیز و باورنکردنی است''. لوئیس روچا دارنده‌ی مدرک CISSP / ISSAP، مدیر فنی، شبکه و امنیت اطلاعات در شرکت Verizon.
- ''سپاس برای این کار بزرگ''. جوریس ون دی ویس، SAP Netweaver و متخصص امنیت.
- ''مقدمه‌ای معقول و منطقی، برای برخی تکنیک‌های ارائه شده''. مایک استی، معلم در مرکز آموزش اجرای قانون فدرال در ایالت جورجیا.
- ''این کتاب را دوست دارم! برخی از دانشجویانم آن را خوانده‌اند و برای استفاده از آن در دوره‌ی فوق لیسانس، برنامه‌ریزی کرده‌اند''. سرگی براتس، استادیار تحقیقات در دپارتمان علوم کامپیوتر در کالج دارتموث.
- ''دنیس یوریچو یک کتاب قابل توجه (و رایگان)! درباره‌ی مهندسی معکوس منتشر کرده است''. تانل پادر، متخصص تنظیم عمل‌کرد Oracle RDBMS.

درباره‌ی برگردان کره‌ای

در ژانویه‌ی سال 2015، شرکت چاپ و نشر (Acorn بلوط (www.acornpub.co.kr)) در کره‌ی جنوبی، برای برگردان و چاپ کتاب من به زبان کره‌ای، کار بزرگی انجام داد. این کتاب هم‌اکنون در وب‌سایت آنها (http://www.acornpub.co.kr/book/reversing-for-beginners) موجود است. مترجم این کتاب Byungho Min است (https://twitter.com/tais9). جلد کتاب را دوست هنرمندم Andy Nechaevsky طراحی کرد (https://www.facebook.com/andydinka). ناشر کره‌ای حق تکثیر برگردان کره‌ای را برای خود محفوظ نگه‌داشته است. بنابراین اگر می‌خواهید یک کتاب واقعی به زبان کره‌ای در کتابخانه‌ی خود داشته باشید و از کار من نیز حمایت کنید، این کتاب اینک برای خرید، در دسترس است.

بخش نخست

الگوهای کد

هنگامی که نویسنده‌ی این کتاب، برای نخستین بار، یادگیری زبان C و پس از آن C++ را آغاز کرد، کد کوچکی را نوشت و آن را کامپایل کرد. سپس به خروجی زبان اسمبلی نگاه کرد. این کار باعث شد درک کدی که او نوشته است، برایش بسیار آسان شود. او این کار را، چندین بار انجام داد تا ارتباط میان کدهای C\C++ و آنچه کامپایلر تولید می‌کند، به صورت عمیق در ذهنش حک شود. تصور فوری و کلی از محتوای کدهای C و کارکرد آنها، برایش آسان بود. شاید این روش، بتواند برای افراد دیگر نیز مفید باشد.

در اینجا، گاهی برای رسیدن به کوتاه‌ترین (یا ساده‌ترین) کد ممکن، از کامپایلرهای قدیمی استفاده می‌شود.

هنگامی که نویسنده‌ی این کتاب، زبان اسمبلی را مطالعه می‌کرد، اغلب توابع کوچک به زبان C را کامپایل و سپس آنها را به‌تدریج به زبان اسمبلی، بازنویسی می‌کرد. او تلاش می‌کرد، کدهای خود را تا آنجا که شدنی است، کوتاه کند. شاید امروزه، در دنیای واقعی این کار ارزش انجام نداشته باشد، چون از نظر بهره‌وری، رقابت آن با کامپایلرهای مدرن دشوار است. هرچند، این یک راه بسیار خوب، برای به دست آوردن درک بهتری از زبان اسمبلی است. بنابراین، به آسانی می‌توانید هر کدی را که می‌خواهید، از این کتاب بردارید و تلاش کنید که آن را کوتاه‌تر کنید. اما آزمایش آنچه را که نوشته‌اید، فراموش نکنید.

سطوح بهینه‌سازی و اشکال‌زدایی اطلاعات

کد منبع (Source code) را می‌توان با کامپایلرهای متفاوت و سطوح گوناگون بهینه‌سازی، کامپایل کرد. معمولا کامپایلرها، سه سطح دارند که سطح صفر به معنی غیرفعال کردن بهینه‌سازی است. همچنین بهینه‌سازی می‌تواند حجم کد یا سرعت کد را هدف قرار دهد.

کامپایلر بهینه‌سازی نشده، سریع‌تر است و کدهای قابل فهم‌تری (البته طولانی) تولید می‌کند. اما یک کامپایلر بهینه‌سازی شده، کندتر است و تلاش می‌کند کدهایی تولید کند که سریع‌تر اجرا شوند (اما حتما فشرده‌تر نیست).

افزون بر سطوح بهینه‌سازی و هدایت، کامپایلر می‌تواند دربرگیرنده‌ی برخی از اطلاعات اشکال‌زدایی در فایل خروجی باشد که در نتیجه، کد تولیدشده را برای اشکال‌زدایی آسان می‌کند.

یکی از ویژگی‌های مهم اشکال‌زدایی کد، شامل ارتباط میان هر خط از کد منبع و آدرس کد ماشین مربوطه است. از سوی دیگر، بهینه‌سازی کامپایلرها، تمایل به تولید خروجی دارد که در آن همه‌ی خطوط کد منبع، می‌توانند به طور پیوسته بهینه‌سازی شوند و حتی کد ماشین نیز در خروجی حاصل از آن، وجود نداشته باشد.

مهندسان معکوس می‌توانند با هر دو نسخه روبه‌رو شوند، زیرا برخی از توسعه‌دهندگان، پرچم‌های مربوط به بهینه سازی کامپایلرها را روشن می‌کنند و برخی دیگر این کار را نمی‌کنند. به همین دلیل، تلاش خواهیم کرد در نمونه‌های موجود در این کتاب، تا جایی که شدنی است، هم بر روی نسخه‌ی اشکال‌زدایی و هم بر روی نسخه‌ی آزاد کار کنیم.

فصل 1

مقدمه‌ای کوتاه برای CPU

CPU دستگاهی است که کد ماشین یک برنامه‌ی دربردارنده‌ی آن کد را اجرا می‌کند.

واژه‌نامه‌ی کوتاه

دستور (**Instruction**): یک دستور آغازین CPU است. ساده‌ترین نمونه‌ها عبارت‌اند از: انتقال داده میان ثبات‌ها، کار با حافظه و محاسبات آغازین. به‌عنوان یک قانون، هر CPU مجموعه دستورهای معماری (ISA) مربوط به خود را دارد.

کد ماشین (**Machine code**): کدی که CPU به‌طور مستقیم آن را پردازش می‌کند. هر دستور معمولا با چند بایت، کدگذاری می‌شود.

زبان اسمبلی (**Assembly language**): کدهای مربوط به حافظه و برخی از الحاقات مانند ماکروها که برای آسان‌تر کردن کار برنامه‌نویس، در نظر گرفته شده‌اند.

ثبات **CPU (CPU register)**: هر پردازنده، مجموعه‌ای ثابت از ثبات‌های همه منظوره (GPR) را دارد. در معماری x86 شمار آن 8 عدد، در معماری x86-64 شمار آن 16 عدد و در معماری ARM نیز، شمار آن 16 عدد است. ساده‌ترین راه برای درک یک ثبات این است که به عنوان یک متغیر موقت و بدون نوع، به آن بنگریم. تصور کنید که در حال کار با یک زبان برنامه‌نویسی سطح بالا هستید و تنها می‌توانید از هشت متغیر 32 بیتی (یا 64 بیتی) استفاده کنید. با این‌حال می‌توان با استفاده از آنها، کارهای بسیاری را انجام داد!

شاید تعجب کنید که چرا باید میان کد ماشین و یک زبان برنامه‌نویسی تفاوت وجود داشته باشد؟ پاسخ در این واقعیت نهفته است که انسان‌ها و پردازنده‌ها یکسان نیستند. برای انسان‌ها، استفاده از یک زبان برنامه‌نویسی سطح بالا همانند C/C++، Java، Python و غیره ساده‌تر است، اما برای یک CPU استفاده از یک زبان با سطح بسیار پایین، آسان‌تر است. شاید اختراع یک CPU که بتواند کدهای زبان برنامه‌نویسی سطح بالا را اجرا کند، شدنی باشد اما این امر می‌تواند پیچیده‌تر از پردازنده‌هایی که ما امروزه می‌شناسیم، باشد. در یک روش مشابه، برای انسان‌ها، برنامه‌نویسی به زبان اسمبلی چون بسیار سطح پایینی دارد و دشوار است، بدون اشتباه‌های

آزاردهنده‌ی فراوان، ناخوشایند است. برنامه‌ای که کدهای نوشته شده به زبان برنامه‌نویسی سطح بالا را به اسمبلی تبدیل می‌کند، کامپایلر نامیده می‌شود.

1-1 چند واژه درباره‌ی ISAهای گوناگون

همیشه x86 ISA یک آپ‌کد (opcode یا کد عمل‌کرد) با طول متغیر دارد، بنابراین هنگامی که وارد عصر پردازنده‌های 64 بیتی شدیم، پسوند x64 تأثیر بسیار چشم‌گیری بر ISA نگذاشت. در حقیقت، x86 ISA هنوز دربردارنده‌ی مقدار بسیاری از دستورها بود که برای نخستین بار در پردازنده‌های 16 بیتی 8086 آشکار شده بودند و هنوز هم در پردازنده‌های امروزی وجود دارند. ARM یک پردازنده بر اساس معماری کم دستور (RISC) است که با در نظر گرفتن آپ‌کد با طول ثابت طراحی شده است و در گذشته مزایایی داشت. در آغاز، تمام دستورهای ARM در 4 بایت کدگذاری می‌شدند. امروزه "حالت ARM" نامیده می‌شود.

سپس فهمیدند به اندازه‌ای که آنها در آغاز تصور می‌کردند، به صرفه نیست. در حقیقت، بیشتر دستورهای پردازنده را که برنامه‌های کاربردی از آن استفاده می‌کنند، می‌توان با استفاده از اطلاعات کمتر، رمزگذاری کرد. بنابراین آنها ISA دیگری را افزودند که Thumb نامیده می‌شد و در آن، هر دستور تنها در 2 بایت رمزگذاری می‌شد. اکنون "حالت Thumb" نامیده می‌شود. هر چند، تمام دستورهای ARM را نمی‌توان تنها در دو بایت رمزگذاری کرد و مجموعه دستورهای Thumb تا حدودی محدود است. گفتنی است، کد کامپایل‌شده برای حالت ARM و حالت Thumb شاید هم‌زمان در یک برنامه وجود داشته باشد.

سازندگان ARM فکر می‌کردند که Thumb می‌تواند توسعه داده شود و به همین دلیل Thumb-2 به وجود آمد و در ARMv7 آشکار شد. Thumb-2 هنوز از دستورهای 2 بایتی استفاده می‌کرد، اما برخی از دستورهای جدید، اندازه‌ی 4 بایتی داشتند. یک تصور غلط و رایج این است که Thumb-2 ترکیبی از ARM و Thumb است. این اشتباه است؛ بلکه Thumb-2 به‌طور کامل از تمام ویژگی‌های پردازنده پشتیبانی می‌کند و می‌تواند با حالت ARM رقابت کند. یک هدف آشکار این است که بیشتر برنامه‌های کاربردی برای iPod/iPhone/iPad با مجموعه دستورهای Thumb-2 کامپایل شده‌اند (به دلیل این واقعیت که Xcode به‌طور پیش‌فرض این کار را انجام می‌دهد). بعدها نسخه‌ی 64 بیتی ARM عرضه شد. این ISA آپ‌کدهای 4 بایتی دارد و نیاز به هیچ حالت Thumb اضافی ندارد. هر چند، نیازهای 64 بیتی، ISA را تحت تأثیر قرار می‌دهد و باعث شده است که اکنون ما، سه مجموعه دستور ARM داشته باشیم: حالت ARM، حالت Thumb (از جمله Thumb-2) و ARM64. این ISAها تا حدودی همدیگر را پوشش می‌دهند، اما می‌توان گفت به جای اینکه انواع گوناگونی از یک مدل باشند، ISAهای گوناگونی هستند. بنابراین باید تلاش کنیم قطعاتی از کد را برای هر سه ARM ISA در کتاب بیافزاییم.

هر چند، بسیاری از ISAها از نوع RISC و با طول ثابت آپ‌کد 32 بیتی وجود دارند؛ مانند PowerPC، MIPS و Alpha AXP.

فصل 2

ساده‌ترین تابع

ساده‌ترین تابع ممکن، تابعی است که تنها یک مقدار ثابت را برگرداند. به نمونه‌ی زیر توجه کنید:

مثال 2-1: کد C\C++

```
int f()
{
        return 123;
};
```

اجازه دهید آن را کامپایل کنیم!

2-1 x86

در اینجا خروجی تولیدشده بر روی پلتفرم x86 را برای هر دو کامپایلر بهینه‌سازی شده GCC و MSVC می‌بینیم:

مثال 2-2: GCC/MSVC بهینه‌سازی شده (خروجی اسمبلی)

```
f:
        mov     eax, 123
        ret
```

تنها دو دستور در اینجا وجود دارد: خط نخست مقدار 123 را به داخل ثبات EAX منتقل می‌کند که طبق قرارداد، برای ذخیره‌سازی مقدار بازگشتی استفاده می‌شود و خط دوم RET است که برگشت به فراخوان را انجام می‌دهد.

2-2 ARM

تفاوت‌های کمی در پلتفرم ARM وجود دارد:

مثال 2-3: خروجی اسمبلی بهینه‌سازی شده به‌وسیله‌ی محیط توسعه Keil 6/2013 (حالت ARM)

```
f PROC
        MOV     r0,#0x7b ; 123
        BX      lr
        ENDP
```

ARM از ثبات R0 برای برگرداندن نتیجه‌ی تابع استفاده می‌کند، بنابراین 123 در درون R0 کپی شده است. آدرس برگشتی بر روی پشته محلی در ARM ISA ذخیره نشده، بلکه در ثبات پیوند ذخیره شده است. بنابراین دستور BX LR باعث اجرای پرش به آن آدرس می‌شود- به‌طور کارآمد اجرا را به فراخوان برمی‌گرداند.

گفتنی است که MOV یک نام گمراه‌کننده برای دستور در ISAهای x86 و ARM است. در حقیقت، داده منتقل نمی‌شود، بلکه کپی می‌شود.

MIPS 2-3

هنگام نام‌گذاری ثبات‌ها در دنیای MIPS، از دو قاعده نام‌گذاری استفاده می‌شود: با عدد (از $0 تا $31) یا با شبه نام ($A0, $V0 و غیره). خروجی اسمبلی GCC در زیر، ثبات‌ها را با عدد فهرست کرده است:

مثال 2-4: GCC 4.4.5 بهینه‌سازی شده (خروجی اسمبلی)

```
        j       $31
        li      $2,123                  # 0x7b
```

در حالی که IDA (Interactive Disassembler and debugger) این کار را به‌وسیله‌ی شبه نام انجام می‌دهد:

مثال 2-5: GCC 4.4.5 بهینه‌سازی شده (IDA)

```
jr      $ra
li      $v0, 0x7B
```

ثبات $2 (یا $V0) برای ذخیره‌ی مقدار بازگشتی تابع استفاده می‌شود. LI مخفف "بارگذاری فوری"و معادل MIPS آن MOV است. دستورهای دیگر، دستورهای پرش هستند (J یا JR) که جریان اجرا را به فراخوان برمی‌گردانند و به آدرس موجود در ثبات $31 (یا $RA) پرش می‌کنند. این ثبات همانند LR در ARM است. شاید تعجب کنید که چرا موقعیت دستور بارگذاری (LI) و دستور پرش (J یا JR) جابه‌جا می‌شوند؟ این به خاطر یکی از ویژگی‌های RISC است که "شاخه اسلات دیرکرد" نامیده می‌شود. delay slot یک اسلات دستور است که بدون اثر گذاشتن روی دستور پیشین اجرا می‌شود. چرایی آن، تغییر ناگهانی در معماری برخی از ISAهای RISC است و این برای هدف ما مهم نیست. تنها در MIPS باید به خاطر داشته باشیم که دستوری که از یک دستور پرش یا شاخه (brunch) پیروی می‌کند، پیش از خود دستو پرش/شاخه اجرا می‌شود. در نتیجه، دستور شاخه همیشه محل را با دستوری که باید پیشاپیش اجرا شود عوض می‌کند.

2-3-1 یک نکته درباره‌ی نام‌های دستور/ثبات MIPS

در دنیای MIPS نام‌های دستورها و ثبات‌ها به‌طور سنتی با حروف کوچک نوشته می‌شوند. به خاطر انسجام، از حروف بزرگ استفاده نمی‌کنیم و این به‌عنوان قراردادی است که به‌وسیله‌ی تمام ISAهای دیگر در این کتاب دنبال می‌شود.

فصل 3

سلام دنیا!

اجازه دهید از یک نمونه‌ی معروف در کتاب "زبان برنامه‌نویسی C" استفاده کنیم:

```
#include <stdio.h>

int main ()
{
     printf("hello, world\n");
     return 0;
}
```

3-1 x86

3-1-1 MSVC

اجازه دهید آن را در MSVC 2010 کامپایل کنیم:

```
C1 1.cpp /Fa1.asm
```

(گزینه‌ی Fa/ به کامپایلر دستور می‌دهد که فایل نمونه‌ی اسمبلی را تولید کند).

مثال 3-1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
 ;Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC فایل نمونه را به صورت ترکیب نحوی اینتل تولید می‌کند. تفاوت میان ترکیب نحوی اینتل و ترکیب نحوی AT&T در بخش (3-1-3) بحث خواهد شد.

کامپایلر فایل 1.obj را تولید می‌کند و به فایل 1.exe متصل می‌شود. درباره‌ی ما، فایل شامل دو بخش است: CONST (برای داده‌های ثابت) و TEXT_ (برای کد).

رشته‌ی hello, world در زبان C\C++ دارای نوع const char[] است اما نام مربوط به خود را ندارد. کامپایلر نیاز به سروکار داشتن با رشته دارد، به‌گونه‌ای که نام داخلی $SG3830 را برای آن تعریف می‌کند.

به همین دلیل، نمونه به صورت زیر بازنویسی می‌شود:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n;"

int main ()
{
     printf($SG3830);
     return 0;
}
```

اجازه دهید به نمونه‌ی اسمبلی برگردیم. همان‌گونه که می‌بینیم، رشته با یک بایت صفر پایان یافته است و این استانداردی برای رشته‌ها در زبان C\C++ است. در بخش (1-1-57) درباره‌ی رشته‌ها در زبان C بیشتر یاد خواهیم گرفت.

در بخش کد که TEXT_ است، تاکنون تنها یک تابع وجود دارد: ()main. تابع ()main با کد آغاز و با کد پایان می‌یابد (تقریبا مانند هر تابع دیگر).

پس از مقدمه‌ی تابع، فراخوانی تابع ()printf را می‌بینیم: CALL _printf. پیش از فراخوانی آدرس رشته (یا یک اشاره‌گر به آن) که دربردارنده‌ی پیام تبریک است، آدرس با کمک دستور PUSH بر روی پشته جای گرفته است.

هنگامی که تابع ()printf کنترل را به تابع ()main برمی‌گرداند، آدرس رشته (یا یک اشاره‌گر به آن) هنوز در پشته جای دارد. از آنجا که دیگر به آن نیازی نداریم، اشاره‌گر پشته (ثباتESP) باید اصلاح شود ADD ESP, 4. به معنی این است که 4 را به مقدار ثبات ESP اضافه کن. اما چرا 4؟ از آنجا که این یک برنامه‌ی 32 بیتی است، دقیقا باید 4 بایت را برای آدرس به پشته رد کنیم. اگر این یک کد x64 بود، آنگاه نیاز به 8 بایت داشتیم. در واقع ADD ESP, 4 به‌طور غیررسمی برابر با ثبات POP است، اما بدون استفاده از هرگونه ثبات.

برخی از کامپایلرها (مانند Intel C++ Compiler) شاید برای همان هدف، POP ECX را به جای ADD تولید کنند (برای نمونه، چنین الگویی می‌تواند در کد Oracle RDBMS که با کامپایلر Intel C++ کامپایل شده است، دیده شود). این دستور تقریبا همان اثر را دارد، اما محتویات ثبات ECX بازنویسی خواهد شد. از آنجا که آپ‌کد این

دستور کوتاه‌تر از ADD ESP, x (1 بایت برای POP در برابر 3 بایت برای ADD) است، کامپایلر Intel C++ از POP ECX استفاده می‌کند.

در اینجا یک نمونه از استفاده‌ی POP به جای ADD در Oracle RDBMS وجود دارد:

مثال 2-3: Oracle RDBMS 10.2 Linux (فایل app.o)

```
.text:0800029A                push    ebx
.text:0800029B                call    qksfroChild
.text:080002A0                pop     ecx
```

پس از فراخوانی printf()، کد اصلی C\C++ شامل return 0 است. return 0 به عنوان نتیجه‌ای برای تابع main() است. در کد تولیدشده، این با دستورهای XOR EAX, EAX پیاده‌سازی شده است. در واقع XOR تنها "OR یکتا" است، اما کامپایلرها اغلب از آن به جای MOV EAX, 0 استفاده می‌کنند. یکبار دیگر، این به دلیل آپ کد کمی کوتاه‌تر است (2 بایت برای XOR در برابر 5 بایت برایMOV).

برخی از کامپایلرها SUB EAX, EAX را منتشر کرده‌اند که به معنی کم کردن مقدار موجود در EAX از مقدار موجود در EAX است و در هر صورت نتیجه‌ی آن صفر است.

آخرین دستور RET است و کنترل را به فراخوان برمی‌گرداند. معمولا این کتابخانه‌ی زمان اجرای C/C++ است که به نوبه‌ی خود، کنترل را به سیستم عامل برمی‌گرداند.

GCC 3-1-2

اینک اجازه دهید همان کد به زبان C\C++ را در کامپایلر GCC 4.4.1 در لینوکس بررسی کنیم: gcc 1.c -o 1 و در گام بعد اجازه دهید با کمکIDA disassembler ببینیم که چگونه تابع main() ایجاد شده است. IDA همانند MSVC از ترکیب نحوی اینتل استفاده می‌کند.

مثال 3-3: کد در IDA

```
main            proc near

var_10          = dword ptr -10h

                    push    ebp
                    mov     ebp, esp
                    and     esp, 0FFFFFFF0h
                    sub     esp, 10h
                    mov     eax, offset aHelloWorld ; "hello, world\n"
                    mov     [esp+10h+var_10], eax
                    call    _printf
                    mov     eax, 0
                    leave
                    retn
main            endp
```

نتیجه تقریبا یکسان است. آدرس رشته hello, world (ذخیره شده در بخش داده) نخست در ثبات EAX بارگذاری شده و سپس در بالای پشته ذخیره شده است. افزون بر این، مقدمه‌ی تابع شامل AND ESP, 0FFFFFFF0h است. این دستور مقدار ثبات ESP را در یک مرز 16 بایتی جای می‌دهد. این نتایج در تمام مقادیر موجود در پشته، به همین شیوه هم‌تراز شده است (اگر مقادیری که با آن سروکار داریم، در آدرسی از حافظه قرار داشته باشند که در یک مرز 4 یا 16 بایتی جای گرفته باشند، CPU بهتر انجام وظیفه می‌کند).

SUB ESP, 10h مقدار 16 بایت در پشته را به خود اختصاص داده است. اگرچه همان‌گونه که می‌بینیم، تنها 4 بایت در اینجا نیاز است. به این دلیل است که اندازه پشته اختصاص داده شده نیز، در مرز 16 بایتی جای گرفته است. سپس آدرس رشته (یا یک اشاره‌گر به رشته) بدون استفاده از دستور PUSH به‌طور مستقیم در بالای پشته ذخیره شده است. Var_10 – یک متغیر محلی و همچنین یک آرگومان برای ()printf است. درباره‌ی آن در زیر مطالعه کنید. سپس تابع ()printf فراخوانی می‌شود.

برخلاف MSVC، هنگامی که GCC بدون بهینه‌سازی فعال شده کدی را کامپایل می‌کند، آنگاه به جای یک آپ کد کوتاه‌تر، MOV EAX, 0 را منتشر می‌کند.

آخرین دستور LEAVE است که معادل هر دو دستور MOV ESP, EBP و POP EBP است. به دیگر سخن، این دستور اشاره‌گر پشته (ESP) را به عقب برگردانده و ثبات EBP را به حالت آغازین آن بازیابی می‌کند. از آنجا که لازم داریم مقادیر این ثبات‌ها (ESP وEBP) را در آغاز تابع تغییر دهیم (با اجرای MOV EBP, ESP و AND ESP, 0FFFFFFF0h) این کار لازم است.

3-1-3 GCC: ترکیب نحوی AT&T

اجازه دهید ببینیم چگونه می‌توان این را در زبان اسمبلی، با ترکیب نحوی AT&T بیان کرد. این ترکیب نحوی، بیشتر در دنیای UNIX بسیار محبوب است.

مثال 3-4: کامپایل آن در GCC 4.7.3

```
gcc -s 1_1.c
```

خروجی زیر را می‌گیریم:

مثال 3-5: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world\n"
        .text
        .globl main
        .type   main, @function
main:
```

```
.LFB0:
	.cfi_startproc
	pushl	%ebp
	.cfi_def_cfa_offset 8
	.cfi_offset 5, -8
	movl	%esp, %ebp
	.cfi_def_cfa_register 5
	andl	$-16, %esp
	subl	$16, %esp
	movl	$.LC0, (%esp)
	call	printf
	movl	$0, %eax
	leave
	.cfi_restore 5
	.cfi_def_cfa 4, 4
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
	.ident "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
	.section	.note.GNU-stack,"",@progbits
```

این نمونه، شامل ماکروهای بسیاری است (با نقطه آغاز می‌شوند). اکنون اینها برای ما جالب نیستند و به خاطر ساده‌سازی، می‌توانیم از آنها چشم‌پوشی کنیم (به جز ماکروی string که همانند ترتیب یک رشته در زبانC ، با کاراکتر تهی (null) پایان می‌یابد). سپس نمونه‌ی زیر را می‌بینیم:

مثال 3-6: GCC 4.7.3

```
.LC0:
	.string"hello, world\n"
main:
	pushl	%ebp
	movl	%esp, %ebp
	andl	$-16, %esp
	subl	$16, %esp
	movl	$.LC0, (%esp)
	call	printf
	movl	$0, %eax
	leave
	ret
```

برخی از تفاوت‌های اصلی میان ترکیب نحوی Intel و AT&T عبارت‌اند از:

- عملوندهای مبدا و مقصد، مخالف یکدیگر نوشته می‌شوند:

  در ترکیب نحوی Intel <دستور> <عملوند مقصد> <عملوند مبدا>

  در ترکیب نحوی AT&T <دستور> <عملوند مبدا> <عملوند مقصد>

یک راه آسان برای به خاطر سپردن تفاوت وجود دارد: هنگام سروکار داشتن با ترکیب نحوی اینتل، می‌توانید تصور کنید که یک نماد تساوی (=) میان عملوندها وجود دارد و هنگام سروکار داشتن با ترکیب نحوی AT&T می‌توانید تصور کنید که یک فلش به سمت راست (→) وجود دارد.

- AT&T: پیش از نام ثبات یک نماد درصد (%) و پیش از عدد، یک نماد دلار ($) باید نوشته شود. به جای براکت، از پرانتز استفاده می‌شود.
- AT&T: برای تعریف اندازه‌ی عملوند، یک پسوند به دستور افزوده است:
  - ✓ q– چهارگانه (64 بیت).
  - ✓ L– بلند (32 بیت).
  - ✓ W– واژه (16 بیت).
  - ✓ B– بایت (8 بیت).

اجازه دهید به نتیجه‌ی کامپایل‌شده برگردیم: این دقیقا با آنچه در IDA دیدیم یکسان است، اما با یک تفاوت نامحسوس: 0FFFFFFF0h به‌صورت 16-$ معرفی شده است. این همانند 16 در سیستم ده‌دهی است که برابر با 0x10 در مبنای شانزده است. همچنین 0X10- برابر با 0xFFFFFFF0 (برای یک نوع داده 32 بیتی) است.

یک تفاوت دیگر این که مقدار بازگشتی با استفاده از MOV معمولی و نه XOR، روی مقدار 0 تنظیم شده است. MOV تنها مقدار را به یک ثبات بارگذاری می‌کند. نام آن بی‌معناست (داده منتقل نمی‌شود و تنها کپی می‌شود). در معماری‌های دیگر، این دستور "LOAD" یا "STORE" یا چیزی همانند آن نام‌گذاری شده است.

3-2 x86-64

3-2-1 MSVC—x86-64

اجازه دهید MSVC 64-bit را نیز بررسی کنیم:

مثال 3-7: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

در x86-64 تمام ثبات‌ها به 64 بیت توسعه داده شده‌اند و هم‌اینک نام آنها، یک پیشوند R دارد. بیشتر برای استفاده از پشته کمتر (به دیگر سخن، اغلب برای دسترسی به حافظه خارجی/کش کمتر)، یک راه رایج برای رد کردن آرگومان‌های تابع با ثبات‌ها وجود دارد (fastcall در بخش 3-64). برای نمونه، بخشی از آرگومان‌های تابع به ثبات‌ها رد می‌شوند و بقیه با پشته. در ویندوز 64 بیتی چهار آرگومان تابع به ثبات‌های RCX، RDX، R8 و R9 رد شده‌اند. این همان چیزی است که در اینجا می‌بینیم. هم‌اکنون برای printf() یک اشاره‌گر به رشته، به پشته رد نشده است، اما در ثبات RCX وجود دارد.

اینک اشاره‌گرها 64 بیتی هستند و بنابراین آنها به ثبات‌های 64 بیتی (که پیشوند R دارند) رد شده‌اند. گرچه، برای سازگاری، هنوز دسترسی به بخش‌های 32 بیتی با استفاده از پیشوند E شدنی است.

در جدول زیر، چگونگی نمایش ثبات‌های RAX/EAX/AX/AL در x86-64 آمده است:

7th (شمار بایت)	6th	5th	4th	3rd	2nd	1st	0th
RAX[x64]							
				EAX			
						AX	
						AH	AL

تابع main() یک مقدار از نوع عدد صحیح را برمی‌گرداند که برای سازگاری و قابلیت حمل بهتر، هنوز در C\C++، 32بیتی است و به همین دلیل ثبات EAX در پایان تابع (برای نمونه بخش 32 بیتی ثبات) به جای RAX پاک می‌شود. همچنین 40 بایت اختصاص داده شده، در پشته محلی وجود دارد. این "فضای هاشورزنی" نامیده می‌شود که درباره‌ی آن در بخش 1-2-8 صحبت می‌کنیم.

3-2-2 GCC—x86-64

اجازه دهید GCC را در لینوکس 64 بیتی نیز آزمایش کنیم:

مثال 3-8: GCC 4.4.6 x64

```
.string"hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

این یک روش برای پاس دادن آرگومان‌های تابع به ثبات‌هاست که در Linux، BSD* و Mac OS X نیز استفاده می‌شود. شش آرگومان نخست، به ثبات‌های RDI، RSI، RDX، RCX، R8 و R9 رد شده‌اند و بقیه به‌وسیله‌ی پشته رد می‌شوند. بنابراین اشاره‌گر به رشته به EDI (بخش 32 بیتی ثبات) رد شده است. اما چرا از بخش 64 بیتی RDI استفاده نشد؟

مهم است که به یاد داشته باشید، تمام دستورهای MOV در حالت 64 بیتی، چیزی را در بخش 32 بیتی پایین می‌نویسند و همچنین 32 بیت بالایی را پاک می‌کنند. برای نمونه MOV EAX, 011223344h یک مقدار را به درستی در RAX می‌نویسد و در نتیجه، بیت‌های بالایی پاک خواهند شد. اگر فایل کامپایل شده شئ (o.) را باز کنیم، می‌توانیم تمام دستورهای آپ‌کد را ببینیم:

مثال 9-3: GCC 4.4.6 x64

```
.text:00000000004004D0                                  main  proc
near
.text:00000000004004D0 48 83 EC 08                        sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00                     mov     edi, offset
format ; "hello, world\n"
.text:00000000004004D9 31 C0                             xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF                    call    _printf
.text:00000000004004E0 31 C0                                xor     eax, eax
.text:00000000004004E2 48 83 C4 08                          add     rsp, 8
.text:00000000004004E6 C3                                      retn
.text:00000000004004E6                                  main  endp
```

همان‌گونه که می‌بینید، دستور را در ثبات EDI در آدرس 0x4004D4 می‌نویسد و 5 بایت را اشغال می‌کند. همین دستور اگر یک مقدار 64 بیتی را در ثبات RDI بنویسد، 7 بایت را اشغال می‌کند. به نظر می‌رسد، GCC در تلاش است مقداری از فضا را ذخیره کند. افزون بر این، می‌توان مطمئن شد، بخش داده که شامل رشته است، در آدرسی بیش از 4 گیگابایت، ذخیره نخواهد شد.

همچنین دیدیم که ثبات EAX پیش از فراخوانی تابع printf() پاک شد. به این دلیل که شماری از ثبات‌های برداری استفاده شده طبق استاندارد، به ثبات EAX رد شده‌اند: "اطلاعات، درباره‌ی شمار ثبات‌های برداری استفاده شده به همراه آرگومان‌های متغیر، پاس داده می‌شود".

3-3 GCC– یک چیز بیشتر

این واقعیت که یک رشته‌ی ناشناس در زبان C نوع ثابت دارد و رشته‌های اختصاص داده شده در بخش ثابت، در زبان C تضمین شده‌اند که تغییرناپذیر باقی بمانند، یک نتیجه‌ی جالب دارد: کامپایلر شاید از یک بخش ویژه از رشته استفاده کند.

اجازه دهید نمونه‌ی زیر را آزمایش کنیم:

```
#include <stdio.h>

int f1()
{
       printf ("world\n");
}

int f2()
{
       printf ("hello world\n");
}

int main()
{
       f1;()
       f2;()
}
```

کامپایلرهای رایج C\C++ (ازجمله MSVC) دو رشته را اختصاص می‌دهند، اما اجازه دهید ببینیم که GCC 4.8.1 چه کاری انجام می‌دهد:

مثال 10-3: GCC 4.8.1 + مثال IDA

```
f1              proc near

s               = dword ptr -1Ch

                    sub     esp, 1Ch
                    mov     [esp+1Ch+s], offset s ; "world\n"
                    call    _puts
                    add     esp, 1Ch
                    retn
f1              endp

f2              proc near

s               = dword ptr -1Ch

                    sub     esp, 1Ch
                    mov     [esp+1Ch+s], offset aHello ; "hello"
                    call    _puts
                    add     esp, 1Ch
                    retn
f2              endp

aHello          db 'hello'
s               db 'world',0xa,0
```

هنگامی که رشته‌ی "hello world" را چاپ می‌کنیم، این دو حرف در حافظه در کنار یکدیگر جای می‌گیرند و puts() که از تابع f2() فراخوانی شده است، از این قضیه که رشته تقسیم شده است آگاه نیست؛ یعنی تقسیمی صورت نگرفته است و در این لیست، تقسیم تنها به صورت "مجازی" انجام شده است.

هنگامی که puts() از f1() فراخوانی می‌شود، از رشته‌ی "world" به‌علاوه‌ی صفر بایت استفاده می‌کند. puts() از این قضیه که چیزی پیش از این رشته وجود دارد، آگاه نیست!

این ترفند هوشمندانه اغلب به‌وسیله‌ی GCC استفاده می‌شود و می‌تواند در مصرف حافظه، صرفه‌جویی کند.

3-4 ARM

من برای آزمایش پردازنده‌های ARM از چندین کامپایلر استفاده کردم:

- محبوب‌ترین آن در حوزه‌ی جاسازی شده (embedded) بود که زمان انتشار آن 6/2013 است.
- محیط برنامه‌نویسی Apple Xcode 4.6.3 (با کامپایلر LLVM-GCC 4.2).
- GCC 4.9 (Linaro) برای ARM64 و در قالب فایل اجرای ویندوز در آدرس http://go.yurichev.com/17325 در دسترس است.

در همه‌ی نمونه‌های این کتاب، از کدهای 32 بیتی ARM (ازجمله حالت‌های Thumb و Thumb-2) استفاده می‌شود، اگرچه اشاره‌ای به 32 بیت بودن آن نمی‌کنیم، اما هنگامی که از نسخه‌ی 64 بیتی ARM صحبت کنیم، آن را به شکل ARM64 بیان می‌کنیم.

3-4-1 Keil 6/2013 بهینه‌سازی نشده (حالت ARM)

اجازه دهید با کامپایل کردن نمونه در Keil آغاز کنیم:

```
Armcc.exe --arm --c90 -00 1.c
```

کامپایلر armcc نمونه‌های اسمبلی را در ترکیب نحوی اینتل تولید می‌کند اما یک پردازشگر ARM سطح بالا مربوط به ماکروها دارد و برای ما مهم‌تر است که دستورها را در "شرایط موجود" ببینیم. پس بیایید نتیجه‌ی کامپایل شده را در IDA ببینیم.

مثال 3-11: مربوط به Keil 6/2013 (در حالت ARM) بهینه‌سازی نشده IDA

```
.text:00000000             main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      __2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
main+4
```

در این نمونه، به آسانی می‌توانیم ببینیم که هر دستور، اندازه‌ی 4 بایتی دارد. در حقیقت، کد را برای حالت ARM کامپایل کردیم، نه برای Thumb.

نخستین دستور که STMFD SP!, {R4,LR} است، همانند دستور PUSH در x86 کار می‌کند و مقدار دو ثبات (R4وLR) را در پشته می‌نویسد. در این نمونه، خروجی کامپایلر armcc، به دلیل ساده‌سازی دستور PUSH {r4,lr} نشان داده شده است، اما این دقیق نیست. دستور PUSH تنها در حالت Thumb در دسترس است، بنابراین برای اینکه گیج نشوید، این را در IDA انجام می‌دهیم.

این دستور، نخست SP (اشاره‌گر پشته) را کاهش می‌دهد، بنابراین به محلی در پشته اشاره می‌کند که برای ورودی تازه آزاد است. سپس این مقادیر، ثبات‌های R4 و LR (ثبات پیوند) را در آدرس ذخیره‌شده در SP تغییریافته، ذخیره می‌کند.

این دستور (که همانند دستور PUSH در حالت Thumb است) توانایی ذخیره‌سازی مقادیر چندین ثبات را به صورت یکباره دارد و می‌تواند بسیار مفید باشد. به هر روی، این هیچ معادلی در معماری x86 ندارد. همچنین می‌توان اشاره کرد که دستور STMFD تعمیم یافته‌ی دستور PUSH (که ویژگی‌های آن گسترش یافته) است؛ زیرا نه تنها با SP بلکه با هر ثباتی می‌تواند کار کند. به دیگر سخن، STMFD را می‌توان برای ذخیره‌سازی بسته‌ای از ثبات‌ها در آدرس مشخصی از حافظه استفاده کرد.

دستور ADR R0, aHelloWorld مقداری را که در ثبات PC (شمارنده‌ی برنامه) جای دارد، از آفست جایی که رشته‌ی hello,world در آن جای دارد، اضافه یا کم می‌کند. شاید بپرسید که چگونه ثبات PC در اینجا استفاده می‌شود؟ این به اصطلاح "کد مستقل از موقعیت" نامیده می‌شود. چنین کدی را می‌توان در یک آدرس غیر ثابت در حافظه اجرا کرد. به بیانی دیگر، این یک آدرس‌دهی وابسته به PC است. دستور ADR اختلاف میان آدرس این دستور و آدرس جایی را که رشته در آن جای گرفته است، در نظر می‌گیرد. این تفاوت (آفست) بدون توجه به اینکه کد در چه آدرسی توسط سیستم عامل بارگذاری شده است، همیشه یکسان است. به همین دلیل، برای به دست آوردن آدرس مطلق حافظه که مربوط به رشته در زبان C است، تمام چیزی که نیاز داریم، افزودن آدرس دستور کنونی (از PC) است.

دستور BL __2printf تابع printf() را فراخوانی می‌کند. نحوه‌ی کار کردن این دستور به این شکل است:

- آدرس دستور BL (0xC) را در LR (ثبات پیوند) ذخیره می‌کند.
- سپس با نوشتن آدرس خود در ثبات PC کنترل را به printf() رد می‌کند.

هنگامی که اجرای printf() به پایان می‌رسد، باید اطلاعاتی درباره‌ی جایی که نیاز دارد کنترل را به آن برگرداند، داشته باشد. به همین دلیل، هر تابع کنترل را به آدرس ذخیره‌شده در ثبات LR رد می‌کند.

این یک تفاوت میان پردازنده‌های از نوع RISC "محض" مانند ARM و پردازنده‌های CISC مانند x86 است که آدرس برگشتی معمولا در پشته ذخیره می‌شود.

به هر روی، یک آدرس 32 بیتی مطلق یا آفست نمی‌تواند در یک دستور 32 بیتی BL رمزگذاری شود، زیرا تنها برای 24 بیت فضای کافی دارد. همان‌گونه که به یاد دارید، همه‌ی دستورها در حالت ARM اندازه‌ی 4 بایتی (32 بیتی) دارند. از این رو، آنها تنها می‌توانند بر روی آدرس‌ها در مرز 4 بایتی جای بگیرند؛ یعنی 2 بیت آخر از آدرس دستور (که همیشه بیت صفر است) شاید حذف شده باشد. به‌طور خلاصه، 26 بیت برای رمزگذاری آفست داریم. این برای رمزگذاری PC ـ جاری ± تقریبا 32M کافی است. (Current_PC ± ≈ 32M)

گام بعد که دستور MOV R0, #0 است، تنها 0 را در ثبات R0 می‌نویسد. دلیلش این است که تابع C مقدار 0 را برمی‌گرداند و مقدار بازگشتی در ثبات R0 جای داده شده است.

آخرین دستور LDMFD SP!, R4,PC معکوس دستور STMFD است که مقادیر را برای ذخیره در R4 و PC از پشته (یا هر جای دیگر حافظه) بارگذاری می‌کند و اشاره‌گر پشته SP را افزایش می‌دهد. آن در اینجا مانند POP کار می‌کند. توجه داشته باشید که نخستین دستور STMFD هر دو ثبات‌های R4 و LR را در پشته ذخیره می‌کند، اما R4 و PC در طول اجرای LDMFD دوباره بازسازی می‌شوند.

همان‌گونه که می‌دانیم، آدرس جایی که هر تابع باید کنترل را به آن برگرداند، معمولا در ثبات LR ذخیره شده است. نخستین دستور، مقدارش را در پشته ذخیره می‌کند، زیرا در هنگام فراخوانی printf() همان ثبات توسط تابع main() استفاده می‌شود. در پایان تابع، این مقدار می‌تواند به‌طور مستقیم در ثبات PC نوشته شود و در نتیجه، کنترل را به جایی که تابع فراخوانی شده است، رد می‌کند.

از آنجا که main() معمولا تابع اصلی در زبان C/C++ است، کنترل به بارگذار سیستم عامل یا نقطه‌ای در CRT (کتابخانه زمان اجرای C) یا چیزی مانند آن برگشت داده خواهد شد. همه‌ی آنها اجازه می‌دهند که حذف دستورهای BX LR در پایان تابع انجام شود.

DCB یک رهنمود (دستور) زبان اسمبلی است که یک آرایه از بایت‌ها یا رشته ASCII را تعریف می‌کند و همانند دستور DB در زبان اسمبلی x86 است.

3-4-2 Keil 6/2013 (در حالت Thumb) بهینه‌سازی نشده

اجازه دهید که با استفاده از Keil در حالت Thumb همان نمونه را کامپایل کنیم:

```
Armcc.exe --thumb --c90  -00 1.c
```

در: IDA

مثال 12-3: Keil 6/2013 (در حالت Thumb) بهینه‌سازی نشده + IDA

```
.text:00000000                main
```