نشریه علمی «علوم و فناوری‌های پدافند نوین»
سال دهم، شماره ۴، زمستان ۱۳۹۸؛ ص ۳۹۲-۳۸۳

پیاده‌سازی عملی حمله نوین کانال جانبی Flush+Reload بر روی AES

مهدی اصفهانی[۱]، هادی سلیمانی[۲*]، محمدرضا عارف[۳]

۱- دانشجوی دکتری دانشگاه آزاد اسلامی واحد کرج، ۲- استادیار، دانشگاه شهید بهشتی، ۳- استاد، دانشگاه صنعتی شریف

(دریافت: ۹۷/۰۵/۱۵، پذیرش: ۹۷/۱۰/۱۸)

چکیده

با توجه به آن‌که دسترسی‌های متعدد به حافظه زمان‌بر هستند، پردازنده‌ها از یک حافظه کوچک به نام حافظه نهان (Cache) به‌منظور بهینه‌سازی در زمان اجرا استفاده می‌کنند. وجود حافظه نهان منجر به ایجاد تغییرات زمانی در اجرای یک برنامه شده و یکی از مهم‌ترین منابع نشت اطلاعات کانال جانبی زمان محسوب می‌شود. حمله Flush+Reload دسته‌ای از حملات کانال جانبی مبتنی‌بر حافظه نهان است که از مهم‌ترین ویژگی‌های این حمله می‌توان به شناسایی دسترسی به یک خط حافظه خاص و مورد هدف بودن پایین‌ترین سطح حافظه نهان (LLC) اشاره کرد که این ویژگی‌ها منجر به افزایش دقت در حمله و کاربردی بودن آن می‌شود. در این مقاله یک حمله جدید Flush+Reload از نوع متن انتخابی، بر روی پیاده‌سازی الگوریتم رمزنگاری استاندارد AES که در کتابخانه OpenSSl پیاده‌سازی شده، ارائه شده است. درحالی‌که حمله پیشین Flush+Reload بر روی AES نیاز به حدود ۴۰۰,۰۰۰ عمل رمزنگاری دارد، در حمله ارائه‌شده در این مقاله مهاجم می‌تواند با مشاهده تنها حدود ۱۰۰ عمل رمزنگاری کلید را به‌صورت کامل بازیابی کند. حمله توصیف‌شده در این مقاله به‌صورت عملی پیاده‌سازی شده و نتایج عملی صحت حمله را تائید می‌کند.

کلیدواژه‌ها: حافظه نهان، حملات کانال جانبی زمان، حمله Flush+Reload، AES

Practical Implementation of a New Flush+Reload Side Channel Attack on AES

M. Esfahani, H. Soleimany*, M. R. Aref
Shahid Beheshti University
(Received: 06/08/2018; Accepted: 08/01/2019)

Abstract

Since multiple memory accesses are time consuming, processors use cache to optimize runtime. The cache leads to temporal changes in the implementation of a program and is one of the most important source of information leakage in the timing side channel. Flush+Reload attack is a series of Cache Side Channel attack that the most important characteristics of this attack can be used to identify access to a particular memory line and target being the lowest level cache (LLC) noted that these features lead to increased precision of attack and its usability. In this paper, a new Flush+Reload attack (of the chosen plaintext attack) on the AES implemented in the OpenSSL is presented. While the previous Flush+Reload attack on AES requires about 400000 encryption operations, the attack presented in this paper, required only about 100 encryption operations to fully recover encryption keys. The attack described in this paper is implemented in practice and the actual results confirm the attack's integrity.

Keywords: Cache, Timing Side Channel Attack, Flush+Reload Attack, AES

*Corresponding Author E-mail: h_soleimany@sbu.ac.ir

۱. مقدمه

الگوریتم‌ها و پروتکل‌های رمزنگاری به‌تنهایی امنیت اطلاعات را فراهم نمی‌کنند، بلکه آن‌ها نیاز به یک بستر دیجیتال برای اجرای امن و کارآمد [۱] دارند. هم‌اکنون یکی از مهم‌ترین چالش‌ها در حوزه رمزنگاری کاربردی، برآورد کردن امنیت الگوریتم‌های رمزنگاری در مقابل حملات کانال جانبی است. برخلاف تحلیل‌های ریاضی که از ضعف‌های ساختاری الگوریتم‌های رمزنگاری بهره می‌برند، حملات کانال جانبی از اطلاعاتی که از نحوه پیاده‌سازی الگوریتم رمزنگاری نشت می‌کند، استفاده می‌کنند. پیاده‌سازی ناامن یک طرح رمزنگاری که به لحاظ ریاضیاتی امن است ممکن است سبب ایجاد ضعف در مقابل حملات کانال جانبی شده و به‌صورت عملی شکسته می‌شود.

یکی از مهم‌ترین منابع نشت اطلاعات کانال جانبی، زمان ناشی از اجرای محاسبات است. دسترسی‌ها به حافظه و وجود انشعاب‌ها[1] در برنامه، در زمان اجرا هزینه‌بر هستند، بنابراین، پردازنده‌ها برای کاهش این هزینه، از حافظه نهان[2] و پیشگوی انشعاب[3] استفاده می‌کنند. این بهینه‌سازی در زمان اجرا، منجر به ایجاد تغییرات زمانی در اجرای یک برنامه می‌شود؛ بنابراین، پیاده‌سازی الگوریتم رمز بدون داشتن دانش حملات کانال جانبی زمانی و روش‌های ریاضیاتی جهت مقابله با این حملات، می‌تواند عاملی جهت نشت اطلاعات مخفی باشد.

اندازه‌گیری آسان و عدم نیاز به ابزار سخت‌افزاری خاص برای اندازه‌گیری، از ویژگی‌های خاص حملات کانال جانبی مبتنی بر زمان است. در یک کانال جانبی زمانی مبتنی بر حافظه نهان، رویدادهای ریزپردازنده مانند برخورد حافظه نهان[4] و فقدان حافظه نهان[5] با استفاده از اندازه‌گیری‌های زمان متمایز می‌شوند. درصورتی‌که اطلاعات موردنیاز پردازنده در حافظه نهان موجود باشد، اطلاعات به‌سرعت توسط پردازنده پردازش می‌شود و اصطلاحاً برخورد حافظه نهان اتفاق می‌افتد. در مقابل، اگر اطلاعات موردنیاز پردازنده در حافظه نهان موجود نباشد، اطلاعات باید از حافظه اصلی خوانده شود که منجر به افزایش زمان دسترسی و کاهش سرعت پردازش اطلاعات می‌شود. در این حالت اصطلاحاً گفته می‌شود که *فقدان* حافظه نهان اتفاق افتاده است. تفاوت زمانی بین رخدادهای برخورد حافظه نهان و فقدان حافظه نهان سبب نشت اطلاعات می‌شود چراکه برای مهاجم روشن می‌سازد که آیا اطلاعات در حال پردازش پیش از این استفاده شده‌اند یا خیر.

برای اندازه‌گیری زمان نیاز به منبع کلاک خیلی دقیقی است. بیشتر پردازنده‌ها، کلاک‌های دقیقی به فرم شمارنده مهر زمانی[6] دارند که می‌توان از آن‌ها به‌منظور اندازه‌گیری زمان با دقت بسیار بالا و سربار کم استفاده کرد. با توجه به اندازه‌گیری زمان دسترسی به حافظه نهان توسط امکانات موجود در پردازنده‌ها، می‌توان گفت حافظه نهان در حملات کانال جانبی زمانی چالش‌برانگیزتر و کاربردی‌تر است [۲].

بر اساس میزان دسترسی مهاجم، حملات کانال جانبی زمان مبتنی بر حافظه نهان را می‌توان به دودسته Time Driven و Access Driven تقسیم‌بندی کرد. در حمله Time Driven، مهاجم صرفاً توانایی رصد کل زمان اجرای الگوریتم رمزنگاری را دارد [۶-۳]. در مقابل در حملات Access Driven، فرض می‌شود که مهاجم توانایی ایجاد تغییرات در حافظه نهان را نیز دارد [۹-۷].

استخراج اطلاعات حساس توسط حافظه نهان، اولین بار توسط هو [۱۰] انجام شد. کسلی، اشنایر، وانگر و هال [۱۱]، امکان به‌کارگیری حافظه نهان را به‌عنوان منبعی برای اجرای حملات مبتنی بر نرخ برخورد حافظه نهان موردبررسی قراردادند. تأثیر رفتار حافظه نهان در حملات کانال جانبی زمان مبتنی بر جدول مبنا،[7] توسط تی‌سانو، سیتو، سوزاکی و شیگرای [۱۲] موردبررسی قرار گرفت و اولین نتایج عملی برای اعمال حملات Time Driven علیه DES صورت گرفت. برنشتاین [۱۳] با مشاهده غیرثابت بودن زمان اجرای الگوریتم رمز، توانست حمله‌ای را جهت استخراج کلید AES پیاده‌سازی کند. این حمله بر روی نسخه پیاده‌سازی‌شده رمز AES در کتابخانه OpenSSl انجام گرفت. در ادامه اولین نتایج عملی برای حملات حافظه نهان Time Driven علیه AES در سال‌های ۲۰۰۷-۲۰۰۴ ارائه شد [۳، ۵ و ۱۳] و در سال ۲۰۰۷، یک مدل مقاوم‌سازی رمزهای متقارن علیه حملات ارائه شد [۱۴]. پرسیوال و اسویک [۷] به ترتیب، پیشگام در حملات Access Driven بر روی RSA و AES بودند.

هرچند نتایج حاصل از حملات ارائه‌شده نسبت به حملات پیشین به‌طور قابل‌توجهی بهبود یافته‌اند [۷]، میزان عملیاتی بودن این حملات به‌طور دقیق مشخص نیست. همان‌طور که ذکر شد در حملات Access Driven، فرض می‌شود که مهاجم توانایی ایجاد تغییرات در حافظه نهان را دارد. این فرض سبب شده است که کاربرد حملات Access Driven با محدودیت‌هایی مواجه شود. در همین راستا برای اولین بار یاروم [۲۵] حمله جدیدی به نام Flush+Reload بر روی RSA ارائه کرد که مبتنی‌بر مشخصه‌های

[1] Branch
[2] Cache
[3] Branch
[4] Cache Hit
[5] Cache Miss

[6] Time Stamp Counter
[7] Look Up Table

به اشتراک گذاشته‌شده در حافظه نهان است و مهاجم بدون دسترسی به هسته پردازنده که توسط کاربر به‌کار گرفته می‌شود، می‌تواند تغییراتی در حافظه نهان ایجاد کند. این حمله به‌طور گسترده موردتوجه سایر محققین قرار گرفت. ایده حمله Flush+Reload بر روی AES، ابتدا توسط گولاچ، بانگرتر و کرن [۱۵] معرفی شد. سپس حمله Flush+Reload علیه دور آخر AES انجام و با ۴۰۰۰۰۰ بار عمل رمزنگاری تمام ۱۲۸ بیت کلید استخراج شد [۱۶]. همچنین گولمزوگلو، سینان، انسی، ایرازوکی و آیزنبرگ در یک سناریوی متفاوت مبتنی بر حمله Flush+Reload موفق به استخراج کل ۱۲۸ بیت کلید رمز AES با ۳۰۰۰ بار عمل رمزنگاری شدند [۲۸].

حمله Flush+Reload ارائه‌شده در تحقیق‌های پیشین در مقایسه با سایر حملات از این مزیت برخوردار است که مهاجم نیازی به دانستن متون اصلی ندارد و صرفاً با دیدن متن رمزشده، می‌تواند کلید را بازیابی کند. در مقابل مهم‌ترین محدودیت حمله این است که مهاجم باید تعداد زیادی عمل رمزنگاری را رصد کند (حدود ۴۰۰,۰۰۰ عمل رمزنگاری) [۱۶]. در این مقاله، ما یک حمله Flush+Reload جدید بر روی AES در مدل متن آشکار منتخب ارائه می‌کنیم که بر اساس آزمایش‌های عملی صورت گرفته مهاجم می‌تواند با رصد تعداد بسیار کمتری عملیات رمزنگاری AES (حدود صد متن منتخب) کلید را بازیابی کند.

در این مقاله، سناریوی حمله مبتنی بر Flush+Reload بوده که علیه دور اول AES از کتابخانه OpenSSl نسخه 1.1.0f در پردازنده Intel i5-2450M با فرکانس 2.50GHz پیاده‌سازی شده است. همچنین این حمله بر روی تمام پردازنده‌های مشابه قابل اعمال است. با توجه به آنکه حمله ارائه‌شده در این مقاله از نوع Flush+Reload است، نسبت به حملات پیشین Access Driven به‌مراتب کاراتر است. در حملات پیشین Access Driven بر روی AES، لایه اول حافظه نهان هدف حمله مهاجم است و در فرآیند حمله و قربانی می‌بایست روی یک هسته پردازنده، پردازش شوند؛ این در حالی است که در حمله Flush+Reload ارائه‌شده در این مقاله، پایین‌ترین سطح حافظه نهان که بین همه هسته‌های پردازنده به اشتراک گذاشته می‌شود، مورد استفاده قرار می‌گیرد. درنتیجه، در فرآیند حمله، مهاجم و قربانی می‌توانند روی دو هسته مجزا از پردازنده پردازش خود را انجام دهند. باید توجه کرد که عموماً در کاربردهای عملی، فرآیندهای مهاجم و قربانی هیچ‌گاه روی یک هسته پردازنده پردازش نمی‌شوند. بنابراین، سناریوی حمله Flush+Reload می‌تواند به‌صورت واقعی در سیستم‌های رمزنگاریِ فضای ابر و ماشین‌های مجازی نیز قابل‌استفاده و کاملاً عملی باشد.

همچنین، باید توجه کرد که در فرآیند حملات قبلی Time Driven، تنها مقادیر پرارزش بایت‌های کلید بازیابی می‌شوند، درصورتی‌که در سناریوی حمله ارائه‌شده در این مقاله تمامی مقادیر کلید بازیابی می‌شوند.

در بخش ۲، حملات کانال جانبی زمان مبتنی‌بر حافظه نهان را مورد بررسی قرار خواهیم داد. در این بخش ضمن بررسی ساختار حافظه نهان در پردازنده‌های مختلف، حمله Flush+Reolad را معرفی خواهیم کرد. در بخش ۳، ضمن معرفی پیاده‌سازی نرم‌افزاری کتابخانه OpenSSl نسخه 1.1.0f، سناریوی یک حمله جدید Flush+Reolad بر روی AES را شرح می‌دهیم. در بخش ۴، نتایج عملی پیاده‌سازی حمله را شرح می‌دهیم. درنهایت در بخش ۵ مسائل باز و اولویت‌های پژوهشی در این حوزه برای تحقیقات آتی را ارائه می‌کنیم.

۲. حملات کانال جانبی زمان مبتنی‌بر حافظه نهان

حافظه نهان یکی از اجزای پردازنده‌های مدرن است که می‌تواند موجب نشت اطلاعات زمانی، بالاخص برای رمزهای قالبی باشد. حملاتی که از حافظه نهان استفاده می‌کنند، نیازمند این هستند که دسترسی‌ها به حافظه در اجرای رمز در مکان‌هایی رخ دهد که وابسته به کلید مخفی باشند. همچنین این حملات از این حقیقت استفاده می‌کنند که فقدان حافظه نهان در مقایسه با برخورد حافظه نهان زمان بیشتری را صرف می‌کند [۱۷]. وقتی پردازنده به یک داده یا دستورالعملی از حافظه اصلی نیاز داشته باشد، ابتدا در حافظه نهان جستجو می‌کند، اگر آن را یافت به این عمل برخورد حافظه نهان و در غیراین‌صورت به آن فقدان حافظه نهان می‌گویند. در این حالت پردازنده، داده یا دستورالعمل را از حافظه اصلی می‌خواند و همچنین آن‌ها را به حافظه نهان نگاشت می‌کند.

نشت اطلاعات زمانی در سیستم‌های رمزنگاریِ فضای ابر [۱۸]، ماشین‌های مجازی (VMs) [۱۶]، سیستم‌های چندکاربره مانند سیستم‌عامل اندروید در موبایل‌های هوشمند [۱۹] و کتابخانه رمزنگاری OpenSSL [۲۰]، نشانگر تهدید جدی و عملی حملات زمانی علیه سامانه‌های امنیتی نسبت به سایر حملات کانال جانبی است. ما در ادامه ابتدا ساختار حافظه نهان و نحوه استفاده از آن در پردازنده را مورد بررسی قرار می‌دهیم. سپس مدل کلی حمله Flush+Reload را معرفی می‌کنیم.

۲-۱. حافظه نهان

در پردازنده‌های مدرن، حافظه نهان سرعت دسترسی به حافظه را افزایش می‌دهد که این کار را به‌وسیله نگهداری داده‌ها و

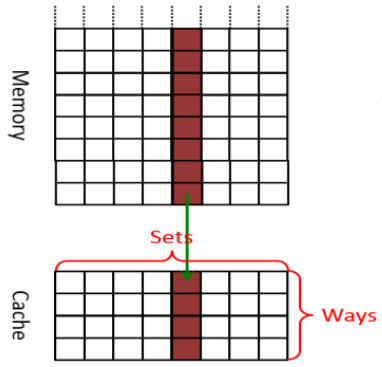
دستورالعمل‌هایی که اخیراً مورد دسترسی قرار گرفتند، انجام می‌دهد. حافظه نهان، یک بانک از حافظه با سرعت بالا بوده که نسبت به حافظه اصلی رایانه سریع‌تر و گران‌تر است [۲۱].

هر بخش از حافظه نهان از بخش‌هایی به نام خطوط کش[1] تشکیل شده است. هر خط از سه عنصر بلوک داده، تگ و بیت اعتبارسنجی تشکیل شده است. بلوک داده شامل اطلاعاتی است که هنگام دسترسی پردازنده به آن، از حافظه اصلی به حافظه نهان کپی می‌شود. تگ، آدرسی از حافظه اصلی است که داده در آن قرار دارد و بیت اعتبارسنجی یک فیلد تک‌بیتی است که اعتبارِ داده قرارگرفته در خط کش را می‌سنجد.

در حالتی که فقدان حافظه نهان رخ دهد پردازنده، داده یا دستورالعمل را از حافظه اصلی می‌خواند و آن را در حافظه نهان بارگذاری می‌کند. سه روش رایج برای نگاشت داده از حافظه اصلی به حافظه نهان وجود دارد. در روش اول، حافظه اصلی به بلوک‌های مساوی تقسیم می‌شود و هرکدام از خانه‌های حافظه نهان فقط به یکی از بلوک‌های حافظه تخصیص می‌یابد که به آن نگاشت مستقیم حافظه نهان[2] می‌گویند. حداقل زمان جستجو[3] از مزایا و بدترین عملکرد[4] به‌دلیل استفاده نشدن احتمالی قسمت‌هایی از حافظه نهان، از معایب این روش است. در روش دوم، مقدار یک خانه از حافظه اصلی، در هر خانه دلخواه از حافظه نهان می‌تواند ذخیره شود که به آن نگاشت شرکت‌پذیر کامل حافظه نهان[5] می‌گویند. بهترین عملکرد از مزایا و بدترین حالت در زمان جستجو از معایب این روش است.

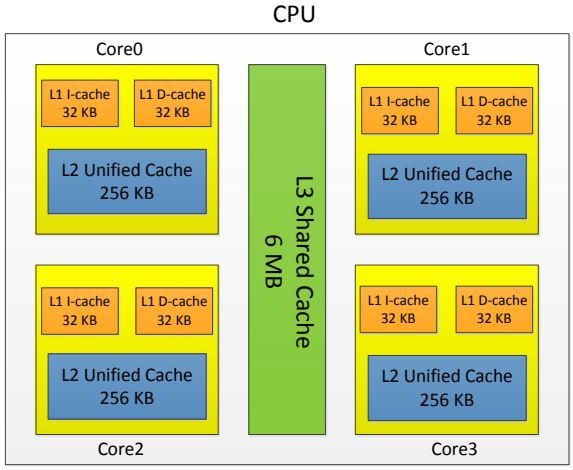
در روش سوم، هر خانه از حافظه اصلی فقط می‌تواند در یک تعداد مشخصی از خانه‌های حافظه نهان ذخیره شوند که به آن مجموعه شرکت‌پذیر حافظه نهان[6] می‌گویند. بهترین حالت برای زمان جستجو و عملکرد حافظه نهان نسبت به دو روش قبلی، موجب شده ریزپردازنده‌های مدرن از روش سوم استفاده کنند [۲۲]. حافظه نهان به یک سری مجموعه‌هایی[7] تقسیم‌بندی می‌شود و هر مجموعه شامل یک تعدادی سوی[8] است که یک بلوک از حافظه می‌تواند در هر سوی از یک مجموعه ذخیره شود (شکل ۱).

در پردازنده‌هایی همچون پردازنده‌های اینتل، سلسله‌مراتب در حافظه نهان شامل سه سطح است. بالاترین سطح را L1 می‌نامند که نزدیک‌ترین سطح به هسته بوده و کم‌حجم‌ترین و بالاترین سرعت را دارد. پایین‌ترین سطح حافظه نهان LLC[9] است که کندترین سرعت و بیش‌ترین حجم را دارا است [۲۳].



شکل ۱. ساختار حافظه نهان [۲۹]

لایه L1 به دو قسمت تقسیم می‌شود: L1-D[10] که داده‌های برنامه در آن ذخیره می‌شود و L1-I[11] که دستورالعمل‌ها و کدهای اجرای برنامه در آن ذخیره می‌شود. در پردازنده‌های چندهسته‌ای، معمولاً هر هسته شامل لایه‌های مجزا L1 و L2 بوده و همچنین لایه LLC بین همه هسته‌ها به اشتراک گذاشته می‌شود (شکل ۲) [۲۳]. در بیشتر پردازنده‌های اینتل رابطه بین لایه‌های حافظه نهان بدین‌صورت است که تمامی مقادیر ذخیره‌شده در L1 در لایه‌های L2 و LLC نیز وجود دارند. همچنین تمامی مقادیر ذخیره‌شده در L2 در لایه LLC نیز وجود دارند. به این خاصیت اصطلاحاً inclusive می‌گویند.



شکل ۲. پردازنده چهار هسته‌ای

در بخش بعد بررسی می‌کنیم که چگونه از خاصیت inclusive بودن لایه‌های حافظه نهان و همچنین مشترک بودن پایین‌ترین سطح حافظه نهان (یعنی LLC) بین همه هسته‌ها، می‌توان برای طراحی سناریوهای حملات Flush+Reload استفاده کرد.

[1] Cache Line
[2] Direct Mapped Cache
[3] Hit Time
[4] Hit Rate
[5] Fully Associative Cache
[6] Set Associative Cache
[7] Cache Set
[8] Way
[9] Last-Level Cache
[10] L1 Data Cache
[11] L1 Instruction Cache

۲-۲. ساختار حمله Flush+Reload

حافظه‌های نهان، منابع به اشتراک گذاشته‌شده هستند. اگر یک فرآیند در یک سیستم اجرا شود، یک بلوک از داده‌ها در حافظه نهان واکشی می‌شود، داده تا زمانی که از حافظه نهان خارج نشود در آن باقی می‌ماند. اجرا می‌تواند به‌وسیله هر فرآیندی در سیستم انجام شود و می‌تواند منجر به یک کانال زمانی پنهانی شود. حملات علیه الگوریتم‌های رمزنگاری مبتنی‌بر مشخصه‌های به اشتراک گذاشته‌شده در حافظه‌های حافظه نهان هستند (به‌طور مثال در فضای ابر همه کاربرها از پردازنده (حافظه نهان) به اشتراک گذاشته‌شده استفاده می‌کنند). این حملات به Access-Driven Cache شهرت دارند و مهاجم باید به حافظه نهان قربانی دسترسی داشته باشد [۲۶]. همان‌طور که پیش‌ازاین در بخش مقدمه اشاره شد، دسترسی هم‌زمان مهاجم به حافظه نهان مورداستفاده توسط کاربر امری چالش‌برانگیز است. بر همین اساس حمله Flush+Reload مبتنی بر مشخصه‌های به اشتراک گذاشته‌شده در حافظه نهان ارائه شد که مهاجم بدون دسترسی به هسته پردازنده، می‌تواند تغییراتی در حافظه نهان ایجاد کند. این حمله بر روی پردازنده‌هایی قابل اعمال است که حافظه نهان در آن‌ها ویژگی inclusive را دارا هستند [۲۵].

اجرای حمله Flush+Reload، شامل چهار مرحله است:

مرحله اول: نگاشت یک منبع به اشتراک گذاشته‌شده در حافظه نهان (این منبع را قربانی و مهاجم هر دو دارند).

مرحله دوم: مهاجم خط به اشتراک گذاشته‌شده در حافظه نهان را خالی یا اصطلاحاً تخلیه[1] می‌کند (با استفاده از دستور clflush).

مرحله سوم: در این مرحله مهاجم منتظر می‌ماند تا قربانی از مکان‌هایی از حافظه که در مرحله قبل تخلیه شده، استفاده کند. قربانی داده‌های مربوط به رمزنگاری را بارگذاری می‌کند.

مرحله چهارم: مهاجم داده به اشتراک گذاشته‌شده در مرحله اول را دوباره بارگذاری کرده و زمان دسترسی به داده را اندازه‌گیری می‌کند.

درصورتی‌که دسترسی مهاجم به داده در مرحله چهارم سریع باشد بدین معنی است، قربانی داده موردنظر را که در مرحله دوم تخلیه شده بود دوباره فراخوانی کرده است (برخورد حافظه نهان)؛ بنابراین، در این حالت مهاجم با اندازه‌گیری، زمان کمتری از بارگذاری را نسبت به زمانی که داده از حافظه اصلی مورد دسترسی قرار می‌گیرد، مشاهده خواهد کرد. درصورتی‌که دسترسی مهاجم به داده در مرحله چهارم سریع نباشد، بدین معنی است که قربانی داده فلاش شده در مرحله اول را مورد دسترسی قرار نداده است و بنابراین، داده در حافظه نهان موجود نیست و درنتیجه هنگام بارگذاری از حافظه به حافظه نهان (در مرحله چهارم)، مهاجم بالاترین زمان بارگذاری را در این مرحله مشاهده می‌کند.

به دو دلیل روش Flush+Reload، نسبت به حملات کانال جانبی قبلی که مبتنی‌بر ریزمعماری پردازنده‌ها بودند، قوی‌تر است. دلیل اول این است که برخلاف حملات قبلی که مهاجم دسترسی به کلاس‌هایی از مکان‌های بزرگ‌تر حافظه را شناسایی می‌کند (مانند مجموعه‌های حافظه نهان)؛ با استفاده از این روش مهاجم می‌تواند دسترسی به یک خط حافظه خاص را شناسایی کرده و با استفاده از دستور ()clflush آن خط خاص را فلاش کند و یا آن را مورد دسترسی قرار دهد، درنتیجه دقت حمله افزایش می‌یابد. مزیت دوم استفاده از روش Flush+Reload، به خاطر تمرکز این روش روی پایین‌ترین سطح حافظه نهان (LLC) است که بین همه هسته‌های پردازنده به اشتراک گذاشته می‌شود. درحالی‌که حملات قبلی نیز می‌توانند از LLC استفاده کنند ولی دقت حمله آن‌ها پایین‌تر است.

۳. سناریوی حمله جدید Flush+Reload بر روی AES

در این بخش ابتدا نحوه پیاده‌سازی نرم‌افزاری AES که به‌صورت گسترده مورد استفاده قرارگرفته است، بررسی می‌شود. سپس یک حمله جدید بر اساس روش Flush+Reload توصیف می‌شود که برخلاف حمله قبلی که مبتنی‌بر این روش بوده است به تعداد بسیار کمتری عملیات رمزنگاری برای بازیابی کلید نیاز دارد.

۳-۱. پیاده‌سازی نرم‌افزاری AES مبتنی‌بر جداول مبنا

رمز AES نمونه‌ای از ساختار شبکه جانشینی- جایگشتی در رمزهای قالبی است که در سال ۲۰۰۰ تحت عنوان Rijndael در مسابقه AES انتخاب و در سال ۲۰۰۱ تحت عنوان رمز AES، توسط موسسه ملی استاندارد و فنّاوری[2] (NIST) آمریکا به‌عنوان یک الگوریتم استاندارد انتخاب شد. رمز AES از کلیدهایی به طول ۱۲۸، ۱۹۲ و ۲۵۶ بیتی برای رمزگذاری و رمزگشایی استفاده می‌کند که در این مقاله همانند مقالات پیشین تنها حمله به الگوریتم رمزنگاری AES با طول کلید ۱۲۸ را موردبررسی قرار می‌دهیم.

در AES داده‌ها به‌صورت بایتی نمایش داده می‌شوند که هر

[1] Flush

[2] National Institute Of Standard And Technology

بایت درواقع عضوی از میدان $GF(2^8)$ است. ورودی رمز AES-128، به‌صورت یک ماتریس 4×4 بایتی مرتب می‌شود که به آن ماتریس حالت[1] می‌گویند. AES با طول کلید ۱۲۸ از ده دور تشکیل شده است. تابع دور از چهار تبدیل جانشینی بایتی[2]، شیفت سطری[3]، مخلوط‌سازی ستونی[4] و تبدیل جمع با زیرکلید دور[5] تشکیل شده است.

برای افزایش سرعت و کارآیی در اجرای الگوریتم‌های رمزنگاری، روش‌های پیاده‌سازی در سخت‌افزار و نرم‌افزار تفاوت دارند. به‌عنوان یک روش کارا و پرکاربرد، در پیاده‌سازی نرم‌افزاری رمزهای قالبی، ترکیب جعبه‌های جانشینی و لایه خطی با استفاده از جداول مبنا[6] پیاده‌سازی می‌شوند. از مزایای استفاده از این جداول در پیاده‌سازی نرم‌افزاری، می‌توان به انعطاف‌پذیر بودن بخش‌هایی از پیاده‌سازی و سرعت بالای اجرا اشاره کرد. در رمز AES، چهار جدول مبنا برای ترکیب لایه جانشینی بایتی و لایه خطی مخلوط‌ساز ستونی به‌صورت رابطه (۱) ساخته می‌شوند [۲۷] که در مقالات و گزارش‌های علمی عموماً تحت عنوان T-table نام‌گذاری می‌شوند و در کتابخانه‌های معروف رمزنگاری مانند OpenSSL مورداستفاده قرار می‌گیرند.

$$T_0[z] = \begin{bmatrix} 02.S(z) \\ S(z) \\ S(z) \\ 03.S(z) \end{bmatrix}, T_1[z] = \begin{bmatrix} 03.S(z) \\ 02.S(z) \\ S(z) \\ S(z) \end{bmatrix},$$
$$T_2[z] = \begin{bmatrix} S(z) \\ 03.S(z) \\ 02.S(z) \\ S(z) \end{bmatrix}, T_3[z] = \begin{bmatrix} S(z) \\ S(z) \\ 03.S(z) \\ 03.S(z) \end{bmatrix} \quad (۱)$$

با توجه به جداول ارائه‌شده در رابطه (۱)، نه دور اول AES می‌تواند به‌صورت رابطه (۲) محاسبه شود.

$$\begin{aligned} &T_0\left[s_0^{(r)}\right] \oplus T_1\left[s_5^{(r)}\right] \oplus T_2\left[s_{10}^{(r)}\right] \oplus T_3\left[s_{15}^{(r)}\right] \oplus [k_0^{(r)} k_1^{(r)} k_2^{(r)} k_3^{(r)}]|| \\ &T_0\left[s_4^{(r)}\right] \oplus T_1\left[s_9^{(r)}\right] \oplus T_2\left[s_{14}^{(r)}\right] \oplus T_3\left[s_3^{(r)}\right] \oplus [k_4^{(r)} k_5^{(r)} k_6^{(r)} k_7^{(r)}]|| \\ &T_0\left[s_8^{(r)}\right] \oplus T_1\left[s_{13}^{(r)}\right] \oplus T_2\left[s_2^{(r)}\right] \oplus T_3\left[s_7^{(r)}\right] \oplus [k_8^{(r)} k_9^{(r)} k_{10}^{(r)} k_{11}^{(r)}]|| \\ &T_0\left[s_{12}^{(r)}\right] \oplus T_1\left[s_1^{(r)}\right] \oplus T_2\left[s_6^{(r)}\right] \oplus T_3\left[s_{11}^{(r)}\right] \oplus \left[k_{12}^{(r)} k_{13}^{(r)} k_{14}^{(r)} k_{15}^{(r)}\right] \end{aligned} \quad (۲)$$

در رابطه (۲)، ماتریس حالت در دور rام را با $s^{(r)}$، امین بایت از ماتریس حالت در دور rام با نماد $s_i^{(r)}$ نشان داده می‌شود، که در آن، $0 \le r \le 9$ و $0 \le i \le 15$ هستند.

به‌دلیل نبود عملگر مخلوط‌ساز ستونی در دور آخر (دور دهم الگوریتم)، نمی‌توان به‌صورت ذکرشده در رابطه (۲) از این جداول استفاده کرد. در پیاده‌سازی‌های استاندارد عموماً دو راه‌کار برای پیاده‌سازی دور آخر استفاده می‌شود. یک راه استفاده از یک جدول مبنای مانند T_4 است. راه‌کار دیگر استفاده از جداول ذکرشده در رابطه (۱) به نحوی متفاوت است. از آنجائی که دور آخر تنها شامل جعبه جانشانی و شیفت سطری است، می‌توان مقادیر $S(s_{12}^9)$، $S(s_8^9)$، $S(s_4^9)$ و $S(s_0^9)$ را با فراخوانی T_0 و استفاده از (تنها) بایت دوم آن محاسبه کرد. به‌طور مشابه می‌توان سایر بایت‌ها را با فراخوانی جداول T_1، T_2 و T_3 محاسبه کرد. لازم به ذکر است که پیاده‌سازی مورداستفاده در کتابخانه OpenSSl نسخه 1.1.0f از روش دوم به‌منظور پیاده‌سازی دور آخر استفاده کرده است. به‌عبارت‌دیگر، هر یک از جداول T، در هر دور ۴ بار و درمجموع به ازای یک عمل رمزنگاری ۴۰ بار فراخوانی می‌شوند.

۳-۲. توصیف حمله جدید

در این بخش، سناریو حمله Flush+Reload جدید را ارائه می‌شود. ابتدا مشاهدات اصلی که حمله بر اساس آن طرح‌ریزی شده است تشریح شده و سپس مراحل حمله به‌صورت جزئی ارائه می‌شود.

۳-۲-۱. مشاهدات اصلی

هر یک از جداول T توصیف‌شده در رابطه (۱)، یک بایت z را به ۴ بایت خروجی نگاشت می‌کنند. با توجه به اینکه هر بایت ۲۵۶ حالت دارد، هر جدول مبنا T نیاز به ۱۰۲۴= ۴×۲۵۶ بایت برای ذخیره‌سازی دارد. اگر هر خط از حافظه نهان ۶۴ بایتی باشد، ذخیره کامل یک جدول T در حافظه نهان ۱۶ خط را اشغال می‌کند. همان‌طور که پیش‌ازاین در بخش ۳-۱ ذکر شد، هر جدول T در طول اجرای عملیات رمزنگاری ۴۰ بار فراخوان می‌شود. در زمان فراخوانی هرکدام از جداول، درصورتی‌که مقادیر موردنظر در حافظه نهان موجود نباشند، از حافظه اصلی فراخوانی شده و به همراه مقادیری که در یک سطر موجود هستند، در یک سطر از حافظه نهان قرار می‌گیرند. بنابراین، با پایان عملیات رمزنگاری تمامی مقادیر فراخوانی‌شده در حافظه نهان باقی می‌مانند و این در حالی است که ممکن است برخی از مقادیر فراخوانی نشده باشند و درنتیجه در حافظه نهان موجود نباشد. احتمال آنکه یک خط خاص از جدول T (مثلاً خط iام) پس از اجرای یک‌بار عمل رمزنگاری در حافظه نهان قرار نگرفته باشد، به‌صورت زیر محاسبه می‌شود:

$$\Pr\{\text{no access to } T[i]\} = \left(1 - \frac{16}{256}\right)^{40} \cong 7.5\%$$

[1] State
[2] Subbyte
[3] Shiftrows
[4] Mixcolumns
[5] Addroundkey
[6] Look-Up Table

معادله فوق بدین‌صورت محاسبه شده است: احتمال آن‌که یک خط خاص از جدول فراخوان شود $\frac{16}{256}$ است چرا که از ۲۵۶ حالت ممکن تنها ۱۶ مقدار در یک خط وجود دارند. بنابراین، احتمال عدم فراخوانی یک خط در هر بار فراخوانی جدول برابر $1-\frac{16}{256}$ است. با توجه به آن‌که هر جدول ۴۰ بار فراخوانی می‌شود، احتمال آن‌که یک خط خاص در طول اجرای الگوریتم فراخوانی نشده باشد برابر $\left(1-\frac{16}{256}\right)^{40}$ است.

حمله ارائه‌شده در این مقاله بر اساس این واقعیت است که اگر عملیات رمزنگاری تعدادی متن را در نظر بگیریم که در همه آن‌ها مقدار یک بایت خاص متن اصلی (مثلاً بایت P_0) ثابت باشد، در این صورت با توجه به ثابت بودن مقدار کلید (یعنی k_0)، همیشه مقدار بایت ورودی به دور اول ثابت و برابر $s_0^{(0)} = p_0 \oplus k_0$ است. بنابراین، در زمان عملیات رمزنگاری تمامی متن‌های مذکور همیشه مقدار $T_0(p_0 \oplus k_0)$ فراخوان می‌شود. این در حالی است که سایر مقادیر ممکن برای جدول T_0 لزوماً فراخوانی نمی‌شوند. به عبارت دقیق‌تر سایر خطوط جدول T_0 با احتمال $\left(1-\frac{16}{256}\right)^{40}$ فراخوانی نمی‌شوند.

هدف ما این است که از مشاهده مذکور استفاده کرده و با استفاده از حمله Flush+Reload دریابیم که کدام خط از جدول T_0 در دور اول فراخوانی شده است. با یافتن خط فراخوانی‌شده، مقادیر ممکن برای $p_0 \oplus k_0$ را یافته و بر اساس آن می‌توان کاندیداهای ممکن برای k_0 را پیدا کرد (با توجه به آنکه p_0 مشخص است). تکرار حمله برای تعداد دفعات محدود می‌توان مقدار بایت k_0 کلید را به‌صورت یکتا پیدا کرد. بدیهی است که این سناریو را می‌توان به‌طور مشابه برای سایر بایت‌های کلید تکرار کرده و تمام کلید را به‌صورت یکتا به‌دست آورد.

۳-۲-۲. مراحل اجرای حمله

سناریوی حمله در سه مرحله به شکل زیر قابل‌اجرا است:

مرحله اول: بازیابی آفست جداول T

قبل از اجرای حمله، مهاجم باید آفست هر یک از جداول T را نسبت به شروع کتابخانه رمزنگاری بازیابی کند. با توجه به اینکه هر یک از جداول T در ۱۶ خط حافظه ۶۴ بایتی ذخیره می‌شود، مهاجم با آگاهی از آفست هر یک از جداول T، می‌تواند به هر خط حافظه مراجعه کند. روش مراجعه به هر خط حافظه به‌طور کامل در بخش ۴ بحث شده است.

مرحله دوم: اجرای حمله Flush+Reload و اندازه‌گیری زمان‌ها

در این مرحله، حمله Flush+Reload با در نظر گرفتن دور اول رمز AES انجام می‌شود. ابتدا مهاجم هر ۱۶ خط کش ۶۴ بایتی مورداستفاده برای هر $T_\ell\left[s_i^{(0)}\right], i \equiv 1\ (mo\ 4), \ell = 0,1,3,4$ را با استفاده از دستور clflush(offset[j]) فلاش می‌کند که $s_i^{(0)} = p_i \oplus k_i$، $i = 0, \dots, 15$ بوده و offset[j] به ازای $j = 0, \dots, 15$، امین آفست از خط حافظه‌ای است که جدول T در آن ذخیره شده است. بعد از هر مرحله فلاش کردن offset[j]، مهاجم رمز AES را اجرا کرده و پس از بارگذاری offset[j] زمان را اندازه‌گیری می‌کند. برای جلوگیری از نویز، مهاجم ℓ بار فرآیند رمزنگاری را اجرا کرده و زمان بارگذاری هر آفست را به زمان‌های بارگذاری قبلی آن اضافه می‌کند.

مرحله سوم: بازیابی کلید

مهاجم برای بازیابی اولین بایت کلید k_0، رمز AES را روی متن آشکاری که p_0 آن ثابت و مابقی p_iها به‌صورت تصادفی مقداردهی شده‌اند، اجرا کرده و زمان دسترسی را طبق مرحله دوم برای هر خط کش T_0 اندازه‌گیری می‌کند. پس از ℓ بار تکرار این فرآیند، خط حافظه متناظر با کمترین زمان، به احتمال زیاد مورد دسترسی قرارگرفته و در آن برخورد حافظه نهان اتفاق افتاده است. با توجه به آشکار بودن محتویات خط‌کش متناظر و با به‌کار بردن رابطه (۳)، می‌توان به ۱۶ کاندید برای کلید k_0 رسید.

$$k_0 = p_0 \oplus T_l\left[s_0^{(0)}\right] \ , l \equiv 0 \left(\mathrm{mod}\, 4\right) \qquad (۳)$$

با پیاده‌سازی فرآیند فوق برای p_0های مختلف، کاندیداهای متعددی از k_0 را می‌توان محاسبه نمود که با اشتراک‌گیری، مقدار صحیح بایت اصلی کلید k_0 بازیابی می‌شود (شکل ۳).

برای بازیابی بایت‌های دیگر کلید، باید p_i متناظر با k_i را ثابت گرفته و بعد فرآیند فوق را با به‌کارگیری رابطه (۳) با $T_\ell[s_i^0]$ های مناسب اجرا کرد.

۴. نتایج پیاده‌سازی عملی حمله

برای آزمایش سناریوی حمله به‌صورت عملی، از کتابخانه OpenSSl نسخه 1.1.0f تحت سیستم‌عامل Ubuntu 16:04 استفاده شده است که فرآیند حمله و رمز روی دو هسته مجزا از پردازنده Intel i5-2450M با فرکانس 2.50GHz پردازش

می‌شوند. همچنین فرض می‌شود سیستم رمزکننده قربانی در اختیار مهاجم قرارگرفته به‌طوری‌که می‌تواند هر متن آشکار دلخواهی را با کلید نامعلوم رمز کند. کلید توسط قربانی از قبل مقداردهی شده است.

```
input: P_0          \\Reload vector for plaintext byte 0
offset[j]
  \\Reload addresses of each memoryline form T_0 table
output: K_0
begin
sum_j = 0
forall d_a ∈ P_0        \\d_a is constant and d_a ∈ [0,255]
  for l number of encryption do
      P ← (d_a ‖ P_1 ‖ P_2 ‖ ⋯ ‖ P_15) \\P_i are random and P_i ∈ [0,255]
      for (j = 0:15)

flush (offset[j])  \\ Flush each memoryline from T_0 table
        t1_j ← Time (Encryption using P)
        t2_j ← Time (Reload (offset[j]) )

       delta_j ← t2_j − t1_j  \\
  Access timing for each memoryline form T_0 table
        sum_j = sum_j + delta_j
     end
  end
 if (sum_j equal to Minvalue) then
   b = j

K_{d_a} = d_a ⊕ (The content of the memory line with the address off
\\Generate 16 key candidates correspondign d_a value
end
return ∩ K_{d_a}          \\Retrieve byte K_0
                          end
```

شکل ۳. شبه کد مراحل اجرای حمله

در این حمله مهاجم ابتدا فایل libcrypto.so که روش‌های مختلف رمزنگاری مورداستفاده در کتابخانه OpenSSl مانند AES، DES و ... را برای کاربر فراهم می‌کند، با استفاده از دستور

readelf -a libcrypto.so > ~/aeslib.txt

در ترمینال لینوکس، به یک فایل متنی تبدیل و سپس آفست همه جداول T را بازیابی می‌کند. سناریوی حمله Flush+Reload طبق توضیحات بخش ۳-۲ در چهار مرحله اجرا می‌شود.

در مرحله اول مهاجم با استفاده از دستور $mmap$ ، فایل باینری libcrypto.so را به حافظه مجازی در دیسک سخت نگاشت کرده و با توجه به اینکه از آفست‌های آدرس هر جدول مبنا در فایل باینری آگاهی دارد، اقدام به محاسبه آدرس‌های مجازی هر خط حافظه نهان از جدول مبنا می‌نماید. همچنین با فعال کردن آرگومان MAP_SHARED در دستور $mmap$، ناحیه‌ای از فضای مجازی را که فایل باینری در آن نگاشت شده، بین فرایندهای دیگر به اشتراک می‌گذارد (به‌طور مثال بین قربانی و مهاجم). هرگونه تغییرات در داخل فایل باینری توسط هر فرایند، موجب تغییرات فایل در حافظه مشترک نیز می‌شود. در مرحله دوم مهاجم با توجه به آگاهی از آدرس مجازی هر خط حافظه نهان از جدول مبنای T_0 ، محتوای هر خط را تخلیه می‌کند. در مرحله سوم مهاجم رمز AES را با متن آشکار ۱۶ بایتی (یا ۱۲۸ بیتی) که در آن، بایت p_0 مقداری ثابت بوده و بقیه بایت‌ها به‌صورت تصادفی مقداردهی شده‌اند را اجرا می‌کند. در مرحله چهارم مهاجم داده مربوط به جدول مبنای T_0 را بارگذاری کرده و زمان دسترسی به داده را برای هر خط‌کش اندازه‌گیری می‌کند.

چهار مرحله فوق توسط مهاجم ۱۰۰ مرتبه انجام شده که درنهایت خط حافظه نهانی که برای ذخیره مقدار $T_0(p_0 \oplus k_0)$ مورد استفاده قرارگرفته آشکار می‌شود.

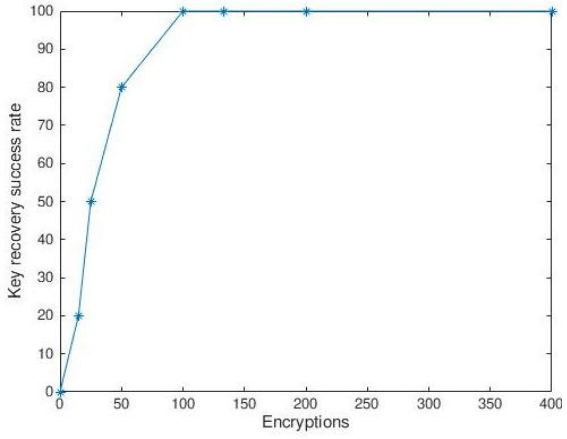
با توجه به مقادیر ممکن ۰...۱۵، ۱۶...۳۱، ۳۲...۴۷ و.....و ۲۴۰...۲۵۵ برای $s_0^{(0)}$ ، در پیاده‌سازی رمز AES از کتابخانه OpenSSl نسخه 1.1.0f، خروجی جدول مبنا می‌تواند به ترتیب در خطوط اول، دوم، ...، شانزدهم نگاشت شوند. با مشخص شدن خط حافظه نهان، به یک مجموعه کاندید ۱۶ عضوی برای $s_0^{(0)}$ می‌رسیم. درنهایت طبق توضیحات بخش ۳-۲-۲، مقدار صحیح بایت اصلی کلید k_0 بازیابی می‌شود.

طبق جدول (۱)، این سناریوی حمله نیاز به تعداد نمونه‌های کمتری از رمزنگاری نسبت به حملات مشابه دارد. همچنین تمام ۱۲۸ بیت کلید با ۱۰۰ نمونه رمزنگاری، در زمان بلادرنگ استخراج می‌شود. طبق شکل (۴) با افزایش تعداد نمونه‌های رمزنگاری، درصد موفقیت بازیابی کلید نیز افزایش می‌یابد.

از مهم‌ترین نقاط قوت سناریوی حمله پیشنهادشده نسبت به روش‌های استفاده‌شده در گزارش‌های قبلی [۹ و ۲۴]، استخراج کل کلید است درصورتی‌که در آن گزارش‌ها فقط مقادیر پرارزش بایت‌های کلید بازیابی می‌شوند.

جدول ۱. مقایسه تعداد نمونه‌های موردنیاز رمزنگاری در حملات مشابه

مرجع مقاله	نوع حمله	تعداد نمونه‌های مورد نیاز برای حمله
مقاله [۱۶]	Flush+Reload	۴۰۰۰۰۰
مقاله [۲۸]	Flush+Reload	۳۰۰۰
این مقاله	Flush+Reload	۱۰۰



شکل ۴. درصد موفقیت بازیابی کلید برحسب تعداد نمونه‌های رمزنگاری

۵. روش‌های مقابله

برای مقابله با سناریوی حمله، روش‌های مختلفی وجود دارد، ازجمله پیاده‌سازی رمز با استفاده از روش AES-NI[1] به‌جای استفاده از روش جدول مبنا، استفاده از روش پیش‌فرض حافظه نهان[2] و استفاده از روش تخلیه حافظه نهان[3].

در روش اول برای پیاده‌سازی رمز AES به‌جای استفاده از جدول مبنا و دسترسی به حافظه نهان از دستورالعمل‌های سخت‌افزاری خاص استفاده می‌شود. بنابراین، نشت اطلاعات زمانی که به‌دلیل دسترسی به حافظه نهان رخ می‌دهد، در روش AES-NI وجود ندارد. با این‌حال استفاده از روش ذکرشده فقط برای مقابله با نشت اطلاعات زمانی هنگام اجرای رمز AES امکان‌پذیر است و برای سایر رمزهای قالبی امکان‌پذیر نیست.

در روش دوم استفاده از پیش‌فرض جداول مبنای قبلی برای اجرای هر دور رمز AES، می‌تواند نشت اطلاعات زمانی را کاهش دهد. اگر همه مقادیر جدول مبنا قبل از اجرای رمز در حافظه نهان بارگذاری شود، مهاجم توانایی رصد کردن تعداد دفعات دسترسی به حافظه نهان را برای خط‌های مختلف نخواهد داشت. با این‌حال استفاده از روش ذکرشده موجب افزایش زمان عملیات رمزنگاری نیز می‌شود.

در روش سوم مقادیر جدول مبنا بعد از هر اجرای رمز AES از حافظه نهان تخلیه می‌شوند. زمانی که مهاجم بخواهد یک خط حافظه نهان را مورد دسترسی قرار دهد، جداول مبنا را در حافظه نهان نخواهد یافت. بنابراین، رصد اختلاف زمانی هنگام اجرای عملیات رمزنگاری برای مهاجم امکان‌پذیر نخواهد بود. بااین‌حال، نتایج روش ذکرشده مشابه با روش دوم بوده و موجب افزایش زمان عملیات رمزنگاری می‌شود [۱۶].

۶. نتیجه‌گیری

در این مقاله با توجه به به‌کارگیری حمله Flush+Reload برای هر خط‌کش مورداستفاده T-table در دور اول رمز AES، بازیابی کل کلید با کمترین نویز نسبت به سایر حملات مشابه انجام می‌گیرد. بنابراین، حمله می‌تواند برای هر رمز قالبی دیگری که به‌صورت T-table پیاده‌سازی شده، اجرا شود. در این حمله نیازی به دانستن متن رمزی نیست و فقط مهاجم برای اجرای رمز از متن آشکار معلوم استفاده می‌کند. همچنین برخلاف حملات دیگر، فرایندهای مهاجم و قربانی در حمله Flush+Reload می‌توانند در هسته‌های مجزای پردازنده، پردازش شوند. با توجه به اینکه سناریوی حمله از خاصیت inclusive بودن و همچنین مشترک بودن پایین‌ترین سطح حافظه نهان بین همه هسته‌ها استفاده می‌کند این سناریوی حمله قابل پیاده‌سازی در سیستم‌های رمزنگاریِ فضای ابر و همچنین ماشین‌های مجازی است.

۷. مرجع‌ها

[1] Jahanbani, M.; Noroozi, Z.; Bagheri, N. “FPGA Implementation of Cryptographic Systems Based on Tate Pairing on Binary Field”; J. Adv. Defence Sci. Technol. 2016, 7, 95-106.

[2] Rebeiro, C.; Mukhopadhyay, D.; Bhattacharya, S. “Timing Channel Cryptography “; Springer, 2015.

[3] Aciicmez, O.; Schindler, W.; Koc, C. K. “Cache Based Remote Timing Attack on the AES”; Proc. Int. Conf. CT-RSA, 2007, 271-286.

[4] Aly, H.; ElGayyar, M. “Attacking AES Using Bernstein’s Attack on Modern Processors”; Proc. Int. Conf. AFRICACRYPT, 2013, 127-139.

[5] Neve, M.; Seifert, J. P.; Wang, Z. “A Refined Look at Bernstein’s AES Side-Channel Analysis”; Proc. Int. Conf. ASIACCS, 2006.

[6] Bonneau, J.; Mironov, I. “Cache-Collision Timing Attacks against AES”; Proc. Int. Conf. CHES, 2006.

[7] Percival, C. “Cache Missing for Fun and Profit”; 2005.

[8] Neve, M.; Seifert, J. P. “Advances on Access-Driven Cache Attacks on AES”; International Workshop on Selected Areas in Cryptography: Selected Areas in Cryptography 2006, 147-162.

[1] AES New Instruction
[2] Cache Prefetching
[3] Cache Flushing

[9] Tromer, E.; Osvik, D. A.; Shamir, A. "Efficient Cache Attacks on AES and Countermeasures"; J. Cryptology 2010, 23, 37–71.

[10] Hu, W. M. "Lattice Scheduling and Covert Channels"; Proc. IEEE Computer Soc. Symp. Res. Security and Privacy 1992, 52.

[11] Kelsey, J.; Schneier, B.; Wagner, D.; Hall, C. "Side Channel Cryptanalysis of Product Ciphers"; J. Computer Security 2000, 8, 141-158.

[12] Tsunoo, Y.; Saito, T.; Suzaki, T.; Shigeri, M. "Cryptanalysis of DES Implemented on Computers with Cache"; Proc. Int. Conf. CHES 2003, 62-76.

[13] Bernstein, D. J. "Cache-Timing Attacks on AES"; 2004.

[14] Tiri, K.; Aciicmez, O.; Neve, M.; Andersen, F. "An Analytical Model for Time-Driven Cache Attacks"; Proc. Int. Conf. FSE 2007, 399-413.

[15] Gullasch, D.; Bangerter, E.; Krenn, S. "Cache Games Bringing Access-Based Cache Attacks on AES"; IEEE Symposium on Security and Privacy 2011, 490-505.

[16] Irazoqui, G.; Sinan Inci, M.; Eisenbarth, T.; Sunar, B. "Wait a minute! A fast, Cross-VM Attack on AES"; Int. Workshop on Recent Advances in Intrusion Detection 2014, 299-319.

[17] Rebeiro, C.; Mukhopadhyay, D.; Bhattacharya, S. "Timing Channels in Cryptography: A Micro-Architectural Perspective"; Springer, 2014.

[18] Inci, M. S.; Gulmezoglu, B.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. "Cache Attacks Enable Bulk Key Recovery on the Cloud"; Int. Conf. Cryptographic Hardware and Embedded Systems 2016, 368-388.

[19] Lipp, M.; Gruss, D.; Spreitzer, R.; Maurice, C.; Mangard, S. "ARMageddon: Cache Attacks on Mobile Devices"; USENIX Security Symposium 2016, 549-564.

[20] Yarom, Y.; Benger, N. "Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-Channel Attack"; IACR Cryptology ePrint Archive, 2014, 140.

[21] Ge, Q.; Yarom, Y.; Li, F.; Heiser, G. "Contemporary Processors are Leaky–and there's nothing You Can Do about It"; The Computing Research Repository arXiv. 2016.

[22] Brumley B. B. "Covert Timing Channels, Caching, and Cryptography"; Ph.D. Thesis, Aalto University, 2011.

[23] Yarom, Y.; Genkin, D.; Heninger, N. "CacheBleed: A Timing Attack on OpenSSL Constant-Time RSA"; J. Cryptographic Eng. 2017, 7, 99-112.

[24] Osvik, D. A.; Shamir, A.; Tromer, E. "Cache Attacks and Countermeasures: the Case of AES"; Cryptographers' Track at the RSA Conference, 2006, 1-20.

[25] Yarom, Y.; Falkner, K. "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack"; USENIX Security Symposium, 2014, 1, 22-25.

[26] Rebeiro, C.; Mukhopadhyay, D.; Bhattacharya, S. "Access-Driven Cache Attacks on Block Ciphers"; Timing Channels in Cryptography 2015, 109-24.

[27] Daemen, J.; Rijmen, V. "The Design of Rijndael: AES-the Advanced Encryption Standard"; Springer Science & Business Media, 2013.

[28] Gulmezoglu, B.; Sinan, B.; Inci, M. S.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. "A Faster and More Realistic Flush+Reload Attack on AES"; Proc. Int. Conf. COSADE 2015, 111–126.

[29] Yarom, Y. "Microarchitectural Side-Channel Attacks"; Proc. Int. Conf. CHES 2016, Tutorial Part 2.