# پردازنده‌های دوقلو در مقابله با حمله کانال جانبی

احسان اعرابی[1]، بهاره افشین‌پور[2]

۱ عضو هیات علمی دانشگاه صنعتی همدان، aerabi@hut.ac.ir

۲عضو هیات علمی گروه مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه پیام نور

تاریخ دریافت: ۹۳/۱۲/۱۶ تاریخ پذیرش: ۹۴/۱۰/۹

## چکیده:

*در طول دو دهه گذشته کانون‌های پژوهشی روش‌های متعددی را برای استخراج اطلاعات حساس توسط نشت از کانال جانبی و همچنین روش مقابله با آن ارائه داده‌اند. این مقاله به ایده‌ای نو در حیطه ریزپردازنده‌ها می‌پردازد با نام «پردازنده دوقلو» یا Twin Microprocessor (TuP) و همچنین یک بسته نرم‌افزاری را ارائه می دهد که به ما اجازه می‌دهد که کدها را برای اجرا روی TuP دوباره ترجمه کنیم. TuP یک واحد محاسبه و منطق (ALU) و یک فضای حافظه اما دو بانک ثبات، دو پشته و دو اشاره‌گر پشته دارد. فرض کنیم که $Q_0$ و $Q_1$ دو زیربرنامه از یک الگوریتم رمزنگاری باشند. در هر سیکل ساعت TuP به صورت تصادفی یکی از زیربرنامه‌ها را انتخاب می‌کند تا اجرا شود. بنابراین با برنامه نویسی ماهرانه $Q_0$ و $Q_1$، دو زیربرنامه می توانند با همکاری هم یک فضای حافظه را تغییر دهند و یک وظیفه مشخص را به اتمام برسانند. ارزیابی‌های آزمایشگاهی نشان می‌دهد که ایده TuP در مقابله با حمله کانال جانبی اثربخش است.*

## کلیدواژه:

*حمله کانال جانبی، آنالیز تفاضلی توان، ریزپردازنده*

## مقدمه

در طول دو دهه گذشته کانون‌های پژوهشی روش‌های متعددی را برای استخراج اطلاعات حساس توسط نشت از کانال جانبی و همچنین مقابله با این حمله ارائه داده‌اند. فرض بر این است که خواننده این مقاله با حمله های کانال جانبی مانند SPA، DPA[6] و CPA[1] آشناست و در این متن آن‌ها را تشریح نخواهیم کرد.

این مقاله به ایده‌ای نو در حیطه ریزپردازنده‌ها می‌پردازد با نام «پردازنده دوقلو» یا Twin Microprocessor (TuP) و همچنین یک بسته نرم‌افزاری را ارائه می‌دهد که به ما اجازه می‌دهد که کدها را برای اجرا روی TuP ترجمه دوباره کنیم. TuP یک واحد محاسبه و منطق (ALU) و یک فضای حافظه اما دو بانک ثبات، دو پشته و دو اشاره گر پشته دارد. فرض کنیم که $Q_0$ و $Q_1$ دو زیربرنامه از یک الگوریتم رمزنگاری باشند. در هر سیکل ساعت TuP به صورت تصادفی یکی از برنامه ها را انتخاب می‌کند تا اجرا شود. بنابراین با برنامه نویسی ماهرانه $Q_0$ و $Q_1$، دو برنامه می‌توانند با همکاری هم یک فضای حافظه را تغییر دهند و یک وظیفه مشخص را به اتمام برسانند.

پروسه حمله در CPA یا DPA به طور عمده بر روی توان مصرفی در یک سیکل ساعت هدف در میان همه سیکل‌های ساعت یک الگوریتم رمزنگاری تمرکز می‌کند که به دفعات زیاد اجرا شده است. در این سیکل هدف بیشترین همبستگی[1] مابین داده‌های مخفی (معمولا کلید رمزنگاری) که در حال پردازش است و توان مصرفی دستگاه وجود دارد. با فرض ثابت بودن زمان اجرای الگوریتم (که معمولا برای جلوگیری از حمله زمانی [10] لازم است)، نفوذگر می تواند به راحتی توان مصرفی تعداد زیادی از اجراهای متعدد و پشت سرهم الگوریتم را زیر هم تراز کند و روی یک سیکل هدف متمرکز شود. سپس با تحلیل آماری توان مصرفی دستگاه می‌تواند تشخیص دهد که فرضیه او برای قسمت کوچکی از داده مخفی (معمولا یک بایت) درست بوده است یا خیر. حدس درست بیشترین تشابه را مابین توان مصرفی مورد انتظار و توان مصرفی واقعی دستگاه در سراسر اجراهای متعدد الگوریتم ایجاد خواهد کرد.

ایده TuP بر این سعی استوار است که حمله مذکور را با انتخاب تصادفی مابین $Q_0$ و $Q_1$ خنثی کند که باعث می‌شود که ترتیب دستورات درهم آمیخته شود و از این رو سیکل ساعت هدف که در

1 Correlation

آن سیکل، بایت‌های داده مخفی مورد پردازش قرار می‌گیرند، در هر اجرای الگوریتم جابجا شود. بدین ترتیب توان مصرفی در سیکل ساعت هدف نه تنها تحت تاثیر اجرای دستورات غیرحساس دیگر قرار م‌گیرد بلکه دیگر بایت‌های داده مخفی نیز آن را تحت تاثیر قرار می‌دهند. ما انتظار داریم که این ایده تلاش مورد نیاز نفوذگر را برای کشف داده‌های حساس از درون دستگاه بالاتر ببرد.

مزایای معماری TuP بدین ترتیب است:

- یک نفوذگر که قصد سوء استفاده از نشت اطلاعات کانال جانبی را دارد با مشکل عدم تفکیک درست عملیات محاسباتی در حوزه زمان مواجه خواهد شد. به علاوه توان مصرفی در زمان t نه تنها به $opcode_t$ بلکه به $opcode_{t-1}$ نیز بستگی دارد. به شرطی که این دو دستورالعمل به دو $Q_i$ متفاوت متعلق باشند، توان مصرفی دو برنامه یکدیگر را تحت تاثیر قرار داده و ردیابی توان[2] را مبهم و تار می کند.

- اضافه کردن تصادفی دستورات تاخیری (wait-states) یک روش مقابله مرسوم است[2,3,4] اما این تاخیرها کارایی سیستم را کاهش داده و سرعت اجرا را کند می‌کند(برای مثال زمانی که سیستم دستورات NOP را اجرا می کند تا نفوذگر را گمراه کند زمان ناگزیر هدر می رود). اجرای نوبتی و متناوب دو برنامه‌ی کاملا مفید باعث از دست رفتن زمان نخواهد شد. بعلاوه کارهای قبلی مرتبط با تصادفی سازی دستورات عمدتا بر یک الگوریتم خاص تمرکز می‌کنند ([12,13,14]) و یا منابع زیادی از سیستم (مانند توان مصرفی و مساحت تراشه) را در زمان اجرا طلب می‌کنند[15]. از طرفی محدودیت روش‌های سخت‌افزاری که فقط بر یک الگوریتم خاص (مثلا AES) متمرکزند را ندارد[17]. در ضمن تنها برروی موازی سازی بلوک‌ها استوار نیست و می‌تواند در سطح دستورالعمل نیز موازی سازی را انجام دهد[18].

- از آن‌جا که TuP یک نسخه (و نه دو نسخه) ALU و حافظه دارد (که گران‌ترین بخش‌های تراشه از حیث مساحت هستند)معماری آن نسبتا ارزان تمام می‌شود.

- نهایتا با سازگاری کامل با نسخه پیش از خود، با خاموش کردن یکی از دو Qi، کدهای معمولی مانند سایر پردازنده‌ها روی TuP اجرا می‌شود.

کامپایلرها روش‌های موازی‌سازی برنامه‌ها را به منظور افزایش کارایی استفاده می‌کنند. ما از روش [7] به منظور موازی سازی خودکار کدهای باینری خود استفاده کرده‌ایم. این روش الگوریتم-های کامپایل (مخصوصا[5]) را به گونه‌ای تغییر داده است که بتوان بر محدودیت‌های کدهای باینری مانند نبود اطلاعات سمبلیک و اندیس‌های توابع غلبه کرد. در این مقاله همچنین روشی جدید برای موازی سازی کدهای مستقیم در بخش ۳ ارائه شده است.

2 Power Trace

این مقاله بدین ترتیب تنظیم شده است: بخش دوم پیش زمینه های فنی مورد نیاز را ذکر می‌کند. بخش ۳ روش‌های موازی‌سازی کدها را بررسی می‌کند. بخش ۴ به ایده TuP می‌پردازد. بخش ۵ و ۶ نیز نتایج آزمایشگاهی را مرور خواهد کرد.

## پیش‌زمینه های مرتبط

### محدودیت‌ها در تغییر ترتیب دستورات وابسته

TuP به منظور اجرای خارج از ترتیب، غیرقابل پیش بینی و لابه-لای یکدیگر دو (یا بیشتر از دو) جریان دستورالعمل[3] طراحی شده است. این مسئله مستلزم آن است که همه حالت‌های ممکن لابه لا اجرا کردن کدها معادل هم باشند. برای اطمینان از معادل بودن دو نمونه مختلف از اجرای موازی یک الگوریتم، ما تئوری (۱) را از [5] استفاده خواهیم کرد:

«تئوری ۱: هرگونه تغییر $\phi$ در ترتیب اجرای دستوالعمل‌های یک برنامه که همه وابستگی‌های برنامه را حفظ می‌می‌کند، معنی آن برنامه را نیز حفظ خواهد کرد.»

در این تئوری معنای وابستگی به رابطه بین دستورالعمل‌ها از نوع نوشتن-بعداز-نوشتن (WaW)، نوشتن-بعداز-خواندن (WaR) و خواندن-بعداز-نوشتن (RaW) بازمی‌گردد. لازم به یادآوری است که WaW، WaR و RaW شرایطی از مخاطره[4] مابین داده‌های نوشته شده و خوانده شده در یک آدرس یکسان حافظه هستند، هنگامی که دو دستورالعمل به صورت موازی سعی در خواندن یا نوشتن در آن آدرس از حافظه می‌کنند. ما این مخاطرات را با یک مثال ساده در جدول ۱ نشان داده‌ایم، با فرض این مسئله که برای انجام یک عمل محاسباتی صحیح opcode1 می‌بایست قبل از opcode2 اجرا شود. متغیر مشکل ساز در این مثال x است.

جدول ۱. انواع مختلف وابستگی

| | RaW | WaR | WaW |
|---|---|---|---|
| $opcode_1$ | x:=a | a:=x | x:=a |
| $opcode_2$ | b:=x | x:=b | x:=b |

[5] مجموعه روش‌هایی را برای آزمایش وابستگی در حلقه‌ها به منظور موازی اجرا کردن دورهای آنان ارائه می‌کند که برای هدف ما کامل و مناسب است. [7] زیرمجموعه‌ای از این روش‌ها را بکار گرفته است تا کدهای باینری را موازی کند. از این رو ما [7] و [5] را برای موازی سازی حلقه‌ها (که در الگوریتم‌های رمزنگاری بسیار استفاده می‌شوند) در TuP استفاده کردیم.

۳ در این مقاله «جریان دستورالعمل‌ها» با عنوان دیگری مانند صف‌ها یا برنامه‌ها یا کدها و همچنین $Q_0$ و $Q_1$ ذکرگردیده است.

4 Hazard

### Motorola 68HC05

ما ایده TuP را با تغییر ریزپردازنده Motorola 68HC05 ارزیابی کرده‌ایم. این ریزپردازنده دارای یک انباره ۸ بیتی با نام A، یک ثبات ۸ بیتی اندیس با نام X، یک شمارنده برنامه ۱۶ بیتی با نام PC و چندین پرچم پایه‌ای[5] است. پردازنده می‌تواند ۶۴کیلوبایت حافظه را آدرس دهی کند که با M[0] , …M[64k-1] نشان داده می‌شود.

## موازی سازی

### بخش های حلقه و غیرحلقه در کد

برای موازی سازی برنامه Q با تقسیم برنامه به قسمت‌ها حلقه و غیر حلقه آغاز می کنیم.

یک حلقه یک دسته برنامه است که به صورت پشت سر هم تکرار می‌شود تا یک شرط خاص برای خروج از آن برقرار شود و معمولا شروع، بدنه و پایان دارد. هرآن‌چه خارج از حلقه قرار می‌گیرد را با عنوان بخش‌های غیرحلقه ذکر می‌کنیم.

هر دستورالعمل در بخش غیرحلقه بسته به وابستگی‌هایش با دستوراتی که قبلا به صف‌های Qi i∈{0,1} اضافه شده‌اند به $Q_0$ یا $Q_1$ ملحق می‌شود. الحاق[6] توسط الگوریتمی انجام می‌شود که بعدا در این مقاله ذکر خواهد شد.

حلقه‌ها و حلقه های تودرتو در کد می توانند توسط [5,7] موازی سازی شوند که عمدتا بر بررسی امکان موازی کردن پیمایش‌های حلقه‌ها تمرکز دارند.

### همگام سازی صف‌های دستورالعمل

به منظور اطمینان از این مسئله که دستورالعمل‌های وابسته به هم در دو صف هرگز ترتیب وابستگی را نقض نمی‌کنند، گاه به گاه صف $Q_i$ باید متوقف شود تا دستوالعملی خاص در صف مقابل یعنی $Q_{1-i}$ اجرا شود. این مسئله مستلزم این است که دستورات همگام سازی جدیدی به TuP اضافه شود. صف‌های دستورالعمل بوسیله دستورات ماشین جدید barrier(brr) و carrier(crr) با هم همگام می‌شود. در متن دستورات ماشین brr و crr با اختصار xrr نمایش داده خواهند شد.

• brr برای وادار کردن برنامه به ترتیب درست اجرای دو دستوالعمل متعلق به دو صف مختلف Qi بکار می‌رود. در ادامه‌ی هر دستورالعمل brr یک آدرس حافظه به عنوان پارامتر ذکر می‌شود مانند brr X که X مشخص می‌کند که صفی که brr به آن متعلق است باید متوقف شود تا صف مقابل دستوراتش را تا مکان X اجرا کند. جدول ۲ یک مثال 68HC05 از brr را نمایش می‌دهد. فرض کنید lda mem1 در صف Q0 باید قبل از sta mem1 در صف $Q_1$ اجرا شود. brr صف $Q_1$ را متوقف می‌کند تا $Q_0$ به نقطه LABEL1 برسد.

• crr برای ارسال مقدار درون یک ثبات از یک صف به صف دیگر بکار می‌رود. هر دستور crr با یک آدرس حافظه X و یک نام ثبات Y همراه است مانند crr X,Y که X مشخص می‌کند که صفی که crr به آن متعلق است (صف فعلی) باید متوقف شود تا صف دیگر اجرا را تا محل X کامل کند و سپس محتوای ثبات Y از صف دیگر به صف فعلی منتقل شود. جدول ۳ نحوه استفاده از crr را نشان می‌دهد. فرض کنیم که دستور add در صف $Q_1$ نیازمند مقداری بارگذاری شده در ثبات A از مکان حافظه mem1 است تا با 0x20 جمع کند. crr منتظر می‌ماند تا دستور lda کامل شود و سپس مقدار A را به $Q_1$ منتقل کرده و دستور add اجرا خواهد شود.

جدول ۲. دستورالعمل همگام سازی brr

| $Q_0$ | $Q_1$ |
|---|---|
| ...<br>lda mem1<br>LABEL1:<br>... | ...<br>brr LABEL1<br>sta mem1<br>... |

جدول ۳. دستورالعمل همگام سازی crr

| $Q_0$ | $Q_1$ |
|---|---|
| ...<br>lda mem1<br>LABEL2:<br>... | ...<br>crr LABEL2,A<br>add 0x20<br>... |

### الگوریتم برای کدهای غیرحلقه

برای شرح الگوریتمی که به منظور موازی سازی کدهای غیرحلقه ارائه کرده‌ایم، دستورالعمل k-ام در برنامه را با opcode[k] نمایش داده و خصوصیات آن را با مجموعه (D[k],S[k],Qi[k],R[k],n[k]) D[k] و S[k] به ترتیب مجموعه همه دستورالعمل‌های مقصد[7] و مبدا[8] است که به opcode[k] وابسته‌اند. اگر opcode[j] به علت وابستگی باید قبل از opcode[k] اجرا شود، ما opcode[j] را یک مبدا برای opcode[k] می‌نامیم و برعکس اگر باید بعد از opcode[k] اجرا شود ما opcode[j] را یک مقصد برای opcode[k] می‌نامیم. هر عضو در مجموعه D[k] یا S[k] به صورت (r,J) است. J اندیس دستورالعمل وابسته (مبدا یا مقصد) و r توصیف منبعی است که وابستگی دو دستورالعمل به علت دسترسی به آن منبع می‌باشد (نام ثبات یا آدرس حافظه). Qi[k] اندیس صفی است که opcode[k] نهایتا بعد از اجرای الگوریتم به آن الحاق می‌گردد(مقدار پیش فرض آن -1 است). R[k] اندیس «صف پیشنهادی»[9] از طرف مبداهای opcode[k] است. نهایتا n[k] تعداد دستورالعمل‌های وابسته است که باید قبل از اجرای opcode[k]

5 (C: Carry, Z: Zero, N: Negative and H: Half carry)
6 Assign

7 Destination
8 Source
۹ در ادامه توضیح داده می شود.

اجرا شوند. n[k] با تعداد مبداهای opcode[k] مقداردهی اولیه می‌گردد و هر گاه یکی از این مبداها به صفی الحاق می‌شود یک واحد از آن کسر می‌گردد. هرگاه که n[k] به صفر برسد، opcode[k] می‌تواند اجرا گردد زیرا کلیه مبداهایش به یکی از صف‌ها الحاق شده‌اند.

برای شرح این الگوریتم، کد شکل ۱ را در نظر بگیرید که نشان دهنده هفت دستورالعمل است که ثبات‌های انباره A و اندیس X را تغییر می‌دهند و گراف وابستگی آن‌ها در شکل ۲-الف دیده می‌شود. هر نود یک دستورالعمل است و هر یال جهت دار یک وابستگی از یک دستورالعمل مبدا به یک دستورالعمل مقصد. incx در خط چهارم خاصیت D[4] = {(X,6),(X,7)} را دارد چرا که نوشتن در ثبات X باید قبل از خطوط ۶ و ۷ اجرا شود و lda در خط اول خاصیت D[1]={(mem1,6) , (A,3)}.

```
1   lda mem1 ;      A ← mem1
2   ldx mem2 ;      X ← mem2
3   inca     ;      A ← A+1
4   incx     ;      X ← X+1
5   sta mem2 ;      mem2 ← A
6   stx mem1 ;      mem1 ← X
7   mul      ;      X:A = 256*X+A ← A*X
```

شکل ۱. نمونه برنامه

پروسه الحاق در الگوریتم ۱ کد نویسی شده است. هر پیمایش الگوریتم با پیدا کردن نودی آغاز می‌شود که هیچ نود ورودی ندارد یا این‌که تمام نودهای مبدا همسایه‌اش الحاق شده‌اند. این نود را « نود جاری» می‌نامیم. سپس الگوریتم نود جاری را به یکی از صف‌ها الحاق می‌کند و زمانی پایان می‌پذیرد که همه نودها الحاق شده باشند. انتخاب صف مناسب برای هر نود بر اساس دو شرط است:

۱. دستورالعمل‌هایی که قبلا به یکی از صف‌ها الحاق شده‌اند می توانند مقصدهای خود را راهنمایی کنند تا به همان صف ملحق شوند. این مسئله را «پیشنهاد»[10] نامیده‌ایم. هرگاه نودی به صفی الحاق می‌شود، اندیس آن صف به نودهای مقصد پیشنهاد می‌شود. البته پیشنهاد زمانی انجام می‌پذیرد که نوع وابستگی از جنس ثبات (و نه حافظه) باشد. این مسئله تعداد دستورات crr را کمینه می‌کند چرا که دستورالعمل‌های وابسته‌ی ثباتی در صف مشترک قرار می‌گیرند و نیاز به انتقال مقدار ثبات از صفی به صف دیگر از بین می‌رود.

۲. اگر نودی هیچ پیشنهادی نداشته باشد، به صف کوتاه تر اضافه می‌گردد.

بعداز مشخص کردن صف مناسب و الحاق نود جاری به آن، الگوریتم ۱ با توصیه صف انتخابی به مقصدهای نود جاری ادامه پیدا می‌کند. سپس برای این که مشخص شود آیا xrr نیاز است یا خیر، الگوریتم ۱ نودهای مبدا را بررسی می‌کند تا ببیند آیا یکی از این نودها به صف مقابل الحاق شده است یا خیر. اگر چنین است، دستورالعمل مناسب crr یا brr را درج می‌کند.

الگوریتم ۱. موازی سازی بخش‌های غیر حلقه

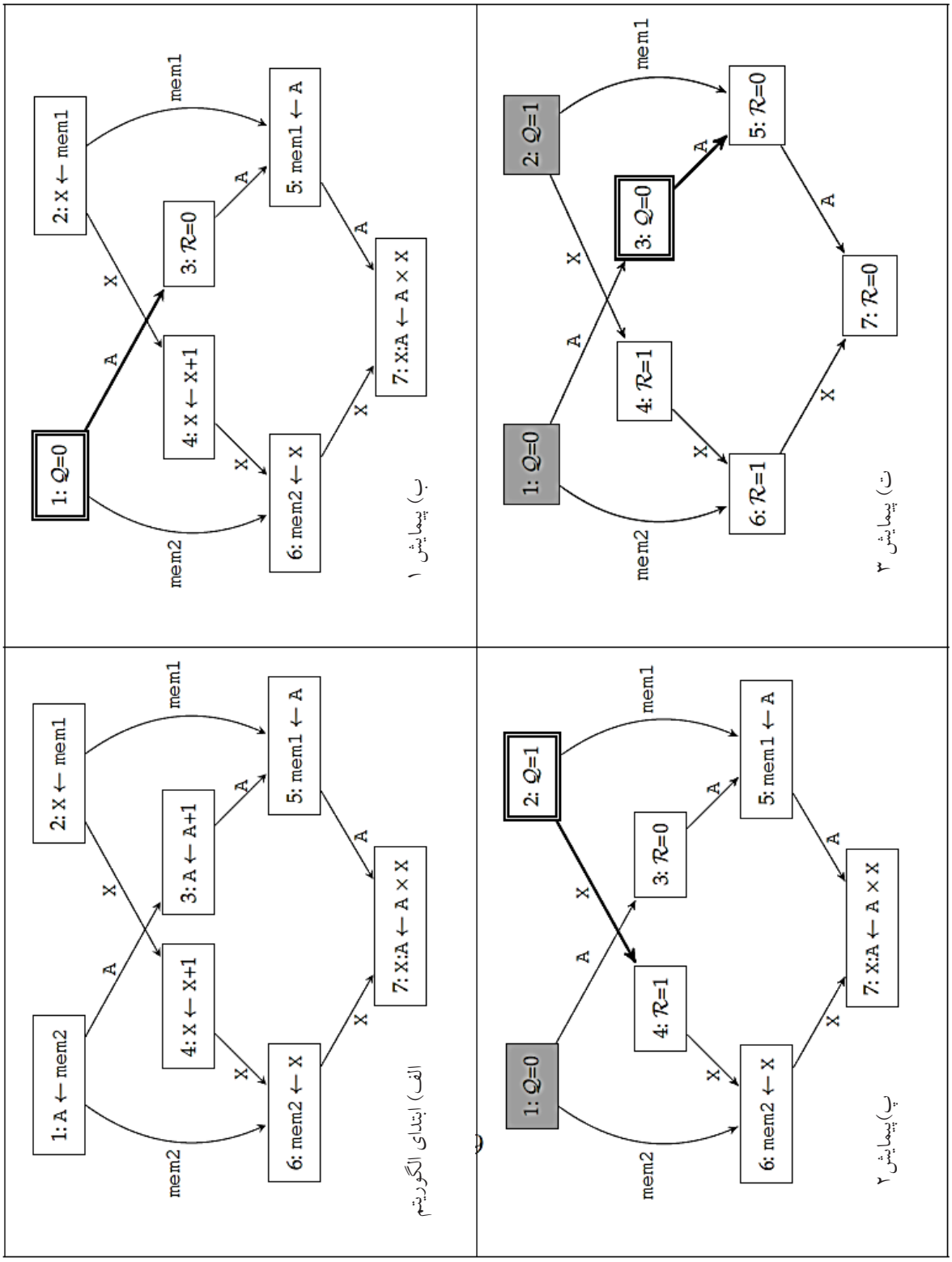
```
ParallelizationFunction( input: Program P, outputs: Queues Q0 and Q1 )
{
//initialization
for ∀opcode[k] k ∈ {1, 2, ..., size of program P} do
    D[k] ← all Destination of opcode[k];
    S[k] ← all Sources of opcode[k];
    n[k] ← |S[k]|; //number of unassigned sources of
    opcode[k]
    Qi[k] ← −1; //assigned queue
    Ri[k] ← −1; //recommendation queue
end for
//assigning nodes
while there is an unassigned instruction in the program do
    find an opcode[k] for which n[k] = 0 and
    Qi[k] = −1;
    if R[k] ≠ −1 then
        Qi[k] ← R[k];
    else
        Assign instruction opcode[k] to the shorter
        queue;
    end if
    if ∃(R, J) ∈ S[k] such that Qi[J] ≠ R[k] then
        Insert required ?rr instructions before
        opcode[k] in R[k];
    end if
    Recommend Qi[k] to all nodes in D[k] if
    dependency is register type between opcode[k]
    and each member in D[k];
    ∀(r, l) ∈ D[k], n[l]--;
end while
}
```

## یک مثال

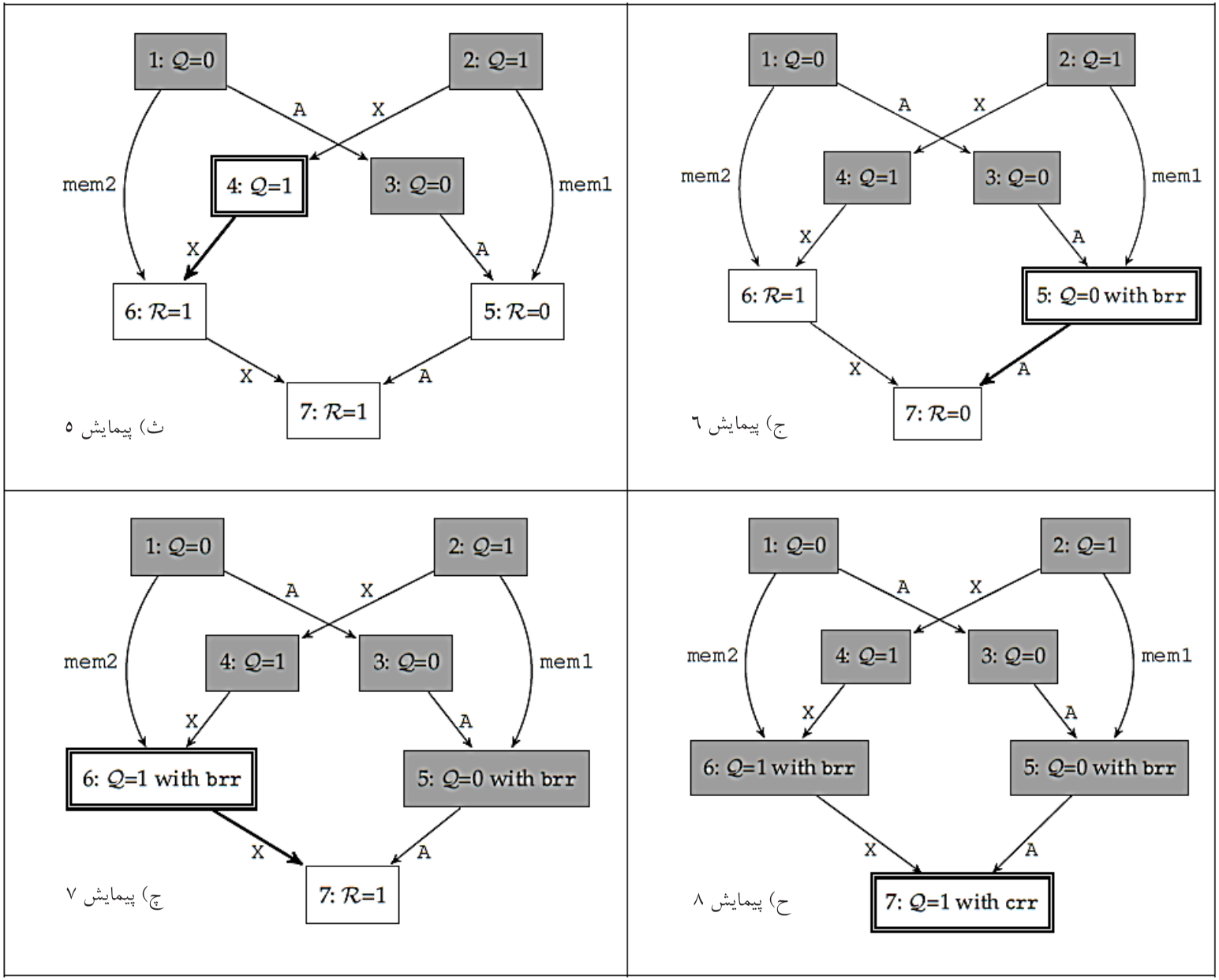
برای این‌که نشان دهیم این الگوریتم چگونه کار می‌کند، ما آن را بر کدی که در شکل ۱ آورده شده است و همچنین گراف وابستگی نظیر آن در شکل ۲-الف اعمال می‌کنیم. در شکل۲ مستعطیل‌های ضخیم نشان دهنده دستورالعملی است که در هر پیمایش پردازش می‌شود. بردارهای ساده نشان دهنده وابستگی مابین دستورالعمل‌هاست. بردارهای ضخیم نشان دهنده مسیر توصیه و نودهای خاکستری نشان دهنده دستوالعمل‌هایی است که الحاق شده‌اند و الگوریتم دیگر آن‌ها را پردازش نخواهد کرد.

در ابتدا اولین دستورالعمل در خط ۱ به صف $Q_0$ الحاق می‌شود و این صف به دستورالعمل سوم توصیه می‌شود (شکل ۲-ب). نود بعدی بدون مبدا، دستورالعمل دوم است و به صف کوتاه تر $Q_1$ الحاق می‌شود (شکل ۲-پ). اکنون دستورالعمل‌های سوم و چهارم به ترتیب به صف $Q_0$ و $Q_1$ پیشنهاد شده‌اند. بنابراین الگوریتم دستور سوم را به صف صفر و دستور چهارم را به صف یک الحاق می‌کند. در نظر گرفتن این حقیقت که دستور سوم و مبداش دستور اول هر دو به یک صف یکسان الحاق شده‌اند باعث می‌شود که هیچ دستور xrr نیاز نباشد(شکل ۲.ت). این مسئله برای دستورالعمل چهارم نیز صحیح است(شکل ۲-ث). از آن‌جا که صف $Q_0$ به دستورالعمل پنجم توصیه شده است (R[5]=0) این دستورالعمل به صف $Q_0$ می‌پیوندد اما یک مبدا آن(دستور دوم) در صف دیگر قرار گرفته است.

10 Recommendation




**شکل ۲- پیمایش‌های اجرای الگوریتم ۱ روی کد شکل ۱**



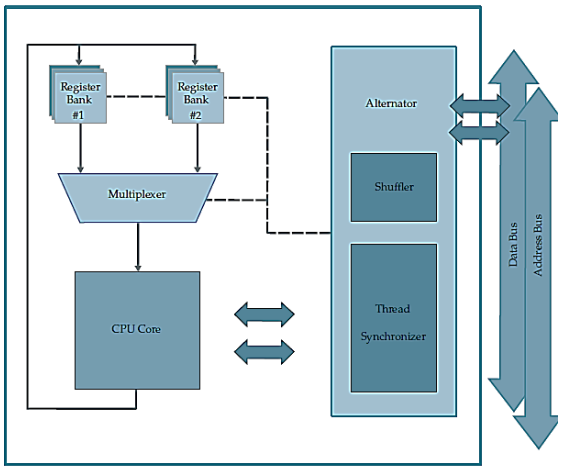
این بدین معنی است که یک دستور brr نیاز است تا مطمئن شویم که دستورالعمل ۲ قبل از دستورالعمل ۵ کامل شده است (شکل ۲-ج). همین عمل برای دستورالعمل ۶ اتخاذ می‌گردد (شکل ۲-چ). نهایتا دستورالعمل هفتم به صف $Q_0$ الحاق می‌شود. از آن‌جا که این دستور به مقدار ثبات‌های A و X نیازمند است که هر کدام در صفی متفاوت محاسبه می‌شوند یک دستور crr نیاز است تا در زمان اجرای دستورالعمل هفتم محتوای ثبات A را از $Q_1$ به $Q_0$ منتقل کند(شکل ۲-ح). جدول ۴ هر دو صف دستورالعمل‌ها را بعد از اتمام الگوریتم نشان می‌دهد. دو دستورالعمل brr و یک crr درج شده است. دستورالعمل brr LABEL_1_1 در صف $Q_0$ ما را مطمئن می‌کند که این صف منتظر خواهد ماند تا دستور ldx در صف $Q_1$ پردازش شود. brr LABEL_0_1 در $Q_1$ اثر مشابهی بر $Q_1$ دارد. دستورالعمل crr LABEL_0_2,A محتوای ثبات A را از $Q_0$ بعد از اجرای inca منتقل می‌کند.

جدول ۴. صف‌ها بعد از اجرای الگوریتم ۱

| Instruction | $Q_0$ | $Q_1$ |
|---|---|---|
| 1 | lda mem1 | ldx mem2 |
| 2 | LABEL_0_1: | LABEL_1_1: |
| 3 | inca | incx |
| 4 | LABEL_0_2: | brr LABEL_0_1 |
| 5 | brr LABEL_1_1 | stx mem1 |
| 6 | sta mem2 | crr LABEL_0_2,A |
| 7 | | mul |

## معماری TuP

TuP (شکل ۳) دو بانک ثباتی جداگانه دارد، یک تناوب‌گر[11] و یک هسته معمول شامل ALU و منطق کنترل.




شکل۳. معماری TuP

هر بانک ثبات، دو نسخه از ثبات‌های X،A، پرچم‌ها، نشان‌گر پشته و شمارنده برنامه را می‌سازد. مالتی پلکسر می‌تواند مسیر داده را بسته به این که کدام صف در حال اجراست، مابین دو بانک تغییر

11 Alternator

دهد. تناوب‌گر نیز شامل یک سوئیچ تصادفی[12] است که بین دو جریان دستورالعمل سوئیچ می‌کند و یک همگام کننده نخ[13] که بررسی می‌کند دستورالعمل واکشی شده xrr هست یا خیر. تناوب-گر با استفاده از xrr-ها دو صف را کنترل می‌کند تا به ترتیب درستی اجرا شوند. از آنجا که تصادفی بودن انتخاب صف ها اهمیت زیادی در مقاومت روش ارائه شده دارد برای دستیابی به سوئیچ تصادفی از [1] و [16] کمک گرفته شده است.

## پیاده سازی

### انتخاب هسته پردازنده

TuP با زبان VHDL پیاده‌سازی شد. طراحی شامل هسته 68HC05، تناوب‌گر، ROM، دو کپی از بانک ثبات‌ها و یک LFSR ۳۲ بیتی(به عنوان سوئیچ تصادفی). از آن‌جا که ما TuP را روی ModelSim6.3 شبیه سازی کرده‌ایم، هیچ پورت ورودی خروجی نیاز نداشتیم. همه ورودی‌ها و خروجی‌ها در دردون فایلی در شبیه ساز VHDL به سیستم داده می‌شد. پیاده‌سازی برای تولید توان مصرفی شبیه سازی شده اجرا شد تا بتوان روی آن ارزیابی کانال جانبی را به عمل آورد( بخش ۵ را ببینید). ما از تغییرات فاصله همینگ[14] همه ثبات‌ها به عنوان مدلی از توان مصرفی پویا استفاده کردیم[8,11]. فاصله همینگ تعداد تغییرات منطقی از صفر به یک و بالعکس در مولفه‌های مختلف پردازنده است که با توان مصرفی دستگاه رابطه مستقیم دارد.

همچنین ابزار نرم‌افزاری موازی سازی با C++ پیاده سازی شد. این ابزار یک برنامه 68HC05 را می پذیرفت و همه وابستگی‌های آن را در قالب چندین گراف استخراج می‌کرد سپس روش‌های موازی سازی را روی آن‌ها بکار می‌گرفت و نهایتا دو جریان باینری از دستورالعمل‌ها را تولید می‌کرد که بعدا در درون ROM سیستم قرار می‌گرفت.

### مقایسه مساحت و سرعت

جدول ۵ نتایج شبیه سازی یک هسته 68HC05 و همچنین TuP را روی Xilinx Spartan-6 FPGA نشان می‌دهد. جدول ۵ تنها به مساحت و سرعت و نه به امنیت پیاده‌سازی می‌پردازد. افزودن یک بانک ثبات اضافه و مدار تناوب‌گر تصادفی باعث شده است که مساحت حدود ۲۰٪ در ثبات‌های مصرفی و ۷٪ در LUTها اضافه گردد. همچنین سرعت تراشه حدود ۴٪ کاهش یافته است.

12 Shuffler Switch
13 Thread Synchronizer
14 Hamming Distance

جدول ۵. مساحت و سرعت روی Xilinx xc6slx4-3tqg144

| طراحی | ثبات | LUT | فرکانس |
|---|---|---|---|
| **Standard 68HC05** | **137** | **1288** | **120 MHz** |
| **TuP 68HC05** | **165** | **1375** | **115 MHz** |
| نسبت مابین طراحی‌ها | **1.02** | **1.07** | **0.96** |

## آزمایش روی AES

به منظور اطمینان از این مسئله که پیاده سازی AES روی 68HC05 استاندارد به CPA[۱۵] آسیب پذیر است، ما به TuP هنگامی که مانند همه پردازنده‌ها با یک صف کار می‌کرد حمله کردیم. پروسه حمله از [9] الهام گرفته شده که در آن تابع AddRoundKey در دور آخر رمزنگاری AES مورد حمله قرار می‌گرفت. AddRoundKey یک زیر برنامه در AES است که در آن کلید رمزنگاری با داده‌ی درحال رمزشدن یای انحصاری (xor) می شود. به علت این‌که ما از جزئیات پیاده سازی خود باخبر بودیم مشخص بود که در چه پالسی از ساعت اطلاعات کلید توسط تابع xor با داده یای انحصاری می‌شود. از آن‌جا که 68HC05 یک پردازنده هشت بیتی است، ۱۲۸ بیت زیرکلید مرحله نهایی AESتوسط یک حلقه ۱۶ دوری با داده یای انحصاری می‌شد که هر دور هشت بیت (از ۱۲۸ بیت) را پردازش می‌کرد. ما هدف را کشف اولین هشت بیت این کلید قرار دادیم.

با تمرکز بر طرح پیاده‌سازی شده به منظور کشف سیکل ساعت هدف مشخص شد که اولین xor درسیکل ۱۵۲۳۲-ام اتفاق می‌افتد و سپس با پانزده xor دیگر دنبال می‌شود که با فاصله ۲۴۵ پالس از یکدیگر رخ می‌دهند. ما در هر دور سعی کردیم که هشت بیت از کلید را با ارزیابی درستی ۲۵۶ (۲۸) حالت ممکن آن پیدا کنیم. برای ارائه شرح مختصری از پروسه حمله فرض کنید که D و C به ترتیب داده و متن رمزشده قبل و بعد از دور آخر AES هستند و K(0:7) اولین هشت بیت کلید (۱۲۸ بیتی) دور آخر باشند. ShiftRows و SubBytes نیز دیگر زیربرنامه‌ها دورآخر AES (به غیر از AddRoundKey) هستند که به ترتیب اجرا می‌شوند. پس:

$$C(0:7) = \text{ShiftRows}\left(\text{SubBytes}\left(D(0:7)\right)\right) \oplus K(0:7) \qquad (۱)$$

ما ۲۰۰ نمونه از عملیات رمزنگاری را اجرا کرده که هر نمونه داده متفاوت اما کلید یکسانی داشت و تمامی نمونه توان‌های مصرفی شبیه سازی شده را ذخیره کردیم. بنابراین در نمونه اجرای i، توان مصرفی Pi در سیکل ساعت هدف (۱۵۲۳۲-ام) در دور آخر الگوریتم AES که در آن دور Di (ورودی دور آخر) به Ci (خروجی دور) تبدیل می‌شود را می توان با رابطه زیر محاسبه کرد:

$$Pi = HD(Ci, Di) \qquad (۲)$$

برای هر یک از ۲۵۶ کلید هشت بیتی مورد حدس Kj(0:7) ،j∈{0,…,255}، تعداد ۲۵۶ داده ورودی محتمل Dij(0:7) را با استفاده از رابطه (۱) شکل می‌توان داد. فرض که فاصله همینگ بین Ci(0:7) و Dij(0:7) برای اجرای i و همچنین کلید مورد حدس j را hij بنامیم و توان مصرفی روی کلاک ۱۵۲۳۲-ام آن نمونه را Pi. هدف یافتن کلیدی است که بیشترین شباهت را میان توان مصرفی شبیه سازی شده دستگاه و فاصله‌های همینگ محاسبه شده را در تمامی اجراهای الگوریتم داشته باشد. ما همبسته‌ترین کلید مورد حدس را با استفاده از رابطه همبستگی Pearson بدست می‌آوریم:

$$\rho_j = \frac{1}{(n-1)\sigma_h\sigma_p}\sum_{i=0}^{200}(h_i - \bar{h})(P_i - \bar{P}) \qquad (۳)$$

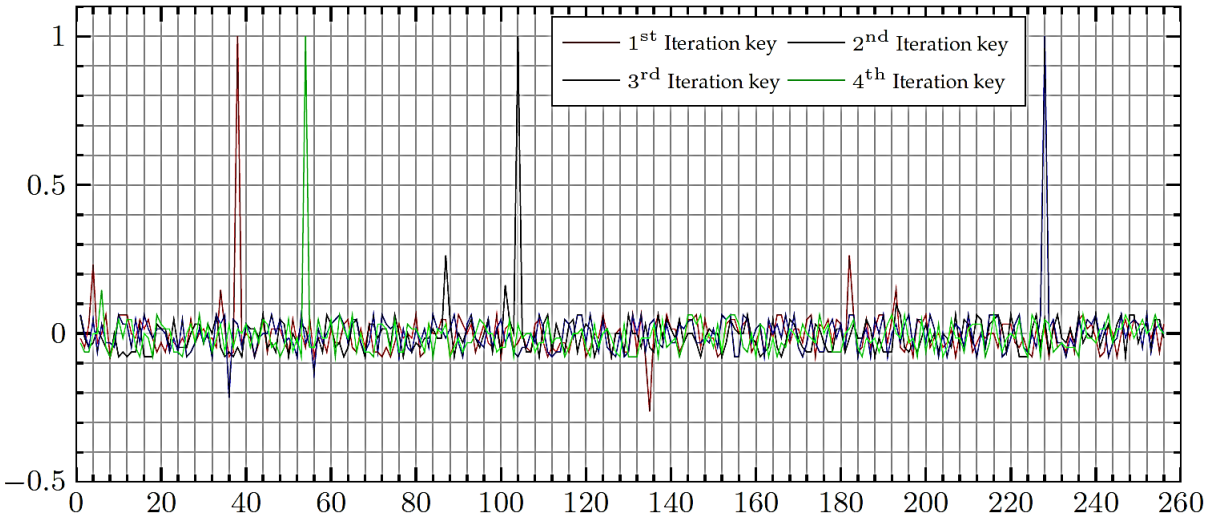
در این معادله، $\bar{P}$ میانگین همه نمونه توان‌های مصرفی در کلاک ۱۵۲۳۲-ام است و σp انحراف استاندارد همه Pi-ها است. $\bar{h}$ نیز میانگین همه فاصله همینگ های اجراهای مختلف با انحراف استاندارد σh است.

همان گونه که در شکل ۵ می‌بینیم، کلید صحیح یک قله متمایز در مقدار ρ تولید می‌کند. در این شکل چهار بایت از زیر کلید در رنگ‌های مختلف نمایش داده شده اند. هر قله نماینده همبسته ترین مقدار کلید برای هر یک از چهار بایت است.

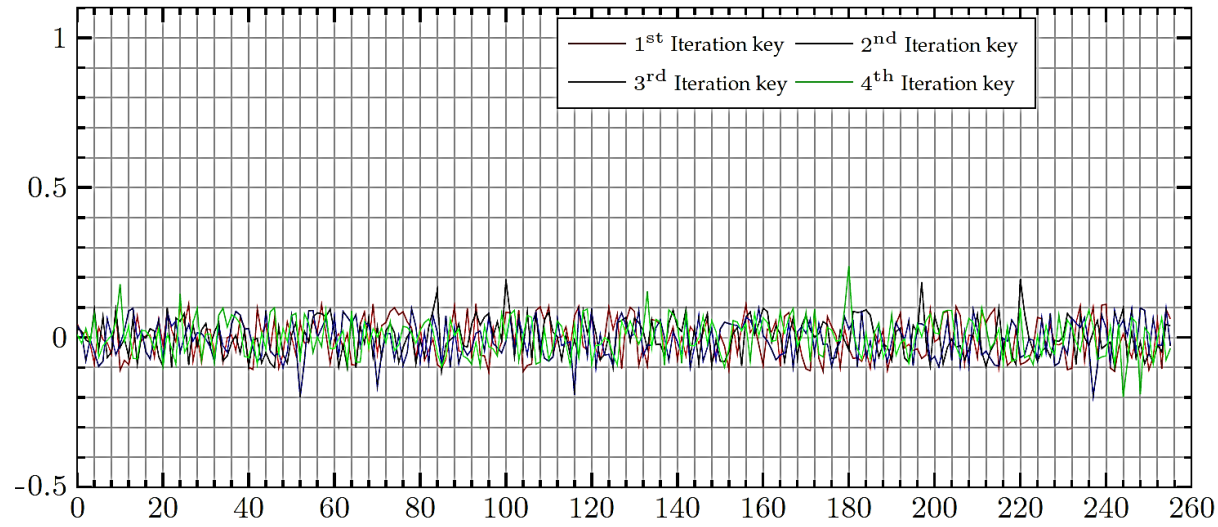
بعد از این، همان حمله روی TuP انجام شد. شکل ۶ نمایشگر CPA به چهار بایت از کلید نهایی در TuP بوسیله همان داده‌ها و همان کلید است. برخلاف سیستم حفاظت نشده، هیچ قله‌ای برای TuP قابل مشاهده نیست. شکل ۶ نیز نشان می‌دهد که چگونه سه عمل eor از دور نهایی AES (اول، دوم و هشتم) برای ۲۰۰ نمونه اجرا در بین سیکل‌های پردازنده پخش شده اند. اجرای اولین Exclusive-OR در حوالی زمان قبلی خود اتفاق می‌افتد اما راهی نیست که بتوان دقیقا گفت که کجا اتفاق می‌افتد و این درحالی است که eor-های دیگر ممکن است قبل یا بعد از آن اتفاق بیافتند که خود باعث محو و درهم شدن الگوی توان مصرفی دستگاه می‌شود. بکارگیری حمله مذکور ظاهرا نتایج غیرهبسته‌ای را برای همه ۲۵۶ کلید ممکن تولید می‌کند. بنابراین TuP می‌تواند در مقابل CPA موثر باشد.

برای ارزیابی این‌که نهایت مقاومت TuP تا چه حد ادامه خواهد یافت، تعداد نمونه‌های اجرای الگوریتم را بالا بردیم تا مقدار همبستگی قابل ملاحظه‌ای را در پیرامون سیکل ساعت هدف مشاهده کنیم.

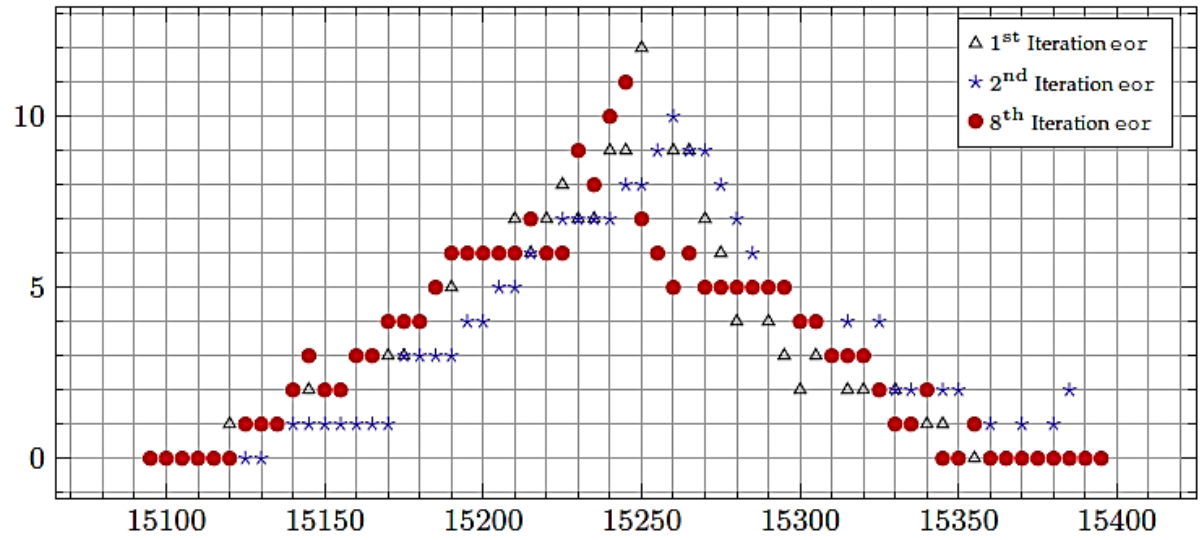
15 Correlation Power Analysis




شکل ۴ . CPA : حدس صحیح چهار کلید هشت بیتی (68HC08 استاندارد)




شکل ۵ . CPA: سعی در پیدا کردن کلید صحیح برای چهار کلید هشت بیتی (TuP 68HC05)




شکل ۶ . توضیح زمانی سه eor در دور نهای AES ۲۰۰ نمونه اجرا

[11] نشان می‌دهد که اگر ضریب همبستگی در یک حمله CPA به مقدار Tn بار کاهش پیدا کند، تعداد اجراهای مورد نیاز برای یک حمله موفق به مقدار $Tn^2$ برابر افزایش خواهد یافت. همان‌گونه که شکل ۴ و شکل ۵ قابل مشاهده است ضریب همبستگی میانگین برای چهار کلید از ۱ به ۰/۰۶ کاهش پیدا کرده است. ما انتظار داشتیم که تعداد اجراهای مورد نیاز برای حمله موفق به TuP به میزان ۲۷۷ برابر افزایش یابد(۲۰۰×۲۷۷= ۵۵۴۰۰ نمونه اجرا). جدول ۶ تعداد کلیدهایی که با موفقیت حدس زده شده‌اند را درکنار تعداد نمونه اجراهای مورد نیاز برای کشف آن‌ها، نشان می‌دهد. کاملا واضح است که برای این پیاده‌سازی حداقل ۶۰۰۰۰ اجرا نیاز است تا بتوان بخش‌هایی از کلید را کشف کرد و برای کشف کل کلید حداقل ۹۰۰۰۰ نمونه اجرا لازم است. بدین ترتیب TuP میزان تلاش نفوذگر را برای حمله CPA به طور قابل ملاحظه‌ای افزایش می‌دهد.

جدول ۶. ارزیابی نهایت مقاومت TuP

| تعداد کلید کشف شده | تعداد نمونه اجرای **AES** |
|---|---|
| ۰ | ۵۰۰۰۰ |
| ۲ | ۶۰۰۰۰ |
| ۵ | ۷۰۰۰۰ |
| ۱۴ | ۸۰۰۰۰ |
| ۱۶ | ۹۰۰۰۰ |

## نتیجه گیری

در این مقاله ما یک معماری پردازنده را پیشنهاد دادیم که می‌تواند دو صف از دستورالعمل‌ها را لابه‌لای هم به صورت تصادفی اجرا کند که این دوصف هر دو با هم یک الگوریتم مشترک را تکمیل خواهند کرد. از آن‌جا که ترتیب اجرای دستورالعمل‌ها از هر اجرا تا اجرای دیگر تفاوت می‌کند، الگوی توان مصرفی قابل پیش بینی نیست و می‌تواند در مقابل CPA مقاومت کند. همچنین ما یک مجموعه از روش‌های کامپایلری را بکار بردیم تا ابزاری بسازیم که بتواند یک برنامه نرم‌افزاری را بپذیرد و دو صف از دستورالعمل‌ها را تولید کند که اجرای لابه‌لای آن‌ها معادل برنامه اولیه است. ما توانمندی روش‌مان را با راه‌اندازی یک حمله آنالیز توان تفاضلی آزمودیم.

## مرجع‌ها


[1] E. Brier, H. Handschuh, C. Tymen. A fast primitives for internal data scrambling in tamper resistant hardware, Cryptographic Hardware and Embedded Systems - CHES 2001, volume 2162 of Lecture Notes in Computer Science, pp. 16-28, Springer, 2001.

[2] C. Clavier, J-S. Coron, N. Dabbous. Differential power analysis in the presence of hardware countermeasures, Cryptographic Hardware and Embedded Systems - CHES 2000, volume 1965 of Lecture Notes in Computer Science, pp. 252-263. Springer, 2000.

[3] J-S. Coron, I. Kizhvatov. An efficient method for random delay generation in embedded software, Cryptographic Hardware and Embedded Systems - CHES 2009, volume 5747 of Lecture Notes in Computer Science, pp. 156-170. Springer, 2009.

[4] J-S. Coron, I. Kizhvatov. Analysis and improvement of the random delay countermeasure of CHES 2009, Cryptographic Hardware and Embedded Systems - CHES 2010, volume 6225 of Lecture Notes in Computer Science, pp. 95-109. Springer, 2010.

[5] K. Kennedy, J. Allen. Optimizing compilers for modern architectures: a dependence-based approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[6] P. Kocher, J. Jaffe, B. Jun. Differential power analysis. Advances in Cryptology : Proceedings of CRYPTO '99, volume 1666 of Lecture Notes in Computer Science, pp. 388-397. Springer-Verlag, 1999.

[7] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, R. Barua. Automatic parallelization in a binary rewriter. Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43), pp. 547-557, 2010.

[8] K. Tiri, I. Verbauwhede. Simulation models for side-channel information leaks, Proceedings of the 42nd annual Design Automation Conference – DAC '05 pp. 228-233, 2005.

[9] Brier,E.,Clavier,C.,Andolivier. Correlation power analysis with a leakage model. InProceedings of Cryptographic Hardware and Embedded Systems (CHES). 16–29. 2004.

[10] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. Advances in Cryptology, CRYPTO '96, Springer LNCS 1109, pp 104–113, 1996.

[11] Mangard, S., Oswald, E., Popp, T. Power Analysis Attacks - Revealing the Secrets of Smart Cards. 2007.

[12] Kamal, A. A., Youssef, A. M. An area-optimized implementation for AES with hybrid countermeasures against power analysis. In Proceedings of the International Symposium on Signals, Circuits and Systems (ISSCS). 1–4. 2009.

[13] Madlener, F., Stoettinger, M., and Huss, S. A. Novel hardening techniques against differential power analysis for multiplication in GF(2n). In Proceedings of the International Conference on Field-Programmable Technology –ICFPT.pp 328–334. 2009.

[14] Tillich, S.,Herbst, C., and Mangard, S. Protecting AES software implementations on 32-bit processors against power analysis. In Proceedings of Applied Cryptography and Network Security (ACNS). 141–157. 2007.

[15] May, D., Muller, H. L., and Smart, N. P. Non-deterministic processors. In Proceedings of Information Security and Privacy (ACISP). volume. 2119, 115–129. 2001.

[16] V. Bagini, G. Morgari. Keyed permutations for fast data scrambling, European Transactions on Telecommunications, volume 17, issue 1, pp. 1-9, 2006.

[17] Yu Bo,Li Xiangyu,Chen Cong,Sun Yihe,Wu Liji and Zhang Xiangmin.An AES chip with DPA resistance using hardware-based random order execution[J]. Journal of Semiconductors, 33(6):065009-8. 2012.

[18] A. Galip Bayrak, N. Velickovic, P. Ienne and W. Burleson. An architecture independent instruction shuffler to protect against side-channel attacks. In TACO, 8(4), 2012.