بسمه تعالی

دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

گزارش سمینار کارشناسی ارشد

تشخیص دژافزار با استفاده از اجرای پویانمادین

نگارش:

محمود اقوامی پناه

استاد راهنما:

دکتر بابک صادقیان

تابستان ۱۳۹۶

چکیده: دژافزار کدی مخرب است که باهدف آسیب‌رسانی به سامانه‌های رایانه‌ای یا انجام اعمال ناخواسته نظیر سرقت اطلاعات شخصی یا مالی، نمایش تبلیغات ناخواسته، جاسوسی و باج‌گیری طراحی می‌شود. با گسترش استفاده از دستگاه‌های رایانه‌ای و گوشی‌های هوشمند تولید و پیچیدگی دژافزار ها نیز افزایش چشم‌گیری داشته است. برای تشخیص دژافزار ها و تحلیل آن‌ها، دو رویکرد متفاوت تحلیل ایستا و پویا موجود است.

تحلیل ایستا روشی سریع، ارزان و سبک برای یافتن الگوهای مخرب در برنامه‌های کاربردی بدون نیاز به اجرای آن‌ها است. اما این تحلیل به دلیل محدودیت‌های ذاتی که در اثر اجرا نکردن برنامه دارد، دارای مثبت کاذب بالایی است. در تحلیل پویا، برنامه‌های کاربردی در یک محیط کنترل‌شده اجراشده و رفتارشان مورد ارزیابی قرار می‌گیرد. تحلیل پویا اگرچه با نقاط ضعف تحلیل ایستا روبه‌رو نیست ولی تنها قادر است در هر بار اجرا یک مسیر اجرایی از برنامه را مورد ارزیابی قرار دهد. این چالش موجب می‌شود تا مسیرهای اجرایی مختلف برنامه مورد تحلیل قرار نگیرند و یک کاربر بدخواه بتواند دژافزار خود را به‌عنوان یک برنامه‌ی کاربردی معرفی کند.

از سویی دیگر در سال‌های اخیر،روش‌های جدید آزمون نرم‌افزار برای تحلیل برنامه‌ها موردتوجه قرارگرفته است. به‌عنوان نمونه اجرای پویانمادین که در سال ۲۰۰۵ برای اولین بار در ابزار آزمون DART معرفی شد پس‌ازآن توجه بسیاری را به خود جلب کرد. پس از نوآوری علمی ابزار DART تحقیقات فراوانی برای استفاده از اجرای پویانمادین و بهبود آن انجام گرفت. این تحقیقات شامل اجرای پویانمادین برای آزمون نرم‌افزار، کشف آسیب‌پذیری، تشخیص نقض حریم خصوصی، تشخیص و تحلیل دژافزار می‌شد. چالش علمی اجرای مسیرهای محدود برنامه در تحلیل پویا با استفاده از اجرای پویانمادین قابل‌حل است اگرچه استفاده از اجرای پویانمادین خود چالش‌های علمی جدیدی را موجب می‌شود.

ما در این گزارش در ابتدا به بررسی اجرای نمادین و روش‌های اجرای نمادین سنتی و نوین پرداخته‌ایم. در بررسی اجرای نمادین نوین، روش‌های پویانمادین و EGT را شرح داده و چالش‌های پیش رو برای آن‌ها را بررسی کرده‌ایم. پس‌ازآن با بررسی ابزارهای پویانمادین نظیر,DART Mayhem وDriller به مقایسه این ابزارها و نوآوری‌های علمی هریک پرداخته‌ایم. در ادامه با اشاره به چالش‌های علمی موجود برای تحلیل برنامه‌های کاربردی اندروید در فصلی جداگانه به بررسی و مقایسه ابزارهای ACTEV ، Condroid، SIG-droidو AppIntent که اجرای پویانمادین را برای پلتفرم اندروید پیاده‌سازی کرده‌اند پرداخته‌ایم. در فصل پنجم به بررسی روش‌های تشخیص دژافزار با تمرکز بر دژافزار های اندرویدی و در فصل ششم به بررسی ابزارهای پیشینی که از اجرای پویانمادین برای تشخیص یا تحلیل دژافزار بهره برده اندپرداخته ایم.

در فصل هفتم مباحث مطرح‌شده را جمع‌بندی نموده و بابیان پروژه‌ی کارشناسی ارشد به راهکارهای پیش رو برای حل مسئله‌ی باز مطرح‌شده پرداخته‌ایم.

واژه‌های کلیدی: اجرای نمادین، اجرای پویانمادین، تشخیص دژافزار، دژافزار های اندرویدی

فهرست مطالب

فهرست شکل‌ها

فهرست جداول

فصل اول :

مقدمه

۱-۱-پیشینه موضوع

در سال‌های اخیر گسترش استفاده از سیتم های رایانه‌ای و گوشی‌های هوشمند موجب گسترش دژافزار ها در این حوزه شده است. روش ابتدایی تشخیص این دژافزار ها، تحلیل ایستا بود، این تحلیل با بررسی کد دژافزار بدون هرگونه اجرا سعی می‌کرد تا رفتارهای مخرب یک دژافزار را از یک برنامه‌ی عادی متمایز نماید.

در ابتدا و زمانی که دژافزار های رایانه‌ای بسیار ساده بودند، استفاده از تحلیل ایستا برای تشخیص آن‌ها بسیار مؤثر بود.اما محدودیت‌های ذاتی موجود در تحلیل ایستا موجب می‌شد تا قادر به شناسایی دژافزار های پیچیده نباشد. به‌عنوان‌مثال اگر دژافزار از تکنیک مبهم سازی استفاده می‌کرد و یا کد مخرب خود را در زمان اجرا و به‌صورت پویا بارگذاری می‌نمود، تحلیل ایستا قادر به تشخیص دژافزار نبود.

تحلیل پویا در راستای برطرف کردن نقاط ضعف تحلیل ایستا ارائه شد و در سال‌های بعد از ظهور تحلیل ایستا به‌عنوان یک نوآوری علمی موردتوجه قرار گرفت. با توجه به آن‌که در تحلیل ایستا دژافزار اجرا می‌شد، مثبت کاذب ناشی از روش‌های مبهم سازی و یا بارگذاری پویای کد به‌طورکلی از بین می‌رفت. اما نقطه‌ضعف اصلی ابزارهای تحلیل پویا در این بود که در هر بار اجرا تنها یک مسیر اجرایی را می‌آزمودند. این مسیر اجرایی در ساده‌ترین حالت بر اساس ورودی‌های دلخواه انتخاب و اجرا می‌شد ازاین‌رو احتمال انتخاب مسیرهای اجرایی متفاوت بسیار کم بود. درواقع دژافزار های بسیاری وجود داشت که اگرچه مسیرهای اجرایی دارای کد مخرب را شامل می‌شدند اما در تحلیل پویا به‌عنوان برنامه‌ی کاربردی شناخته می‌شدند زیرا مسیرهای اجرایی محدودی از آن‌ها اجرا می‌شد و باقی مسیرهای کد مورد ارزیابی قرار نمی‌گرفت.

از سویی دیگر مهندسین آزمون نرم‌افزار چهار سطح مختلف آزمون را برای آزمون یک برنامه در نظر می‌گیرند. این چهار سطح عبارت‌اند از:

۱. آزمون واحد[۱]
۲. آزمون تجمیع[۲]
۳. آزمون اطمینان از کیفیت[۳]

[۱] Unit Testing
[۲] Integration Testing
[۳] Quality Assurance Testing

۴. آزمون تائید کاربر[۱]

فارغ از سطح آزمون برای یک نرم‌افزار، ما نیازمند آن هستیم که داده‌های تولیدشده برای آزمون به‌گونه‌ای هوشمندانه تولید شوند که پوشش کد مناسبی را تأمین نمایند. برای این کار روش غیرهوشمندانه، تولید ورودی دلخواه خواهد بود که لزوماً کارا نیست. ازاین‌رو در دهه‌ی ۹۰ میلادی روش آزمون نمادین مطرح شد. در این روش ورودی‌های برنامه به‌صورت نمادین در نظر گرفته می‌شد و شروط موجود در یک مسیر اجرا استخراج می گشت. با ارائه‌ی این شروط به حل‌کننده‌ی قید، مسیر اجرایی جدیدی تولید می‌شد که از امکان‌پذیر بودن آن مطمئن بودیم. در فصل دوم، برای اجرای نمادین[۲] به‌طور دقیق، موردبررسی قرار خواهد گرفت.

مشکل عمده‌ای که در مورداجرای نمادین وجود داشت ناتوانی حل‌کننده‌های قید برای حل قیود پیچیده و عدم قابلیت گسترش‌پذیری بود. پس‌ازآن در سال ۲۰۰۵ میلادی در راستای حل مشکلات اجرای نمادین و در یک نوآوری علمی اجرای پویانمادین معرفی شد. در این روش اجرای نمادین با استفاده از اجرای عددی کاراتر شده بود. درواقع هر جا که حل‌کننده‌ی قید قادر به حل قید مسیر نبود، مسیر به‌طور عددی اجرا می‌شد. به‌جز آزمون نرم‌افزار که هدف اصلی اجرای پویانمادین بود، کاربردهای دیگری نیز موردبررسی قرار گرفت. تشخیص آسیب‌پذیری‌های نرم‌افزاری، تشخیص نقض حریم خصوصی، تشخیص و تحلیل دژافزار از این موارد بودند.

ما در این گزارش ، اجرای پویانمادین برای تشخیص دژافزار را بررسی کرده‌ایم و سیر تاریخی و علمی لازم برای بیان این مطلب را موردبحث قرار داده‌ایم.

۱-۲-ساختار گزارش

این پژوهش در هفت فصل تدوین‌شده است. در فصل اول مقدمات موضوع موردبحث قرارگرفته است.

در فصل دوم اجرای نمادین شرح داده‌شده است.هم‌چنین دو روش نوین در اجرای نمادین بانام‌های روش EGT و روش اجرای پویانمادین مطرح می‌شوند.

[۱] User Acceptance Testing
[۲] Symbolic Execution

در فصل سوم، در اجرای پویانمادین برای تحلیل برنامه‌ها بررسی می‌شوند. ابزارهای بیان‌شده عبارت‌اند از: DART، Mayhem و Driller. این ابزار در زیر بخش آخر این فصل با یکدیگر مقایسه شده و مزایا و معایب و نوآوری‌های علمی هریک بیان می‌شود.

در فصل چهارم، در اجرای پویانمادین برای تحلیل برنامه‌های اندروید بررسی می‌شوند.علت اینکه این ابزارها یک‌فصل جداگانه را به خود اختصاص داده‌اند این است که تحلیل برنامه‌های اندروید و اجرای پویانمادین برای پلتفرم اندروید با چالش‌های علمی جدیدی مواجه است که در ابتدای همین فصل شرح داده‌شده است. ابزارهای بیان‌شده در این فصل عبارت‌اند از: ACTEV، Condroid ، Sig-Droidو APPIntent. این ابزارها در زیر بخش آخر این فصل با یکدیگر مقایسه شده و مزایا و معایب و نوآوری‌های علمی هریک بیان می‌شود.

در فصل پنجم، بیان می‌شوند. روش‌های تحلیل ایستا و پویا، شرح داده‌شده و در پایان فصل نیز به‌طور خاص روش‌های تشخیص دژافزار های اندرویدی موردمطالعه قرار می‌گیرد.

در فصل ششم، در اجرای پویانمادین برای تحلیل و تشخیص دژافزار بررسی می‌شوند. در این فصل سه کار پژوهشی مورد ارزیابی قرارگرفته‌اند. در کار اول برواملی و همکاران[۲۲] به ارائه‌ی یک روش برای تشخیص خودکار بات نت ها با استفاده از اجرای پویانمادین پرداخته‌اند. در کار دوم کروگل و همکاران [۲۳] با ارائه یک راهکار مبتنی بر اجرای نمادین بهبودیافته و استفاده از شبیه‌ساز به ارائه‌ی بک روش نو برای تحلیل دژافزار پرداخته‌اند. در کار سوم که Bitscope نام دارد با استفاده از اجرای نمادین چارچوبی برای تحلیل دژافزار ارائه‌شده که می‌تواند مسیرهای متفاوت یک دژافزار را پیمایش کند.

در پایان در فصل هفتم، به ۷-۱- می‌پردازیم. در مورد آینده بحث، مسائل باز موجود در این حوزه و همچنین پروژه کارشناسی ارشد و مراحل پژوهش آن مطالبی عنوان خواهد شد. درنهایت استفاده‌شده در فصل هشتم معرفی می‌شوند.

فصل دوم:

اجرای پویا نمادین

۲-۱- اجرای نمادین[۱] سنتی

اجرای نمادین در سال‌های اخیر توجه زیادی را به خود جلب کرده است. این روش به‌عنوان یک روش مؤثر برای تولید مورد آزمون[۲] و پوشش کد[۳] برای پیداکردن خطاهای عمیق در نرم‌افزارهای پیچیده مورداستفاده قرار می‌گیرد. ایده کلیدی اجرای نمادین بیش از سه دهه قبل معرفی‌شده است[۲] اما در سال‌های اخیر با توجه به پیشرفت قابل‌توجه در ارضای قیود[۴] و مقیاس‌پذیری[۵] رویکردهای پویا این روش عملی شده و موردتوجه قرارگرفته است.

در این فصل، ما بر اساس پژوهش[۱] یک مرور کلی بر روی اجراهای نمادین ارائه می‌دهیم. سپس تکنیک‌های اجرای نمادین سنتی و مدرن را موردبررسی قرار می‌دهیم و چالش‌های آن‌ها در مورد اکتشاف مسیر،حل مسئله قیود و مدل‌سازی حافظه را موردبررسی قرار می‌دهیم.

هدف کلیدی اجرای نمادین درزمینهٔ تست نرم‌افزار این است که در یک مدت‌زمان قابل‌قبول مسیرهای مختلف برنامه را کشف و اجرا نماید. اجرای نمادین باید به ازای هر مسیر اجرایی دو مورد زیر را بررسی نماید.

۱. مجموعه‌ای از مقادیر ورودی را به‌گونه‌ای تولید کند تا آن مسیر خاص اجرا گردد.
۲. وجود انواع خطاها را در آن مسیر بررسی نماید.خطاهایی نظیر استثنائات[۶] غیرقانونی، وجود آسیب‌پذیری‌ها، و خرابی حافظه[۷].

ایده کلیدی در اجرای نمادین، استفاده از مقادیر نمادین به‌جای مقادیر داده‌های ورودی و نشان دادن مقادیر متغیرهای برنامه با استفاده از عبارات نمادین است. درنتیجه مقدار خروجی محاسبه‌شده توسط یک برنامه به‌صورت یک تابع از مقادیر نمادین ورودی بیان می‌شود. درواقع در تست نرم‌افزار از اجرای نمادین برای تولید داده‌ی آزمون ورودی جهت اجرای یک مسیر مشخص از برنامه استفاده می‌شود.

---

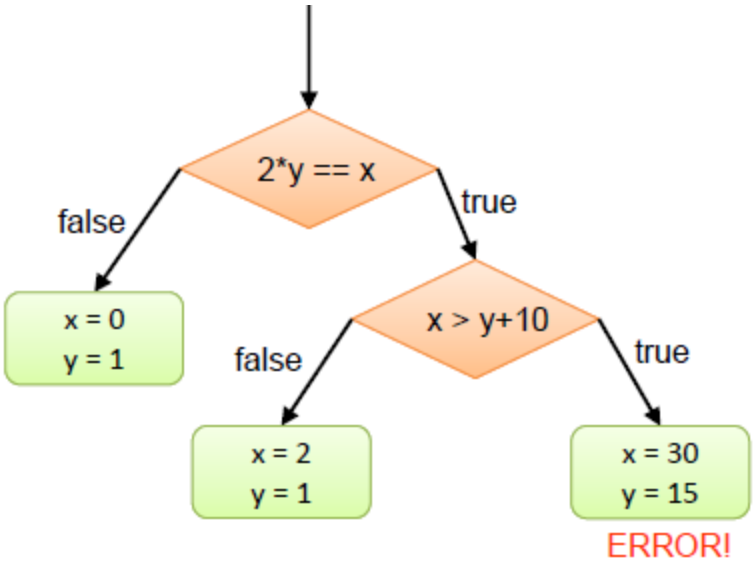
۱ Symbolic Execution
۲ Test case
۳ Code coverage
۴ Constraint satisfiability
۵ Scaleability
۶ Exception
۷ Memory corruption

هر مسیر اجرایی از برنامه یک دنباله از مقادیر بولی درست و غلط برای قیود برنامه است.، که در آن مقدار بولی درست در موقعیت i ام نشان‌دهنده‌ی اجراشدن بخش then در iامین شرط برنامه است. همه مسیرهای اجرایی یک برنامه را می‌توان با استفاده از یک درخت، به نام درخت اجرا نشان داد.

به‌عنوان‌مثال در شکل ۱ تابع Testme() سه مسیر اجرایی مختلف دارد. این سه مسیر را می‌توان در درخت اجرای شکل۲ مشاهده کرد. مجموعه‌ی ورودی‌های {x = ۰, y = ۱}, {x = ۲, y = ۱} {x = ۳۰, y = ۱۵} برای پیمودن تمام‌مسیرهای اجرای مختلف شکل۱ کافی است. هدف نهایی اجرای نمادین آن است که این مجموعه داده‌های ورودی را در زمان قابل‌قبول تولید کند و با اجرای برنامه در همه ی مسیرهای اجرا از عملکرد صحت آن مطمئن شود.

```
1   int twice (int v) {
2           return 2*v;
3   }
4
5   void testme (int x, int y) {
6           z = twice (y);
7           if (z == x) {
8                   if (x > y+10)
9                           ERROR;
10                  }
11          }
12  }
13
14  /* simple driver exercising testme() with sym inputs */
15  int main() {
16          x = sym_input();
17          y = sym_input();
18          testme(x, y);
19          return 0;
20  }
```

شکل ۱ -مثالی از یک تابع ساده برای نمایش اجرای نمادین



شکل ۲-درخت اجرای مثال شکل ۱

اجرای نمادین، یک وضعیت نمادین[۱] از سیستم را نگه می‌دارد که با متغیر σ نمایش داده می‌شود. این وضعیت نشان می‌دهد که چگونه متغیرها به مقادیر نمادین نگاشت شده‌اند. به‌علاوه اجرای نمادین، یک قید مسیر نمادین[۲] دارد که با استفاده از PC نمایش داده می‌شود. PC یک فرمول مرتبه اول برپایه ی عبارات نمادین است. در ابتدای اجرای نمادین، σ با یک جدول خالی از نگاشت‌ها مقداردهی اولیه می‌شود. PC نیز در ابتدا با مقدار True مقداردهی اولیه می‌شود. σ و PC در طول اجرای نمادین به‌روزرسانی می‌شوند. در پایان یک اجرای نمادین از یک مسیر اجرا، PC با استفاده از یک حل‌کننده‌ی قید[۳] حل می‌شود تا بتوان با استفاده از آن مجموعه‌ی داده‌های ورودی را تولید کرد.

به‌عنوان‌مثال اجرای نمادین برای تابع ساده‌ی شکل۱ با یک σ خالی و PC حاوی مقدار True آغاز می‌شود. به ازای هر دستور خواندن مقدار ورودی، در برنامه یعنی دستوری مشابه قالب var=sym input() ، اجرای نمادین یک نگاشت از مقدار متغیر به مقدار نمادین « var -> s » را به σ اضافه می‌کند. به‌عنوان‌مثال اجرای نمادین برای دو خط ابتدایی تابع main() از کد شکل ۱ منجر به‌اضافه شدن دو نگاشت به مقادیر نمادین در σ می‌شود.

$$\sigma = \{x \mapsto x_0, y \mapsto y_0\}$$

[۱] Symbolic state
[۲] Symbolic path constraint
[۳] Constraint solver

پس‌ازآن در هر خط از کد که متغیر x مورداستفاده قرار گیرد، مقدار نمادین x۰ به‌جای آن نگاشت می‌شود. مثلاً پس از اجرای خط ۶ از تابع مذکور، σ به‌صورت زیر خواهد بود.

$$\sigma = \{x \mapsto x_0, y \mapsto y_0, z \mapsto 2y_0\}$$

پس از اجرای خط ۷، دو نمونه[۱] از اجرای نمادین ایجاد می‌گردد. یکی از آن‌ها با شرط $x_0 = 2y_0$ و دیگری با شرط $x_0 \neq 2y_0$ می‌باشد. به‌طور مشابه پس از خط ۸ برنامه نیز دو نمونه از اجرای نمادین با شرط $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ و $(x_0 = 2y_0) \wedge (x_0 <= y_0 + 10)$ ایجاد می‌شود.

اگر نمونه‌ی اجرای نمادین با دستور *Terminate* و یا *error* مواجه شود، این نمونه متوقف‌شده و با استفاده از یک حل‌کننده‌ی قید، یک مجموعه داده‌ی ورودی برای رسیدن به این مسیر اجرا تولید می‌گردد.

اجرای برنامه‌هایی که دارای حلقه یا توابع بازگشتی باشند ممکن است منجر به تولید نامتناهی مسیر اجرا شود.به‌عنوان‌مثال اگر شرط پایان حلقه به‌صورت نمادین در خود حلقه مقداردهی گردد نامتناهی مسیر اجرا تولید می‌شود. شکل۳ نشان‌دهنده‌ی یک مثال با نامتناهی مسیر اجرا است. هر یک از این مسیرهای اجرا شامل تعداد دلخواهی مقدار *True* و یک مقدار پایانی *False* می‌باشد.

```
1   void testme_inf () {
2           int sum = 0;
3           int N = sym_input();
4           while (N > 0) {
5               sum = sum + N;
6               N = sym_input();
7           }
8   }
```

شکل ۳ - یک تابع ساده با نامتناهی مسیر اجرا

یک نقطه‌ضعف اساسی در اجرای نمادین این است که اگر فرمول نمادین به‌دست‌آمده از یک مسیر اجرا به‌درستی توسط حل‌کننده‌ی قید قابل‌حل نباشد، نمی‌توان ورودی‌های مناسبی برای اجرای آن مسیر تولید کرد. به‌عنوان‌مثال اگر تابع twice() به‌جای تابع شکل۱، تابع موجود در شکل ۴ باشد در آن صورت حل‌کننده‌ی قید نمی‌تواند عبارات غیرخطی را حل کند بنابراین اجرای نمادین نمی‌تواند مقادیر داده‌ی ورودی را برای مسیر اجرای twice() را تولید کند.

[۱] instance

```
1   int twice (int v) {
2           return (v*v) % 50;
3   }
```

شکل ۴-تابع twice

۲-۲- اجرای نمادین نوین

در این بخش دو روش از اجرای نمادین نوین را بررسی می‌کنیم. مزیت این روش‌ها نسبت به روش‌های سنتی ترکیب اجرای نمادین با تکنیک های عددی[۱]می‌باشد.

۲-۲-۱-روش پویانمادین[۲]

واژه Concolic از ترکیب دو واژه Concrete و Symbolic به وجود آمده است. در اجرای پویانمادین دو حالت نگهداری می‌شود، یکی حالت نمادین و دیگری حالت عددی. در حالت عددی مقادیر عددی هر متغیر نگهداری می‌شود در حالت نمادین هم‌مقدار نمادین هر متغیر یا عبارت نگهداری می‌شود.

در اجرا پویانمادین به‌صورت همزمان برنامه به‌صورت عددی و نمادین اجرا می‌گردد. برای اجرای عددی لازم است تا ورودی‌های برنامه مقداردهی اولیه شوند این مقادیر می‌تواند به‌صورت دلخواه انتخاب شوند. درطی اجرای اولیه، بیان‌های شرطی قیدهای نمادین نگهداری می‌شوند. درنهایت این قیدها به حل‌کننده‌ی قید داده می‌شوند تا ورودی عددی برای اجرای بعدی فراهم گردد. این اجراها تا زمانی که همه مسیرها پیمایش شوند یا اینکه زمان مقرر به پایان برسد ادامه پیدا می‌کند.

برای مثال در کد شکل ۱**Error! Reference source not found.** مقدار دلخواه {x=۲۲, y=۷} انتخاب می‌شود و برنامه هم برای اولین بار به‌صورت نمادین و هم‌عددی اجرا می‌گردد. این اجرا در خط ۷ برنامه شاخه else را انتخاب می‌کند، اجرای نمادین نیز قید $(2y. \neq x.)$ را ایجاد می‌کند. در حل‌کننده‌ی قید عبارت نهایی مکمل می‌شود و شرط جدید $(2y. = x.)$ ایجاد می‌گردد. سپس حل‌کننده‌ی قید آن را حل می‌کند و {x=۲, y=۱} به‌عنوان ورودی جدید برای اجرای بعدی تولید می‌گردد. با این ورودی جدید در خط ۷، شاخه then اجرا و در خط ۸ شاخه else انتخاب می‌شود. به‌این‌ترتیب مسیر جدیدی از برنامه ایجاد و اجرا می‌گردد. در این حالت

[۱] Concrete
[۲] Concolic

$(x. > y. + ۱۰) \wedge (۲y. = x.)$ به‌عنوان PC جدید به حل‌کننده قید داده می‌شود و درنهایت ورودی {x=۳۰, y=۱۵} به دست می‌آید که به‌وسیله آن مسیر منتهی به ERROR اجرا می‌گردد. ابزارهای DART، CUTE و EXE از اجرای پویانمادین بهره می‌برند که در ادامه بررسی خواهند شد.

۲-۲-۲-روش EGT

در این روش به‌صورت پویا در هر عبارت این بررسی صورت می‌گیرد که آیا تمام متغیرهای موجود در برنامه عددی هستند یا خیر. اگر همه‌ی متغیرهای موجود عددی باشند، عبارت به‌صورت عادی و به‌صورت عددی اجرا می‌گردد. ولی اگر حتی یکی از متغیرها نمادین باشند عبارت به‌صورت نمادین اجرا و قید مربوط به PC اضافه می‌شود. اگر در خط ۱۷ عبارت به شکل y=۱۰ تغیر کند، خط ۶ به‌صورت عددی اجرا می‌گردد چون ورودی تابع twice مقدار عددی ۱۰ خواهد بود. در خط ۷ عبارت شرطی به شکل if(x==۲۰) درمی‌آید. در این خط عبارت به‌صورت نمادین اجرا می‌شود. با این حالت هر دو مسیر then و else بررسی می‌شوند. در شاخه then در خط ۸ عبارت به شکل if(x>۲۰) درمی‌آید. با توجه به اینکه x برابر مقدار ۲۰ است بنابراین شاخه then قابل‌اجرا نخواهد بود و اجرا بدون اینکه به شاخه ERROR برسد پایان می‌پذیرد.

با استفاده از روش‌های نوین می‌توان محدودیت‌های مربوط به حل‌کننده‌ی قید و کدهای کتابخانه‌های خارجی را تا حدی برطرف نمود. برای مثال در برنامه‌های کاربردی واقعی اگر ورودی یک تابع کتابخانه خارجی نمادین باشد، برای ادامه اجرا لازم است تا ورودی عددی گردد. برای این منظور درروش EGT قید موجود در آن عبارت توسط حل‌کننده‌ی قید حل می‌شود یا درروش اجرای پویانمادین از حالت عددی موجود مقدار عددی آن ورودی محاسبه می‌گردد. به‌این‌ترتیب اجرا می‌تواند ادامه پیدا کند.

مثلاً در همان کد نمونه شکل ۴ اگر تابع twice از یک کتابخانه خارجی دریافت شود و یا این تابع به‌صورت غیرخطی (v*v)٪۵۰ که در شکل ۴ دیدیم درآید ، هنگامی‌که اجرا به خط ۶ از برنامه می‌رسد ازآنجاکه حل‌کننده‌ی قید نمی‌تواند قید مربوطه را حل کند اجرا متوقف می‌گردد. در این حالت در اجرای پویانمادین مقدار نمادین برای متغیر y با مقدار عددی آن مثلاً ۷ جایگزین می‌شود و اجرا ادامه می‌یابد. در این حالت قید مربوط به خط ۷ به شکل if(x==۴۹) درمی‌آید و اجرا ادامه پیدا می‌کند. در اجرای نمادین سنتی چون حل‌کننده‌ی قید قادر به حل این‌گونه از قیدها نبود، اجرا متوقف می‌شد.

۲-۳-چالش‌ها

در این قسمت چالش‌هایی که در رابطه با اجرای پویانمادین مطرح می‌شود را به‌طور خلاصه بیان می‌کنیم.

۲-۳-۱-انفجار مسیرها

در دنیای واقعی تعداد خطوط برنامه‌ها بسیار زیاد است و تعداد مسیرهایی که در آن‌ها قابل پیمایش است به‌صورت نمایی افزایش می‌یابد. همین موضوع باعث می‌شود تا در اجرای پویانمادین با کمبود منابع محاسباتی به‌خصوص کمبود حافظه مواجه شویم.

۲-۳-۲-حل قیدها

یکی از نقاط چالش‌برانگیز در این حوزه حل کردن قیدهای مسیر اجرا است. اجرای نمادین در برنامه‌های واقعی باعث می‌شود تا قیدهایی تولید شوند که حل‌کننده‌ی قیدها توانایی حل کردن آن‌ها را ندارند یا اینکه حل آن‌ها در زمان کوتاه قابل انجام نیست. برای رفع این منظور لازم است تا بهبودهایی در پیاده‌سازی این حل‌کننده‌ی قیود صورت پذیرد.

۲-۳-۳-مدل‌سازی حافظه

نحوه‌ی تبدیل عبارات برنامه به عبارات نمادین تأثیر مستقیمی بر میزان پوشش کد در زمان اجرای پویانمادین دارد. درواقع نحوه‌ی برخورد با حافظه و مدل کردن آن یکی از چالش‌های مطرح این حوزه است. به‌طور مثال یک متغیر int می‌تواند به شکل یک‌خانه واحد حافظه یا به شکل ۴ خانه یک بایتی در حافظه در نظر گرفته شود. در حالت دوم خطاهایی مانند سرریز بافر را می‌توان بررسی نمود.

به طریق مشابه اشاره‌گرها در DART به‌عنوان یک مقدار عددی int در نظر گرفته می‌شوند ولی در CUTE با مدل خاص پیاده‌سازی شده می‌توان برابری یا نابرابری دو اشاره‌گر را بررسی و قید مربوط به آن را حل کرد.

در ابزار EXE از تئوری آرایه‌ها استفاده می‌شود که حل‌کننده‌های قیدی مانند Z۳ و STP می‌توانند قیدهای مربوط به آن‌ها را حل کنند.

۲-۳-۴-همروندی

برنامه‌های امروزی به‌صورت توزیع‌شده توسعه می‌یابند. معمولاً کاربرهای مختلف به‌صورت همروند و چند نخی می‌توانند این برنامه‌ها را اجرا کنند. نحوه آزمون برنامه‌های همروند از دیگر چالش‌های این حوزه است.

فصل سوم:

ابزارهای اجرای پویانمادین برای تحلیل برنامه

۳-۱-ابزار DART

۳-۱-۱-مقدمه

ابزار DART [۳] که سرنام : Directed Automated Random Testing می‌باشد، درسال ۲۰۰۵ میلادی باهدف آزمون خودکار نرم‌افزار ارائه‌شده است. این ابزار از سه تکنیک مختلف بهره می‌برد که در ذیل مشاهده می شود:

۱. استخراج خودکار رابط[۱] یک برنامه با محیط خارج از آن با استفاده از تحلیل ایستا
۲. تولید خودکار داده‌ی آزمون برای رابط برنامه باهدف پوشش هرچه بیش‌تر متن برنامه
۳. تحلیل پویا و بررسی چگونگی رفتار یک برنامه به ازای داده‌ی ورودی خودکار آزمون

نقطه‌ی قوت ابزار DART نسبت به ابزارهای دیگر آزمون نرم‌افزار این است که بدون نیاز به نوشتن کد راه‌انداز[۲] یا افزودن کدی دیگر، به‌طور کاملاً خودکار نرم‌افزار را مورد ارزیابی قرار می‌دهد.

نمایشان روش‌های مختلف آزمون نرم‌افزار یکی از روش‌های رایج، آزمون واحد[۳] می‌باشد. در آزمون واحد هر یک از اجزای نرم‌افزار به‌طور مستقل مورد ارزیابی قرار می‌گیرد ازاین‌رو پس از انجام این آزمون می‌توان با قطعیت در مورد صحت منطق هر یک از اجزای نرم‌افزار و پوشش ۱۰۰ درصدی کد برنامه اظهارنظر کرد. اما به علت پرهزینه بودن و سختی این روش، آزمون واحد در عمل به‌ندرت مورداستفاده قرار می‌گیرد. واضح است که در آزمون واحد برای ارزیابی مستقل یک جز از برنامه نیاز داریم تا قبل و بعدازآن خطوط دیگری را با عنوان کد آزمون به برنامه اضافه کنیم. به‌عنوان‌مثال لزوماً بایستی قبل از جز مورد آزمون، کد راه‌انداز و پس‌ازآن شرط‌هایی برای چک کردن خروجی مورد انتظار نوشته شود. نوشتن کد اضافی برای آزمون یک برنامه هزینه‌بر است. درعین‌حال آزمون واحد جایگزین آزمون کل برنامه با راهبرد جعبه‌ی سیاه[۴] نخواهد بود. زیرا ممکن است کد و منطق تمامی اجزای برنامه به‌صورت جدا از هم صحیح باشد ولی در ترکیب این اجزا خطا[۵] داشته باشیم.

ابزار DART ، نقاط ضعف آزمون واحد نرم‌افزار را ندارد زیرا نیاز به نوشتن کد اضافه برای آزمون نیست و تمامی مراحل به‌طور خودکار انجام می‌گیرد. این ابزار برای آزمون برنامه‌های نوشته‌شده به زبان C توسعه‌یافته است و

[۱] Interface
[۲] Driver
[۳] Unit testing
[۴] Black box
[۵] Bug

برای این کار به کد منبع برنامه نیاز دارد. در پایان ابزار می‌تواند خطاهای استاندارد نظیر خرابی برنامه[۱]، نقض بیانیه‌ها[۲] و پایان نیافتن برنامه را تشخیص دهد.

یکی از نقاط قوت ابزار DART در مقایسه با ابزارهای فازینگ[۳] که با استفاده از ورودی‌های دلخواه به تحلیل پویا برنامه می‌پردازند پوشش کد[۴] بیش‌تر است. به‌عنوان‌مثال اگر عبارت شرطی "if (x==10) then …" با استفاده از روش‌های فازینگ مورد آزمون قرار بگیرد و فرض کنیم که متغیر int x یک‌خانه‌ی ۳۲بیتی از حافظه است تنها به‌احتمال یک از $۲^{۳۲}$ مقدار ممکن شاخه‌ی then عبارت شرطی اجرا می‌شود. این به این معناست که با استفاده از روش‌های فازینگ نمی‌توان به پوشش کد مناسبی دست‌یافت. با استفاده از ابزار DART اما احتمال اجرای شاخه‌ی then و else با یکدیگر برابر و برابر ۰,۵ می‌باشد.

درواقع روش اجرای پویانمادین که برای اولین بار در پژوهش DART مطرح شد می‌تواند به‌عنوان روش جایگزین تحلیل ایستا مطرح گردد. این روش درزمینهٔ ی آزمون کد کتابخانه‌های خارج از برنامه و تشخیص خطاهای موجود در زیر برنامه‌ها بسیار بهتر از تحلیل ایستا عمل می‌کند.

۳-۱-۳-طرح کلی DART با استفاده از دو مثال

در شکل ۵ کد ساده مربوط به مثال اول را مشاهده می‌کنید که قصد داریم تا با استفاده از ابزار DART آن را مورد تحلیل قرار دهیم.

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y)
     if (f(x) == x + 10)
        abort();        /* error */
  return 0;
}
```

شکل۵ -یک‌تکه کد ساده برای آزمون با ابزار DART-مثال اول

[۱] program crashe
[۲] assertion violation
[۳] Fuzzing
[۴] Code coverage

همان‌طور که مشاهده می‌کنید تابع h دارای خطا می‌باشد زیرا یکی از مسیرهای اجرایی آن می‌تواند به عبارت abort() ختم شود؛ اما با استفاده از روش فازینگ احتمال بسیار کمی دارد که با اجرای برنامه ورودی‌های دلخواه به‌گونه‌ای داده شود که برنامه به عبارت abort() برسد.

اگر تکه کد شکل ۵ را با استفاده از ابزار DART مورد آزمون قرار دهیم. فرض کنید که ورودی‌های دلخواه اولیه x=۲۶۹۱۶۷۳۴۹ , y=۸۸۹۸۰۱۵۴۱ باشد، در آن صورت شاخه‌ی then اولین عبارت شرطی اجرا خواهد شد ولی دومین عبارت شرطی به شاخه‌ی else خواهد رفت بنابراین در اولین اجرا خطایی یافت نخواهد شد.

در حین اجرای این مسیر اولیه عبارات شرطی $y_0 != x_0$ و $2x_0 != x_0+10$ با استفاده از عبارات نمادین x_0,y_0 استخراج می‌شوند. در اولین مسیر اجرا شرط مسیر[۱] به‌صورت« $x_0 != y_0 , 2x_0 != x_0+10$ » بوده است.اکنون در اجرای دوم با استفاده از حل ارائه‌شده توسط حل‌کننده‌ی قید مسیر مربوط به « $x_0 != y_0 , 2x_0 != x_0+10$ » اجرا می‌گردد و برنامه با رسیدن به عبارت abort به خطا می‌رسد.

درواقع ابزار DART ابتدا با مقادیر دلخواه ورودی برنامه را اجرا می‌نماید. پس‌ازآن در حین اجرای یک مسیر از برنامه شرط‌های آن را مسیر را به‌طور نمادین استخراج می‌نماید. با استفاده از این شرط‌ها و خروجی حاصل از حل‌کننده‌ی قید مسیر بعدی اجرای برنامه انتخاب می‌گردد و متناسب با آن ورودی‌هایی تولید می‌شود. این فرآیند تا رسیدن به خطا یا پایان برنامه ادامه می‌یابد.

درواقع زمانی که به یک شرط در مسیر اجرای برنامه می‌رسیم دو مقدار کلی برای آن در نظر گرفته می‌شود مقدار یک نشان‌دهنده‌ی اجرای زیرشاخه‌ی then از آن شرط است و مقدار صفر اجرای زیرشاخه‌ی else را نشان می‌دهد.

ویژگی اصلی ابزار DART دو مورد پیش رو است که بر اساس قضیه‌ی صفحه‌ی بعد بیان می‌شود.

۱. مانعیت[۲](نسبت به خطاهای یافت شده)
۲. جامعیت[۳]

[۱] Path Constraint
[۲] Soundness
[۳] Completness

قضیه:

یک برنامه بانام P را در نظر بگیرید. سه حالت زیر ممکن است رخ دهد:

۱. اگر اجرای ابزار DART منجر به تولید خروجی "Bug found" گردد؛ در آن صورت لزوماً مقادیر عددی به‌عنوان ورودی وجود خواهند داشت که منجر به تولید خطا شوند.
۲. اگر اجرای ابزار DART بدون نمایش هیچ پیغام خطایی اتمام یابد در آن صورت لزوماً هیچ مقدار ورودی وجود نخواهد داشت که منجر به خطا گردد. درواقع تمامی مسیرهای برنامه توسط ابزار پیموده شده و خطایی یافت نشده است.
۳. اجرای ابزار DART برای زمان نامتناهی ادامه خواهد یافت.

واضح است که ابزار DART با اجرای برنامه، مسیر منتهی به خطا را می‌یابد بنابراین برخلاف تحلیل‌های ایستا که مثبت کاذب بسیار زیادی را شامل می‌شوند لزوماً هر خطای یافت شده توسط DART به‌طور واقعی در برنامه وجود خواهد داشت.

در شکل ۶ تکه کد مربوط به مثال دوم را مشاهده می‌کنید.

```
int f(int x, int y) {
  int z;
  z = y;
  if (x == z)
     if (y == x + 10)
        abort();
  return 0;
}
```

شکل ۶- یک‌تکه کد ساده برای آزمون با ابزار DART-مثال دوم

در این تکه کد نوشته‌شده به زبانC ، در ابتدا بردار آدرس به‌صورت $M_0=(m_x , m_y)$ می‌باشد. درواقع m_x وm_y آدرس‌های متفاوت پارامترهای ورودی تابع f [۱] در حافظه سیستم هستند. فرض می‌کنیم که ورودی بی‌قاعده‌ی ابتدایی برای پارامتر$x=۱۲۳۴۵۶$ و برای پارامتر $y=۶۵۴۳۲۱$ باشد. در این صورت حافظه‌ی عددی سیستم در ابتدا به‌صورت$M = [mx \rightarrow ۱۲۳۴۵۶, my \rightarrow ۶۵۴۳۲۱]$ و حافظه‌ی نمادین سیستم نیز به‌صورت $S = [mx \rightarrow mx , my \rightarrow my]$ خواهد بود.

[۱] f(x,y)

با این ورودی‌های اولیه، مسیر اجرای اول به زیرشاخه‌ی else عبارت if اول می‌رود. سپس به دستور return ۰ می‌رسد و این مسیر پایان می‌یابد. شرط مسیر حاصل از این اجرای اولیه $\neg(mx = my)$ می‌باشد. اکنون پس از پایان اجرای اول شرایط حافظ عددی و نمادین به‌صورت زیر است:

$M = [mx \rightarrow ۱۲۳۴۵۶, my \rightarrow ۶۵۴۳۲۱, mz \rightarrow ۶۵۴۳۲۱]$

$S = [mx \rightarrow mx, my \rightarrow my, mz \rightarrow my]$

اکنون نقیض شرط مسیر به‌دست‌آمده در اجرای قبل به حل‌کننده‌ی قید داده می‌شود تا امکان‌پذیر بودن آن را بررسی کند و در صورت امکان‌پذیر بودن ورودی عددی متناسب با آن را تولید کند. حل‌کننده‌ی قید به‌عنوان حل شرط($mx = my$) ، خروجی ($mx \rightarrow ۰, my \rightarrow ۰$) را ارائه می‌دهد که با استفاده از آن در اجرای دوم زیرشاخه‌ی then شرط if اول اجراشده و به‌شرط جدید ($mx = my, my = mx + ۱۰$) می‌رسد. در اینجا نیز به حل‌کننده‌ی قید رجوع می‌شود تا امکان‌پذیر بودن شرط مسیر جدید را بررسی نماید. اما با توجه به آنکه ازنظر منطقی این شرط نمی‌تواند صحیح باشد بنابراین مسیر یافت شده غیر امکان‌پذیر تشخیص داده می‌شود و اجرای ابزار DART بدون اعلام هرگونه پیام خطا پایان می‌یابد. بنابراین اگرچه دستورالعمل abort() در متن کد موجود بود ولی مسیر منتهی به آن امکان‌پذیر نبوده و کد مثال۲ هیچ خطایی را در برندارد.

۳-۲-ابزار Mayhem

۳-۲-۱- مقدمه

MAYHEM [۵] ابزاری برای یافتن آسیب‌پذیری‌ها در فایل‌های اجرایی یا به‌بیان‌دیگر فایل‌های باینری است. ابزار MAYHEM ، به ازای هر آسیب‌پذیری که کشف می‌کند کد بهره‌بردار مربوط به آن را نیز در اختیار تحلیل‌گر می‌گذارد. ابزار MAYHEM برای ارزیابی توانایی‌های خود موفق شد ۲۹ مورد آسیب‌پذیری در پلتفرم‌های ویندوز و لینوکس را کشف و کد بهره‌بردار برای آن‌ها را تولید نماید. دو مورد از این آسیب‌پذیری‌ها تا پیش‌ازاین گزارش نشده بودند.

درواقع ابزار MAYHEM از اجرای نمادین بهبودیافته بهره می‌برد به این صورت که در جایگاه‌های مشکوک کد باینری شرط‌هایی را به اجرای نمادین اضافه می‌کند. این شرط‌ها نشان‌دهنده‌ی موارد مهم برای مهاجم هستند به‌عنوان‌مثال آدرس جایی که پس از اجرا اشاره‌گر IP به آن برمی‌گردد. با استفاده از این شروط فرمول جدیدی

برای آن مسیر اجرا به دست می‌آید. اگر این فرمول توسط حل‌کننده‌ی قید قابل‌حل باشد در آن صورت ابزار MAYHEM به یک مسیر اجرا برای کد بهره‌برداری از آسیب‌پذیری دست‌یافته است .

برای یافتن کد بهره‌بردار آسیب‌پذیری، لازم است تا فضای اجرایی بزرگی از برنامه جست‌وجو شود. درMAYHAM برای مقابله با این موضوع چهاراصل کلیدی پیش‌بینی‌شده است:

۱) سیستم باید بتواند برای مدت طولانی رو به جلو پیش رود. درواقع به‌طور ایدئال سیستم باید بتواند همواره اجرای خود را ادامه دهد بدون اینکه از منابع اختصاص داده‌شده بیش‌ازحد مجاز استفاده کند.
۲) باهدف بالا بردن کارایی سیستم، نباید یک کار تکراری دوباره اجرا گردد.
۳) سیستم نباید یک کار یا نتیجه آن را رها کند بلکه باید در هر اجرا از نتایج اجرای گذشته استفاده کند.
۴) سیستم باید بتواند در مورد مقدار نمادین یک حافظه که خواندن و نوشتن در آن بستگی به کاربر دارد، تصمیم‌گیری کند.

به‌طورکلی دو نوع مختلف را برای ابزارهای تحلیل کد می‌توان در نظر گرفت:

۱) ابزارهای آفلاین:
ابزارهای تحلیلی که ابتدا یک مسیر از کد را به‌صورت عددی اجرا می‌کنند و سپس با حل مجموعه قیدهای تولیدشده در آن میسر به‌صورت نمادین، سعی می‌کنند تا ورودی‌هایی را برای مسیر اجرایی جدید تولید کنند. به‌عنوان‌مثال ابزار SAGE این‌گونه است.
۲) ابزارهای آنلاین:
در این دسته از ابزارهای تحلیل سعی می‌شود تا تمامی مسیرهای اجرای ممکن در یک اجرای واحد ، اجرا شوند. به‌عنوان‌مثال ابزار S۲E این‌گونه است.

در ابزارهای تحلیل آفلاین، اصول ۱ و ۳ برقرارند. زیرا در هر باریک اجرا و مستقل از اجراهای دیگر صورت می‌گیرد پس از نتایج اجراهای قبلی، می‌توان دوباره استفاده کرد.اما در این ابزارها، اصل ۲ برقرار نیست، زیرا هر مسیر اجرا بایستی از ابتدا اجرا شود پس لزوماً برخی از دستورات در تمام‌مسیرها چندباره اجرا می‌شوند.

برخلاف ابزارهای تحلیل آفلاین، تحلیلگرهای آنلاین مثل KLEE و S۲E در هر نقطه از شروع شاخه جدید از fork استفاده می‌کنند، بنابراین اصول ۱ و ۳ در مورد این ابزارها برقرار نیست. علت آن است که با هر fork جدید، دستورات گذشته دوباره اجرا نمی‌شوند. اما دستور fork سربار زیادی را ازنظر حافظه به ابزار تحمیل می‌کند.

۳-۲-۲-دستاوردهای جدید MAYHEM:

ابزارMAYHEM دستاوردهای جدیدی را ارائه داده است:

- اجرای ترکیبی آفلاین و آنلاین برای حفظ حافظه و سرعت اجرا
- مدل مبتنی بر نمایه برای حافظه
- تحلیل فایل‌های باینری با ارائه‌ی کد بهره‌بردار برای آسیب‌پذیری

نکات کلی زیر در مورد آسیب‌پذیری‌ها و ابزارهای تحلیل کد مطرح است:

۱) نیاز است تا کد برنامه در سطح کد ماشین اجرا شود چون آسیب‌پذیری‌های زیادی مانند سرریز پشته وجود دارد که برای تشخیص آن‌ها الزاماً باید ساختار پشته هم موردبررسی قرار گیرد.
۲) تعداد مسیرهای ممکن خیلی زیاد است زیرا هر دستور if یک شاخه‌ی جدید را در درخت اجرا ایجاد می‌کند.
۳) هرچه تعداد پوشش کد بیش‌تر باشد یعنی تعداد مسیر بیشتری از برنامه بررسی گردد، بهتر است.
۴) اگر ابتدا کد برنامه، عددی اجرا شود و در صورت نیاز به اجرای نمادین تغییر کند تحلیل کد بهتری خواهیم داشت.

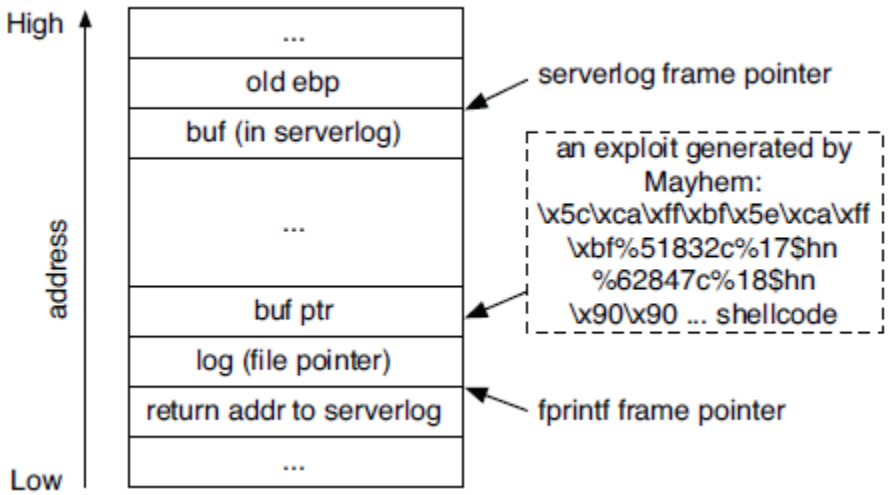
۳-۲-۳-ساختار کلی ابزار MAYHEM

در این زیر بخش ما قصد داریم تا ساختار کلی ابزار MAYHEM و نحوه‌ی عملکرد آن را نمایش دهیم. در شکل ۷ تکه کد مربوط به برنامه‌ی آسیب‌پذیر orzHttpd آمده است. در این برنامه هر ارتباط جدید http به تابع http_read_request فرستاده می‌شود و در خط ۲۹ با فراخوانی تابع static_buffer_read رشته‌ی درخواست http کاربر دریافت می‌شود. ورودی کاربر در یک بافر ۴۰۹۶ بایتی به نام conn->read_buf.buf ذخیره می‌گردد. برای جلوگیری از آسیب‌پذیری سرریز پشته درصورتی‌که ورودی بیش از ۴۰۹۶ بایت باشد و کاراکتر پایان خط در http نیامده باشد، خطای ۴۰۰ بازخواهد گشت و از سرریز پشته جلوگیری خواهد شد.

آسیب‌پذیری موجود در این برنامه در serverlog موجود است، در آنجا تابع fprintf با یک ورودی ساختاریافته از دریافت http کاربر فراخوانی می‌شود. در شکل ۸ وضعیت پشته درزمانی که آسیب‌پذیری format string تشخیص داده می‌شود نشان داده‌شده است.

```
1  #define BUFSIZE 4096
2
3  typedef struct {
4    char buf[BUFSIZE];
5    int used;
6  } STATIC_BUFFER_t;
7
8  typedef struct conn {
9    STATIC_BUFFER_t read_buf;
10   ... // omitted
11 } CONN_t;
12
13 static void serverlog(LOG_TYPE_t type,
14                        const char *format, ...)
15 {
16   ... // omitted
17   if(format != NULL) {
18     va_start(ap, format);
19     vsprintf(buf, format, ap);
20     va_end(ap);
21   }
22   fprintf(log, buf); // vulnerable point
23   fflush(log);
24 }
25
26 HTTP_STATE_t http_read_request(CONN_t *conn)
27 {
28   ... // omitted
29   while(conn->read_buf.used < BUFSIZE) {
30     sz = static_buffer_read(conn, &conn->
          read_buf);
31     if(sz < 0) {
32       ...
33       conn->read_buf.used += sz;
34       if(memcmp(&conn->read_buf.buf[conn->
            read_buf.used] - 4, "\r\n\r\n", 4) ==
            0)
35       {
36         break;
37       }
38   }
39   if(conn->read_buf.used >= BUFSIZE) {
40     conn->status.st = HTTP_STATUS_400;
41     return HTTP_STATE_ERROR;
42   }
43   ...
44   serverlog(ERROR_LOG,
45             "%s\n",
46             conn->read_buf.buf);
47   ...
48 }
```

شکل ۷- تکه کد مربوط به برنامه‌ی آسیب‌پذیر orzHttpd



شکل ۸- وضعیت پشته برای برنامه‌ی آسیب‌پذیر orzHttpd

معماری ابزارMAYHEM در شکل ۹ مشاهده می‌شود. این ابزار از دو بخش همروند برای تحلیل کد بانام CEC [۱] یا همان کلاینت اجراکننده‌ی عددی و SES [۲]یا همان سرور اجراکننده‌ی نمادین تشکیل‌شده است. اگر بخواهیم از دیدگاهی با انتزاع بالا سخن بگوییم، CEC بر روی یک سیستم هدف مشخص اجرا می‌شود اما SES بر روی هر پلتفرمی و بنابر ارجاعات CEC اجرا می‌گردد. ابزار MAYHEM برای تولید کد بهره‌بردار[۴] آسیب‌پذیری مراحلی را انجام می‌دهد که به شرح زیر است:

۱) ابزار MAYHEM با تعریف یک پورت کار خود را شروع می‌کند.کد برنامه‌ی مورد تحلیل از طریق همین پورت دریافت می‌شود و در ابتدا اجرای نمادین بر روی آن اعمال می‌گردد.
۲) واحد CEC برنامه‌ی آسیب‌پذیر را دریافت می‌کندَ و به SES متصل می‌شود تا مقداردهی‌های اولیه نمادین برای ورودی‌ها صورت پذیرد. سپس کد به‌صورت عددی در واحد CEC اجرا می‌شود. در حین اجرای عددیinstrumentation کد باینری و تحلیل آلایش[۷] پویا نیز اجرا می‌گردد. تحلیل آلایش پویا در اینجا با استفاده از زیر بخش Taint tracker که در شکل ۹ نیز آمده است انجام می‌شود.
۳) زمانی که CEC با یک بلاک کد آلوده یا یک پرش آلوده روبه‌رو می‌شود(منظور از پرش آلوده، جایی از کد برنامه است که برای ادامه‌ی کار به یک ورودی از مهاجم نیاز دارد.)،اجرای عددیCEC موقتاً متوقف می‌گردد و شاخه‌ی آلوده‌ی کد به SES برای اجرای نمادین ارسال می‌شود. SES به‌طورکلی برای این شاخه از کد مشخص می‌کند که آیا اجرای آن امکان‌پذیر است یا خیر؟
۴) واحد SES به‌صورت موازی با CEC اجراشده و بلاک‌های کد را دریافت می‌کند. این بلاک‌ها به زبان میانی تبدیل می‌شوند و به‌صورت نمادین اجرا می‌گردند. پس‌ازآن در صورت لزوم مقادیر عددی موردنیاز از CEC دریافت می‌شود.

واحد CEC دو نوع فرمول فراهم می‌نماید:

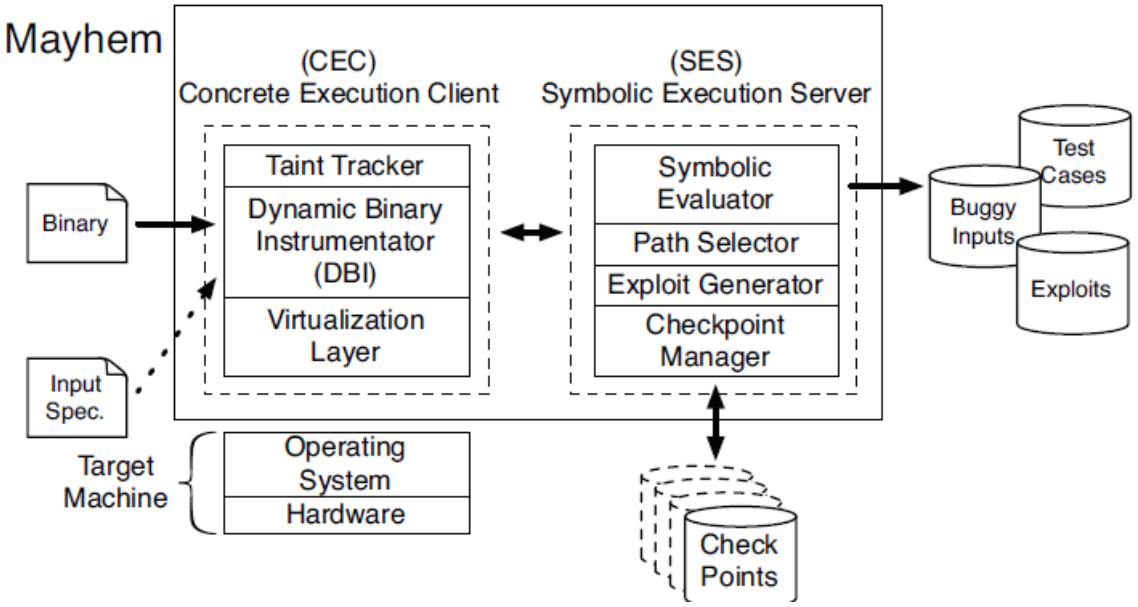
۱- فرمول مسیر[۸]

فرمول مسیر نشان‌دهنده‌ی شرط‌هایی است که با استفاده از آن‌ها می‌توان به خط معینی از کد برنامه رسید.

فرمول قابلیت کد بهره‌بردار[۱]

[۱] Concrete Executor Client
[۲] Symbolic Execution Server
[٤] Exploit
[۷] Taint Analysis
[۸] Path Formula

فرمول قابلیت کد بهره‌بردار، نشان‌دهنده‌ی آن است که آیا مهاجم می‌تواند کنترل اشاره‌گر دستورالعمل[۲] را به دست بگیرد و آیا می‌تواند کد مخرب خود را اجرا نماید؟



شکل ۹- معماری MAYHEM

۵) ابزار وقتی به یک پرش آلوده می‌رسد SES بر اساس پرس‌وجویی از حل‌کننده‌ی قید تصمیم می‌گیرد که آیا دستور fork لازم هست یا خیر. اگر fork لازم باشد اجراهای جدید اولویت‌بندی شده و به ترتیب اجرا می‌شوند. در طی اجراهای جدید اگر منابع سیستم به پایان برسند، واحدSES رویه‌ی بازگشت را اجرا می‌کند. درنهایت بعد از اتمام اجرای یک پردازه، تعدادی مورد آزمون بر اساس آن تولید می‌شوند.

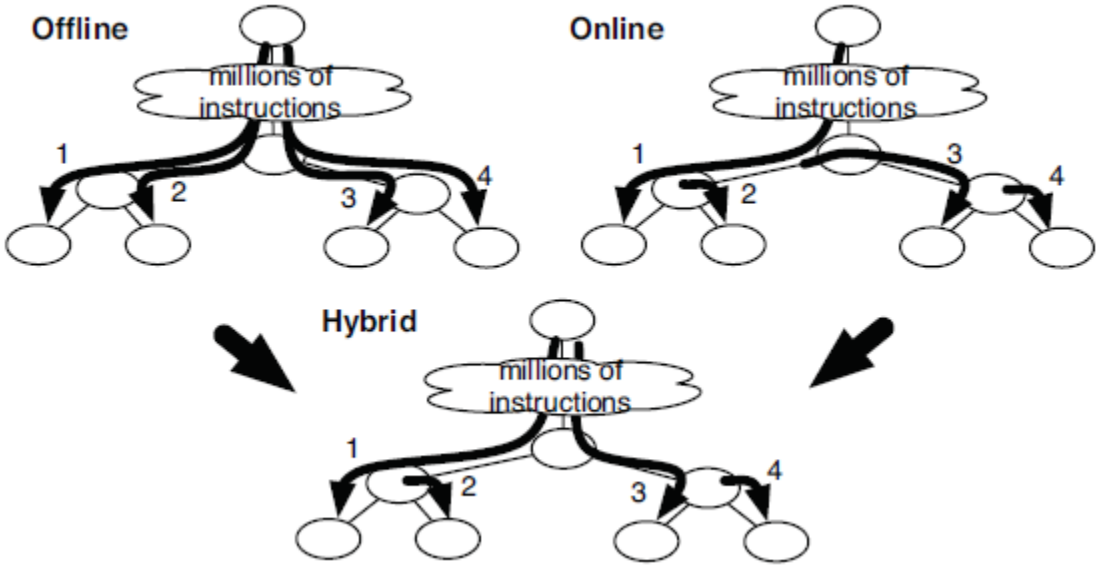
۶) در پرش‌های آلوده یک فرمول قابلیت کد بهره‌بردار تولید و به واحدSES داده می‌شود. اگر این فرمول قابل ارضا باشد یعنی کد برنامه از این مسیر آسیب‌پذیر و. قابل بهره‌برداری توسط مهاجم است.

برای اجرای نمادین لازم است تا کد باینری به کد اسمبلی تبدیل شود. برای این منظور از BAP [۳۲] استفاده می‌شود. سپس کد اسمبلی به شکل نمادین اجرا می‌شود.

همان‌گونه که در بخش ۳-۲-۱ مطرح شد، ابزارهای تحلیل کد به دودسته‌ی آنلاین و آفلاین تقسیم می‌شوند اما ابزار MAYHEM در یک نوآوری علمی از اجرای پویانمادین هیبرید استفاده می‌نماید. این به آن معناست که

[۱] Exploitability Formula
[۲] instruction pointer: اشاره گری که نشان دهنده‌ی خط بعدی برنامه برای اجرا است

ابزار ابتدا کد برنامه را به‌صورت آنلاین اجرا می‌کند. اما زمانی که اجرای ابزار به محدوده‌ی منابع سیستم برسد، اجرای آنلاین متوقف‌شده و پس‌ازآن اجرا به شکل آفلاین ادامه می‌یابد.



شکل ۱۰- انواع اجراهای پویانمادین

۳-۲-۴-اجرای هیبرید پویانمادین در ابزار MAYHEM

ابزار MAYHEM در اجرای هیبرید چهار مرحله اصلی را طی می‌نماید:

۱) **مقداردهی اولیه:** زمانی که ابزارMAYHEM برای اولین بار کد یک برنامه را بارگذاری می‌نماید، پایگاه داده CHECKPOINTها، محل نگهداری مورد آزمون‌ها و مدیر CHECKPOINTها را مقداردهی اولیه می‌کند.

۲) **جست‌وجوی آنلاین:** در این مرحله کد برنامه مورداجرای نمادین آنلاین قرار می‌گیرد.در پایان این مرحله مورد آزمون نیز تولید می‌گردد.

۳) **مرحله CHECKPOINTING**: در این مرحله مدیر CHECKPOINT بر اجرای آنلاین نظارت می‌کند. درواقع زمانی که حافظه مصرفی برنامه یا تعداد اجراهای حاضر به حد از پیش تعیین‌شده‌ای برسد، از میان اجراهای حاضر یکی از آن‌ها انتخاب وCHECKPOINT مربوط به آن تولید می‌گردد. این CHECKPOINT شامل وضعیت اجرای نمادینی است که متوقف‌شده و زمانی که منابع سیستم

آزاد گردد مجدداً ادامه می‌یابد. وضعیت اجرای نمادین شامل شرط مسیر، آمارها و موارد دیگری می باشد.

۴) **بازسازی با CHECKPOINT**[۱]: یکی از CHECKPOINTها با توجه به یک هیوریستیک از پیش تعیین‌شده‌ای انتخاب می‌شود. ابزار باید حالت عددی قبل از متوقف شدن را از CHECKPOINT بازسازی کند. برای این کار از مقادیری استفاده می‌کند که به ازای آن‌ها شرط مسیر قابل ارضا هست. به این وسیله حالت عددی تا جایی که اجرا معلق شده ساخته می‌شود و از این به بعد اجرا به‌صورت آنلاین ادامه می‌یابد.

۳-۲-۵-معماری و پیاده‌سازی CEC

واحد CEC به‌عنوان ورودی، فایل باینری و یک CHECKPOINT دلخواه را دریافت می‌کند. این واحد ابتدا کد برنامه را به‌صورت عددی اجرا می‌کند. هم‌زمان با اجرای عددی، تحلیل آلایش نیز انجام می‌شود. اگر ابزار به بلاک آلوده‌ای از کد برنامه برسد، بلاک را به واحد SES می‌دهد تا به‌صورت نمادین آن را تحلیل کند. خروجی آن آدرس پرش به یک بلاک کد یا ایجاد یک CHECKPOINT است. واحد CEC اجرا را تا زمانی که همه مسیرهای ممکن پیمایش شوند ادامه می‌دهد.

لایه مجازی‌سازی در واحد CEC، مسئولیت شبیه‌سازی OS را بر عهده دارد. هر اجراکننده عددی برای خود یک وضعیت[۲] جداگانه دارد که از دید دیگر اجراکننده‌ها پنهان است. به‌عنوان‌مثال اگر یک اجراکننده روی فایل می‌نویسد بقیه‌ی اجراکننده‌ها یک کپی از این فایل رادارند و می‌توانند هم‌زمان روی آن بنویسند.

۳-۲-۶-معماری و پیاده‌سازی SES

قسمت SES وظیفه مدیریت محیط نمادین و انتخاب مسیر بعدی برای CEC را به عهده دارد. محیط نمادین شامل اجراکننده نمادین، انتخاب‌کننده مسیر و مدیر CHECKPOINT است. واحد SES ظرفیتی محدود دارد که با رسیدن به محدوده‌ی آن، به اجرای نمادین مسیر ادامه نخواهد داد. در این حالت یک CHECKPOINT تولید می‌شود. هرCHECKPOINT شامل حالت‌هایی از اجراکننده نمادین است که در اجرای اول به دلیل محدودیت حافظه اجرانشده‌اند. هر حالت اجرای نمادین شامل دو مقدار است. اولین مقدار، مقادیر ثبات‌های نمادین است و مقدار دوم، مقادیر حافظه نمادین می‌باشد. در هر CHECKPOINT این حالت‌های اجرایی نگهداری می‌شوند.

---

[۱] Checkpoint Restoration
[۲] State

در زیر بخش پیش‌بینی اجرای نمادین[۱]،کاربر به‌عنوان ورودی می‌تواند تعدادی قید را بر روی ورودی‌های برنامه تعریف کند به‌عنوان‌مثال می‌توان قیدی را بر روی طول ورودی تعریف کرد تا فضای جست‌وجو را کاهش دهد. اگر هیچ قیدی بر روی ورودی‌های برنامه تعریف نشود، واحد SES تلاش می‌کند تا تمام فضای کد برنامه را پوشش دهد.

در زیر بخش انتخاب مسیر، ابزار MAYHEM برای انتخاب مسیر بعدی واحد SES از هیوریستیک‌هایی استفاده می‌کند. این ابزار سه قاعده‌ی هیورستیک را برای رتبه‌بندی مسیرها به کار می‌برد:

۱) جست‌وجوی بلاک جدیدی از کد (در برابر جست‌وجوی تکراری بلاک‌های پیمایش شده از کد)
۲) اجراکننده‌هایی که از حافظه نمادین استفاده می‌کنند در اولویت هستند.
۳) مسیرهای اجرایی که اشاره‌گر دستورالعمل‌های نمادین[۲] در آن‌ها تشخیص داده می‌شوند، در اولویت هستند.

۳-۲-۷-مدل‌سازی حافظه در MAYHEM

ابزار MAYHEM به‌صورت جزئی حافظه را مدل‌سازی می‌کند. به این معنی که آدرس دستورات read می‌تواند به‌صورت نمادین یا عددی باشند ولی در دستورات write لزوماً آدرس باید عددی باشد. برای تعریف حافظه نمادین MAYHEM از مفهوم Memory Object استفاده می‌کند که با M نشان داده می‌شود. هر بار که آدرس حافظه در کد به شکل نمادین باشد، یک M تازه ایجاد می‌شود که مجموعه‌ای از آدرس‌هایی با عبارت نمادین قابل‌دسترسی می‌باشد. این حافظه برخلاف u غیرقابل تغییر است و صرفاً یک تصویر از u می‌باشد.

در بدترین حالت M شامل تمام 2^{32} حالت ممکن است که این فضای حالت بسیار بزرگی است. برای کاهش آن از یک جست‌وجوی دودویی استفاده می‌شود و سعی می‌شود که کران بالا و پایین Memory Object به‌صورت محافظه‌کارانه‌ای مشخص گردد. برای تعیین کران‌ها از SMT استفاده می‌شود. برای مثال اگر نصف اول کل بازه قابل ارضا باشد، کران پایین در نصف اول خواهد بود.

کش تصفیه[۳]: برای تصفیه و به دست آوردن محدوده‌ی موردنیاز، لازم است تا تعدادی پرس‌وجو به SMT زده شود. برای کاهش این‌گونه درخواست‌ها یک کش قرار داده می‌شود که قدم‌ها و نتیجه تصفیه نهایی را در خود

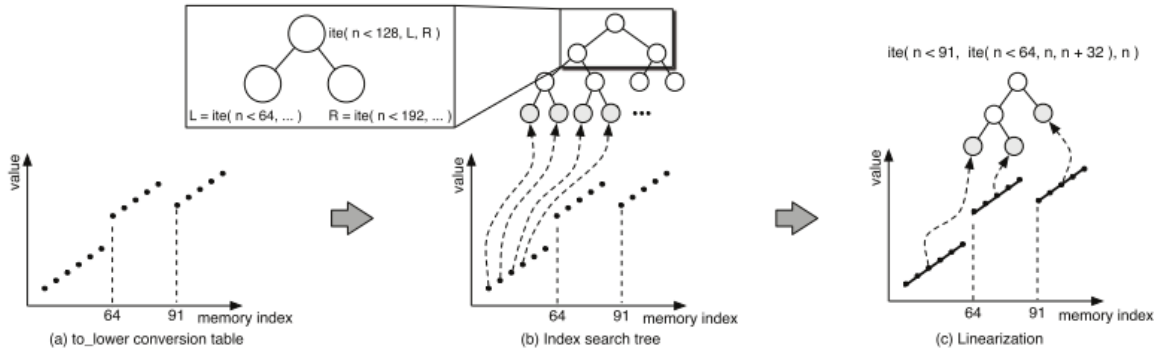
[۱] Preconditioned Symbolic Execution
[۲] symbolic instruction pointer
[۳] Refinement Cache

نگهداری می‌کند. برای هر قدم جدید ابتدا درخواستی به این کش ارسال می‌شود در صورت وجود، یک درخواست به SMT ارسال می‌شود تا ارضا پذیری آن و میزان دقت آن برای فرمول جدید را بررسی نماید.

کش Lemma: باوجود کش تصفیه هر بار لازم است تا درخواستی به SMT داده شود و میزان دقت تصفیه موجود بررسی گردد. برای بهبود این فرآیند یک کش جدید قرار داده می‌شود که شکل کانونی[۱] هر فرمول(F) و نتیجه آن(Q) به‌صورت F→Q در آن نگهداری می‌شود. با استفاده از کش Lemm می‌توان۹۶درصد پرس‌وجوهای حافظه را کاهش داد.

درخت‌های جست‌وجوی ایندکس[۲]: برای بهبود جست‌وجو در Memory Object باهدف یافتن عبارات، MAYHEM از یک درخت دودویی IST [۳] استفاده می‌کند.

$$IST(E)= ite(i<addr(E_{right}),E_{left},E_{right}))$$



شکل ۱۱-درخت دودویی IST و خطی سازی آن

E عبارت مورد جست‌وجو است و مقدار راست و چپ آن با E_{right} و E_{left} نشان داده می‌شود.

برای اینکه تعداد گره‌های درخت IST کاهش یابد از ایده خطی سازی پاکتی استفاده می‌شود. ایندکس‌ها و مقادیرشان معمولاً در کنار هم در یک خط قرار می‌گیرند. قبل از تولید درخت با یک پیش‌پردازش مجموعه پاکت‌های خطی تولید می‌شود و سپس با استفاده از آن‌ها درخت تولید می‌شود با این کار به‌عنوان‌مثال تعداد گره‌ها از ۲۵۶ به ۳ کاهش خواهد یافت.

[۱] canonical representation
[۲] Index Search Trees
[۳] Indexed Search Tree

مدل کردن حافظه و استفاده از Memory Object زمانی مناسب است که اندازه M<<U باشد. اگر اندازه M از یک حد آستانه بیشتر شود، ابزار آدرس‌های نمادین را عددی می‌کند. البته این عددی کردن دلخواه صورت نمی گیرد و بر اساس سه معیار زیر خواهد بود:

۱. اگر امکان‌پذیر است آدرسی تولید شود که برنامه را به قسمت‌های پوشش داده نشده هدایت کند.

۲. درصورتی‌که معیار شماره‌ی۱ قابل‌اعمال نبود،آدرسی تولید شود که به حافظه نمادین اشاره می‌کند. لازم به ذکر است که ابزار قسمت‌های حافظه را به تعدادی منطقه‌ی نمادین[۱] تقسیم می‌کند که این مناطق بر اساس پیچیدگی قیدهای مرتبط به آن‌ها مرتب‌شده‌اند. ابزار سعی می‌کند اشاره‌گر به منطقه‌ای ساده‌تر را ایجاد کند.

۳. اگر موارد بالا قابل‌اجرا نبودند، آدرس عددی معتبر و دلخواهی تولید خواهد شد.

۳-۳-ابزار Drillller

۳-۳-۱-مقدمه

ابزارهای جدید تحلیل برنامه تنها می‌توانند به نقص‌های سطحی دست پیدا کنند. در حقیقت برای هوشمند سازی fuzzerها باید مکانیزم هوشمندانه‌ای برای تولید ورودی‌های جدید داشت. در ساده‌ترین حالت این ابزارها به‌صورت دلخواه این کار را انجام می‌دهند یعنی ورودی‌های جدید کاملاً دلخواه تولید می‌شوند.اما ابزارهای که به اجرای پویانمادین می‌پردازند، ورودی‌های جدید را به کمک حل‌کننده‌ی قید تولید می‌کنند. مشکل ابزارهای اجرای پویانمادین انفجار مسیر در تحلیل برنامه است.از همین رو با استفاده از این ابزارها نمی‌توان به آسیب‌پذیری‌هایی در عمق بالادست یافت.

دستاورد علمی ابزارDriller این است که روش fuzzing را با روش پویانمادین انتخابی[۲] ترکیب کرده است.

۳-۴-مقایسه‌ی ابزارهای اجرای پویانمادین برای تحلیل برنامه‌ها

[۱] Symbolic region
[۲] selective concolic

جدول ۱-مقایسه‌ی ابزارهای اجرای پویانمادین برای تحلیل برنامه‌ها

ابزار	سال ارائه	زبان	پلتفرم	نوع	ویژگی‌ها
DART	**۲۰۰۵**	**C**	**PC**	**Offline**	اجرای پویانمادین و استفاده از DFS برای انتخاب مسیر محدود به زبان C مدل‌سازی حافظه ندارد از هم‌روندی پشتیبانی نمی‌کند بهینه‌سازی برای ارسال قیدها به حل‌کننده قید ندارد در حل قیدهای مربوط به اشاره‌گرها مشکل دارد
Mayhm	**۲۰۱۲**	**Binary**	**PC**	**Hybrid**	مدل‌سازی حافظه باهدف یافتن آسیب‌پذیری به همراه ارائه‌ی کد بهره‌بردار
Driller	**۲۰۱۶**	**Binary**	**PC**	**Hybrid**	FUZZING، Concolic، تکرار Fuzzer انتخاب‌شده از نوع instrumented-genetic هست. AFL موتور پویانمادین شبیه‌سازی ابزار Mayhem هست.

فصل چهارم:

ابزارهای اجرای پویانمادین برای تحلیل برنامه‌های اندرویدی

۴-۱-چالش‌های علمی برای تحلیل برنامه در اندروید

در فصل قبل ابزارهای اجرای پویانمادین برای تحلیل کد برنامه در زبان‌های برنامه‌نویسی متفاوت را دیدیم. در این فصل در بخش مقدمه به تفاوت تحلیل برنامه‌های کاربردی اندروید با سایر برنامه‌ها می‌پردازیم سپس ابزارهای اجرای پویانمادین در پلتفرم اندروید را مورد بررسی قرار خواهیم داد.

۱-برنامه‌های اندروید بسیار وابسته به کتابخانه‌های چارچوبِ کاری هستند و این موضوع باعث ایجاد مشکل واگرایی مسیر می‌شود. در اجرای نمادین اگر یک مقدار نمادین از زمینه برنامه خارج شود، مثلاً برای انجام یک پردازش به یک کتابخانه داده شود یا در اختیار چارچوبِ کاری قرار گیرد، گفته می‌شود که واگرایی مسیر اتفاق افتاده است. واگرایی مسیر موجب ایجاد دو مشکل می‌شود:

الف) موتور اجرای نمادین ممکن است نتواند کتابخانه خارجی را اجرا کند پس تلاش بیشتری لازم است تا بتوان آن کتابخانه را نیز به‌صورت نمادین اجرا کرد.

ب) در کتابخانه خارجی ممکن است تعدادی قید وجود داشته باشد که در خروجی و حاصل پردازش کتابخانه مؤثر باشند. ازاین‌جهت این قیدها در تولید مورد آزمون‌ها مؤثر خواهند بود و به‌جای آزمون برنامه اصلی، تمرکز به آزمون مسیر واگرا شده در کتابخانه معطوف می‌شود و به‌طور پیوسته لازم است تا قسمتی از سیستم‌عامل اندروید به‌صورت نمادین اجرا شود که در کل موجب ایجاد سربار زیاد در آزمون برنامه می‌شود. یک مثال پرکاربرد از این نوع می‌تواند Intentها باشد که سیستم پیام‌رسانی بین مؤلفه‌های مختلف در اندروید است. به‌وسیله Intent یک مقدار به یک مؤلفه در درون یک برنامه یا به مو لف‌های در برنامه دیگر ارسال می‌شود. Intent بعد از خارج شدن از محدوده برنامه وارد کتابخانه‌های سیستمی‌شده و بعدازآن وارد مؤلفه مقصد می شود.

۲-برنامه‌های اندروید رخدادمحور هستند. به این معنی که در اجرای نمادین موتور اجرا باید منتظر کاربر بماند تا با تعامل با برنامه یک رخداد مثل لمس صفحه‌نمایش ایجاد شود. علاوه بر کاربر برنامه‌های ثانویه هم می‌توانند رخداد تولید کنند مثل رخداد تماس ورودی یا دریافت یک پیام.

۳-برنامه‌های کاربردی اندروید وابسته به مجموعه‌ای از کتابخانه‌های اندروید هستند که خارج از دستگاه اندروید یا شبیه‌ساز اندروید در دسترس نیست. کد اندروید برخلاف کد برنامه‌های جاوا که در ماشین مجازی جاوا اجرا

می‌شوند در ماشین مجازی Dalvik اجرا می‌گردد. پس به‌جای java byte-code برنامه‌های اندروید به Dalvik byte-code کامپایلر می‌شوند.

۴-۲-ابزار ACTEV

۴-۲-۱-مقدمه

ابزار ACTEV[۸] اولین ابزاری بود که اجرای پویانمادین را برای تحلیل خودکار برنامه‌های کاربردی گوشی‌های هوشمند پیاده‌سازی کرد. پلتفرم موردنظر ابزار ACTEV ، سیستم‌عامل اندروید است.

برنامه‌های کاربردی اندروید دارای ویژگی‌هایی منحصربه‌فردی هستند که تحلیل ایستا در آن‌ها دچار چالش‌های جدیدی می‌شود. به‌عنوان‌مثال وجود SDK [۱]، ناهمگامی، تعامل میان پردازه‌ای[۲]، پایگاه داده‌ها و رابط کاربری گرافیکی[۳] موجب می‌شود تا تحلیل ایستا برای برنامه‌های اندروید به کارایی و راحتی دیگر پلتفرم‌ها قابل‌اعمال نباشد.

از سویی دیگر به‌طورکلی در تحلیل پویا، هدف تولید پویای ورودی‌های برنامه است.برنامه‌های اندرویدی علاوه بر ورودی‌های عادی که در هر برنامه‌ای از پلتفرم‌های مختلف وجود دارند مانند اعداد و رشته‌ها، رخداد[۴]ها را نیز به‌عنوان ورودی شامل می‌شوند. یک رویداد ضربه[۵] روی صفحه‌نمایش، فشردن یک کلید از دستگاه و یا ورود یک پیامک ازجمله رخدادهای معمول برنامه‌های اندرویدی است.

برنامه‌های اندرویدی زیرمجموعه‌ای از برنامه‌های مبتنی بر رخداد[۶] هستند. این‌گونه برنامه‌ها دارای محاسباتی حاصل از تعامل با ترتیب نامحدودی از رخدادها و پاسخ به آن‌ها می‌باشند. در آزمون این‌گونه از برنامه‌ها با دو چالش جدید علمی روبه‌رو هستیم:

۱. چطور یک رخداد ایجاد می‌شود؟
۲. چطور ترتیبی از رخدادها ایجاد می‌شود؟

---

[۱] Software Development Kit
[۲] Inter-process communication
[۳] GUI
[۴] Event
[۵] Tap event
[۶] Event-Driven Programs

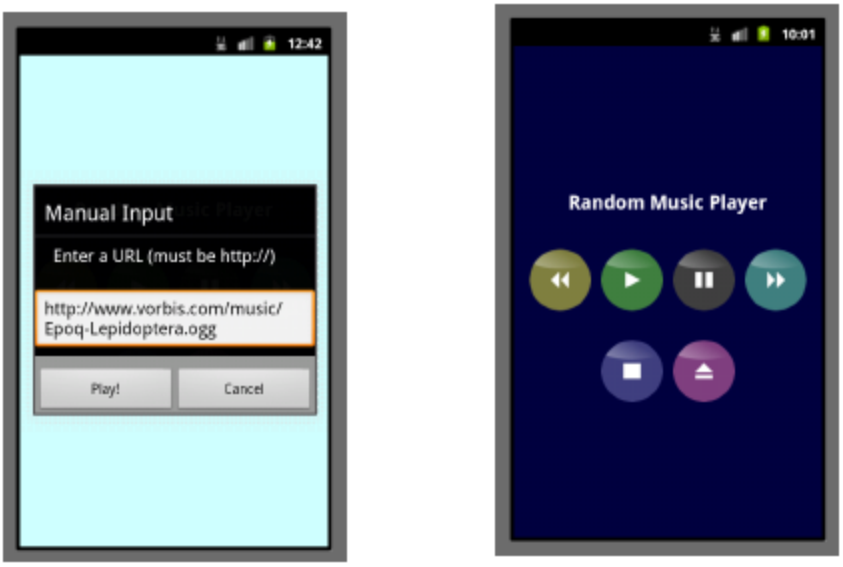
این مقاله از روش اجرای پویانمادین برای تولید رخدادها استفاده می‌کند. درواقع ابزار ACTEV رخدادها را از جایی که تولید می‌شوند تا جایی که کنترل می‌شوند، دنبال می‌کند. در مورد چالش دوم اگر از اجرای پویانمادین مرسوم استفاده شود، در مورد برنامه‌های واقعی با مشکل انفجار مسیر روبه‌رو می‌شویم. درواقع حجم مجموعه رخدادهای ممکن برای اجرای پویانمادین یک برنامه‌ی معمول اندروید به‌قدری زیاد است که منابع سیستم توان پاسخگویی به آن را نخواهند داشت و ابزار در عمل قادر به تست برنامه‌های اندرویدی واقعی نخواهد بود. برای بهبود این موضوع، سعی شده است تا ورودی‌های تولیدی که مجموعه‌ای از رخدادها هستند موردبررسی قرار گیرند. درواقع در مورد هر مجموعه از رخدادها بررسی می‌شود که آیا در مجموعه‌های دیگر ورودی نیز حضور دارند یا خیر. در صورت وجود این مجموعه‌ها در موارد دیگر، این ورودی‌ها اجرا نخواهند شد.

۴-۲-۲-کلیات طرح

برای توضیح طرح از یک برنامه ساده به زبان اندروید استفاده‌شده است. کد مربوط به این برنامه در شکل ۱۲و نمایی از این برنامه اندرویدی نیز در شکل۱۳ آمده است. برنامه‌های اندرویدی خود به‌تنهایی تابع main ندارند. بلکه این برنامه‌ها کامل‌کننده SDK اندروید هستند. درواقع هر برنامه اندرویدی از تعدادی

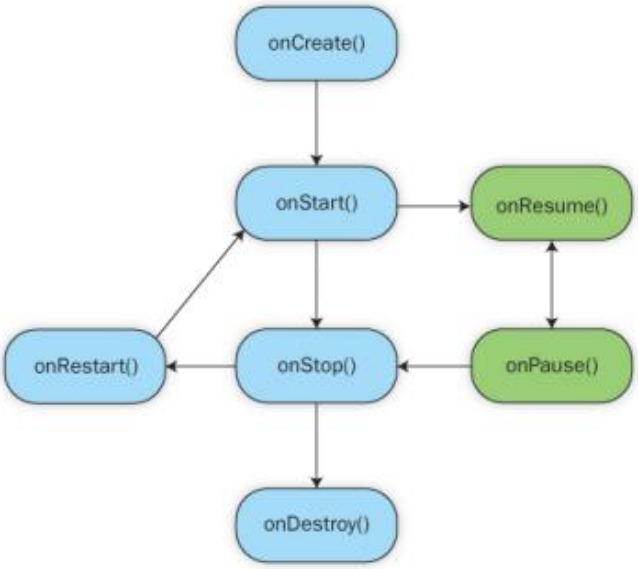
```
public class MainActivity extends Activity {
    Button mRewindButton, mPlayButton, mEjectButton, ...;
    public void onCreate(...) {
        setContentView(R.layout.main);
        mPlayButton = findViewById(R.id.playbutton);
        mPlayButton.setOnClickListener(this);
        ... // similar for other buttons
    }
    public void onClick(View target) {
        if (target == mRewindButton)
            startService(new Intent(ACTION_REWIND));
        else if (target == mPlayButton)
            startService(new Intent(ACTION_PLAY));
        ... // similar for other buttons
        else if (target == mEjectButton)
            showUrlDialog();
    }
}
public class MusicService extends Service {
    MediaPlayer mPlayer;
    enum State { Retrieving, Playing, Paused, Stopped, ... };
    State mState = State.Retrieving;
    public void onStartCommand(Intent i, ...) {
        String a = i.getAction();
        if (a.equals(ACTION_REWIND)) processRewind();
        else if (a.equals(ACTION_PLAY)) processPlay();
        ... // similar for other buttons
    }
    void processRewind() {
        if (mState == State.Playing || mState == State.Paused)
            mPlayer.seekTo(0);
    }
}
```

شکل۱۲- کد برنامه ی اندرویدی برای تست با ابزار ACTEV



شکل ٦- نمایی از برنامه ی اندرویدی برای تست با ابزار ACTEV

Activity تشکیل‌شده است و در ابتدا مشخص نیست که کدام‌یک از این Activity ها فراخوانی می‌شود. هر Activity از چرخه حیات Activity در اندروید پیروی می‌کند. چرخه‌ی حیات یکActivity مطابق شکل ۱۴، شامل توابع onCreate()، onStart()، onResume می‌باشد. در کد موجود در شکل ۱۲



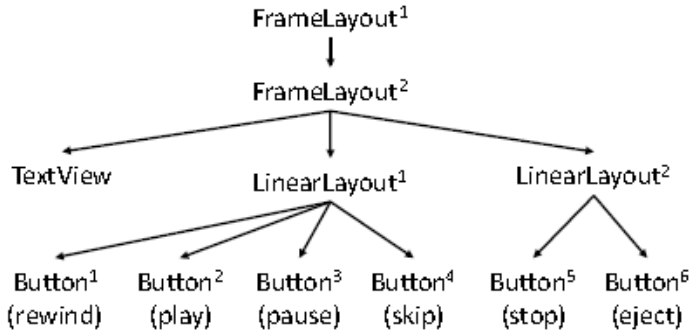
شکل ۸- چرخه ی حیات یک Activity در اندروید

تابع onCreate به همراه تعدادی دکمه تعریف‌شده است. به ازای هریک از این دکمه‌ها، تابع onClickListener مربوطه تعریف‌شده است. کلاس MusicService هم برای اجرای سرویس خاص هر دکمه پیاده‌سازی شده است.

۴-۲-۳-تولید یک رخداد

در برنامه‌ی اندرویدی شکل۱۴ رخداد ضربه[۱] در نظر گرفته‌شده است. شکل۱۶ نمایی از سلسله‌مراتب ویجت[۲]های موجود در صفحه اصلی برنامه را نشان می‌دهد. برگ‌های درخت درنهایت دکمه‌های روی صفحه و نمایشگر متن[۳] هستند. هدف، تولید رخدادهایی است که برای همه این یازده ویجت رخداد ضربه را به‌طور خودکار تولید نماید.

در روش‌های گذشته این درخت یا به‌صورت دستی[۴](Model-based) یا به‌صورت خودکار(Capture-Replay)



شکل ۱۵- سلسله مراتب ویجت ها در صفحه ی اصلی برنامه

تولید می‌شده است. اما درروش خودکار طرح به‌صورت موردی بوده و همه ویجت‌ها، چه ویجت‌های SDK و چه ویجت‌های سفارشی، پشتیبانی نمی‌شوند. در این مقاله از اجرای پویانمادین برای تولید رخدادها استفاده می‌شود. برای این منظور SDK و برنامه تحت آزمون باید تجهیز شوند. سپس در حین اجرای یک رخداد عددی، یک رخداد به‌صورت نمادین هم تولید می‌شود که تمام قیدهای مسیر را در خود نگهداری می‌کند. با این روش می‌توان رخداد ضربه را برای هر یازده ویجت ایجاد کرد.

۱ Tap Event
۲ Wiget
۳ TextView
۴ Manual

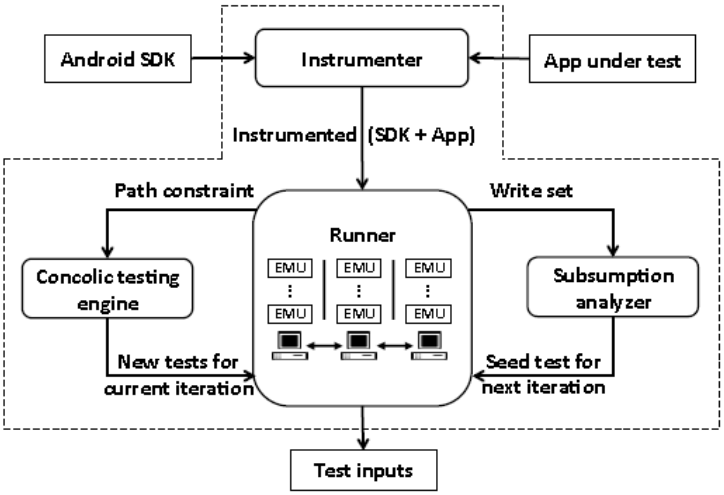
۴-۲-۴-تولید ترتیبی از رخدادها

برای اینکه تمام‌مسیرهای برنامه بررسی شوند، باید تمام رخدادها و ترتیب‌های مختلف از وقوع آن‌ها تولید شوند. استفاده از روش سنتی بدون بهینه‌سازی باعث انفجار مسیر می‌شود که ۲۱ هزار ترتیب چهارتایی از رخدادها را باید تولید کرد.

برای حل مسئله‌ی انفجار مسیر در این مقاله، مجموعه رخدادهای تولیدشده موردبررسی قرارگرفته‌اند. بررسی‌ها نشان می‌دهد که تعداد زیادی از مجموعه رخدادهای تولیدشده تکراری هستند. درواقع برنامه‌ها را می‌توان به شکل یک نمودار حالت[۱] درآورد که با هر رخداد حالت برنامه تغییر می‌کند. تعدادی از رخدادهای تولیدشده توسط اجرای پویانمادین حالت برنامه را تغییر نمی‌دهند و ازآنجایی‌که در حافظه چیزی نمی‌نویسند به آن‌ها رخدادهای فقط خواندنی[۲] گفته می‌شود. برای جلوگیری از تولید این‌گونه رخدادها موارد زیر در نظر گرفته‌شده است:

۱. تعدادی از ویجت‌ها مثل linearLayout و FrameLayout برخلاف دکمه‌ها هنگام رخداد دارای کنش[۳] نیستند. پس تنها شش ویجت از یازده ویجت موجود در صفحه امکان اجرا دارند.
۲. تعدادی از ویجت‌هایی که فعالیت دارند ممکن است به دلایل مشخصی که برنامه‌نویس صلاح دیده است، غیرفعال باشند.
۳. در برنامه‌های دارای رابطه کاربری، هنگامی‌که یک رخداد تولید می‌شود برای جلوگیری از دوباره طراحی صفحه‌نمایش، اگر رخداد، حالت برنامه را تغییر نمی‌دهد، رخداد فقط خواندنی در نظر گرفته می‌شود.
۴. در SDK تعدادی زیادی از رخدادها تعریف‌شده‌اند که ممکن است یک برنامه خاص به آن‌ها واکنش نشان ندهد. در این صورت این‌گونه رخدادها هم فقط خواندنی خواهند بود. مثل رخداد تماس تلفنی ورودی[۴].

[۱] State Transition Diagram
[۲] Read Only
[۳] Action
[۴] Incoming Call



شکل ۱۶-ساختار کلی ابزار ACTEV

درنهایت درروش ارائه‌شده مجموعه رخدادهایی که به یک رخداد فقط خواندنی ختم می‌شوند، حذف خواهند شد. چون عملاً تأثیری در آزمون برنامه نخواهند داشت. حذف این‌گونه ترتیب رخدادها ۲۱ هزار ترتیب رخداد ۴تایی را به حدود ۳هزار ترتیب رخداد ۴تایی کاهش می‌دهد. معماری ابزار ارائه‌شده در شکل ۱۶ نشان داده‌شده است.

۴-۳-ابزار Condroid

۴-۳-۱-مقدمه

در سال‌های اخیر ابزارهای بسیاری باهدف تحلیل برنامه‌های اندروید نوشته‌شده‌اند. یک نقطه‌ضعف اصلی در مورد این ابزارها عدم توانایی رسیدن به نقاط حساس کد است. به‌عنوان‌مثال بخش‌های بارگذاری شده پویا[۱] و یا بخش‌هایی از کد که در آن قسمت داده‌ی خاصی را ترجمه‌ی رمز می‌کنند نقاط حساسی هستند که غالباً از دید ابزارهای تحلیل کد اندروید پنهان می‌ماند. در حقیقت نقطه‌ی ضعف اصلی این ابزارها آن است که لزوماً به بخش‌هایی از کد برنامه که دارای تهدیدات امنیتی است دست نمی‌یابند زیرا این ابزارها شرایط پویایی را که مسیرهای مشخصی از کد برای اجراشدن به آن نیاز دارند تأمین نمی‌کنند.

[۱] dynamically loaded

برای حل این چالش علمی، ابزار Cindroid [۹]یک رویکرد تحلیل ایستای فراخوانی مسیر[۱] و bytecode instrumentation به همراه اجرای نمادین ابتکاری را در پیش‌گرفته است. درواقع هدف از این رویکرد دستیابی به مسیرهای خاص موجود در کد برای تحلیل تهدیدات امنیتی است. این ابزار می‌تواند در حین تحلیل پویا، برنامه را مجبور به نشان دادن کد بارگذاری شده پویا نماید درواقع مهاجمین با استفاده از همین تکنیک ساده‌ی کد بارگذاری شده‌ی پویا می‌توانند[۲] ابزار Google Bouncer را دور بزنند و برنامه‌ی اندروید موذیانه‌ی خود را به‌عنوان یک برنامه‌ی کاربردی اندروید در فروشگاه مهم و جهانی GooglePlay منتشر کنند. در پایان این پژوهش با استفاده از یک مثال ساده‌ی بمب منطقی در اندروید نحوه‌ی کارایی ابزار Condroid را نشان داده‌شده است.

۴-۳-۲-بررسی اجمالی یک مثال

رویکرد این ابزار مبتنی بر تحلیل ایستای جریان کنترل و اجرای هیبرید پویانمادین باهدف بررسی مسیرهای اجرایی از کد که دارای نقاط حساس هستند می‌باشد. در شکل ۱۸ یک بمب منطقی مبتنی بر زمان به همراه کد بارگذاری شده‌ی پویا نشان داده‌شده است. فرض می‌کنیم که تابع این مثال بخشی از یک برنامه‌ی اندرویدی است که به‌عنوان BroadcastReceiver برای پیامک ورودی نوشته‌شده است . بنابراین چهارچوب اندروید هرزمان که پیامکی به سیستم وارد شود این تابع را فراخوانی می‌کند. با توجه به شروط موجود در خط ۳ و ۴ تنها زمانی که تاریخ سیستم از ۲۰۱۷/۰۱/۰۱ فراتر باشد و کد بر روی یک پردازنده‌ی واقعی[۳] و نه یک شبیه‌ساز اجراشده باشد کد موذیانه‌ی مخرب به‌صورت پویا بارگذاری خواهد شد.

ازآنجاکه بارگذاری پویای کد مخرب تنها به ورودی کاربر وابسته نیست بلکه به شرایط محیط اجرای کد نیز وابسته است بنابراین روش‌های fuzzing ورودی هرگز نمی‌توانند شروط این بخش از کد را برآورده کنند و کد مخرب را به‌صورت پویا بارگذاری نمایند. اما ابزار Condroid با instrument کردن برنامه و بدون هرگونه تغییر در چهارچوب اندروید قادر به اجرای مسیر منتهی به بارگذاری پویای کد است.

[۱] static call path analysis

[۲] تا سال ۲۰۱۵ میلادی این امکان وجود داشت زیرا ابزار Google Bouncer که مسئول ارزیابی امنیتی برنامه‌های موجود در Google play می‌باشد. صرفاً از تحلیل ایستا بهره می‌برد. ممکن است نسخه‌های جدید Google Bouncer بهبودیافته باشند که نیازمند بررسی بیش‌تری است.

[۳] Goldfish kernel

```
public class Mallory extends BroadcastReceiver {
 public void onReceive(Context ctx, Intent i) {
  ...
  if (System.currentTimeMillis()>1483228800) {
   if (android.os.Build.BRAND!=null) {
    if (!android.os.Build.BOARD.contains("goldfish")
    ){
     DexClassLoader dcl = new DexClassLoader(libpath
     ,
                 dexOutDir.getAbsolutePath(),
                 null,
                 ClassLoader.getSystemClassLoader());
     Class<?> clazz = dcl.loadClass("SomeClass");
    }
   }
  }
 }
}
```

شکل۱۷- بمب منطقی مبتنی بر زمان به همراه کد بارگذاری شده‌ی پویا

۴-۳-۳-چهارچوب ابزارCondroid

مطابق شکل۱ ابزار Condroid مراحل زیر را برای تحلیل برنامه‌های اندرویدی طی می‌کند:

این تحلیل باهدف یافتن مسیرهایی از برنامه که دارای کد حساس هستند انجام می‌گیرد.

۱. اجرای پویانمادین سازگار

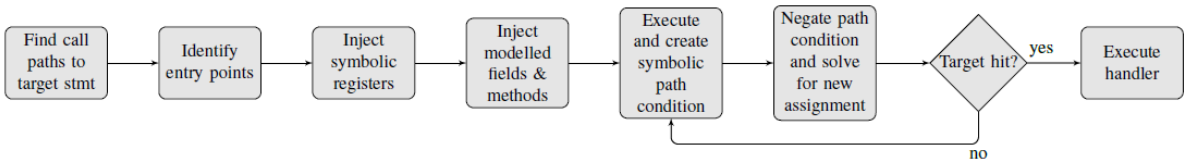
الف) instrument کردن برنامه برای مسیرهای اجرایی منتهی به کد حساس تا بتوان در زمان اجرا ثبات‌ها را بازنویسی کرد.

ب)instrument کردن برنامه باهدف استخراج قید مسیر.

پ) حل‌کننده‌ی قیدی که قید مسیر را به‌عنوان ورودی برای مسیرهای مختلف می‌گیرد و آخرین شرط آن را نقیض می‌کند؛ آنگاه سعی می‌کند تا عبارت شرطی جدید حاصل‌شده را حل نماید.

این حل جدید به این معناست که مسیر جدید به‌دست‌آمده نیز امکان‌پذیر است.

ت)مقادیر ثبات‌ها برای اجرای بعدی که مسیر اجرایی متفاوتی است تنظیم می‌شود.



شکل ۱۸-ساختار کلی ابزار Condroid

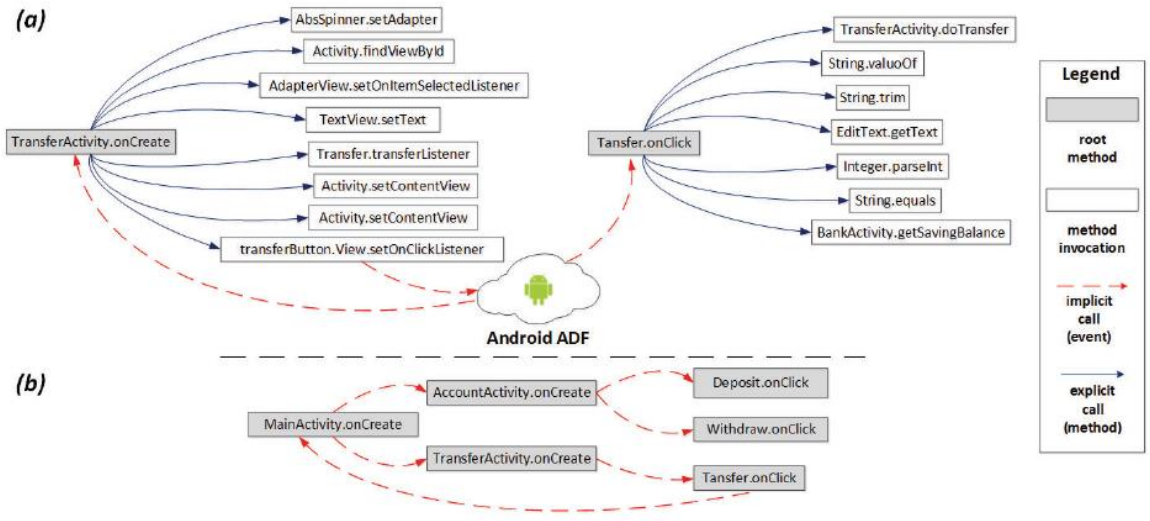
۴-۴-ابزار SIG-Droid

۴-۴-۱-مقدمه

همان طور که در شکل ۱۹ دیده می‌شود، SIG-Droid [۱۰] از سه قسمت اصلی تشکیل‌شده است:

۱. Behavior Model: در این قسمت از ابزار مورد کاربردهای برنامه و رشته‌ای از رخدادهای مرتبط با هر یک استخراج می‌شود. برای این کار با استفاده از ابزار MoDiscoکه ابزار استخراج گراف فراخوانی برنامه جاوا است،گراف فراخوانی توابع استخراج می‌گردد. ازآنجاکه برنامه‌های اندرویدی با سیستم‌عامل دارای تعاملات بسیاری هستند گراف‌های استخراج‌شده از آن‌ها برخلاف گراف‌های برآمده از برنامه‌های جاوا، گراف‌هایی یکپارچه نیستند و از تعدادی زیر گراف تشکیل‌شده‌اند.

۲. SIG-Droid با بررسی کد گراف‌های استخراج‌شده را به هم وصل کرده و گره‌های میانی گراف را حذف می‌کند. نمونه‌ای از این گراف در شکل ۱۹ آمده است.

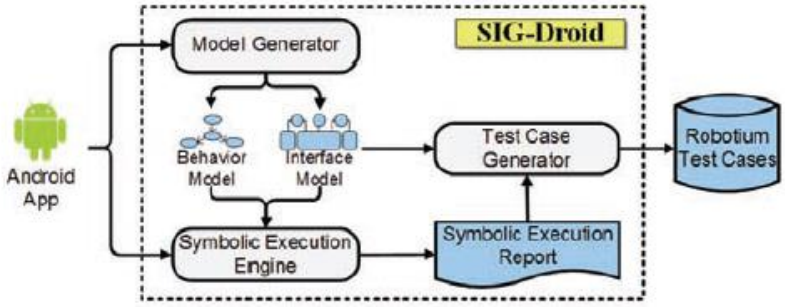


شکل ۱۹- گراف حاصل از دو زیرفراخوانی یک نرم افزار اندرویدی بانکی

۳. Interface Model: در اندروید در کنار کلاس‌ها و کدهایی که به زبان جاوا نوشته می‌شوند، تعدادی فایل به زبان xml وجود دارند که مشخصات Activityها (فایل manifest.xml) و ساختار و چینش اجزای هر Activity در آن‌ها ذخیره می‌شود. با تحلیل ایستای این فایل‌ها، ابزار اطلاعات مربوط به UI و اجزای هر صفحه (widgetها) را استخراج می‌کند.

۴. اجرای نمادین: همان‌طور که در ابتدا گفته شد برنامه‌های اندروید سه تفاوت عمده با برنامه‌های عادی دارند که در اجرای نمادین باید به آن‌ها توجه شود.

- برنامه‌های اندروید به‌جای java byte-code به dalvik byte-code تبدیل می‌شوند. پس برای اینکه بتوان از موتورهای اجرای نمادین مربوط به جاوا استفاده کرد، باید کدهای اندروید را با کامپایلر جاوا کامپایل کرد. اما همان‌طور که گفته شد، کتابخانه‌های اندروید در جاوا وجود ندارند و عملیات کامپایل نیاز دارد که این کتابخانه ها و فراخوانی به آن‌ها با تعدادی کلاس (stub) شبیه‌سازی شوند.
- مشکل دیگر واگرایی مسیر هست. برای حل این موضوع کتابخانه‌های ساختگی(mock) ایجاد



شکل ۲۰-ساختار کلی ابزار SIG-droid

می‌شوند که تنها یک مقدار دلخواه به‌عنوان خروجی تولید می‌کنند. با این کار ابزار درگیر آزمون کتابخانه‌های خارج از برنامه نمی‌شود.

- برای آزمون جنبه‌های مختلف یک برنامه لازم است تا رشته‌ای از رخدادها تولید شود. برای این کار با استفاده از Behavior Model تعدادی کلاس Driver برای این کار نوشته می‌شود. برای تولید رشته‌های مختلف در این ابزار گراف BM با روش DFS پیمایش می‌شود.
- موضوع آخر مشخص کردن ورودی‌های نمادین برنامه هست. به این منظور با استفاده از Interface Model و بررسی ویجت‌های مختلف ورودی‌های نمادین تعیین می‌شوند. به‌عنوان‌مثال اگر در برنامه‌ای یک TextBox وجود داشته باشد، تمام متغیرهایی از کد که

مقدار ورودی در این TextBox را در خود ذخیره می‌کنند به‌عنوان متغیر نمادین در نظر گرفته می‌شود.

۵. مؤلفه تولید مورد آزمون: با استفاده از Interface Model و گزارش موتور اجرای نمادین تعدادی مورد آزمون برای اجرا به‌وسیله Robotium تولید می‌شود.

هدف اصلی SIG-Droid پوشش هر چه بیشتر مسیرهای موجود در برنامه است. برای اندازگیری میزان پوشش کد برنامه‌ی اندرویدی از ابزار متن‌باز EMMA استفاده می‌شود. موتور اجرای نمادین در ابزار Sig-Droid نیز JPF [۱] است که باهدف اجرای نمادین برای برنامه‌های اندروید تغییریافته است.

۴-۵-ابزار APPINTENT

AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection [۱۱]

APPINTENT : تحلیل داده‌های حساس انتقالی در اندروید برای تشخیص نقض حریم خصوصی

۴-۵-۱-مقدمه

امروزه با گسترش گوشی‌های اندرویدی و وجود داده‌های حساس افراد در آن‌ها،حریم خصوصی کاربران اندروید با تهدید دژافزار های اندروید روبه‌رو است. با استفاده از روش‌های شناخته‌شده در نوشتگان این حوزه‌ی علمی می‌توان انتقال داده از گوشی تلفن همراه به خارج از آن را تشخیص داد. اما نسبت به انتقال داده‌های حساس درون‌گوشی تلفن همراه بررسی صورت نگرفته است. این انتقال لزوماً نشان‌دهنده‌ی نقض حریم خصوصی نیست. درواقع تنها در صورتی نقض حریم خصوصی رخ‌داده است که این انتقال داده‌های حساس در درون‌گوشی همراه و بدون اطلاع و خواست کاربر باشد. اکنون ما با این چالش علمی مواجه هستیم که از کجا بدانیم انتقال داده‌ها با خواست کاربر انجام‌گرفته با بدون اطلاع او؟ به‌عنوان را حل این چالش علمی ابزار APPINTENT ارائه‌شده است. با استفاده از این ابزار برای هر انتقال داده‌ای دنباله‌ای از تغییرات رابط کاربری گرافیکی که نشان‌دهنده‌ی دنباله‌ای از رویدادها برای انجام این انتقال است، رسم می‌شود. این دنباله‌ی تغییرات در رابط کاربری گرافیکی به تحلیل‌گر کمک می‌کند تا راحت‌تر تصمیم بگیرد که یک انتقال داده با خواست کاربر صورت گرفته است یا خیر؟

[۱] Java Path Finder

ایده‌ی اصلی ابزار APPINTENT استفاده از اجرای نمادین برای به دست آوردن دنباله‌ی رویدادهایی است که موجب یک انتقال داده‌ی مشخص درون‌گوشی همراه شده‌اند. اما اجرای نمادین در کنار مزایای قابل‌توجه ای که در اختیار می‌گذارد ازنظر مصرف حافظه و زمان بسیار ناکارآمد است. نوآوری علمی ابزار APPINTENT ارائه‌ی بهبودی برای اجرای نمادین با کاهش فضای جست جو در برنامه‌های اندروید و بدون از دست رفتن پوشش کد بالا است. در پایان برای ارزیابی این پژوهش ۷۵۰ برنامه‌ی اندرویدی با APPINTENT مرد تحلیل قرارگرفته‌اند که نتایج قابل‌توجه ای برای تشخیص برنامه‌های ناقض حریم خصوصی به دست آمد.

۴-۵-۲-معماری کلی ابزار APPINTENT

ابزار APPINTENT یک ابزار کاملاً خودکار برای تشخیص انتقال داده‌ی ناخواسته در برنامه‌های اندروید نیست. درواقع تشخیص کاملاً خودکار ناخواسته بودن یک انتقال داده، در عمل ناممکن است. اما این ابزار برای اولین بار در این حوزه‌ی علمی سعی در ارائه‌ی یک دنباله از تغییرات رابط گرافیکی کاربر به ازای هر انتقال داده را دارد. این دنباله از تغییرات به تحلیل‌گر می‌کند تا راحت‌تر خواسته یا ناخواسته بودن انتقال داده‌ها را تشخیص دهد.

در این ابزار دستیابی به سه هدف زیر موردتوجه قرار دارد:

۱- ایجاد داده‌های خاص ورودی که منجر به انتقال داده‌های حساس در برنامه‌های اندرویدی گردد. داده‌های ورودی به برنامه‌های اندرویدی به‌طورکلی به دودسته تقسیم می‌شوند:
   a. الف) داده‌های ورودی که متن خاصی را شامل می‌شوند و از بیرون از برنامه وارد می‌گردند.
   b. ب)رویدادهای ورودی که ناشی از تعامل کاربر با رابط گرافیکی برنامه‌ی کاربردی و یا ارتباطات میان پردازه‌ای سیستم می‌باشد.
۲- پوشش قابل قبولی از کد تأمین گردد. درواقع هدف آن است که تمامی مسیرهای برنامه که ممکن است موجب نقض حریم خصوصی شود، پیمایش گردد. ضمناً ازآنجا دقت ابزار تحلیل حیاتی است، باید مثبت کاذب و منفی کاذب ابزار نیز جزئی باشد.
۳- ابزار باید برای تحلیل‌گر انسانی قابل‌فهم و ساده باشد.

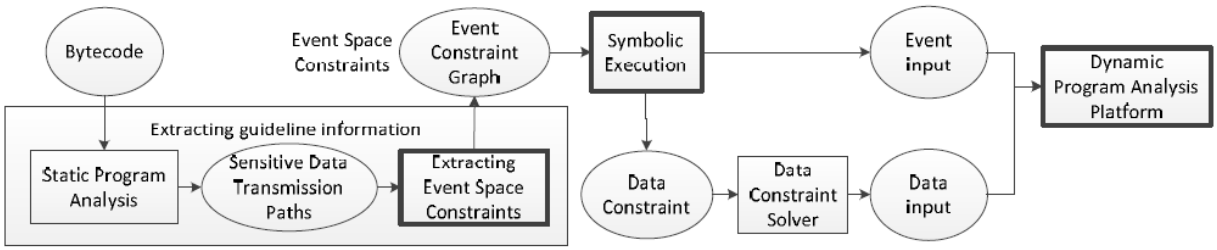
در شکل ۱ معماری کلی ابزار APPINTENT را مشاهده می‌نمایید. در این ابزار یک برنامه‌ی کاربردی اندروید در دو مرحله مورد تحلیل قرار می‌گیرد:

۱- اجرای نمادین هدایت‌شده در فضای رویداد[۱]

هدف از این مرحله، تولید داده‌های ورودی برای اجرای برنامه به‌گونه‌ای است که مسیرهای مختلف انتقال داده‌ها پیمایش شوند. در ابزار APPINTENT از تحلیل آلایش ایستا استفاده‌شده است که با استفاده از آن تمامی انتقال داده‌های حساس و دنباله‌ی رویدادهای مربوط به آن‌ها استخراج می شود. در ادامه با اجرای نمادین هدایت‌شده توسط اطلاعات به‌دست‌آمده از تحلیل آلایش ایستا ، ورودی‌های حساس برای برنامه تولید می‌شود. پوشش کد کافی نیز بنابر ماهیت ذاتی اجرای نمادین به دست می‌آید.

۲- پلتفرم تحلیل پویای برنامه

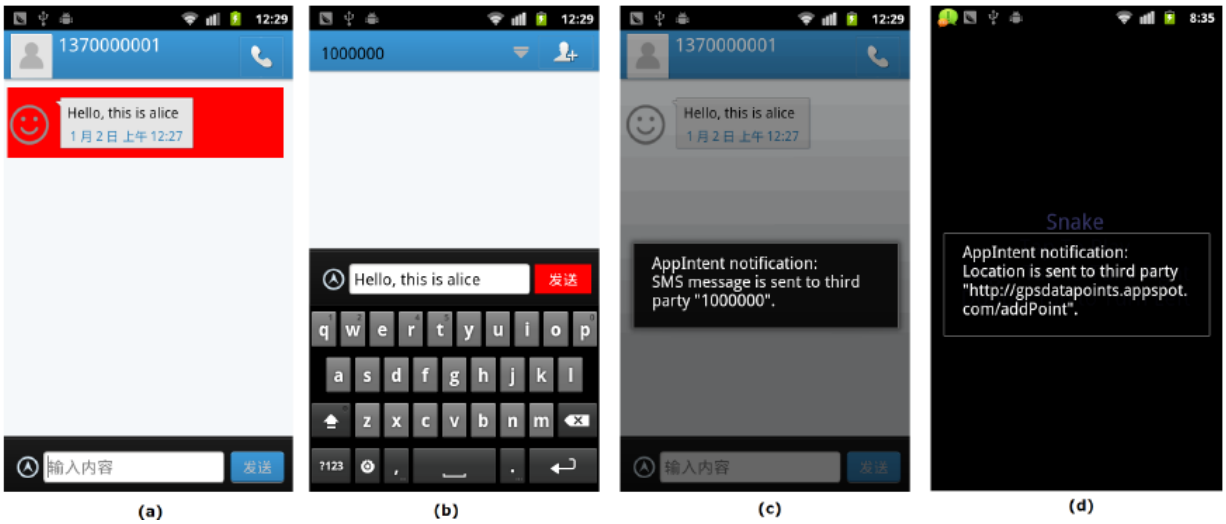
با استفاده از داده‌های ورودی به‌دست‌آمده در مرحله‌ی ۱ در یک پلتفرم تحلیل پویا به اجرای خودکار برنامه‌ی اندرویدی می‌پردازیم. مراحل مختلف انتقال داده‌های حساس به‌صورت گرافیکی ثبت می‌شوند تا تحلیل‌گر راحت‌تر بتواند در مرد خواسته یا ناخواسته بودن انتقال تصمیم‌گیری کند.



شکل ۲۱-معماری کلی ابزار APPINTENT

نمونه‌ای از اجرای ابزار APPINTENT را برای تحلیل یک برنامه‌ی اندرویدی در شکل ۱ مشاهده می‌نمایید.

[۱] Event-space Constraint Guided Symbolic Execution



شکل ۲۲- اسکرین شات از مراحل مختلف یک انتقال داده

۴-۶-مقایسه ابزارهای اجرای پویانمادین برای تحلیل برنامه‌های اندرویدی

جدول ۱-مقایسه‌ی ابزارهای اجرای پویانمادین برای تحلیل برنامه‌های اندرویدی

ابزار	سال ارائه	هدف	ویژگی‌ها
ACTEV	**۲۰۱۲**	**آزمون نرم‌افزار**	چگونه می‌توان یک رویداد را معتبر تولید کرد؟ چگونه می‌توان دنباله‌ی معتبری از رویدادها را تولید کرد؟ محدود به رخداد ضربه نیاز به بهینه‌سازی برای کاهش فضای حالت در ترتیب‌های مختلف از رخدادها انفجار مسیر
APP-Intent	**۲۰۱۳**	**تشخیص نقض حریم خصوصی**	انتقال داده بین برنامه‌های مختلف اندروید آیا با خواست کاربر بوده یا خیر؟ ارائه‌ی یک دنباله از تغییرات رابط گرافیکی کاربر ایجاد داده‌های خاص ورودی که منجر به انتقال دادهای حساس در برنامه‌های اندرویدی گردد
Sig-Droid	**۲۰۱۵**	**آزمون نرم‌افزار**	استخراج گراف کنترل جریان(CFG) با MoDisco ساختmock و stub به‌صورت خودکار
Condroid	**۲۰۱۵**	**تحلیل پویا هدفمند**	تحلیل ایستای مقدماتی– استخراج گراف کنترل جریان(CFG) باهدف به دست آوردن entry point برای تست اجرای پویانمادین برنامه انفجار مسیر

فصل پنجم:

روش‌های تشخیص دژافزار

۵-۱- تحلیل ایستا

در یک محیط تحلیل دژافزار تحلیل ایستا، ساده‌ترین و ابتدایی‌ترین روش تشخیص دژافزار است. از طریق تجزیه‌وتحلیل ایستا، سامانه می‌تواند تعیین کند که آیا فایل دارای الگوهایی که نشان دهد یک دژافزار وجود دارد، هست یا خیر. به‌عنوان‌مثال، آیا اسکریپت‌های اجرایی در فایل تعبیه‌شده و یا اتصال به یک سرور ناشناخته و یا مشکوک وجود دارد. تحلیل ایستا یک‌راه سریع و دقیق برای تشخیص دژ افزارهای شناخته‌شده است که بخش عمده‌ای از حملات رایانه‌ای به سازمان‌ها را شامل می‌شود.

برخی از سامانه‌ها تحلیل ایستا از یادگیری ماشین برای تشخیص استفاده می‌کنند. یادگیری ماشین شامل ایجاد و خودکارسازی یک سامانه برای طبقه‌بندی رفتارهای مخرب به گروه‌های متفاوت است. این گروه‌ها می‌توانند برای شناسایی کد مخرب در آینده بدون نیاز به ساخت الگوی دستی مورداستفاده قرار گیرند.

اگر شباهت‌های بین کد مشکوک به‌اندازه کافی زیاد باشد سامانه به‌طور خودکار یک امضا از دژافزار ایجاد می‌کند و در سراسر شبکه آن را مورد جست‌وجو قرار می‌دهد. هرچه دژ افزارهای بیشتری، به‌عنوان نمونه‌ی آموزشی موردبررسی قرار گیرد و فهرست شود، توانایی سامانه برای تشخیص حملات در طول زمان رشد می‌یابد.

۵-۲-تحلیل پویا

اگر یک فایل مشکوک نتواند از طریق تحلیل ایستا مدیریت شود، باید جزئیات بیشتری از آن همراه با رفتار میزبان و شبکه موردبررسی قرار گیرد. آنچه به‌عنوان تحلیل پویا شناخته می‌شود، معمولاً شامل ارسال یک نمونه مشکوک به یک محیط مبتنی بر ماشین مجازی و سپس فعال کردن آن در یک محیط کاملاً کنترل‌شده[۱] است که رفتار آن مشاهده می‌شود و از آن اطلاعاتی استخراج می‌گردد. در موارد پیچیده ،زمانی که دژافزار بتواند تشخیص دهد که محیط اجرا یک محیط واقعی نیست بلکه یک ماشین مجازی است؛ ممکن است تحلیل عمیق‌تری لازم باشد. تحلیل پویا به‌طور خاص در پیدا کردن بهره‌برداری‌های روز-صفرم در دژافزارها مناسب است.

[۱] Sandbox

ازآنجاکه تحلیل ایستا و یادگیری ماشین هر دو نیاز به آشنایی قبلی با آن خانواده از دژافزار دارند، برای آن‌ها شناسایی فعالیت‌های مخرب کاملاً جدید دشوار است. چالش تحلیل پویا مقیاس‌پذیری آن است که نیاز به محاسبات عظیم، ذخیره‌سازی و خودکارسازی دارد. یکی از ایده‌ها آن است که برای تشخیص دژافزار از هر دو تحلیل ایستا و پویا در کنار هم استفاده شود.

۵-۳-روش‌های تشخیص دژافزار های اندرویدی

کارهای جدی در حوزه‌ی تشخیص دژافزار های اندرویدی از سال ۲۰۰۹ با مقالات [۱۲] و [۱۳] شروع شد. مقاله‌ی [۱۲] اولین کار روی اندروید برای تشخیص دژافزار بوده است.این کار تحلیل ایستا را بر روی فایل‌های elf [۱] باینری انجام می‌دهد. در این فایل‌ها فراخوانی‌های تابعی[۲] و نام فایل‌های تغییریافته ذخیره می‌شوند.صحت تشخیص اعلام‌شده در این روش تا ۹۶ درصد و نرخ مثبت کاذب آن نیز ۱۰ درصد بوده است. مشکل این پژوهش این بود دژافزار هایی که بر روی آن‌ها تست انجام گرفت، دژافزار هایی بودند که توسط خود نویسندگان مقاله نوشته‌شده بود به همین علت داده‌ی کمی برای آزمون روش پیشنهادی‌شان داشتند. از سویی دیگر آزمون بر روی دستگاه‌های اندرویدی انجام‌نشده بود.

کار بعدی [۱۳] در همان سال با استفاده از رویکرد تحلیل ایستا با استفاده از قوانین ترکیبی از مجوزها، انجام‌شده است.در این روش برنامه‌های ناخواسته بلوکه می‌شوند.البته هدف اصلی این روش تشخیص آسیب‌پذیری است.

در سال بعد مقاله‌ی [۱۴] ارائه شد که روش ترکیبی ایستا و پویا را برای تشخیص ارائه داده بود. در این کار در مرحله‌ی اول فایل apk را از حالت کامپایل خارج کرده و به کد جاوا تبدیل می‌کنند. سپس در این کد به دنبال الگوی مخرب می‌گردند و آن‌ها را علامت‌گذاری می‌کنند. سپس تعداد فراخوانی‌های سیستمی در زمان اجرا شمرده می‌شود و از روی این دو ویژگی تشخیص را انجام می‌دهد. مشکل این مقاله درصد صحت پایین و هم‌چنین نبود کاربرد واقعی در آزمایش است.

در همان سال ۲۰۱۰ مقاله‌ی دیگری[۱۵] درزمینهٔ ی تحلیل ایستا ارائه شد که همین مشکل یعنی عدم وجود کاربرد واقعی و شبیه‌سازی بودن را داشت. در این مقاله ویژگی‌ها و رخدادهای موبایل مانند کارکرد پردازشگر،

[۱] Executable and Linkable Format
[۲] Function Call

تعداد بسته‌های ارسالی از wifi، تعداد پردازه‌ها، تعداد فشرده شدن کلید یا لمس صفحه نظارت می‌شوند.سپس از روش یادگیری ماشین استفاده می‌کنند تا داده‌ها را به دودسته‌ی عادی و خطرناک تقسیم کند.

مقاله‌ی مهم بعدی در سال ۲۰۱۱ ارائه شد[۱۶]. روش کار این مقاله تحلیل پویا است. به این صورت که با اجرای برنامه، ،Trace های آن‌ها(برای مثال فراخوانی‌های سیتمی برای مجموعه ای از کاربران در زمان اجرا) را به دست آورده و. در فایل log ذخیره می‌کنند. این فایل برای سرور فرستاده می‌شود. در آن جا باروش k-means خوشه بندی صورت می گیرد تا رفتار طبیعی و مخرب از یکدیگر شناسایی شوند. این مقاله در مقیاس کوچکی کار کرده و مقیاس پذیر نیست.علاوه بر این فرستادن اطلاعات خام به سرور می تواند مشکل آفرین باشد.

مقاله ی[۱۷] از هردو روش ایستا و پویا برای ساخت پروفایل رفتار بهره برده است.از روش مبتنی بر ویژگی و تحلیل مجوز برای دژافزار های شناختهش ده و از روش اکتشافی برای تشخیص دژافزار های ناشناخته استفاده می‌شود. از یک واحد هسته برای ثبت فراخوانی های سیستمی که توسط کد های بهره بردار شناخته شده ی اندروید که از دسترسی مدیر استفاده می‌کنند، استفاده می‌شود.نرخ منفی کاذب این روش به دلیل استفاده از روش اکتشافی بالاست.

در مقاله‌ی[۱۸] تشخیص دژافزار به‌صورت خودکار از روی رفتار شبکه انجام می‌شود.پس از تشخیص اقدام به از بین بردن دژافزار و برگرداندن دستگاه به حالت عادی می‌کند. اصل عملیات تشخیص بر روی شبکه توسط سیستم تشخیص نفوذ انجام می‌شود به همین علت حافظه‌ی زیادی از موبایل اشغال نمی‌گردد. زمانی که مانیتورینگ شبکه ترافیک غیرطبیعی را شناسایی کرد به برنامه اطلاع می‌دهد. عامل داخل دستگاه فرآیندی را که منجر به تولید آن ترافیک شده است را یافته و عملیات بازگردانی را انجام می‌دهد. این کار می‌تواند شامل فیلتر کردن ترافیک مشکوک، sand box کردن یا حذف کردن برنامه و یا بازیابی تنظیمات کارخانه باشد.

بهترین مقاله‌ی ارائه‌شده در این زمینه با رویکرد تحلیل ایستا در سال ۲۰۱۴ اراده شد[۱۹]. در این مقاله ابزاری ارائه‌شده که با تحلیل و استفاده از ویژگی‌های ایستا، تشخیص می‌دهد برنامه دژافزار است یا خیر. این تشخیص در خود موبایل صورت می‌گیرد. ازجمله ویژگی‌هایی که می‌توانیم از آن‌ها برای تحلیل ایستا برنامه استفاده کنیم، می‌توان به مؤلفه‌های سخت‌افزاری که برنامه از آن‌ها استفاده می‌کند، مجوزهای برنامه،مؤلفه‌های برنامه،رابط بین مؤلفه‌ها[۱]، فراخوانی‌های API و آدرس‌های IP مورداستفاده توسط برنامه اشاره کرد. در ادامه در این مقاله با

[۱] Intent

استفاده از روش‌های یادگیری ماشین(SVM خطی) برنامه را به یکی از دو حالت آسیب‌رسان و عادی تقسیم می کند و تشخیص را انجام می‌دهد. این پژوهش هم ازنظر سرعت و هم ازنظر دقت عملکرد بسیار مطلوبی را داشته است. علاوه بر این مجموعه‌ی ویژگی‌های مورداستفاده آن نیز به‌آسانی از هر فایل APK حتی مبهم شده نیز قابل‌دسترسی است.

مقالات بسیار دیگری در سال‌های بعد نظیر [۲۰]و[۲۱] مطرح شدند که بر اساس مجوزهای برنامه کاوش انجام می‌دادند. کار [۲۲]که یک sand box برای تحلیل دژافزار ها ارائه داد و کار [۲۳]که یک چهارچوب برای تشخیص و بهبود گوشی‌های هوشمند ارائه داده است و بسیار از کارهای مشابه دیگر در سال‌های اخیر مطرح شدند اما به دلیل مشابهت از شرح آن‌ها صرف‌نظر می‌کنیم.

فصل ششم:

کارهای پیشین در اجرای پویانمادین باهدف تشخیص یا تحلیل دژافزار

Automatically Identifying Trigger-based Behavior in Malware -۱-۶

۶-۱-۱-مقدمه

برخی از دژافزار ها شامل رفتارهای موذیانه‌ی پنهانی هستند که تنها در شرایط خاصی این رفتارها را به اجرا می گذارند. چند نمونه مشهور از این دژافزار ها عبارت‌اند از:

۱. کرم واره[۱] MyDoom که حملات منع سرویس توزیع‌شده[۲] را در زمان‌های مشخصی صورت می‌دهد.
۲. دژافزار هایkeylogger خاص که تنها گزارش‌ها صفحه‌کلید برای وب‌سایت‌های معین نظیر وب‌سایت‌های بانکی را ثبت می‌کنند.
۳. دژافزار های حملات منع سرویس توزیع‌شده بر روی دستگاه‌های زامبی[۳] که فقط با توجه به دستور خاص از جانب کنترل‌کننده‌ی بات فعال می‌شوند.

ما رفتار موذیانه‌ی مخفی در این دژافزار ها را که فقط در شرایط خاصی فعال می‌شود رفتار مبتنی بر ماشه[۴] می نامیم. این رفتار مبتنی بر ماشه می‌تواند بر اساس یک ماشه[۵] از نوع زمان،رویدادهای سیستمی[۶]و یا ورودی شبکه باشد. در حال حاضر، تجزیه‌وتحلیل رفتار مبتنی بر ماشه اغلب به‌صورت دستی و با روش‌های قدیمی و مبتنی بر صرف زمان توسط تحلیلگر انسانی انجام می‌شود؛ ازاین‌رو حتی کمک اندک در راستای خودکارسازی و تسریع تحلیل این‌گونه از دژافزار ها می‌تواند به میزان قابل‌توجهی مؤثر باشد.

در این مقاله[۲۴]، نویسندگان تجزیه‌وتحلیل خودکار مبتنی بر ماشه را پیشنهاد می‌دهند. به‌طور مشخص در این پژوهش یک رویکرد خودکار برای تشخیص تحلیل رفتار مبتنی بر ماشه با استفاده از instrument کردن پویای کد باینری و اجرای پویانمادین طراحی‌شده است. رویکرد این پژوهش نشان می‌دهد که در بسیاری از موارد ما می‌توانیم:

۱. وجود رفتار مبتنی بر ماشه را تشخیص دهیم
۲. شرایط خاصی را که باعث فعال شدن چنین رفتار پنهانی می‌شود بیابیم.

---

[۱] Worm
[۲] Distributed Denial of Service(DDoS)
[۳] Zombie
[۴] Trigger-based behavior
[۵] Trigger
[۶] System events

۳. ورودی‌هایی را برای برآورده کردن این شرایط ارائه دهیم که با استفاده از آن‌ها بتوان رفتار موذیانه‌ی دژافزار را در یک محیط کنترل‌شده مشاهده کرد.

در این پژوهش ابزارMinSweeper پیاده‌سازی شده است. این ابزار در آزمایش‌ها صورت گرفته نشان داده است که می‌تواند رفتار مبتنی بر ماشه را در دژافزار های واقعی تشخیص دهد. اگرچه چالش‌های بسیاری برای تشخیص کاملاً خودکار رفتارهای مبتنی بر ماشه وجود دارد ولی وجود ابزار MineSweeper نشان می‌دهد که چنین تجزیه‌وتحلیل خودکاری امکان‌پذیر است و درآینده می‌توان آن را انتظار داشت.

۶-۱-۲-تحلیل دستی کد برای تشخیص رفتار مبتنی بر ماشه

نویسندگان مقاله ادعا می‌کنند که تا پیش از سال ۲۰۰۷ و ارائه این پژوهش که منجر به ابزارMinSweeper شد ، پژوهش شاخص دیگری باهدف تشخیص رفتار مبتنی بر ماشه صورت نگرفته است. درواقع در دنیای واقعی تحلیل دستی کد برای تشخیص رفتار مبتنی بر ماشه به‌صورت زیر بوده است:

۱. تحلیل‌گر انسانی دژافزار را در یک ماشین مجازی اجرا می‌کرد، در حقیقت ممکن بود با هیچ رفتار موذیانه‌ای مواجه نشود زیرا احتمال برآورده شدن شرایط ماشه بسیار کم بود.
۲. تحلیل‌گر انسانی سعی می‌کرد تا کد باینری را به کد اسمبلی برگرداند[۱]، تا بتواند با استفاده از کد اسمبلی یک مدل مفهومی نادقیق از اجرای برنامه داشته باشد.
۳. تحلیلگر انسانی در این مرحله سعی می‌کرد تا حدس بزند که تغییر کدام بخش ورودی و یا تنظیم پیکربندی برنامه می‌تواند منجر به فعال‌سازی اجرای ماشه گردد.

این فرآیند ادامه می‌یافت تا زمان در نظر گرفته‌شده برای تحلیل دژافزار پایان یابد و یا تحلیل‌گر انسانی ناامید شود و یا در بهترین حالت تحلیل‌گر موفق شود تا با خوش‌شانسی رفتار مبتنی بر ماشه را اجرا کند و رفتار موذیانه‌ی دژافزار را مشاهده نماید.

این پژوهش با تمرکز بر روی بات نت ها انجام‌شده است.در حقیقت سعی شده تا ابزاری برای تحلیل خودکار کد باینری بات نت ها باهدف یافتن رفتار مبتنی بر ماشه توسعه یابد.

[۱] Disassembeling

۶-۱-۳-رویکرد ابزارMinSweeper

در این زیر بخش رویکرد کلی و ساختار ابزارMinSweeper برای تحلیل خودکار کد باینری را ارائه می‌دهیم. با در نظر گرفتن یک انتزاع کلی می‌توان بیان کرد که رفتارهای مبتنی بر ماشه با استفاده از پرش‌های شرطی[۱] در کد باینری نمایان می‌شوند. این پرش‌های شرطی بر اساس ورودی‌های ماشه که از نوع زمان ، رویدادهای سیستمی، ورودی شبکه یا فشردن یک دکمه‌ی خاص صفحه‌کلید هستند اجرا می‌شوند. درواقع رفتار موذیانه زمانی که یکی از این پرش‌های شرطی برآورده شود و مسیر اجرایی برنامه به مسیر موذیانه تغییر یابد، اجرا می گردد. ما در این پژوهش شرط‌هایی را که منجر به اجرای رفتار مبتنی بر ماشه می‌شوند، شرط ماشه[۲] و مقدار ورودی لازم برای برآورده شدن شرط ماشه را ورودی ماشه[۳] می‌نامیم. ازآنجاکه رفتار مبتنی بر ماشه می‌تواند در جایگاه‌های مختلف برنامه قرار بگیرد بنابراین ما نیاز داریم تا برای تحلیل خودکار کد مسیر های متفاوتی را اجرا نماییم.

در رویکرد ابزارMinSweeper کد باینری دژ افزار و یک مجموعه از انواع ماشه‌های احتمالی به‌عنوان ورودی به ابزار داده می‌شود. در این رویکرد برای تحلیل خودکار از ترکیب اجرای نمادین و اجرای عددی استفاده می‌شود، این ترکیب درواقع بهبودیافته‌ی اجرای نمادین است که اجرای پویانمادین نام دارد. در اجرای پویا نمادین هر یک از ورودی‌های ماشه به عنوان یک نماد در نظر گرفته می‌شود و دستورالعمل های باینری[۴] که براساس این ورودی ها اجرا می گردد به‌صورت نمادین[۵] اجرا می‌شود. بنابراین با اجرای یک مسیر از برنامه یک فرمول نمادین بر روی ورودی ها به دست می آید. این فرمول نمادین حاصل از شرط هایی است که در این مسیر اجرای برنامه وجود داشته است.سپس این فرمول به حل کننده ی قید داده می‌شود تا با حل کردن آن ورودی های متناسب با مسیر های اجرایی جدیدی را ایجاد کند.

تا پیش از این پژوهش، ابزارهای بسیاری برای تحلیل خودکار کد توسعه یافته است اما تمامی این ابزار ها با این فرض بوده است که ابزار به کد منبع برنامه دستری دارد. در فرض این پژوهش ما تحلیل خودکار را باهدف تشخیص دژافزار انجام می دهیم بنابراین فرض دسترسی به کد منبع دژافزار فرضی غیرقابل قبول است.

[۱] Conditional jump
[۲] Trigger condition
[۳] Trige
[۴] Binary instructions
[۵] Symbolically

از این رو ما در ابزارMinSweeper با این فرض که تنها به کد باینری دژافزار دسترسی داریم تحلیل خودکار را براساس اجرای پویانمادین مبتنی بر باینری درنظرگرفتیم. لازم به ذکر است در مورد بسیاری از دژافزار ها علاوه بر عدم دسترسی به کد منبع، خروجی دژافزار نیز مبهم سازی[۱] یا بسته بندی[۲] شده است؛ از این رو استفاده از اجرای پویانمادین مبتنی بر باینری کاملا ضروری است.

۶-۱-۴-شرح مساله تشخیص رفتار مبتنی بر ماشه

در شکل۶ یک نمونه از کد اسمبلی و کد منبع یک دژافزار مشابه کرم واره Mydoom نمایش داده شده است.

```
4012b1: call   401810 <_GetLocalTime@4>
4012b6: add    $0xc,%esp
4012b9: cmpw   $0x9,0xffffffee(%ebp)
4012be: jne    40132d <_main+0xad>
4012c0: cmpw   $0xa,0xfffffff0(%ebp)
4012c5: jne    40132d <_main+0xad>
4012c7: cmpw   $0xb,0xffffffea(%ebp)
4012cc: jne    40132d <_main+0xad>
4012ce: cmpw   $0x6,0xfffffff2(%ebp)
4012d3: jne    40132d <_main+0xad>
4012d5: sub    $0xc,%esp
4012d8: push   $0x404000
4012dd: call   4017a0 <ddos>
4012e2: add    $0x10,%esp
4012e5: jmp    40132d <_main+0xad>
...
40132d: ret
```

```
SYSTEMTIME systime;
GetLocalTime(&systime);
site = ''www.usenix.org'';
if (9 == systime.wDay){
   if (10 == systime.wHour){
      if (11 == systime.wMonth){
         if (6 == systime.wMinute){
            ddos(site);
         }
      }
   }
}
```

شکل ۱۳- نمونه ی کد اسمبلی و کد منبع یک دژافزار مشابه واره ی Mydoom

[۱] Obfuscated
[۲] Packed

در این مثال، حمله ی ممانعت از سرویس توزیع شده تنها زمانی انجام خواهد شد که فراخوانی تابع GetLocalTime مقدار ۱۰:۰۶ ۱۱/۹ را خروجی دهد. بنابراین در این جا حمله ی ممانعت از سرویس یک رفتار مبتنی بر ماشه تلقی می‌شود که تنها در یک شرایط خاص زمانی اجرا خواهد شد.

لازم به ذکر است که ما در این مثال صرفا باهدف توضیح بیش تر و فهم راحت تر کد منبع را ارائه می دهیم، در واقع به طور معمول کد منبع دژافزار برای تحلیلگر در دسترس نیست. ضمن آنکه اغلب دژافزار ها مبهم سازی می شوند تا مانع تبدیل به کد اسمبلی برای تحلیل شده باشند.بنابراین، در یک سناریوی معمول، تحلیلگر فقط دستورالعمل های کد اسمبلی را که قابل اجرا هستند می داند.

در این مثال نوع ماشه را GetLocalTime فرض می‌کنیم. بنابراین خروجی تابع systime، همان ورودی ماشه[۱] موردنظر می باشد. در واقع رفتار مبتنی بر ماشه تحت مسیراجرایی قرار دارد که با فقط با ورودی زیر می‌توان آن را اجرا کرد:

$$\text{systime.wDay} == 9 \wedge \text{systime.wHour} == 10 \wedge \text{systime.wMonth} == 11 \wedge \text{systime.wMinute} == 6$$

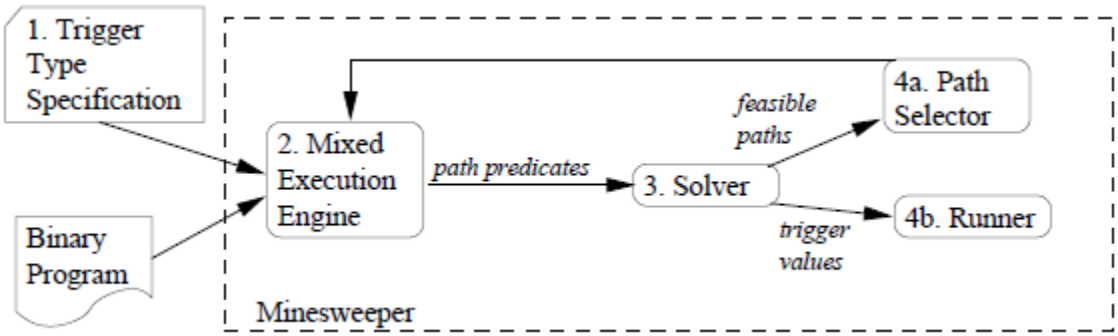
از آنجا که رفتار مبتنی بر ماشه می تواند در هر جایی از کد برنامه نهاده شده باشد، شناسایی خودکار رفتار مبتنی بر ماشه به پیمایش مسیرهای متفاوت برنامه نیاز دارد. این مسیر های مختلف، به ورودی های متفاوت برنامه بستگی دارند. یکی از راه حل های ساده برای این کار این است که به طور بی قاعده اعداد وروی را تولید کنیم و امیدوار باشیم که مسیر های مختلف برنامه پیمایش شوند. اما ازآنجاکه چنین رویکردی درعمل ناکارآمد است ، این پژوهش یک رویکرد مبتنی بر تکرار را با اجرای پویانمادین مورد نظر قرار داده است.

همانطور که در شکل۷ مشاهده می‌شود، ابزارMinSweeper چهار مرحله ی زیر را برای تشخیص خودکار دژافزار طی می کند:

- مرحله ۱: کاربر ابزارMinSweeper در ابتدا بایستی لیستی از انواع ماشه های مورد علاقه برای تحلیل دژافزار را انتخاب کند. این لیست می تواند شامل انواع مختلف ماشه ها نظیر زمان، رویدادهای سیستمی، ورودی های شبکه،هرگونه کتابخانه یا فراخوانی سیستمی باشد.

[۱] Trigger input

- مرحله ی۲: پس از مشخص شدن انواع ماشه ی موردعلاقه در مرحله ی قبل، در این مرحله اجرای پویانمادین صورت می گیرد به این ترتیب با اجرای یک مسیر از برنامه عبارات نمادین مربوط به آن مسیر و شرط ها استخراج می شوند. در حقیقت شرط های استخراج شده از یک مسیر اجرا همان دستورالعمل های پرش شرطی در آن مسیر هستند. با رسیدن به این دستورالعمل های پرش شرطی دو مسند مسیر[۱] متفاوت ایجاد می‌شود. یکی از این مسند ها، مسند مسیری است که به ازای زیرشاخه ی True دستورالعمل شرطی ایجاد می‌شود. مسند مسیر دوم نیز به ازای زیرشاخه ی False دستورالعمل شرطی ایجاد می گردد.
- مرحله ی ۳:در مرحله ی سوم دو مسند مسیر ایجاد شده در مرحله ی قبل به حل کننده ی قید داده می‌شود تا با حل کردن آن امکان‌پذیر بودن آن زیرشاخه ی مربوطه را ارزیابی کند. با امکان‌پذیر بودن هریک از زیرشاخه ها یک مسیر اجرایی جدید از برنامه کشف می‌شود.
- مرحله ی ۴-الف: در این مرحله، جز انتخاب کننده ی مسیر در ابزار یکی از مسیر ها را از مجموعه ی مسیر های امکان‌پذیر انتخاب می کند تا اجرا شود. سپس در طی اجرای این مسیر جدید فرآیند بخش دوم (اجرای پویانمادین) تکرار می گردد. بنابراین با تکرار اجرای پویانمادین، بیش تر مسیر های برنامه پیمایش خواهند شد.
- مرحله ی ۴-ب: جز اجرا کننده[۲] در این بخش از مرحله ی نهایی برنامه را به‌صورت عددی اجرا می کند. این اجرای عددی با استفاده از مقادیر عددی تولید شده در مرحله ی۳ توسط حل کننده ی قید انجام می گیرد.



شکل ۱۴- مراحل مختلف ابزارMinSweeper

[۱] برای هر مسیر اجرایی از یک برنامه مجموعه‌ای از شروط به دست می‌آید که نشان‌دهنده‌ی آن مسیر مشخص از برنامه می‌باشد، به این مجموعه از شروط مسند مسیر یاPath predicate گویند.

[۲] Runner

ابزارMinSweeper چهار جز مختلف را برای تشخیص خودکار دژافزار دارا می باشد. این چهار جز عبارت‌اند از:

۱. موتور اجرای پویانمادین[۱]
۲. حل کننده ی قید
۳. انتخاب کننده ی مسیر
۴. اجرا کننده

این ابزار زمانی که کد باینری ورودی و مجموعه ی ماشه های مورد علاقه را دریافت می کند، با استفاده از موتور اجرای پویانمادین مسیر های مختلفی را کشف می کند. مسند مسیر مربوط به هریک از این مسیر ها به حل کننده ی قید داده می‌شود تا پس از بررسی امکان‌پذیر بودن آن، به مجموعه ی مسیر های امکان‌پذیر اضافه شود و ضمنا داده های ورودی متناسب برای اجرای آن مسیر به دست آید. سپس جز انتخاب کننده ی مسیر یکی از این مسیر ها را برای اجرا برمی‌گزینند. مسیر منتخب با استفاده از ورودی های حاصل از حل کننده ی قید در یک محیط شبیه سازی و کنترل شده اجرا می گردد تا رفتار موذیانه‌ی مبتنی بر ماشه ی دژافزار آشکار گردد.

تفاوت جدی این کار با کار های قبل از آن در اجرای پویانمادین بر روی کد باینری می باشد. درواقع در تمامی کارهای پیشین که بر روی کد منبع صورت گرفته است به طور ایستا کد برنامه بازنویسی می‌شود تا بتوانند اجرای پویانمادین را اجرا کنند. اما برای آن که بتوان اجرای پویا نمادین را بر روی کد باینری برنامه اجرا کرد. لازم است تا یک سیستم شبیه ساز و instrumentation پویا را فراهم کنیم از این رو در این پژوهش نوآوری قابل توجه ای صورت گرفته است.

Exploring Multiple Execution Paths for Malware Analysis-۲-۶

عنوان پژوهش[۲۵]: جست و جوی مسیر های اجرایی مختلف برنامه برای تحلیل دژافزار

[۱] اجرای پویانمادین در این نوشتگان این مقاله با واژه‌ی Mixed execution ذکرشده است.

۶-۲-۱-مقدمه

تحلیل دژافزار فرآیندی است که در طی آن رفتار یک دژافزار مورد ارزیابی قرار می گیرد تا هدف موذیانه‌ی آن کشف شود. فرآیند تحلیل دژافزار برای قدم های بعدی مقابله با دژافزار یعنی توسعه‌ی یک سامانه‌ی تشخیص و یا ابزار مقابله کننده با دژافزار ضروری است.

روش‌های ابتدایی تحلیل دژافزار روش‌های مبتنی بر تحلیل‌گر انسانی بودند که از نظر زمان و هزینه بسیار ناکارآمد تلقی می شدند. پس از آن ابزارهای تحلیل پویا باهدف اجرای دژافزار تحت یک محیط شبیه سازی شده ی محدود معرفی شدند. این ابزارها پس از اجرای دژافزار در شبیه‌ساز لاگ مربوط به فراخوانی های سیستمی دژافزار را ذخیره می کردند که با استفاده از آن تحلیل‌گر قادر به تحلیل رفتار دژافزار بود.

مشکل عمده ی روش‌های تحلیل پویا آن بود که در هر بار اجرا تنها یک مسیر مشخص از برنامه مورد اجرا قرار می گرفت. از این گذشته بسیار محتمل بود که مسیر های خاصی که حاوی کد موذیانه بودند تنها در شرایطی مشخص و بسیار خاص اجرا شوند. به عنوان مثال تنها در صورتی که تاریخ خاصی فرا برسد و یا فایل خاصی در سیستم موجود باشد رفتار موذیانه فعال گردد. بنابراین احتمال آن که یک دژافزار بتواند رفتار موذیانه‌ی خود را از دید ابزار تحلیل پنهان نماید بسیار بالا بود.

در این پژوهش با ارائه‌ی یک راهکار جدید، مسیر های اجرایی مختلف یک دژافزار را اجرا می‌کنیم و مورد بررسی و تحلیل قرار می دهیم.

۶-۲-۲-بررسی اجمالی راهکار پیشنهادی

در راهکار پیشنهادی این پژوهش، روند تغییر یک ورودی برنامه از ابتدا تا انتها مورد بررسی قرار می گیرد. این بررسی در واقع براساس فراخوانی های سیستمی ناشی از کد باینری دژافزار می باشد. در طی این بررسی نقاط خاصی از کد که ورودی برنامه موجب تغییر مسیر اجرایی کد می گردد ثبت می شوند. در این پژوهش، به این گونه نقاط از برنامه که مسیر اجرایی متفاوت برنامه را براساس مقدار ورودی متفاوت موجب می شوند، نقاط انشعابی گفته می‌شود. به عنوان مثال وجود یا عدم وجود یک فایل می تواند دو مسیر اجرای متفاوت را موجب شود. از این رو در این مرحله، در هر نقطه ی انشعابی از اجرای برنامه یک snapshot ثبت می‌شود.

با اتمام این مسیر اجرا برنامه snapshot مربوطه را بازیابی نموده و با تغییر مقدار ورودی تاثیرگذار، مسیرمتفاوتی را برای اجرا آغاز می کند. بنابراین به جای یک مسیر واحد از برنامه، مسیر های مختلفی مورد

تحلیل قرار می‌گیرد. لازم به ذکر است که انواع ورودی های مشخصی برای ابزار به عنوان ورودی های موردتحلیل تعریف شده است و تنها اگر ورودی تاثیر گذار دژافزار یکی از این انواع از پیش تعریف شده باشد مورد تحلیل قرار خواهد گرفت.

۶-۲-۳-اجرای چندین مسیر اجرا

در بخش قبل به طور کلی شرح دادیم که در این پژوهش برای تحلیل هرچه بهتر دژافزار چگونه مسیرهای اجرای مختلف اجرا می گردند. اکنون جزییات این تغییر مسیر اجرا را در این زیربخش توضیح می دهیم.

زمانی که در حین تحلیل کد برنامه با یک تصمیم جریان کنترل مواجه می شویم، اگر ورودی به کار رفته در آن یکی از انواع ورودی های از پیش تعریف شده باشد این ورودی را برچسب گذاری می نماییم. هدف ما آن است تا پس از اتمام این اجرا مقدار این ورودی برچسب گذاری شده را به گونه ای تغییر دهیم که مسیر اجرایی جدید متفاوت از مسیر های قبلی باشد.

۶-۳-مقایسه‌ی ابزارهای اجرای پویانمادین برای تحلیل یا تشخیص دژافزار

در این بخش ابزارهای مطرح‌شده در همین فصل را با یکدیگر مقایسه می‌نماییم:

جدول ۲-مقایسه‌ی ابزارهای اجرای پویانمادین برای تشخیص یا تحلیل دژافزار

ابزار	سال ارائه	زبان	هدف	ویژگی‌ها
Temporal search [۲۷] (TimeBom detection)	**۲۰۰۶**	**Binary**	**تشخیص بمب زمانی**	یک سیستم تحلیل مبتنی بر ماشین مجازی اجرای نمادین محدود برای پیش‌بینی مسندهای زمانی جریان کنترل دژافزار را بررسی نمی‌کند. بخش‌های زیادی از کار مبتنی بر کمک تحلیل‌گر انسانی
[۲۶]Bitscope	**۲۰۰۷**	**Binary**	**تحلیل دژافزار**	یک سیستم تحلیل، پنج جز برای تحلیل دژافزار بر روی Bitscope کنترل جریان(مثلاً حلقه)-رفتار دژافزار-ورودی های برنامه-وابستگی ها اجرای برنامه با مقادیر نمادین تعریف مقدار بازگشت فراخوانی‌های سیستمی به‌صورت نمادین
Mine Sweeper (Botnet Detection)	**۲۰۰۸**	**Binary**	**تشخیص دژافزار**	ورودی فایل باینری+ مجموعه‌ی انواع ماشه‌ی دلخواه ترجمه‌ی کد اسمبلی به یک زبان میانی قراردادی (IR) استفاده از حل‌کننده‌ی قید STP

فصل هفتم:

بحث و نتیجه‌گیری

۷-۱-مقدمه

در این فصل ابتدا بر اساس چالش‌های انفجار مسیر، حل قیدها، مدل‌سازی حافظه، همروندی و چارچوب‌های کاری مختلف، کارها و مقاله‌هایی که در فصل‌های گذشته بررسی شدند، مقایسه می‌شوند و آینده بحث را مشخص می‌کنند. در قسمت بعدی مسائل باز این حوزه عنوان‌شده و درنهایت پروژه کارشناسی ارشد بیان خواهد شد.

۷-۲-مقایسه کارهای پیشین و آینده بحث

در این قسمت چالش‌هایی که در رابطه با این حوزه مطرح می‌شود را به‌طور خلاصه بیان می‌کنیم

۱-**انفجار مسیرها**: در دنیای واقعی تعداد خطوط برنامه‌ها بسیار زیاد است و تعداد مسیرهایی که در آن‌ها قابل پیمایش است، به‌صورت نمایی افزایش می‌یابد. همین موضوع باعث می‌شود تا در اجراها با کمبود حافظه مواجه شویم. یکی از راه‌حل‌هایی که تاکنون استفاده‌شده است، هیوریستیک‌های مختلف هستند که در **Error! Reference source not found.**پویانمادین بیان‌شده است.

۲-حل قیدها: یکی از نقاط چالش‌برانگیز در این حوزه حل کردن قیدهای مسیر هست. اجرای نمادین در برنامه‌های واقعی باعث می‌شود، قیدهایی تولید شوند که حل‌کننده قیدها، توانایی حل کردن آن‌ها را ندارند یا اینکه این برنامه‌ها ازنظر زمانی کارا نیستند و لازم است بهبودهایی در پیاده‌سازی آن‌ها صورت پذیرد

۳-مدل‌سازی حافظه: نحوه برخورد با حافظه و مدل کردن آن، دیگر چالش این حوزه است. به‌طور مثال یک متغیر int به شکل یک‌خانه حافظه در نظر گرفته شود یا اینکه به‌صورت ۴ خانه یک بایتی. که مورد دوم خطاهای مثل سرریزها را می‌تواند بررسی کند. یا در رابطه با اشاره‌گرها در **Error! Reference source not found.** اشاره‌گر به‌عنوان یک مقدار عددی int در نظر گرفته می‌شود. ولی در **Error! Reference source not found.** با مدل خاص خود می‌تواند برابری یا نابرابری دو اشاره‌گر را بررسی و قید مربوط به آن را حل کند.

۴-همروندی: برنامه‌های امروزی به‌صورت توزیع‌شده هستند و معمولاً کاربرهای مختلف به‌صورت همروند و چند نخی اجرا می‌شوند. نحوه آزمون این‌چنین برنامه‌ها از دیگر چالش‌های این حوزه است

۵-چارچوب‌های کاری مختلف: یکی از چالش‌های امروز در مورد آزمون برنامه‌ها، توسعه برنامه برای چارچوب‌های کاری جدید مثل چارچوب اندروید یا برنامه‌های تحت وب هستند. در هر یک از این چارچوب‌های کاری چالش‌های جدیدی وجود دارد. مثلاً در مقاله ACTEV عنوان شد که در این‌گونه برنامه‌ها علاوه بر داده‌های عادی، رخدادها هم باید به‌صورت خودکار تولید شوند تا بتوان تمام‌مسیرهای موجود در برنامه را پوشش داد.

۷-۳-مسائل باز

در این قسمت مسائل باز با توجه به پژوهش‌های پیشین عنوان می‌شوند:

۱. بهبود مسئله انفجار مسیر در اجرای پویانمادین با ارائه هیوریستیک کارا
(روش‌های جستجوی هوشمند، هرس کردن مسیر، کَش کردن پرس‌وجوهای قبلی، ترکیب تحلیل‌های ایستا و پویا، استفاده از روش‌های متن‌کاوی[۱] برای انتخاب کارای مسیرهای برنامه)
۲. بهبود حل‌کننده‌های قید برای ارتقای توان تحلیلی
(به‌عنوان‌مثال پشتیبانی کردن از محاسبات ممیز شناور)
۳. اجرای پویانمادین بروی پلتفرم اندروید
۴. اجرای پویانمادین برای تشخیص دژافزار

۷-۴-پروژه کارشناسی ارشد

در این قسمت پروژه کارشناسی ارشد و مراحل اجرای آن معرفی خواهد شد.

۷-۵-عنوان پروژه

بهبود تشخیص بمب منطقی در دژافزارهای اندروید با اجرای پویا نمادین

۷-۶-توضیح اجمالی پروژه

مقدمه: اندروید محبوب‌ترین سیستم‌عامل حال حاضر گوشی‌های هوشمند می‌باشد که هدف حملات مخرب و دژ افزارهای متنوعی قرارگرفته است. اگرچه روش‌های تحلیل ایستا و پویای موجود می‌توانند به‌طور مؤثری بسیاری از دژ افزارها و نشت اطلاعات ناخواسته را کشف کنند ولی نوع خاصی از دژ افزارها وجود دارند که

[۱] Text Mining

تشخیص آن‌ها با روش‌های کنونی دارای چالش‌های بسیاری است. از این دسته می‌توان به دژ افزارهایی اشاره کرد که صرفاً تحت برآورده شدن شرایط خاصی اعمال مخرب خود را اجرا می‌کنند به این دسته از دژ افزارها بمب منطقی گفته می‌شود. از مکانیزم بمب منطقی در بسیاری از حملات باهدف مشخص[۱]و یا تهدیدات پیشرفته مداوم[۲] استفاده می‌شود. ازآنجاکه این‌گونه دژ افزارها صرفاً تحت شرایط بسیار خاص و در زمان رسیدن به قربانی موردنظر رفتار مخرب خود را به نمایش می‌گذارند، روش‌های موجود برای تشخیص آن‌ها با چالش‌های جدی مواجه است.

مسئله موجود: در سال‌های اخیر دو رویکرد ایستا و پویا برای تشخیص دژ افزارهای اندروید موردتوجه قرارگرفته است.در این میان ابزارهایی نظیر Kirin[۱۳], DroidAPIMiner[۲۸], Flowdroid [۲۹] مطرح‌شده اند که دقت تشخیص قابل قبولی را برای دژ افزارهای سنتی[۳] اندروید ارائه می‌کنند اما این ابزارها در تشخیص بمب منطقی ناکارآمد هستند.[۳۰] در سال۲۰۱۶ میلادی ابزارTriggerScope [۳۰] به‌عنوان یک‌قدم اولیه درزمینه‌ی تشخیص بمب منطقی ارائه شد. اگرچه نوآوری و نگاه جدید در این ابزار باعث شده بود تا بسیاری از انواع بمب منطقی قابل‌تشخیص باشند اما با توجه به رویکرد ایستا به‌کاررفته در آن، این ابزار بسیاری از محدودیت‌های رویکرد ایستا را به ارث برده است. به‌عنوان‌مثال در برابر تکنیک‌های مختلف مبهم سازی[۴] نظیر بارگذاری پویای کد[۵]و کد بومی[۶] کاملاً آسیب‌پذیر است. ازاین‌رو ابزار TriggerScope در تشخیص بسیاری از انواع بمب منطقی کنونی که مبهم سازی شده و یا از تکنیک‌های ضد ایستا دیگری بهره می‌برند ناکارآمد است.از سوی دیگر در ابزار TriggerScope صرفاً به تشخیص ماشه موذیانه[۷] موجود در کد بسنده می‌شود و بررسی کد مخرب[۸] موجود پس از آزاد شدن ماشه بر عهده‌ی تحلیل‌گر انسانی قرار می‌گیرد که باعث نامناسب شدن ابزار برای تحلیل خودکار تعدادی زیادی برنامه می‌شود.

نکته‌ی حائز اهمیت دیگر آن است که در حال حاضر روش‌های پویای تشخیص دژ افزار اندروید، به استخراج الگوی رفتاری دژ افزار در حین اجرا درsandbox می‌پردازند و با استفاده از الگوی رفتاری استخراج‌شده، دژ افزار بودن یا نبودن برنامه اندروید را تشخیص می‌دهند. هنگامی‌که یک بمب منطقی در sandbox اجرا

[۱] Specific targeted attacks
[۲] Advanced Persistent Threat
[۳۳] Traditional malware
[٤] obfuscation
[٥] Dynamic code loading
[٦] Native code
[۷] Trigger
[۸] Payload

می‌گردد الزاماً باید تحت شرایط بسیار خاصی قرار گیرد تا رفتار موذیانه‌ی آن فعال گردد.به‌عنوان‌مثال ممکن است بمب منطقی بررسی کند که لزوماً روی یک پردازنده واقعی در حال اجراست ونه یک نمونه‌ساز[1]و یا برای هدف قرار دادن یک قربانی خاص با شماره سیم‌کارت معین طراحی‌شده باشد و یا بمب منطقی تنها در صورتی رفتار موذیانه خود را آشکار کند که از حضور یک برنامه‌ی کاربردی بانکی آسیب‌پذیر درروی گوشی مطمئن باشد. همان‌طور که در ابتدای این بخش بیان شد، این شرایط خاص باعث شده تا در عمل روش‌های کنونی تشخیص پویای دژ افزار اندروید از تشخیص بمب‌های منطقی بازبمانند.

در سال۲۰۱۵ میلادی ابزار Condroid اجرای پویانمادین را برای اولین بار بروی سیستم‌عامل اندروید به‌صورت یک تحلیل پویای هدفمند مطرح کرده است. در آن پژوهش از دو فاز ایستا و پویا استفاده‌شده است. فاز ایستا باهدف یافتن نقطه ورود به برنامه صورت گرفته است پس‌ازآن اجرای پویانمادین در فاز پویا انجام‌شده است. در مقاله Condroid به‌عنوان ارزیابی،تنها یک مثال ساده از بمب منطقی توسط نویسندگان مقاله مطرح‌شده است که با استفاده از اجرای پویانمادین موفق به برآورده کردن شرایط خاص بمب منطقی مذکور گردیده‌اند. ازآنجاکه این پژوهش در این زمینه یک پژوهش اولیه بوده است،دارای نقاط ضعف جدی است. نقطه‌ضعف اصلی آن ابزار محدودیت‌های مربوط به اجرای پویانمادین خالص، مانند انفجار مسیر است که در خود مقاله موردبررسی قرار نگرفته است.نقطه‌ی ضعف دیگر آن ابزار، عدم وجود راهکار در برابر تکنیک‌های مبهم سازی است. در حقیقت استفاده از یک حلقه نامتناهی در قسمتی که مهاجم می‌خواهد کد مخرب خود را پیاده‌سازی کند باعث می‌شود تا ایده‌ی اجرای پویانمادین خالص در عمل ناکام ماند.

ما در این پژوهش مسئله خود را تشخیص بمب منطقی در دژ افزارهای اندورید قراردادیم ازاین‌رو هدف‌گذاری ما استفاده از اجرای پویانمادین برای این کار بوده است. پژوهش فعلی انجام‌شده در condroid از پویانمادین خالص بهره برده است که طبیعتاً با مشکلاتی نظیر انفجار مسیر روبه‌رو است. قصد داریم تا با ارائه‌ی راهکاری مسئله انفجار مسیر در اجرای پویانمادین را برای تشخیص بمب منطقی بهبود دهیم.(ادامه در پیوست)

ایده حل: یکی از ایده‌های موجود در بهبود مسئله‌ی انفجار مسیر در اجرای پویانمادین هرس کردن درخت اجرا است. به‌عنوان‌مثال در پژوهش Driller از ترکیب فازر و اجرای پویانمادین برای بهبود انفجار مسیر استفاده‌شده است با این توضیح که آن پژوهش مربوط به پلتفرم گوشی‌های هوشمند نیست.

[1] emulator

روش ارزیابی:در این پروژه ما سعی داریم تا با استفاده اجرا پویا نمادین تشخیص بمب منطقی در دژ افزارهای اندروید را بهبود ببخشیم. ازاین‌رو پس از اتمام پژوهش به‌عنوان ارزیابی، تشخیص یک مجموعه داده از دژ افزارهای اندروید را مدنظر قرار می‌دهیم. لازم به ذکر است، اطلاعات فعلی ما نشان می‌دهد که تاکنون یک مجموعه‌ی داده‌ی مجزا از بمب‌های منطقی در اندروید وجود ندارد.

۷-۷-مراحل اجرای پروژه

مراحل اجرای پروژه در زیر آمده است:

۱. مطالعه روش اجرایی پویانمادین و ابزارهای موجود در این حوزه
۲. مطالعه ساختار و معماری اندروید، برنامه‌های اندرویدی و دژافزار های اندرویدی
۳. مطالعه و ارائه روشی برای پوشش تمام‌مسیرهای برنامه‌های اندروید
۴. توسعه‌ی یک ابزار آزمون برای برنامه‌های اندرویدی با پوشش کد مناسب
۵. مطالعه‌ی روش‌های فعلی تشخیص پویای دژافزار اندرویدی
۶. ارائه‌ی یک طرح جامع برای تشخیص دژافزار اندروید مبتنی بر اجرای پویانمادین برنامه‌های اندرویدی
۷. توسعه‌ی یک ابزار جامع برای تشخیص دژافزار اندروید مبتنی بر اجرای پویانمادین برنامه‌های اندرویدی
۸. آزمون روش و ابزار پیشنهادی با استفاده از دژافزار های اندرویدی و بمب منطقی در اندروید
۹. نگارش پایان‌نامه

فصل هشتم:

مراجع

[۱] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." *Communications of the ACM* ۵۶,۲ (۲۰۱۳): ۸۲-۹۰.

[۲] King, James C. "Symbolic execution and program testing." Communications of the ACM ۱۹,۷ (۱۹۷۶): ۳۸۵-۳۹٤.

[۳] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. ٤۰. No. ٦. ACM, ۲۰۰۵.

[۴] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)." *Security and privacy (SP), ۲۰۱۰ IEEE symposium on*. IEEE, ۲۰۱۰.

[۵] Cha, Sang Kil, et al. "Unleashing mayhem on binary code." Security and Privacy (SP), ۲۰۱۲ IEEE Symposium on. IEEE, ۲۰۱۲.

[۶] Stephens, Nick, et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." NDSS. Vol. ۱٦. ۲۰۱٦.

[۷] Choudhary, Shauvik Roy, Alessandra Gorla, and Alessandro Orso. "Automated test input generation for android: Are we there yet?(e)." *Automated Software Engineering (ASE), ۲۰۱۵ ۳۰th IEEE/ACM International Conference on*. IEEE, ۲۰۱۵.

[۸] Anand, Saswat, et al. "Automated concolic testing of smartphone apps." *Proceedings of the ACM SIGSOFT ۲۰th International Symposium on the Foundations of Software Engineering*. ACM, ۲۰۱۲.

[۹] Schütte, Julian, Rafael Fedler, and Dennis Titze. "Condroid: Targeted dynamic analysis of android applications." *Advanced Information Networking and Applications (AINA), ۲۰۱۵ IEEE ۲۹th International Conference on*. IEEE, ۲۰۱۵.

[۱۰] Mirzaei, Nariman, et al. "Sig-droid: Automated system input generation for android applications." *Software Reliability Engineering (ISSRE), ۲۰۱۵ IEEE ۲٦th International Symposium on*. IEEE, ۲۰۱۵.

[۱۱] M Yang, Zhemin, et al. "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection." *Proceedings of the ۲۰۱۳ ACM SIGSAC conference on Computer & communications security*. ACM, ۲۰۱۳.

[۱۲] Schmidt, A-D., et al. "Static analysis of executables for collaborative malware detection on android." Communications, ۲۰۰۹. ICC'۰۹. IEEE International Conference on. IEEE, ۲۰۰۹.Arp.

[۱۳] Enck, William, Machigar Ongtang, and Patrick McDaniel. "On lightweight mobile phone application certification." Proceedings of the ۱٦th ACM conference on Computer and communications security. ACM, ۲۰۰۹.

[۱۴] Bläsing, Thomas, et al. "An android application sandbox system for suspicious software detection." Malicious and unwanted software (MALWARE), ۲۰۱۰ ٥th international conference on. IEEE, ۲۰۱۰.

[۱٥] Shabtai, Asaf, and Yuval Elovici. "Applying behavioral detection on android-based devices." Mobile Wireless Middleware, Operating Systems, and Applications (۲۰۱۰): ۲۳٥-۲٤۹.

[۱۶] Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for android." *Proceedings of the ۱st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, ۲۰۱۱.

[۱۷] Zhou, Yajin, et al. "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets." *NDSS*. Vol. ۲٥. No. ٤. ۲۰۱۲.

[۱۸] Nadji, Yacin, Jonathon Giffin, and Patrick Traynor. "Automated remote repair for mobile malware." *Proceedings of the ۲۱th Annual Computer Security Applications Conference*. ACM, ۲۰۱۱.

[۱۹] Daniel, et al. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." *NDSS*. ۲۰۱٤.

[۲۰] Aswini, A. M., and P. Vinod. "Droid permission miner: Mining prominent permissions for Android malware analysis." *Applications of Digital Information and Web Technologies (ICADIWT), ۲۰۱٤ Fifth International Conference on the*. IEEE, ۲۰۱٤.

[۲۱] Wu, Dong-Jie, et al. "Droidmat: Android malware detection through manifest and api calls tracing." *Information Security (Asia JCIS), ۲۰۱۲ Seventh Asia Joint Conference on*. IEEE, ۲۰۱۲.CM, ۲۰۱٤.

[۲۲] Spreitzenbarth, Michael, et al. "Mobile-sandbox: having a deeper look into android applications." *Proceedings of the ۲۸th Annual ACM Symposium on Applied Computing*. ACM, ۲۰۱۳.

[۲۳] Roshandel, Roshanak, Payman Arabshahi, and Radha Poovendran. "LIDAR: a layered intrusion detection and remediationframework for smartphones." *Proceedings of the ٤th international ACM Sigsoft symposium on Architecting critical systems*. ACM, ۲۰۱۳.

[۲۴] Brumley, David, et al. "Automatically identifying trigger-based behavior in malware." *Botnet Detection* (۲۰۰۸): ٦٥-۸۸.

[۲۵] Moser, Andreas, Christopher Kruegel, and Engin Kirda. "Exploring multiple execution paths for malware analysis." *Security and Privacy, ۲۰۰۷. SP'۰۷. IEEE Symposium on*. IEEE, ۲۰۰۷.

[۲۶] Brumley, David, et al. *Bitscope: Automatically dissecting malicious binaries*. Technical Report CS-۰۷-۱۳۳, School of Computer Science, Carnegie Mellon University, ۲۰۰۷.

[۲۷] Crandall, Jedidiah R., et al. "Temporal search: Detecting hidden malware timebombs with virtual machines." *ACM Sigplan Notices*. Vol. ٤١. No. ١١. ACM, ۲۰۰٦.

[۲۸] Aafer, Yousra, Wenliang Du, and Heng Yin. "Droidapiminer: Mining api-level features for robust malware detection in android." *International Conference on Security and Privacy in Communication Systems*. Springer, Cham, ۲۰۱۳.

[۲۹] Arzt, Steven, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." *Acm Sigplan Notices* ٤٩,٦ (۲۰۱٤): ۲٥۹-۲٦۹.

[۳۰] Fratantonio, Yanick, et al. "Triggerscope: Towards detecting logic bombs in android applications." *Security and Privacy (SP), ۲۰۱٦ IEEE Symposium on*. IEEE, ۲۰۱٦.

[۳۱] Wong, Michelle Y., and David Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." *NDSS*. ۲۰۱٦.

[۳۲] Rasthofer, Siegfried, et al. "Making malory behave maliciously: Targeted fuzzing of android execution environments." *Proceedings of the ۳۹th International Conference on Software Engineering*. IEEE Press, ۲۰۱۷.

Amirkabir University of Technology
(Tehran Polytechnic)

Computer and Information Technology Engineering Department

Seminar Report

Title:
Malware Detection by Concolic Execution

By:
Mahmoud Aghvami panah

Supervisor:
Dr. Babak Sadeghiyan

September ۲۰۱۷