­ 



**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 1396**

**چکیده:** دژافزار کدی مخرب است که با هدف 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 سیستم های رایانه ای و گوشی های هوشمند تولید و پیچیدگی دژافزار ها نیز افزایش چشم گیری داشته است. برای تشخیص دژافزار ها و تحلیل آن ها، 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 محدودیت های ذاتی که دراثر اجرا نکردن برنامه دارد، دارای مثبت کاذب بالایی است. در تحلیل پویا، برنامه‌های کاربردی در یک محیط کنترل شده اجرا شده و رفتارشان مورد ارزیابی قرار می گیرد. 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 می شود تا مسیر های اجرایی مختلف برنامه مورد تحلیل قرارنگیرند و یک کاربر بدخواه بتواند دژافزار خود را به عنوان یک برنامه ی کاربردی معرفی کند.

از سویی دیگر در سال های اخیر،روش های جدید آزمون نرم افزار برای تحلیل برنامه ها مورد توجه قرار گرفته است. به عنوان نمونه اجرای پویانمادین که در سال 2005 برای اولین بار در ابزار آزمون DART معرفی شد پس از آن 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 نرم افزار، کشف آسیب پذیری، تشخیص نقض حریم خصوصی، تشخیص و تحلیل دژافزار می شد. چالش علمی اجرای مسیر های محدود برنامه در تحلیل پویا با استفاده از اجرای پویانمادین قابل حل است اگرچه استفاده از اجرای پویانمادین خود چالش های علمی جدیدی را موجب می شود.

ما در این گزارش در ابتدا به بررسی اجرای نمادین و روش های اجرای نمادین سنتی و نوین پرداخته ایم.  
در بررسی اجرای نمادین نوین، روش­های پویانمادین و EGT را شرح داده و چالش های پیش رو برای آن ها را بررسی کرده ایم. پس از آن با بررسی ابزارهای پویانمادین نظیر DART, Mayhem و Driller به مقایسه این ابزارها و نوآوری های علمی هریک پرداخته ایم. در ادامه با اشاره به چالش های علمی موجود برای تحلیل برنامه های کاربردی اندروید در فصلی جداگانه به بررسی و مقایسه ابزارهای ACTEV، Condroid، SIG-droidو AppIntent که اجرای پویانمادین را برای پلتفرم اندروید پیاده سازی کرده اند پرداخته ایم. در فصل پنجم به بررسی روش های تشخیص دژافزار با تمرکز بر دژافزار های اندرویدی و در فصل ششم به بررسی ابزار های پیشینی که از اجرای پویانمادین برای تشخیص یا تحلیل دژافزار بهره برده اندپرداخته ایم.

در فصل هفتم مباحث مطرح شده را جمع بندی نموده و با بیان پروژه­ی کارشناسی ارشد به راهکارهای پیش رو برای حل مساله­ی باز مطرح شده پرداخته ایم.

واژه های کلیدی: اجرای نمادین، اجرای پویانمادین، تشخیص دزافزار، دژافزار های اندرویدی

فهرست مطالب

[فصل اول:مقدمه 10](#_Toc495715006)

[1-1-پیشینه موضوع 11](#_Toc495715007)

[1-2-ساختار گزارش 12](#_Toc495715008)

[فصل دوم: اجرای پویانمادین 15](#_Toc495715009)

[2-1- اجرای نمادین سنتی 15](#_Toc495715010)

[2-2- اجرای نمادین نوین 19](#_Toc495715011)

[2-2-1-روش پویانمادین 19](#_Toc495715012)

[2-2-2-روش EGT 20](#_Toc495715013)

[2-3-چالش‌ها 21](#_Toc495715014)

[2-3-2-حل قیدها 21](#_Toc495715015)

[2-3-3-مدل‌سازی حافظه 21](#_Toc495715016)

[2-3-4-همروندی 21](#_Toc495715017)

[فصل سوم: ابزار های اجرای پویانمادین برای تحلیل برنامه 23](#_Toc495715018)

[3-1-ابزارDART 23](#_Toc495715019)

[3-1-1-مقدمه 23](#_Toc495715020)

[3-1-3-طرح کلی DART با استفاده از دو مثال 24](#_Toc495715021)

[3-2-ابزار Mayhem 27](#_Toc495715022)

[3-2-1- مقدمه 27](#_Toc495715023)

[3-2-2-دستاوردهای جدید MAYHEM: 29](#_Toc495715024)

[3-2-3-ساختار کلی ابزار MAYHEM 29](#_Toc495715025)

[3-2-4-اجرای هیبرید پویانمادین در ابزار MAYHEM 33](#_Toc495715026)

[3-2-5-معماری و پیاده‌سازی CEC 34](#_Toc495715027)

[3-2-6-معماری و پیاده‌سازی SES 34](#_Toc495715028)

[3-2-7-مدل سازی حافظه در MAYHEM 35](#_Toc495715029)

[3-3-ابزار Drilller 37](#_Toc495715030)

[3-3-1-مقدمه 37](#_Toc495715031)

[3-4-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل برنامه ها 37](#_Toc495715032)

[فصل چهارم: ابزار های اجرای پویانمادین برای تحلیل برنامه های اندرویدی 40](#_Toc495715033)

[4-1-چالش های علمی برای تحلیل برنامه در اندروید 40](#_Toc495715034)

[4-2-ابزار ACTEV 41](#_Toc495715035)

[4-2-1-مقدمه 41](#_Toc495715036)

[4-2-2-کلیات طرح 42](#_Toc495715037)

[4-2-3-تولید یک رخداد 44](#_Toc495715038)

[4-2-4-تولید ترتیبی از رخدادها 45](#_Toc495715039)

[4-3-ابزار Condroid 46](#_Toc495715040)

[4-3-1-مقدمه 46](#_Toc495715041)

[4-3-2-بررسی اجمالی یک مثال 47](#_Toc495715042)

[4-3-3-چهارچوب ابزارCondroid 48](#_Toc495715043)

[4-4-ابزار SIG-Droid 49](#_Toc495715044)

[4-4-1-مقدمه 49](#_Toc495715045)

[4-5-ابزار APPINTENT 51](#_Toc495715046)

[4-5-1-مقدمه 51](#_Toc495715047)

[4-5-2-معماری کلی ابزار APPINTENT 52](#_Toc495715048)

[4-6-مقایسه ابزارهای اجرای پویانمادین برای تحلیل برنامه های اندرویدی 55](#_Toc495715049)

[فصل پنجم: روش‌های تشخیص دژافزار 57](#_Toc495715050)

[5-1- تحلیل ایستا 57](#_Toc495715051)

[5-2-تحلیل پویا 57](#_Toc495715052)

[5-3-روش های تشخیص دژافزار های اندرویدی 58](#_Toc495715053)

[فصل ششم:کارهای پیشین اجرای پویانمادین با هدف تشخیص یا تحلیل دژافزار 62](#_Toc495715054)

[6-1- Automatically Identifying Trigger-based Behavior in Malware 62](#_Toc495715055)

[6-1-1-مقدمه 62](#_Toc495715056)

[6-1-2-تحلیل دستی کد برای تشخیص رفتار مبتنی بر ماشه 63](#_Toc495715057)

[6-1-3-رویکرد ابزارMinSweeper 64](#_Toc495715058)

[6-1-4-شرح مساله تشخیص رفتار مبتنی بر ماشه 65](#_Toc495715059)

[6-2- Exploring Multiple Execution Paths for Malware Analysis 69](#_Toc495715060)

[6-2-1-مقدمه 69](#_Toc495715061)

[6-2-2-بررسی اجمالی راهکار پیشنهادی 69](#_Toc495715062)

[6-2-3-اجرای چندین مسیر اجرا 70](#_Toc495715063)

[6-3-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل یا تشخیص دژافزار 71](#_Toc495715064)

[فصل هفتم: بحث و نتیجه گیری 73](#_Toc495715065)

[7-1-مقدمه 73](#_Toc495715066)

[7-2-مقایسه کارهای پیشین و آینده بحث 73](#_Toc495715067)

[7-3-مسائل باز 74](#_Toc495715068)

[7-4-پروژه کارشناسی ارشد 74](#_Toc495715069)

[7-5-عنوان پروژه 74](#_Toc495715070)

[7-6-توضیح اجمالی پروژه 74](#_Toc495715071)

[7-7-مراحل اجرای پروژه 77](#_Toc495715072)

[فصل هشتم: مراجع 79](#_Toc495715073)

فهرست شکل ها

[شکل 1-درخت اجرای مثال شکل 1 17](#_Toc495714985)

[شکل 2- تابع ساده با نامتناهی مسیر اجرا 18](#_Toc495714986)

[شکل 3- یک تکه کد ساده برای آزمون با ابزار DART-مثال اول 24](#_Toc495714987)

[شکل 4- تکه کد ساده برای آزمون با ابزار DART-مثال دوم 26](#_Toc495714988)

[شکل 5- تکه کد مربوط به برنامه ی آسیب پذیر orzHttpd 30](#_Toc495714989)

[شکل 6- وضعیت پشته برای برنامه ی آسیب پذیر orzHttpd 30](#_Toc495714990)

[شکل 7- انواع اجراهای پویانمادین 33](#_Toc495714991)

[شکل 8-درخت دودویی IST و خطی سازی آن](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714992) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714992)**

[شکل 9- کد برنامه ی اندرویدی برای تست با ابزار ACTEV](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714993) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714993)**

[شکل 10- نمایی از برنامه ی اندرویدی برای تست با ابزار ACTEV](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714995) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714995)**

[شکل 11- چرخه ی حیات یک Activity در اندروید](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714996) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714996)**

[شکل 12-ساختار کلی ابزار ACTEV](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714998) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714998)**

[شکل 13- سلسله مراتب ویجت ها در صفحه ی اصلی برنامه](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714999) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495714999)**

[شکل 14-ساختار کلی ابزار ACTEV](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495715002) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495715002)**

[شکل 15- معماری کلی ابزار Sig-Droid](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495715003) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495715003)**

[شکل 16- گراف حاصل از دو زیرفراخوانی یک نرم افزار اندرویدی بانکی](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495715004) **[Error! Bookmark not defined.](C:\\Users\\mahmoud\\Desktop\\Seminardoc0.9.docx" \l "_Toc495715004)**

[شکل 17- نمونه ی کد اسمبلی و کد منبع یک دژافزار مشابه واره ی Mydoom 65](#_Toc495715005)

فهرست جداول

[جدول 1- مقایسه ی ابزار های اجرای پویانمادین برای تحلیل برنامه ها 38](#_Toc495234463)

[جدول 2-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل برنامه های اندرویدی 55](#_Toc495234464)

**No table of figures entries found.**

**فصل اول :**

**مقدمه**

# فصل اول:مقدمه

## 1-1-پیشینه موضوع

در سال های اخیر گسترش استفاده از سیتم های رایانه ای و گوشی های 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 می کرد تا رفتار های مخرب یک دژافزار را از یک برنامه ی عادی متمایز نماید.

در ابتدا و زمانی که دژافزار های رایانه ای بسیار ساده بودند، استفاده از تحلیل ایستا برای تشخیص آن ها بسیار موثر بود.اما محدودیت های ذاتی موجود در تحلیل ایستا موجب می­شد تا قادر به شناسایی دژافزار های پیچیده نباشد. به عنوان مثال اگر دژافزار از تکنیک مبهم سازی استفاده می­کرد و یا کد مخرب خود را در زمان اجرا و به صورت پویا بارگذاری می­نمود، تحلیل ایستا قادر به تشخیص دژافزار نبود.

تحلیل پویا در راستای برطرف کردن نقاط ضعف تحلیل ایستا ارائه شد و در سال های بعد از ظهور تحلیل ایستا به عنوان یک نوآوری علمی مورد توجه قرار گرفت. با توجه به آن که در تحلیل ایستا دژافزار اجرا می­شد، مثبت کاذب ناشی از روش های مبهم سازی و یا بارگذاری پویای کد به طور کلی از بین می رفت. اما نقطه ضعف اصلی ابزار های تحلیل پویا در این بود که در هربار اجرا تنها یک مسیر اجرایی را می آزمودند. این مسیر اجرایی در ساده ترین حالت براساس ورودی های دلخواه انتخاب و اجرا می شد از این رو احتمال انتخاب مسیر های اجرایی متفاوت بسیار کم بود. در واقع دژافزار های بسیاری وجود داشت که اگرچه مسیر های اجرایی دارای کد مخرب را شامل می شدند اما در تحلیل پویا به عنوان برنامه ی کاربردی شناخته می شدند زیرا مسیر های اجرایی محدودی از آن ها اجرا می­شد و باقی مسیر های کد مورد ارزیابی قرار نمی گرفت.

از سویی دیگر مهندسین آزمون نرم افزار چهار سطح مختلف آزمون را برای آزمون یک برنامه درنظر می­گیرند. این چهار سطح عبارت اند از:

1. آزمون واحد[[1]](#footnote-1)
2. آزمون تجمیع[[2]](#footnote-2)
3. آزمون اطمینان از کیفیت[[3]](#footnote-3)
4. آزمون تایید کاربر[[4]](#footnote-4)

فارغ از سطح آزمون برای یک نرم افزار، ما نیازمند آن هستیم که داده های تولید شده برای آزمون به گونه ای هوشمندانه تولید شوند که پوشش کد مناسبی را تامین نمایند. برای این کار روش غیرهوشمندانه، تولید ورودی دلخواه خواهد بود که لزوما کارا نیست. از این رو در دهه ی90 میلادی روش آزمون نمادین مطرح شد. در این روش ورودی های برنامه به صورت نمادین در نظر گرفته می شد و شروط موجود در یک مسیر اجرا استخراج می گشت. با ارائه ی این شروط به حل کننده ی قید، مسیر اجرایی جدیدی تولید می شد که از امکان پذیر بودن آن مطمئن بودیم. در فصل دوم، برای اجرای نمادین[[5]](#footnote-5) به طور دقیق، مورد بررسی قرار خواهد گرفت.

مشکل عمده ای که درمورد اجرای نمادین وجود داشت ناتوانی حل کننده های قید برای حل قیود پیچیده و عدم قابلیت گسترش پذیری بود. پس از آن در سال 2005 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 هرجا که حل کننده ی قید قادر به حل قید مسیر نبود، مسیر به طور عددی اجرا   
می­شد. به جز آزمون نرم افزار که هدف اصلی اجرای پویانمادین بود، کاربرد های دیگری نیز مورد بررسی قرار گرفت. تشخیص آسیب پذیری های نرم افزاری، تشخیص نقض حریم خصوصی، تشخیص و تحلیل دژافزار از این موارد بودند.

ما در این گزارش ، اجرای پویانمادین برای تشخیص دژافزار را بررسی کرده ایم و سیر تاریخی و علمی لازم برای بیان این مطلب را مورد بحث قرار داده ایم.

## 1-2-ساختار گزارش

این پژوهش در هفت فصل تدوین شده است. در فصل اول مقدمات موضوع مورد بحث قرار گرفته است.

در فصل دوم اجرای نمادین شرح داده شده است.هم چنین دو روش نوین در اجرای نمادین با نام های روش EGT و روش اجرای پویانمادین مطرح می­شوند.

در فصل سوم، در اجرای پویانمادین برای تحلیل برنامه ها بررسی می‌شوند. ابزارهای بیان شده عبارتند از: DART، Mayhem­­ و Driller. این ابزار در زیربخش آخر این فصل بایکدیگر مقایسه شده و مزایا و معایب و نوآوری های علمی هریک بیان می شود.

در فصل چهارم، در اجرای پویانمادین برای تحلیل برنامه های اندروید بررسی می‌شوند.علت اینکه این ابزار ها یک فصل جداگانه را به خود اختصاص داده اند این است که تحلیل برنامه های اندروید و اجرای پویانمادین برای پلتفرم اندروید با چالش های علمی جدیدی مواجه است که در ابتدای همین فصل شرح داده شده است. ابزارهای بیان شده در این فصل عبارتند از: ACTEV، Condroid­­ ، Sig-Droidو APPIntent. این ابزارها در زیربخش آخر این فصل بایکدیگر مقایسه شده و مزایا و معایب و نوآوری های علمی هریک بیان می شود.

در فصل پنجم، بیان می‌شوند. روش‌های تحلیل ایستا و پویا، شرح داده شده و در پایان فصل نیز به طور خاص روش های تشخیص دژافزار های اندرویدی مورد مطالعه قرار می گیرد.

در فصل ششم، در اجرای پویانمادین برای تحلیل و تشخیص دژافزار بررسی می‌شوند. در این فصل سه کار پژوهشی مورد ارزیابی قرارگرفته اند. در کار اول بروملی و همکاران[22] به ارائه ی یک روش برای تشخیص خودکار بات نت ها با استفاده از اجرای پویانمادین پرداخته اند. در کار دوم کروگل و همکاران[23] با ارائه یک راهکار مبتنی بر اجرای نمادین بهبود یافته و استفاده از شبیه ساز به ارائه ی بک روش نو برای تحلیل دژافزار پرداخته اند. در کار سوم که Bitscope نام دارد با استفاده از اجرای نمادین چارچوبی برای تحلیل دژافزار ارائه شده که می تواند مسیرهای متفاوت یک دژافزار را پیمایش کند.

در پایان در فصل هفتم، به 7-1- می‌پردازیم. در مورد آینده بحث، مسائل باز موجود در این حوزه و همچنین پروژه کارشناسی ارشد و مراحل پژوهش آن مطالبی عنوان خواهد شد. در نهایت استفاده شده در فصل هشتم معرفی می‌شوند.

**فصل دوم:**

**اجرای پویا نمادین**

# فصل دوم: اجرای پویانمادین

## 2-1- اجرای نمادین[[6]](#footnote-6) سنتی

اجرای نمادین در سال های اخیر توجه زیادی را به خود جلب کرده است. این روش به عنوان یک روش موثر برای تولید مورد آزمون[[7]](#footnote-7) و پوشش کد[[8]](#footnote-8) برای پیداکردن خطاهای عمیق در نرم افزار های پیچیده مورد استفاده قرار می گیرد. ایده کلیدی اجرای نمادین بیش از سه دهه قبل معرفی شده است[2] اما در سال های اخیر با توجه به پیشرفت قابل توجه در ارضای قیود[[9]](#footnote-9) و مقیاس پذیری[[10]](#footnote-10) رویکرد های پویا این روش عملی شده و مورد توجه قرار گرفته است.

در این فصل، ما براساس پژوهش[1] یک مرور کلی برروی اجراهای نمادین ارائه می دهیم. سپس تکنیک های اجرای نمادین سنتی و مدرن را مورد بررسی قرار می دهیم و چالش های آن ها در مورد اکتشاف مسیر،حل مسئله قیود و مدل سازی حافظه را مورد بررسی قرار می دهیم.

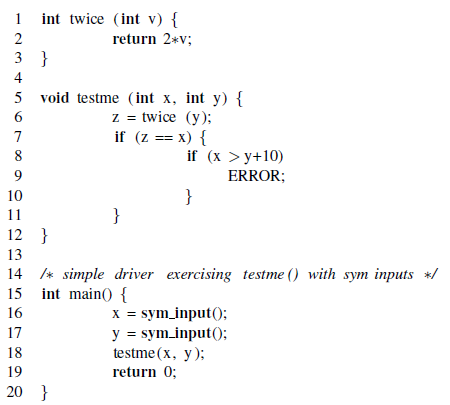
هدف کلیدی اجرای نمادین در زمینه تست نرم افزار این است که در یک مدت زمان قابل قبول مسیرهای مختلف برنامه را کشف و اجرا نماید. اجرای نمادین باید به ازای هر مسیر اجرایی دومورد زیر را بررسی نماید.

1. مجموعه ای از مقادیر ورودی را به گونه ای تولید کند تا آن مسیر خاص اجرا گردد.
2. وجود انواع خطاها را در آن مسیر بررسی نماید.خطاهایی نظیر استثنائات[[11]](#footnote-11) غیرقانونی، وجود آسیب پذیری ها، و خرابی حافظه[[12]](#footnote-12).

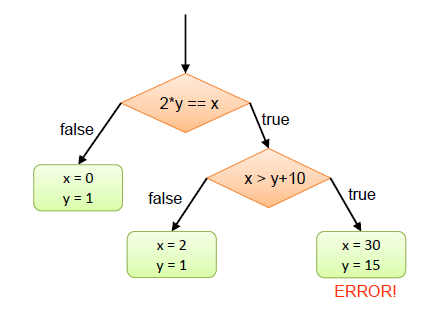
ایده کلیدی در اجرای نمادین، استفاده از مقادیر نمادین به جای مقادیر داده های ورودی و نشان دادن مقادیر متغیرهای برنامه با استفاده از عبارات نمادین است. در نتیجه مقدار خروجی محاسبه شده توسط یک برنامه به صورت یک تابع از مقادیر نمادین ورودی بیان می شود. در واقع در تست نرم افزار از اجرای نمادین برای تولید داده ی آزمون ورودی جهت اجرای یک مسیر مشخص از برنامه استفاده می شود.

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 نشان دهنده ی اجرا شدن بخش then در iامین شرط برنامه است. همه مسیرهای اجرایی یک برنامه را می توان با استفاده ازیک درخت، به نام درخت اجرا نشان داد.

به عنوان مثال در شکل 1 تابع Testme() سه مسیر اجرایی مختلف دارد. این سه مسیر را می توان در درخت اجرای شکل2 مشاهده کرد. مجموعه ی ورودی های {x = 0, y = 1}, {x = 2, y = 1}   
 {x = 30, y = 15} برای پیمودن تمام مسیر های اجرای مختلف شکل1 کافی است. هدف نهایی اجرای نمادین آن است که این مجموعه داده های ورودی را در زمان قابل قبول تولید کند و با اجرای برنامه در همه ی مسیر های اجرا از عملکرد صحت آن مطمئن شود.



شکل 1- مثالی از یک تابع ساده برای نمایش اجرای نمادین



شکل 2-درخت اجرای مثال شکل 1

اجرای نمادین، یک وضعیت نمادین[[13]](#footnote-13) از سیستم را نگه می دارد که با متغیر نمایش داده می شود. این وضعیت نشان می دهد که چگونه متغیر ها به مقادیر نمادین نگاشت شده اند. به علاوه اجرای نمادین،   
یک قید مسیر نمادین[[14]](#footnote-14) دارد که با استفاده از PC نمایش داده می شود. PC یک فرمول مرتبه اول برپایه ی عبارات نمادین است. در ابتدای اجرای نمادین، با یک جدول خالی از نگاشت ها مقداردهی اولیه می شود.PC نیز در ابتدا با مقدار True مقداردهی اولیه می شود. و PC در طول اجرای نمادین به روزرسانی می شوند.  
در پایان یک اجرای نمادین از یک مسیر اجرا، PC با استفاده از یک حل­کننده­ی قید[[15]](#footnote-15) حل می شود تا بتوان با استفاده از آن مجموعه ی داده های ورودی را تولید کرد.

به عنوان مثال اجرای نمادین برای تابع ساده ی شکل1 با یک خالی و PC حاوی مقدار True آغاز می شود.  
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 اضافه می کند. به عنوان مثال اجرای نمادین برای دو خط ابتدایی تابع main() از کد شکل 1 منجر به اضافه شدن دو نگاشت   
 به مقادیر نمادین در می شود.



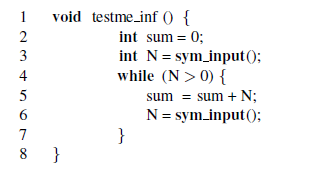
پس از آن در هر خط از کد که متغیر x مورد استفاده قرار گیرد، مقدار نمادینx0 به جای آن نگاشت می شود. مثلا پس از اجرای خط 6 از تابع مذکور، به صورت زیر خواهد بود.



*پس از اجرای خط 7، دو نمونه*[[16]](#footnote-16) *از اجرای نمادین ایجاد می گردد. یکی از آن ها با شرط x0=2y0 و دیگری با شرط x02y0 می باشد. به طور مشابه پس از خط 8 برنامه نیز دو نمونه از اجرای نمادین با شرط  
(x0 = 2y0) ^ (x0 > y0 + 10) و (x0 = 2y0) ^ (x0 <= y0 + 10) ایجاد می شود.*

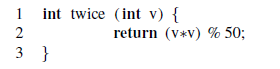
*اگر نمونه ی اجرای نمادین با دستور Terminate و یا error مواجه شود، این نمونه متوقف شده و با استفاده از یک حل کننده ی قید، یک مجموعه داده ی ورودی برای رسیدن به این مسیر اجرا تولید می گردد.*

*اجرای برنامه هایی که دارای حلقه یا توابع بازگشتی باشند ممکن است منجر به تولید نامتناهی مسیر اجرا شود.به عنوان مثال اگر شرط پایان حلقه به صورت نمادین در خود حلقه مقدار دهی گردد نامتناهی مسیر اجرا تولید می شود. شکل3 نشان دهنده ی یک مثال با نامتناهی مسیر اجرا است. هر یک از این مسیر های اجرا شامل تعداد دلخواهی مقدار True و یک مقدار پایانی False می باشد.*



شکل 3- تابع ساده با نامتناهی مسیر اجرا

یک نقطه ضعف اساسی در اجرای نمادین این است که اگر فرمول نمادین به دست آمده از یک مسیر اجرا به درستی توسط حل کننده ی قید قابل­حل نباشد، نمی توان ورودی های مناسبی برای اجرای آن مسیر تولید کرد. به عنوان مثال اگر تابع twice() به جای تابع شکل1، تابع موجود در شکل 4 باشد در آن صورت حل کننده ی قید نمی تواند عبارات غیرخطی را حل کند بنابراین اجرای نمادین نمی تواند مقادیر داده ی ورودی را برای مسیر اجرای twice() را تولید کند.



## 2-2- اجرای نمادین نوین

در این بخش دوروش از اجرای نمادین نوین را بررسی می کنیم. مزیت این روش ها نسبت به روش های سنتی ترکیب اجرای نمادین با تکنیک های عددی[[17]](#footnote-17)می باشد.

### 2-2-1-روش پویانمادین[[18]](#footnote-18)

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 هم مقدار نمادین هر متغیر یا عبارت نگهداری می‌شود.

در اجرا پویانمادین به صورت همزمان برنامه به صورت عددی و نمادین اجرا می‌گردد. برای اجرای عددی لازم است تا ورودی‌های برنامه مقداردهی اولیه شوند این مقادیر می‌تواند به صورت دلخواه انتخاب شوند.   
درطی اجرای اولیه، بیان‌های شرطی قیدهای نمادین نگهداری می‌شوند. در نهایت این قیدها به حل کننده ی 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 **Error! Reference source not found.** مقدار دلخواه {x=22, y=7} انتخاب می‌شود و برنامه هم برای اولین بار به صورت نمادین و هم عددی اجرا می‌ گردد. این اجرا در خط 7 برنامه شاخه else را انتخاب می‌کند، اجرای نمادین نیز قید را ایجاد می‌کند. در حل کننده ی قید عبارت نهایی مکمل می‌شود و شرط جدید ایجاد می‌گردد. سپس حل کننده ی قید آن را حل می‌کند و {x=2, y=1} به عنوان ورودی جدید برای اجرای بعدی تولید می‌ گردد. با این ورودی جدید در خط 7، شاخه then اجرا و در خط 8 شاخه else انتخاب می‌شود. به این ترتیب مسیر جدیدی از برنامه ایجاد و اجرا می‌گردد. در این حالت به عنوان PC جدید به حل کننده قید داده می‌شود و در نهایت ورودی {x=30, y=15} بدست می‌آید که به وسیله آن مسیر منتهی به ERROR اجرا می‌گردد. ابزارهای DART، CUTE و EXE از اجرای پویانمادین بهره می برند که در ادامه بررسی خواهند شد.

### 2-2-2-روش EGT

در این روش به صورت پویا در هر عبارت این بررسی صورت می‌گیرد که آیا تمام متغیرهای موجود در برنامه عددی هستند یا خیر. اگر همه ی متغیر های موجود عددی باشند، عبارت به صورت عادی و به صورت عددی اجرا می گردد. ولی اگر حتی یکی از متغیرها نمادین باشند عبارت به صورت نمادین اجرا و قید مربوط به PC اضافه می‌شود. اگر در خط 17 عبارت به شکل y=10 تغیر کند، خط 6 به صورت عددی اجرا می‌گردد چون ورودی تابع twice مقدار عددی 10 خواهد بود. در خط 7 عبارت شرطی به شکل if(x==20) در می‌آید. در این خط عبارت به صورت نمادین اجرا می‌شود. با این حالت هر دو مسیر then و else بررسی می‌شوند. در شاخه then در خط 8 عبارت به شکل if(x>20) در می‌آید. با توجه به اینکه x برابر مقدار 20 است بنابراین   
شاخه then قابل اجرا نخواهد بود و اجرا بدون اینکه به شاخه ERROR برسد پایان می‌پذیرد.

با استفاده از روش‌های نوین می‌توان محدودیت‌های مربوط به حل کننده ی 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 در روش EGT قید موجود در آن عبارت توسط حل کننده ی قید حل می‌شود یا در روش اجرای پویانمادین از حالت عددی موجود مقدار عددی آن ورودی محاسبه می گردد. به این ترتیب اجرا می‌تواند ادامه پیدا کند.

مثلا در همان کد نمونه شکل 4 اگر تابع twice از یک کتابخانه خارجی دریافت شود و یا این تابع به صورت غیرخطی (v\*v)%50 که در شکل 4 دیدیم در آید ، هنگامی که اجرا به خط 6 از برنامه می‌رسد از آن جا که حل کننده ی قید نمی تواند قید مربوطه را حل کند اجرا متوقف می­گردد. در این حالت در اجرای پویانمادین مقدار نمادین برای متغیر y با مقدار عددی آن مثلا 7 جایگزین می‌شود و اجرا ادامه می یابد. در این حالت قید مربوط به خط 7 به شکل if(x==49) در می‌آید و اجرا ادامه پیدا می‌کند. در اجرای نمادین سنتی چون حل کننده ی قید قادر به حل این گونه از قیدها نبود، اجرا متوقف می‌شد.

## 2-3-چالش‌ها

در این قسمت چالش‌هایی که در رابطه با اجرای پویانمادین مطرح می‌شود را به طور خلاصه بیان می‌کنیم. 2-3-1-انفجار مسیرها

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

### 2-3-2-حل قیدها

یکی از نقاط چالش برانگیز 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 آن ها در زمان کوتاه قابل انجام نیست. برای رفع این منظور لازم است تا بهبودهایی در پیاده‌سازی این حل کننده ی قیود صورت پذیرد.

### 2-3-3-مدل‌سازی حافظه

نحوه­ی تبدیل عبارات برنامه به عبارات نمادین تاثیر مستقیمی بر میزان پوشش کد در زمان اجرای پویانمادین دارد. درواقع نحوه ی برخورد با حافظه و مدل کردن آن یکی از چالش های مطرح این حوزه است. به طور مثال یک متغیر int می تواند به شکل یک خانه واحد حافظه یا به شکل 4 خانه یک بایتی در حافظه در نظر گرفته شود. در حالت دوم خطاهایی مانند سرریز بافر را می توان بررسی نمود.

به طریق مشابه اشاره‌گرها در DART به عنوان یک مقدار عددی int در نظر گرفته می‌شوند ولی در CUTE با مدل خاص پیاده سازی 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 حل کننده های قیدی مانند Z3 و STP می‌توانند قیدهای مربوط به آنها را حل کنند.

### 2-3-4-همروندی

برنامه‌های امروزی به صورت توزیع‌شده توسعه می یابند. معمولا کاربرهای مختلف به صورت همروند و چندنخی می توانند این برنامه ها را اجرا کنند. نحوه آزمون برنامه‌های همروند از دیگر چالش‌های این حوزه است.

**فصل سوم:**

**ابزار های اجرای پویانمادین برای تحلیل برنامه**

# فصل سوم: ابزار های اجرای پویانمادین برای تحلیل برنامه

## 3-1-ابزارDART

### 3-1-1-مقدمه

ابزار DART [3]که سرنام : Directed Automated Random Testing می باشد، درسال 2005 میلادی با هدف آزمون خودکار نرم افزار ارائه شده است. این ابزار از سه تکنیک مختلف بهره می برد که در ذیل مشاهده می شود:

1. استخراج خودکار رابط[[19]](#footnote-19)یک برنامه با محیط خارج از آن با استفاده از تحلیل ایستا
2. تولید خودکار داده ی آزمون برای رابط برنامه با هدف پوشش هرچه بیش تر متن برنامه
3. تحلیل پویا و بررسی چگونگی رفتار یک برنامه به ازای داده ی ورودی خودکار آزمون

نقطه ی قوت ابزار DART نسبت به ابزار های دیگر آزمون نرم افزار این است که بدون نیاز به نوشتن کد   
راه انداز [[20]](#footnote-20)یا افزودن کدی دیگر، به طور کاملا خودکار نرم افزار را مورد ارزیابی قرار می دهد.

درمیان روش های مختلف آزمون نرم افزار یکی از روش های رایج، آزمون واحد[[21]](#footnote-21) می باشد. در آزمون واحد هر یک از اجزای نرم افزار به طور مستقل مورد ارزیابی قرار می گیرد از این رو پس از انجام این آزمون می توان با قطعیت در مورد صحت منطق هر یک از اجزای نرم افزار و پوشش 100 درصدی کد برنامه اظهار نظر کرد. اما به علت پرهزینه بودن و سختی این روش، آزمون واحد در عمل به ندرت مورد استفاده قرار می گیرد. واضح است که در آزمون واحد برای ارزیابی مستقل یک جز از برنامه نیاز داریم تا قبل و بعد از آن خطوط دیگری را با عنوان کد آزمون به برنامه اضافه کنیم. به عنوان مثال لزوما بایستی قبل از جز مورد آزمون، کد راه انداز و پس از آن شرط هایی برای چک کردن خروجی مورد انتظار نوشته شود. نوشتن کد اضافی برای آزمون یک برنامه هزینه بر است. در عین حال آزمون واحد جایگزین آزمون کل برنامه با راهبرد جعبه ی سیاه[[22]](#footnote-22) نخواهد بود. زیرا ممکن است کد و منطق تمامی اجزای برنامه به صورت جدا از هم صحیح باشد ولی در ترکیب این اجزا خطا[[23]](#footnote-23) داشته باشیم.

ابزار DART ، نقاط ضعف آزمون واحد نرم افزار را ندارد زیرا نیاز به نوشتن کد اضافه برای آزمون نیست و تمامی مراحل به طور خودکار انجام می گیرد. این ابزار برای آزمون برنامه های نوشته شده به زبان C توسعه یافته است و برای این کار به کد منبع برنامه نیاز دارد. در پایان ابزار می تواند خطا های استاندارد نظیر خرابی برنامه[[24]](#footnote-24)، نقض بیانیه[[25]](#footnote-25) ها و پایان نیافتن برنامه را تشخیص دهد.

یکی از نقاط قوت ابزار DART درمقایسه با ابزارهای فازینگ[[26]](#footnote-26) که با استفاده از ورودی های دلخواه به تحلیل پویا برنامه می پردازند پوشش کد[[27]](#footnote-27) بیش تر است. به عنوان مثال اگر عبارت شرطی "if (x==10) then …" با استفاده از روش های فازینگ مورد آزمون قرار بگیرد و فرض کنیم که متغیر int x یک خانه ی 32بیتی از حافظه است تنها به احتمال یک از 232  مقدار ممکن شاخه ی then عبارت شرطی اجرا می شود. این به این معناست که با استفاده از روش های فازینگ نمی توان به پوشش کد مناسبی دست یافت. با استفاده از ابزار DART اما احتمال اجرای شاخه ی then و else با یکدیگر برابر و برابر 0.5 می باشد.

در واقع روش اجرای پویانمادین که برای اولین بار در پژهش DART مطرح شد می تواند به عنوان روش جایگزین تحلیل ایستا مطرح گردد. این روش در زمینه ی آزمون کد کتابخانه های خارج از برنامه و تشخیص خطا های موجود در زیربرنامه ها بسیار بهتر از تحلیل ایستا عمل می کند.

### 3-1-3-طرح کلی DART با استفاده از دو مثال

در شکل 5 کد ساده مربوط به مثال اول را مشاهده می کنید که قصد داریم تا با استفاده از ابزار DART آن را مورد تحلیل قرار دهیم.

4-مثال اول

همانطور که مشاهده می کنید تابع h دارای خطا می باشد زیرا یکی از مسیر های اجرایی آن می تواند به عبارت abort() ختم شود؛ اما با استفاده از روش فازینگ احتمال بسیار کمی دارد که با اجرای برنامه ورودی های دلخواه بگونه ای داده شود که برنامه به عبارت abort() برسد.

اگر تکه کد شکل 5 را با استفاده از ابزار DART مورد آزمون قرار دهیم. فرض کنید که ورودی های دلخواه اولیه x=269167349 , y=889801541 باشد، درآن صورت شاخه ی then اولین عبارت شرطی اجرا خواهد شد ولی دومین عبارت شرطی به شاخه ی else خواهد رفت بنابراین در اولین اجرا خطایی یافت نخواهد شد.

درحین اجرای این مسیر اولیه عبارات شرطی x0 != y0و 2·x0 != x0+10 با استفاده از عبارات نمادین x0,y0 استخراج می شوند. در اولین مسیر اجرا شرط مسیر[[28]](#footnote-28) به صورت« x0 != y0 , 2·x0 != x0+10 » بوده­است.اکنون در اجرای دوم با استفاده از حل ارائه شده توسط حل کننده ی قید مسیر مربوط به   
« x0 != y0 , 2·x0 != x0+10 » اجرا می گردد و برنامه با رسیدن به عبارت abort به خطا می رسد.

درواقع ابزار DART ابتدا با مقادیر دلخواه ورودی برنامه را اجرا می نماید. پس از آن در حین اجرای یک مسیر از برنامه شرط های آن را مسیر را به طور نمادین استخراج می نماید. با استفاده از این شرط ها و خروجی حاصل از حل کننده ی قید مسیر بعدی اجرای برنامه انتخاب می گردد و متناسب با آن ورودی هایی تولید می شود. این فرآیند تا رسیدن به خطا یا پایان برنامه ادامه می یابد.

درواقع زمانی که به یک شرط در مسیر اجرای برنامه می رسیم دو مقدار کلی برای آن درنظر گرفته می شود مقدار یک نشان دهنده ی اجرای زیرشاخه ی then از آن شرط است و مقدار صفر اجرای زیرشاخه ی else را نشان می دهد.

ویژگی اصلی ابزار DART دو مورد پیش رو است که بر اساس قضیه ی صفحه ی بعد بیان می شود.

1. مانعیت[[29]](#footnote-29)(نسبت به خطا های یافت شده)
2. جامعیت [[30]](#footnote-30)

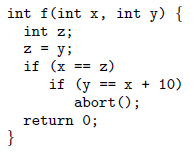
قضیه:

یک برنامه با نام P را درنظر بگیرید. سه حالت زیر ممکن است رخ دهد:

1. اگر اجرای ابزار DART منجر به تولید خروجی "Bug found" گردد؛ در آن صورت لزوما مقادیر عددی به عنوان ورودی وجود خواهند داشت که منجر به تولید خطا شوند.
2. اگر اجرای ابزار DART بدون نمایش هیچ پیغام خطایی اتمام یابد در آن صورت لزوما هیچ مقدار ورودی وجود نخواهد داشت که منجر به خطا گردد. درواقع تمامی مسیر های برنامه توسط ابزار پیموده شده و خطایی یافت نشده است.
3. اجرای ابزار DART برای زمان نامتناهی ادامه خواهد یافت.

واضح است که ابزار DART با اجرای برنامه، مسیر منتهی به خطا را می یابد بنابراین برخلاف تحلیل های ایستا که مثبت کاذب بسیار زیادی را شامل می شوند لزوما هر خطای یافت شده توسط DART به طور واقعی در برنامه وجود خواهد داشت.

در شکل 6 تکه کد مربوط به مثال دوم را مشاهده می کنید.



شکل 5- تکه کد ساده برای آزمون با ابزار DART-مثال دوم

در این تکه کد نوشته شده به زبان C، در ابتدا بردار آدرس به صورت M0=(mx , my) می باشد. در واقع  
 mx وmy آدرس های متفاوت پارامتر های ورودی تابع f [[31]](#footnote-31) در حافظه سیستم هستند. فرض می کنیم که ورودی بی­قاعده ی ابتدایی برای پارامترx=123456 و برای پارامتر y=654321 باشد. در این صورت حافظه ی عددی سیستم در ابتدا به صورت*M* = [*mx →* 123456*, my →* 654321] و حافظه ی نمادین سیستم نیز به صورت *S* = [*mx → mx ,my → my*] خواهد بود.

با این ورودی های اولیه، مسیر اجرای اول به زیرشاخه ی else عبارت if اول می رود. سپس به دستور return0 می رسد و این مسیر پایان می یابد. شرط مسیر حاصل از این اجرای اولیه *￢*(*mx* = *my*) می باشد. اکنون پس از پایان اجرای اول شرایط حافظ عددی و نمادین به صورت زیر است:

*M* = [*mx →* 123456*, my →* 654321*, mz →* 654321]

*S* = [*mx → mx,my → my,mz → my*]

اکنون نقیض شرط مسیر به دست آمده در اجرای قبل به حل کننده ی قید داده می شود تا امکان پذیر بودن آن را بررسی کند و در صورت امکان پذیر بودن ورودی عددی متناسب با آن را تولید کند. حل کننده ی قید به عنوان حل شرط( *mx* = *my* ) ، خروجی (*mx →* 0*,my →* 0) را ارائه می دهد که با استفاده از آن در اجرای دوم زیرشاخه ی then شرط if اول اجرا شده و به شرط جدید (*mx* = *my,my* = *mx* + 10) می رسد. در این جا نیز به حل کننده ی قید رجوع می شود تا امکان پذیر بودن شرط مسیر جدید را بررسی نماید. اما با توجه به آنکه از نظر منطقی این شرط نمی تواند صحیح باشد بنابراین مسیر یافت شده غیر امکان پذیر تشخیص داده می شود و اجرای ابزار DART بدون اعلام هرگونه پیام خطا پایان می یابد. بنابراین اگرچه دستورالعمل abort() در متن کد موجود بود ولی مسیر منتهی به آن امکان پذیر نبوده و کد مثال2 هیچ خطایی را در برندارد.

## 3-2-ابزار Mayhem

### 3-2-1- مقدمه

MAYHEM [5]ابزاری برای یافتن آسیب پذیری‌ها در فایل‌های اجرایی یا به بیان دیگر فایل‌های باینری است. ابزار MAYHEM ، به ازای هر آسیب پذیری که کشف می کند کد بهره بردار مربوط به آن را نیز در اختیار تحلیل گر می گذارد. ابزار MAYHEM برای ارزیابی توانایی های خود موفق شد 29 مورد آسیب پذیری در پلتفرم­های ویندوز و لینوکس را کشف و کد بهره بردار برای آن ها را تولید نماید. دو مورد از این آسیب پذیری ها تا پیش از این گزارش نشده بودند.

درواقع ابزار MAYHEM از اجرای نمادین بهبود یافته بهره می برد به این صورت که در جایگاه های مشکوک کد باینری شرط هایی را به اجرای نمادین اضافه می کند. این شرط ها نشان دهنده ی موارد مهم برای مهاجم هستند به عنوان مثال آدرس جایی که پس از اجرا اشاره گر IP به آن بر می گردد. با استفاده از این شروط فرمول جدیدی برای آن مسیر اجرا به دست می آید. اگر این فرمول توسط حل کننده­ی قید قابل حل باشد در آن صورت ابزار MAYHEM به یک مسیر اجرا برای کد بهره­برداری از آسیب پذیری دست یافته است .

برای یافتن کد بهره بردار آسیب پذیری، لازم است تا فضای اجرایی بزرگی از برنامه جست‌وجو شود. درMAYHAM برای مقابله با این موضوع چهار اصل کلیدی پیش بینی شده است:

1. سیستم باید بتواند برای مدت طولانی رو به جلوپیش رود. در واقع به طور ایده آل سیستم باید بتواند همواره اجرای خود را ادامه دهد بدون اینکه از منابع اختصاص داده شده بیش از حد مجاز استفاده کند.
2. با هدف بالابردن کارایی سیستم، نباید یک کار تکراری دوباره اجرا گردد.
3. سیستم نباید یک کار یا نتیجه آن را رها کند بلکه باید در هر اجرا از نتایج اجرای گذشته استفاده کند.
4. سیستم باید بتواند در مورد مقدار نمادین یک حافظه که خواندن و نوشتن در آن بستگی به کاربر دارد، تصمیم گیری کند.

به طور کلی دو نوع مختلف را برای ابزار های تحلیل‌کد می توان درنظر گرفت:

1. ابزار های آفلاین:

ابزار های تحلیلی‌که ابتدا یک مسیر از کد را به صورت عددی اجرا می‌کنند و سپس با حل مجموعه قیدهای تولید شده در آن میسر به صورت نمادین، سعی می‌کنند تا ورودی هایی را برای مسیر اجرایی جدید تولید کنند. به عنوان مثال ابزار SAGE این گونه است.

1. ابزار های آنلاین:

در این دسته از ابزار های تحلیل سعی می‌شود تا تمامی مسیر های اجرای ممکن در یک اجرا ی واحد ، اجرا شوند. به عنوان مثال ابزار S2E این گونه است.

در ابزار های تحلیل آفلاین، اصول‌ 1 و 3 برقرارند. زیرا در هربار یک اجرا و مستقل از اجراهای دیگر صورت   
می گیرد پس از نتایج اجراهای قبلی، می‌توان دوباره استفاده کرد.اما در این ابزارها، اصل 2 برقرار نیست، زیرا   
هر مسیر اجرا بایستی از ابتدا اجرا شود پس لزوما برخی از دستورات در تمام مسیرها چندباره اجرا می‌شوند.

برخلاف ابزار های تحلیل آفلاین، تحلیلگرهای آنلاین مثل KLEE و S2E در هر نقطه از شروع شاخه جدید از fork استفاده می‌کنند، بنابراین اصول‌ 1 و 3 درمورد این ابزار ها برقرار نیست. علت آن است که با هر fork جدید، دستورات گذشته دوباره اجرا نمی شوند. اما دستور fork سربار زیادی را از نظر حافظه به ابزار تحمیل   
می کند.

### 3-2-2-دستاوردهای جدید MAYHEM:

ابزار MAYHEM دستاورد های جدیدی را ارائه داده است:

* اجرای ترکیبی آفلاین و آنلاین برای حفظ حافظه و سرعت اجرا
* مدل مبتنی بر ایندکس برای حافظه
* تحلیل فایل‌های باینری با ارائه ی کد بهره بردار برای آسیب پذیری

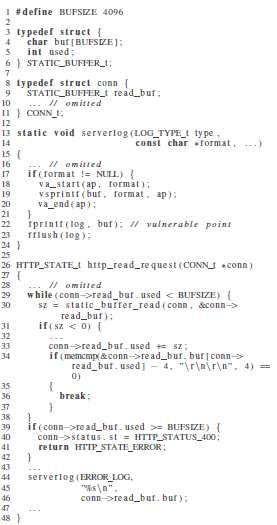
نکات کلی زیر در مورد آسیب پذیری‌ها و ابزار های تحلیل کد مطرح است:

1. نیاز است تا کد برنامه در سطح کد ماشین اجرا شود چون آسیب پذیری‌های زیادی مانند سرریز پششته وجود دارد که برای تشخیص آن ها الزاما باید ساختار پشته هم مورد بررسی قرار گیرد.
2. تعداد مسیرهای ممکن خیلی زیاد است زیرا هر دستور if یک شاخه­ی جدید را در درخت اجرا ایجاد می‌کند.
3. هرچه تعداد پوشش کد بیش تر باشد یعنی تعداد مسیر بیشتری از برنامه بررسی گردد، بهتر است.
4. اگر ابتدا کد برنامه، عددی اجرا شود و در صورت نیاز به اجرای نمادین تغییر کند تحلیل کد بهتری خواهیم داشت.

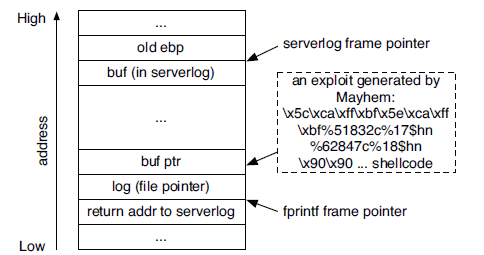
### 3-2-3-ساختار کلی ابزار MAYHEM

در این زیربخش ما قصد داریم تا ساختار کلی ابزار MAYHEM و نحوه ی عملکرد آن را نمایش دهیم. در شکل 7 تکه کد مربوط به برنامه ی آسیب پذیر orzHttpd آمده است. در این برنامه هر ارتباط جدید http به تابع http\_read\_request فرستاده می شود و در خط 29 با فراخوانی تابع static\_buffer\_read رشته ی درخواست http کاربر دریافت می شود. ورودی کاربر در یک بافر 4096 بایتی به نام conn->read\_buf.buf ذخیره می گردد. برای جلوگیری از آسیب پذیری سرریز پشته در صورتی که ورودی بیش از 4096 بایت باشد و کاراکتر پایان خط در http نیامده باشد، خطای 400 بازخواهد گشت و از سرریز پشته جلوگیری خواهد شد.

آسیب پذیری موجود در این برنامه در serverlog موجود است، در آن جا تابع fprintf با یک ورودی ساختاریافته از دریافت http کاربر فراخوانی می شود. در شکل 8 وضعیت پشته در زمانی که آسیب پذیری format string تشخیص داده می شود نشان داده شده است.



شکل 6- تکه کد مربوط به برنامه ی آسیب پذیر orzHttpd



شکل 7- وضعیت پشته برای برنامه ی آسیب پذیر orzHttpd

معماری ابزار MAYHEM در شکل 9 مشاهده می شود. این ابزار از دو بخش همروند برای تحلیل کد با نام CEC [[32]](#footnote-32) یا همان کلاینت اجراکننده ی عددی و [[33]](#footnote-33) SESیا همان سرور اجراکننده­ی نمادین تشکیل شده است. اگر بخواهیم از دیدگاهی با انتزاع بالا سخن بگوییم، CEC برروی یک سیستم هدف مشخص اجرا می شود اما SES برروی هر پلتفرمی و بنابر ارجاعات CEC اجرا می گردد. ابزار MAYHEM برای تولید کد بهره بردار[[34]](#footnote-35) آسیب پذیری مراحلی را انجام می دهد که به شرح زیر است:

1. ابزار MAYHEM با تعریف یک پورت کار خود را شروع می‌کند.کد برنامه ی مورد تحلیل از طریق همین پورت دریافت می‌شود و در ابتدا اجرای نمادین برروی آن اعمال می گردد.
2. واحد CEC برنامه ی آسیب‌پذیر را دریافت می‌کندَ و به SES متصل می‌شود تا مقداردهی‌های اولیه نمادین برای ورودی­ها صورت پذیرد. سپس کد به صورت عددی در واحد CEC اجرا می‌شود. در حین اجرای عددی instrumentationکد باینری وتحلیل آلایش[[35]](#footnote-38) پویا نیز اجرا می گردد. تحلیل آلایش پویا در این جا با استفاده از زیربخشTaint tracker که در شکل 9 نیز آمده است انجام می شود.
3. زمانی که CEC با یک بلاک کد آلوده یا یک پرش آلوده رو به رو می شود(منظور از پرش آلوده، جایی از کد برنامه است که برای ادامه ی کار به یک ورودی از مهاجم نیاز دارد.)،اجرای عددیCEC موقتاً متوقف می گردد و شاخه ی آلوده ی کد به SES برای اجرای نمادین ارسال می‌شود. SES به طور کلی برای این شاخه از کد مشخص می‌کند که آیا اجرای آن امکان پذیر است یا خیر؟
4. واحد SES به صورت موازی با CEC اجرا شده و بلاک‌های کد را دریافت می‌کند. این بلاک‌ها به زبان میانی تبدیل می‌شوند و به صورت نمادین اجرا می­گردند. پس از آن در صورت لزوم مقادیر عددی مورد نیاز از CEC دریافت می‌شود.

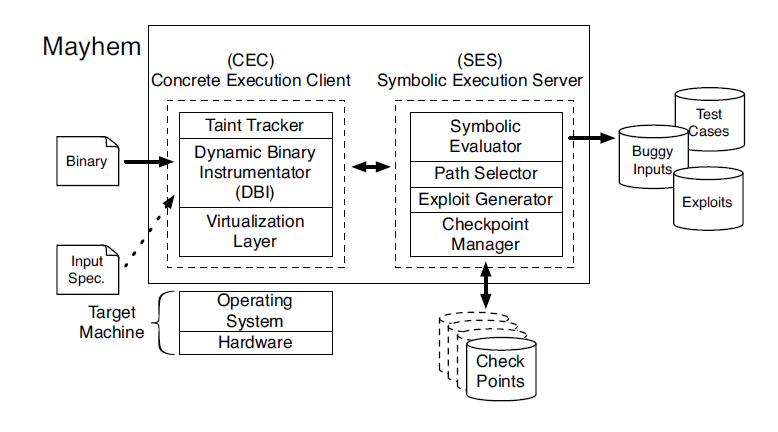
واحدCEC ­دو نوع فرمول فراهم می نماید:

1. فرمول مسیر[[36]](#footnote-39)

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

فرمول قابلیت کد بهره بردار[[37]](#footnote-40)

فرمول قابلیت کد بهره بردار، نشان­دهنده ی آن است که آیا مهاجم می تواند کنترل اشاره گر دستورالعمل[[38]](#footnote-41) را به دست بگیرد و آیا می تواند کد مخرب خود را اجرا نماید؟

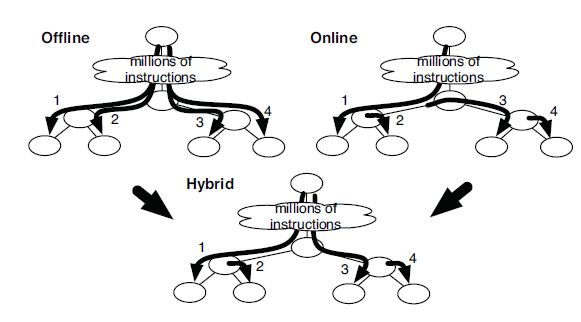


شکل 8- معماری MAYHEM

1. ابزار وقتی به یک پرش آلوده می‌رسد SES بر اساس پرس و جویی از حل کننده­ی قید تصمیم می‌گیرد که آیا دستور fork لازم هست یا خیر. اگر fork لازم باشد اجراهای جدید اولویت بندی شده و به ترتیب اجرا می‌شوند. در طی اجرا های جدید اگر منابع سیستم به پایان برسند، واحدSES­ رویه­ی بازگشت را اجرا می‌کند. در نهایت بعد از اتمام اجرای یک پردازه، تعدادی موردآزمون براساس آن تولید می‌شوند.
2. در پرش‌های آلوده یک فرمول قابلیت کد بهره بردار تولید و به واحدSES داده می‌شود. اگر این فرمول قابل ارضا باشد یعنی کدبرنامه از این مسیر آسیب‌پذیر و. قابل بهرهبردااری توسط مهاجم است.

برای اجرای نمادین لازم است تا کد باینری به کد اسمبلی تبدیل شود. برای این منظور از ] BAP32[ استفاده می‌شود. سپس کد اسمبلی به شکل نمادین اجرا می‌شود.

همانگونه که در بخش 3-2-1 مطرح شد، ابزار های تحلیل کد به دو دسته ی آنلاین و آفلاین تقسیم می شوند اما ابزار MAYHEM در یک نوآوری علمی از اجرای پویانمادین هیبرید استفاده می­نماید. این به آن معناست که ابزار ابتدا کد برنامه را به صورت آنلاین اجرا می‌کند. اما زمانی که اجرای ابزار به محدوده­ی منابع سیستم برسد، اجرای آنلاین متوقف شده و پس از آن اجرا به شکل آفلاین ادامه می یابد.



شکل 9- انواع اجراهای پویانمادین

### 3-2-4-اجرای هیبرید پویانمادین در ابزار MAYHEM

ابزار MAYHEM در اجرای هیبرید چهار مرحله اصلی را طی می نماید:

1. **مقداردهی اولیه:** زمانی که ابزار MAYHEMبرای اولین بار کد یک بر نامه را بارگذاری می نماید، پایگاه‌داده CHECKPOINTها، محل نگهداری موردآزمون‌ها و مدیر CHECKPOINTها را مقدار دهی اولیه می‌کند.
2. **جست و جوی آنلاین:** در این مرحله کد برنامه مورد اجرای نمادین آنلاین قرار می گیرد.در پایان این مرحله موردآزمون نیز تولید می گردد.
3. **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 تعین‌شده ای برسد، از میان اجراهای حاضر یکی از آن ها انتخاب وCHECKPOINT مربوط به آن تولید می‌گردد. این CHECKPOINT شامل وضعیت اجرای نمادینی است که متوقف شده و زمانی که منابع سیستم آزاد گردد مجددال ادامه می یابد. وضعیت اجرای نمادین شامل شرط مسیر، آمارها و موارد دیگری می باشد.
4. **بازسازی با CHECKPOINT** [[39]](#footnote-43)**:** یکی از CHECKPOINTها با توجه به یک هیوریستیک از پیش تعیین شده ای 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 به صورت آنلاین ادامه می‌یابد.

### 3-2-5-معماری و پیاده‌سازی CEC

واحد CEC به عنوان ورودی، فایل باینری و یک CHECKPOINT دلخواه را دریافت می‌کند. این واحد ابتدا کد برنامه را به صورت عددی اجرا می‌کند. همزمان با اجرای عددی، تحلیل آلایش نیز انجام می­شود. اگر ابزار به بلاک آلوده ای از کد برنامه برسد، بلاک را به واحد SES می‌دهد تا به صورت 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، مسولیت شبیه سازی OS را بر عهده دارد. هر اجراکننده عددی برای خود یک وضعیت[[40]](#footnote-44) جداگانه دارد که از دید دیگر اجرا کننده‌ها پنهان است. به عنوان مثال اگر یک اجراکننده روی فایل می‌نویسد بقیه ی اجراکننده‌ها یک کپی از این فایل را دارند و می‌توانند همزمان روی آن بنویسند.

### 3-2-6-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، انتخاب کننده 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 حالت هایی از اجراکننده نمادین است که در اجرای اول به دلیل محدودیت حافظه اجرا نشده اند. هر حالت اجرای نمادین شامل دو مقدار است. اولین مقدار، مقادیر ثبات­های نمادین است و مقدار دوم، مقادیر حافظه نمادین می باشد. در هر CHECKPOINT این حالت­های اجرایی نگهداری می‌شوند.

در زیربخش پیش بینی اجرای نمادین[[41]](#footnote-45)،کاربر به عنوان ورودی می‌تواند تعدادی قید را برروی ورودی‌های برنامه تعریف کند به عنوان مثال می توان قیدی را برروی طول ورودی تعریف کرد تا فضای جست و جو را کاهش دهد. اگر هیچ قیدی برروی ورودی های برنامه تعریف نشود، واحد SES تلاش می‌کند تا تمام فضا ی کد برنامه را پوشش دهد.

در زیر بخش انتخاب مسیر، ابزار MAYHEM برای انتخاب مسیر بعدی واحد SES از هیوریستیک هایی استفاده می‌کند. این ابزار سه قاعده ی هیورستیک را برای رتبه بندی مسیر ها به کار می برد:

1. جست و جوی بلاک جدیدی از کد (در برابر جست و جوی تکراری بلاک های پیمایش شده از کد )
2. اجراکننده‌هایی که از حافظه نمادین استفاده می‌کنند در اولویت هستند.
3. مسیرهای اجرایی که اشاره گر دستورالعمل های نمادین[[42]](#footnote-46) در آن ها تشخیص داده می شوند، در اولویت هستند.

### 3-2-7-مدل سازی حافظه در MAYHEM

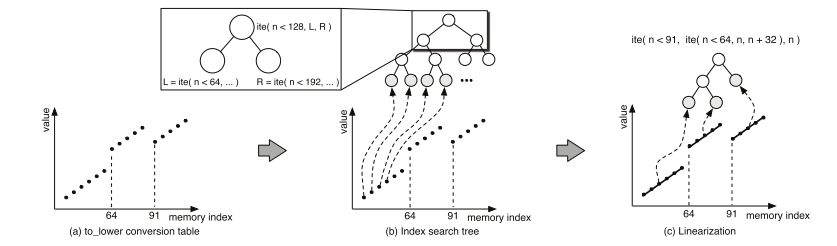
ابزار MAYHEM به صورت جزئی حافظه را مدل‌سازی می‌کند. به این معنی که آدرس دستورات read می‌تواند به صورت نمادین یا عددی باشند ولی در دستورات write لزوما آدرس باید عددی باشد. برای تعریف حافظه نمادین MAYHEM از مفهوم Memory Object استفاده می‌کند که با M نشان داده می‌شود. هربار که آدرس حافظه در کد به شکل نمادین باشد، یک M تازه ایجاد می‌شود که مجموعه ای از آدرس‌هایی با عبارت نمادین قابل دسترسی می باشد. این حافظه برخلاف u غیر قابل تغییر است و صرفا یک تصویر از u می باشد.

در بدترین حالت M شامل تمام 232 حالت ممکن است که این فضای حالت بسیار بزرگی است. برای کاهش آن از یک جست و جوی دودویی استفاده می‌شود و سعی می‌شود که کران بالا و پایین Memory Object به صورت محافظه کارانه ای 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**[[43]](#footnote-47)**:** برای تصفیه و بدست آوردن محدوده ی مورد نیاز، لازم است تا تعدادی پرس و جو به SMT زده شود. برای کاهش این گونه درخواست‌ها یک کش قرار داده می‌شود که قدم­ها و نتیجه تصفیه نهایی را در خود نگهداری می‌کند. برای هر قدم جدید ابتدا درخواستی به این کش ارسال می‌شود در صورت وجود، یک در خواست به SMT ارسال می‌شود تا ارضاپذیری آن و میزان دقت آن برای فورمول جدید را بررسی نماید.

**کش Lemma**: با وجود 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[[44]](#footnote-48) هر فرمول(F) و نتیجه آن(Q) به صورت F→Q در آن نگهداری می‌شود. با استفاده از کش Lemm می توان96درصد پرس و جو های حافظه را کاهش داد.

**درخت‌های جست‌وجوی ایندکس**[[45]](#footnote-49)**:** برای بهبود جست و جو در Memory Object با هدف یافتن عبارات، MAYHEM از یک درخت دودویی IST [[46]](#footnote-50) استفاده می‌کند.

IST(E)= ite(i<addr(Eright),Eleft,Eright))

شکل 10-درخت دودویی IST و خطی سازی آن

E عبارت مورد جست و جو است و مقدار راست و چپ آن با Eright و Eleft نشان داده می‌شود.

برای اینکه تعداد گره های درخت IST کاهش یابد از ایده خطی سازی پاکتی استفاده می شود. ایندکس‌ها و مقادیرشان معمولا در کنار هم در یک خط قرار می گیرند. قبل از تولید درخت با یک پیش پردازش مجموعه پاکت های خطی تولید می شود و سپس با استفاده از آنها درخت تولید می شود با این کار به عنوان مثال تعداد گره ها از 256 به 3 کاهش خواهد یافت.

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 آدرس های نمادین را عددی می کند. البته این عددی کردن دلخواه صورت نمی گیرد و بر اساس سه معیار زیر خواهد بود:

1. اگر امکان پذیر است آدرسی تولید شود که برنامه را به قسمت های پوشش داده نشده هدایت کند.

2. در صورتی که معیار شماره­ی1 قابل اعمال نبود،آدرسی تولید شود که به حافظه نمادین اشاره می­کند.  
 لازم به ذکر است که ابزار قسمت های حافظه را به تعدادی منطقه­ی نمادین[[47]](#footnote-51) تقسیم می کند که این مناطق بر اساس پیچیدگی قیدهای مرتبط به آن­ها مرتب شده اند. ابزار سعی می کند اشاره‌گر به منطقه‌ای ساده‌تر را ایجادکند.

3. اگر موارد بالا قابل اجرا نبودند، آدرس عددی معتبر و دلخواهی تولید خواهد شد.

## 3-3-ابزار Drilller

### 3-3-1-مقدمه

ابزارهای جدید تحلیل برنامه تنها می‌توانند به نقص‌های سطحی دست پیدا کنند. در حقیقت برای هوشمند سازی fuzzerها باید مکانیزم هوشمندانه ای برای تولید ورودی‌های جدید داشت. در ساده ترین حالت این ابزارها به صورت دلخواه این کار را انجام می‌دهند یعنی ورودی های جدید کاملا دلخواه تولید می­شوند.اما ابزارهای که به اجرای پویانمادین می پردازند، ورودی‌های جدید را به کمک حل­کننده ی قید تولید می­کنند. مشکل ابزارهای اجرای پویانمادین انفجار مسیر در تحلیل برنامه است.از همین رو با استفاده از این ابزار ها نمی توان به آسیب‌پذیری‌هایی در عمق بالا دست یافت.

دستاورد علمی ابزارDriller این است که روش fuzzing را با روش پویانمادین انتخابی[[48]](#footnote-52) ترکیب کرده است.

## 3-4-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل برنامه ها

جدول 1-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل برنامه ها

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ابزار | سال ارائه | زبان | پلتفرم | نوع | ویژگی ها |
| DART | **2005** | **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 حل کننده قید ندارد  در حل قیدهای مربوط به اشاره‌گرها مشکل دارد |
| Mayhm | **2012** | **Binary** | **PC** | **Hybrid** | مدل سازی حافظه  با هدف یافتن آسیب پذیری به همراه ارائه ی کد بهره بردار |
| Driller | **2016** | **Binary** | **PC** | **Hybrid** | FUZZING، Concolic، تکرار  Fuzzer انتخاب شده از نوع instrumented-genetic هست.  AFL موتور پویانمادین شبیه سازی ابزار Mayhem هست. |

**فصل چهارم:**

**ابزار های اجرای پویانمادین   
برای تحلیل برنامه های اندرویدی**

# فصل چهارم: ابزار های اجرای پویانمادین برای تحلیل برنامه های اندرویدی

## 4-1-چالش های علمی برای تحلیل برنامه در اندروید

در فصل قبل ابزار های اجرای پویانمادین برای تحلیل کد برنامه در زبان های برنامه نویسی متفاوت را دیدیم. در این فصل در بخش مقدمه به تفاوت تحلیل برنامه‌های کاربردی اندروید با سایر برنامه‌ها می پردازیم سپس ابزار های اجرای پویانمادین در پلتفرم اندروید را مورد برسی قرار خواهیم داد.

1-برنامه‌های اندروید بسیار وابسته به کتابخانه‌های چارچوبه کاری هستند و این موضوع باعث ایجاد مشکل واگرایی مسیر می‌شود. در اجرای نمادین اگر یک مقدار نمادین از زمینه برنامه خارج شود، مثلا برای انجام یک پردازش به یک کتابخانه داده شود یا در اختیار چارچوبه 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 به صورت نمادین اجرا کرد.

ب) در کتابخانه خارجی ممکن است تعدادی قید وجود داشته باشد که در خروجی و حاصل پردازش کتابخانه موثر باشند. از این جهت این قیدها در تولید مورد‌آزمون‌ها موثر خواهند بود و به جای آزمون برنامه اصلی، تمرکز به آزمون مسیر واگرا شده در کتابخانه معطوف می‌شود و به طور پیوسته لازم است تا قسمتی از سیستم عامل اندروید به صورت 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 مولفه‌های مختلف در اندروید است. به وسیله Intent یک مقدار به یک مولفه در درون یک برنامه یا به مولفه‌ای در برنامه دیگر ارسال می‌شود. Intent بعد از خارج شدن از محدوده برنامه وارد کتابخانه‌های سیستمی‌شده و بعد از آن وارد مولفه مقصد می‌شود.

2-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 صفحه نمایش ایجاد شود. علاوه بر کاربر برنامه‌های ثانویه هم می‌توانند رخداد تولید کنند مثل رخداد تماس ورودی یا دریافت یک پیام.

3-برنامه‌های کاربردی اندروید وابسته به مجموعه ای از کتابخانه‌های اندروید هستند که خارج از دستگاه اندروید یا شبیه‌ساز اندروید دردسترس نیست. کد اندروید برخلاف کد برنامه‌های جاوا که در ماشین مجازی جاوا اجرا می‌شوند در ماشین مجازی Dalvik اجرا می‌گردد. پس به جای java byte-code برنامه‌های اندروید به Dalvik byte-code کامپایل می‌شوند.

## 4-2-ابزار ACTEV

### 4-2-1-مقدمه

ابزار [8]ACTEV اولین ابزاری بود که اجرای پویانمادین را برای تحلیل خودکار برنامه‌های کاربردی گوشی‌های هوشمند پیاده سازی کرد. پلتفرم مورد نظر ابزار ACTEV ، سیستم عامل اندروید است.

برنامه های کاربردی اندروید دارای ویژگی‌هایی منحصر به فردی هستند که تحلیل ایستا در آن­ها دچار چالش های جدیدی می‌شود. به عنوان مثال وجود SDK [[49]](#footnote-56)، ناهمگامی، تعامل میان‌پردازه‌ای[[50]](#footnote-57)، پایگاه‌داده‌ها و  
 رابط کاربری گرافیکی[[51]](#footnote-58) موجب می شود تا تحلیل ایستا برای برنامه های اندروید به کارایی و راحتی دیگر پلتفرم ها قابل اعمال نباشد.

از سویی دیگر به طور کلی در تحلیل پویا، هدف تولید پویای ورودی‌های برنامه است.برنامه های اندرویدی علاوه بر ورودی‌های عادی که در هر برنامه‌ای از پلتفرم های مختلف وجود دارند مانند اعداد و رشته‌ها، رخداد[[52]](#footnote-59)ها را نیز به عنوان ورودی شامل می شوند. یک رویداد ضربه[[53]](#footnote-60) روی صفحه نمایش، فشردن یک کلید از دستگاه و یا ورود یک پیامک‌ از جمله رخدادهای معمول برنامه های اندرویدی است.

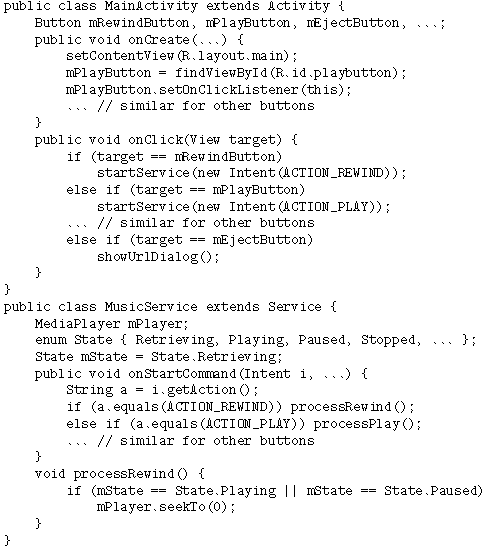
برنامه های اندرویدی زیرمجموعه‌ای از برنامه‌های مبتنی بر رخداد[[54]](#footnote-61) هستند. این گونه برنامه‌ها دارای محاسباتی حاصل از تعامل با ترتیب نامحدودی از رخدادها و پاسخ به آن­ها می­باشند. در آزمون این گونه از برنامه‌ها با دو چالش جدید علمی رو‌به‌رو هستیم:

1. چطور یک رخداد ایجاد می­شود؟
2. چطور ترتیبی از رخدادها ایجاد می­شود؟

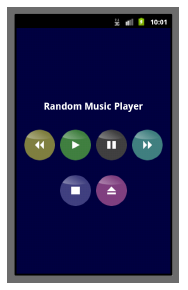
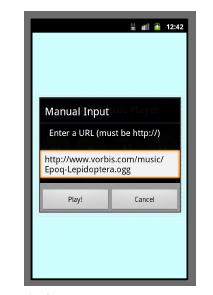
این مقاله از روش اجرای پویانمادین برای تولید رخدادها استفاده می‌کند. درواقع ابزار ACTEV رخدادها را از جایی که تولید می‌شوند تا جایی که کنترل می‌شوند، دنبال می‌کند. در مورد چالش دوم اگر از اجرای پویانمادین مرسوم استفاده شود، در مورد برنامه‌های واقعی با مشکل انفجار مسیر رو‌به‌رو می‌شویم. درواقع حجم مجموعه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

### 4-2-2-کلیات طرح

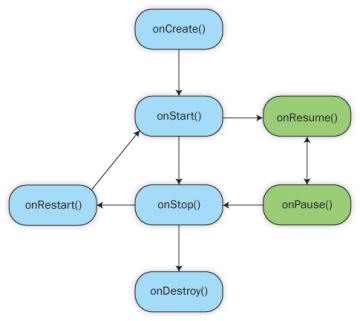
برای توضیح طرح از یک برنامه ساده به زبان اندروید استفاده شده است. کد مربوط به این برنامه در شکل 12و نمایی از این برنامه اندرویدی نیز در شکل13 آمده است. برنامه‌های اندرویدی خود به تنهایی تابع main ندارند. بلکه این برنامه‌ها کامل کننده SDK اندروید هستند. درواقع هر برنامه اندرویدی از تعدادی



شکل 11- کد برنامه ی اندرویدی برای تست با ابزار ACTEV



شکل 13- نمایی از برنامه ی اندرویدی برای تست با ابزار ACTEV

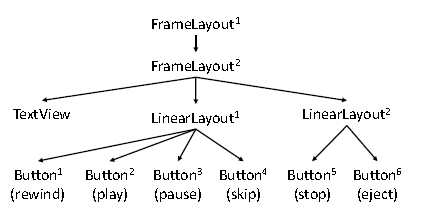


شکل 14- چرخه ی حیات یک Activity در اندروید

Activity تشکیل شده است و در ابتدا مشخص نیست که کدام یک از این Activity ها فراخوانی می­شود.   
هر Activity از چرخه حیات Activity در اندروید پیروی می کند. چرخه ی حیات یک Activity مطابق   
شکل 14، شامل توابع onCreate()، onStart()، onResume می باشد. در کد موجود در شکل 12  
تابع onCreate به همراه تعدادی دکمه تعریف شده است. به ازای هریک از این دکمه ها، تابع onClickListener مربوطه تعریف شده است. کلاس MusicService هم برای اجرای سرویس خاص هر دکمه پیاده‌سازی شده است.

### 4-2-3-تولید یک رخداد

در برنامه­ی اندرویدی شکل14 رخداد ضربه[[55]](#footnote-62) در نظر گرفته شده است. شکل16 نمایی از سلسله مراتب ویجت[[56]](#footnote-63)های موجود در صفحه اصلی برنامه را نشان می‌دهد. برگ‌های درخت در نهایت دکمه‌های روی صفحه و نمایشگر متن[[57]](#footnote-64) هستند. هدف، تولید رخدادهایی است که برای همه این یازده ویجت رخداد ضربه را به طور خودکار تولید نماید.



شکل 17- سلسله مراتب ویجت ها در صفحه ی اصلی برنامه

در روش‌های گذشته این درخت یا به صورت دستی[[58]](#footnote-65)(Model-based) یا به صورت خودکار(Capture-Replay) تولید می‌شده است. اما در روش خودکار طرح به صورت موردی بوده و همه ویجت‌ها، چه ویجت‌های SDK و چه ویجت‌های سفارشی، پشتیبانی نمی شوند. در این مقاله از اجرای چویانمادین رای تولید رخدادها استفاده می‌شود. برای این منظور SDK و برنامه تحت آزمون باید تجهیز شوند. سپس در حین اجرای یک رخداد عددی، یک رخداد به صورت 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.

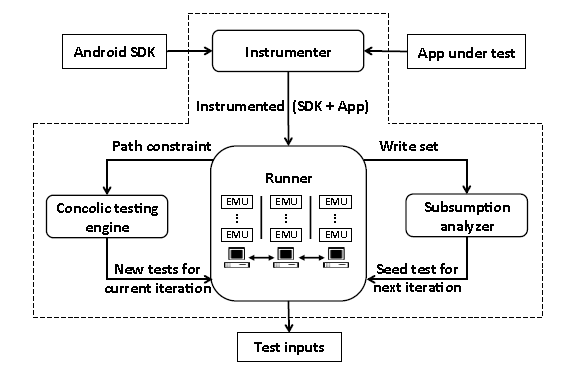
### 4-2-4-تولید ترتیبی از رخدادها

برای اینکه تمام مسیرهای 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 21 هزار ترتیب چهارتایی از رخدادها را باید تولید کرد.

برای حل مساله ی انفجار مسیر در این مقاله، مجموعه رخدادهای تولید شده مورد بررسی قرار گرفته اند. بررسی‌ها نشان می‌دهد که تعداد زیادی از مجموعه رخدادهای تولید شده تکراری هستند. درواقع برنامه‌ها را می‌توان به شکل یک نمودار حالت[[59]](#footnote-66) در آورد که با هر رخداد حالت برنامه تغییر می‌کند. تعدادی از رخدادهای تولید شده توسطاجرای پویانمادین حالت برنامه را تغییر نمی‌دهند و از آنجایی که در حافظه چیزی نمی‌نویسند به آنها رخدادهای فقط خواندنی[[60]](#footnote-67) گفته می‌شود. برای جلوگیری از تولید این گونه رخدادها موارد زیر در نظر گرفته شده است:

1. تعدادی از ویجت‌ها مثل linearLayout و FrameLayout برخلاف دکمه‌ها هنگام رخداد دارای کنش[[61]](#footnote-68) نیستند. پس تنها شش ویجت از یازده ویجت موجود در صفحه امکان اجرا دارند.
2. تعدادی از ویجت‌هایی که فعالیت دارند ممکن است به دلایل مشخصی که برنامه‌نویس صلاح دیده است، غیر فعال باشند.
3. در برنامه‌های دارای رابطه کاربری، هنگامی که یک رخداد تولید می‌شود برای جلوگیری از دوباره طراحی صفحه نمایش، اگر رخداد، حالت برنامه را تغییر نمی‌دهد، رخداد فقط خواندنی در نظر گرفته می‌شود.
4. در SDK تعدادی زیادی از رخدادها تعریف شده‌اند 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[[62]](#footnote-69).

در نهایت در روش ارائه شده مجموعه رخدادهایی که به یک رخداد فقط خواندنی ختم می‌شوند، حذف خواهند شد. چون عملا تاثیری در آزمون برنامه نخواهند داشت. حذف این گونه ترتیب رخدادها 21 هزار ترتیب رخداد 4تایی را به حدود 3هزار ترتیب رخداد 4تایی کاهش می‌دهد. معماری ابزار ارائه شده در شکل 16 نشان اده شده است.



شکل 20-ساختار کلی ابزار ACTEV

## 4-3-ابزار Condroid

### 4-3-1-مقدمه

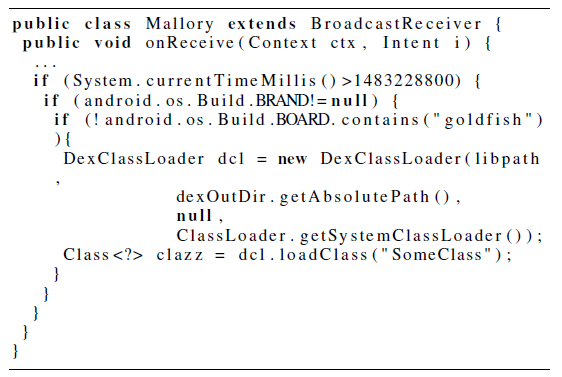
در سال های اخیر ابزار های بسیاری با هدف تحلیل برنامه های اندروید نوشته شده اند. یک نقطه ضعف اصلی در مورد این ابزارها عدم توانایی رسیدن به نقاط حساس کد است. به عنوان مثال بخش­های بارگذاری شده پویا[[63]](#footnote-70)  
و یا بخش هایی از کد که درآن قسمت داده ی خاصی را ترجمه ی رمز می کنند نقاط حساسی هستند که غالبا از دید ابزار های تحلیل کد اندروید پنهان می ماند. درحقیقت نقطه ی ضعف اصلی این ابزار ها آن است که   
لزوما به بخش هایی از کد برنامه که دارای تهدیدات امنیتی است دست نمی یابند زیرا این ابزار ها شرایط پویایی را که مسیر های مشخصی از کد برای اجرا شدن به آن نیاز دارند تامین نمی کنند.

برای حل این چالش علمی، ابزار Cindroid [9]یک رویکرد تحلیل ایستای فراخوانی مسیر[[64]](#footnote-71) و bytecode instrumentation به همراه اجرای نمادین ابتکاری را در پیش گرفته است. در واقع هدف از این رویکرد دستیابی به مسیر های خاص موجود در کد برای تحلیل تهدیدات امنیتی است. این ابزار می تواند در حین تحلیل پویا، برنامه را مجبور به نشان دادن کد بارگذاری شده پویا نماید درواقع مهاجمین با استفاده از همین تکنیک ساده ی کد بارگذاری شده ی پویا می توانند[[65]](#footnote-72) ابزار Google Bouncer را دور بزنند و برنامه ی اندروید موذیانه ی خود را به عنوان یک برنامه ی کاربردی اندروید در فروشگاه مهم و جهانی GooglePlay منتشر کنند. درپایان این پژوهش با استفاده از یک مثال ساده ی بمب منطقی در اندروید نحوه ی کارایی ابزار Condroid را نشان داده شده است.

### 4-3-2-بررسی اجمالی یک مثال

رویکرد این ابزار مبتنی بر تحلیل ایستا ی جریان کنترل و اجرای هیبرید پویانمادین با هدف بررسی مسیر های اجرایی از کد که دارای نقاط حساس هستند می باشد. در شکل 18 یک بمب منطقی مبتنی بر زمان به همراه کد بارگذاری شده ی پویا نشان داده شده است. فرض می کنیم که تابع این مثال بخشی از یک برنامه ی اندرویدی است که به عنوان BroadcastReceiver برای پیامک ورودی نوشته شده است . بنابراین چهارچوب اندروید هرزمان که پیامکی به سیستم وارد شود این تابع را فراخوانی می کند. با توجه به شروط موجود در خط 3 و 4 تنها زمانی که تاریخ سیستم از 2017/01/01 فراتر باشد و کد برروی یک پردازنده ی واقعی[[66]](#footnote-73) و نه یک شبیه­ساز اجرا شده باشد کد موذیانه ی مخرب به صورت پویا بارگذاری خواهد شد.

از آن جا که بارگذاری پویای کد مخرب تنها به ورودی کاربر وابسته نیست بلکه به شرایط محیط اجرای کد نیز وابسته است بنابراین روش های fuzzing ورودی هرگز نمی توانند شروط این بخش از کد را برآورده کنند و کد مخرب را به صورت 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.



شکل17- بمب منطقی مبتنی بر زمان به همراه کد بارگذاری شده ی پویا

### 4-3-3-چهارچوب ابزارCondroid

مطابق شکل1 ابزار Condroid مراحل زیر را برای تحلیل برنامه های اندرویدی طی می کند:

این تحلیل با هدف یافتن مسیر هایی از برنامه که دارای کد حساس هستند انجام می­گیرد.

1. اجرای پویانمادین سازگار

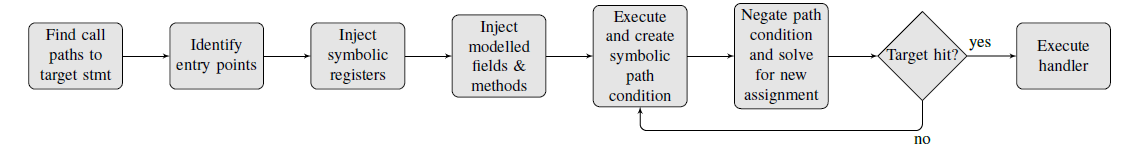
الف) instrument کردن برنامه برای مسیر های اجرایی منتهی به کد حساس تا بتوان در زمان اجرا ثبات ها را بازنویسی کرد.

ب) instrument کردن برنامه با هدف استخراج قید مسیر.

پ) حل کننده ی قیدی که قید مسیر را به عنوان ورودی برای مسیر های مختلف می گیرد و آخرین شرط آن را نقیض می کند؛ آن گاه سعی می کند تا عبارت شرطی جدید حاصل شده را حل نماید.

این حل جدید به این معناست که مسیر جدید به دست آمده نیز امکان پذیر است.

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

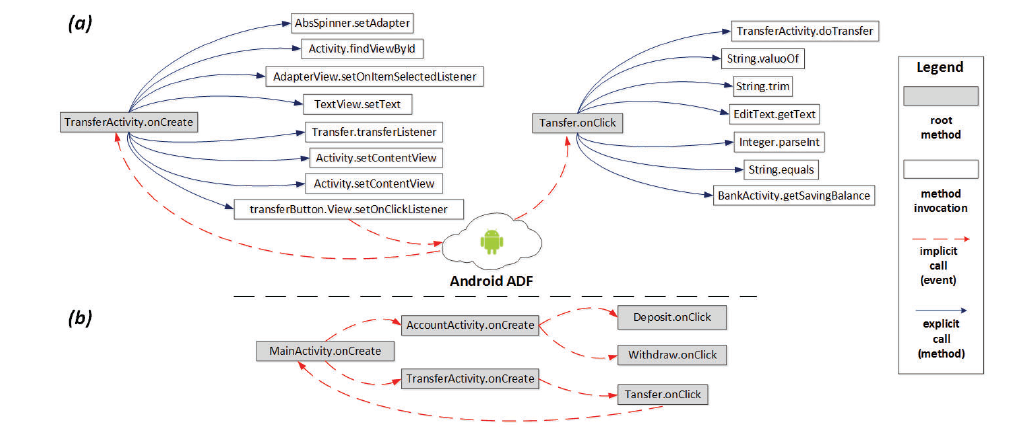


شکل 18-ساختار کلی ابزار Condroid

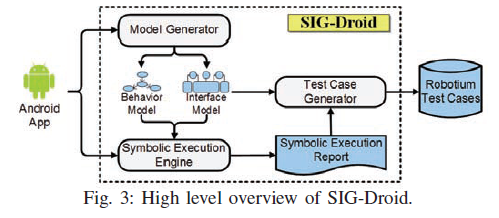
## 4-4-ابزار SIG-Droid

### 4-4-1-مقدمه

همان طور که در شکل 19 دیده می‌شود، SIG-Droid [10]از سه قسمت اصلی تشکیل شده است:

1. Behavior Model: در این قسمت از ابزار موردکاربردهای 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. از آن جا که برنامه های اندرویدی با سیستم عامل دارای تعاملات بسیاری هستند گراف های استخراج شده از آن ها برخلاف گراف های برآمده از برنامه های جاوا، گراف هایی یک پارچه نیستند و از تعدادی زیرگراف تشکیل شده اند.
2.  SIG-Droid با بررسی کد گراف های استخراج شده را به هم وصل کرده و گره های میانی گراف را حذف می کند. نمونه ای از این گراف در شکل 19 آمده است.

شکل 22- گراف حاصل از دو زیرفراخوانی یک نرم افزار اندرویدی بانکی

1. Interface Model: در اندروید در کنار کلاس ها و کد هایی که به زبان جاوا نوشته می شوند، 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.
2. اجرای نمادین: همان طور که در ابتدا گفته شد برنامه های اندروید سه تفاوت عمده با برنامه های عادی دارند که در اجرای نمادین باید به آنها توجه شود.
   * برنامه های اندروید به جای java byte-code به dalvik byte-code تبدیل می شوند. پس برای اینکه بتوان از موتورهای اجرای نمادین مربوط به جاوا استفاده کرد، باید کدهای اندروید را با کامپایلر جاوا کامپایل کرد. اما همان طور 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) شبیه سازی شوند.
   * مشکل دیگر واگرایی مسیر هست. برای حل این موضوع کتابخانه های ساختگی(mock) ایجاد می شوند که تنها یک مقدار دلخواه به عنوان خروجی تولید می کنند. با این کار ابزار درگیر آزمون کتابخانه های خارج از برنامه نمی شود.

شکل 20-ساختار کلی ابزار SIG-droid

* + برای آزمون جنبه های مختلف یک برنامه لازم است تا رشته ای از رخدادها تولید شود. برای این کار با استفاده از Behavior Model تعدادی کلاس Driver برای این کار نوشته می شود. برای تولید رشته های مختلف در این ابزار گراف BM با روش DFS پیمایش می شود.
  + موضوع آخر مشخص کردن ورودی های نمادین برنامه هست. به این منظور با استفاده از Interface Model و بررسی ویجت‌های مختلف ورودی های نمادین تعیین می شوند. به عنوان مثال اگر در برنامه ای یک TextBox وجود داشته باشد، تمام متغیرهایی از کد که مقدار ورودی در این TextBox را در خود ذخیره می کنند به عنوان متغیر نمادین در نظر گرفته می­شود.

1. مولفه تولید موردآزمون: با استفاده از Interface Model و گزارش موتور اجرای نمادین تعدادی مورد آزمون برای اجرا به وسیله Robotium تولید می‌شود.

هدف اصلی SIG-Droid پوشش هر چه بیشتر مسیرهای موجود در برنامه است. برای اندازگیری میزان پوشش کد برنامه ی اندرویدی از ابزار متن‌باز EMMA استفاده می­شود. موتور اجرای نمادین در ابزار Sig-Droid نیز JPF [[67]](#footnote-74) است که با هدف اجرای نمادین برای برنامه های اندروید تغییر یافته است.

## 4-5-ابزار APPINTENT

AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection [11]

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

### 4-5-1-مقدمه

امروزه با گسترش گوشی های اندرویدی و وجود داده های حساس افراد در آن ها،حریم خصوصی کاربران اندروید با تهدید دژافزار های اندروید روبه رو است. با استفاده از روش های شناخته شده در نوشتگان این  
 حوزه­ی علمی می توان انتقال داده از گوشی تلفن همراه به خارج از آن را تشخیص داد. اما نسبت به انتقال داده های حساس درون گوشی تلفن همراه بررسی صورت نگرفته است. این انتقال لزوما نشان دهنده ی نقض حریم خصوصی نمی باشد. در واقع تنها در صورتی نقض حریم خصوصی رخ داده است که این انتقال داده های حساس در درون گوشی همراه و بدون اطلاع و خواست کاربر باشد. اکنون ما با این چالش علمی مواجه هستیم که از کجا بدانیم انتقال داده ها با خواست کاربر انجام گرفته با بدون اطلاع او؟ به عنوان را حل این چالش علمی ابزار APPINTENT ارائه شده است. با استفاده از این ابزار برای هر انتقال داده ای دنباله ای از تغییرات رابط کاربری گرافیکی که نشان دهنده ی دنباله­ای از رویداد ها برای انجام این انتقال است، رسم می شود. این دنباله ی تغییرات در رابط کاربری گرافیکی به تحلیل­گر کمک می کند تا راحت تر تصمیم بگیرد که یک انتقال داده با خواست کاربر صورت گرفته است یا خیر؟

ایده­ ی اصلی ابزار APPINTENT استفاده از اجرای نمادین برای به دست آوردن دنباله ی رویداد هایی است که موجب یک انتقال داده ی مشخص درون گوشی همراه شده اند. اما اجرای نمادین در کنار مزایای قابل توجه ای که در اختیار می گذارد از نظر مصرف حافظه و زمان بسیار ناکارآمد است. نوآوری علمی ابزار APPINTENT ارائه ی بهبودی برای اجرای نمادین با کاهش فضای جست جو در برنامه های اندروید و بدون از دست رفتن پوشش کد بالا است. در پایان برای ارزیابی این پژوهش 750 برنامه ی اندرویدی با APPINTENT مرد تحلیل قرار گرفته اند که نتایج قابل توجه ای برای تشخیص برنامه های ناقض حریم خصوصی به دست آمد.

### 4-5-2-معماری کلی ابزار APPINTENT

ابزار APPINTENT یک ابزار کاملا خودکار برای تشخیص انتقال داده­ی ناخواسته در برنامه های اندروید نیست. درواقع تشخیص کاملا خودکار ناخواسته بودن یک انتقال داده، در عمل ناممکن است. اما این ابزار برای اولین بار در این حوزه ی علمی سعی در ارائه ی یک دنباله از تغییرات رابط گرافیکی کاربر به ازای هر انتقال داده را دارد. این دنباله از تغییرات به تحلیل­گرکمک می­کند تا راحت تر خواسته یا ناخواسته بودن انتقال   
داده­ها را تشخیص دهد.

در این ابزار دستیابی به سه هدف زیر مورد توجه قرار دارد:

1. ایجاد داده های خاص ورودی که منجر به انتقال داد های حساس در برنامه های اندرویدی گردد. داده های ورودی به برنامه های اندرویدی به طور کلی به دو دسته تقسیم می­شوند:
   1. الف) داده های ورودی که متن خاصی را شامل می­شوند و از بیرون از برنامه وارد می گردند.
   2. ب)رویداد های ورودی که ناشی از تعامل کاربر با رابط گرافیکی برنامه ی کاربردی و یا ارتباطات میان پردازه ای سیستم می باشد.
2. پوشش قابل قبولی ازکد تامین گردد. درواقع هدف آن است که تمامی مسیر های برنامه که ممکن است موجب نقض حریم خصوصی شود، پیمایش گردد. ضمنا از آن جا دقت ابزار تحلیل حیاتی است، باید مثبت کاذب و منفی کاذب ابزار نیز جزیی باشد.
3. ابزار باید برای تحلیل گر انسانی قابل فهم و ساده باشد.

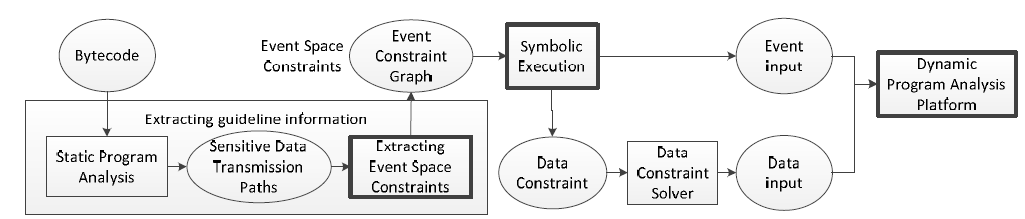
در شکل 1 معماری کلی ابزار APPINTENT را مشاهده می نمایید. در این ابزار یک برنامه ی کاربردی اندروید در دو مرحله مورد تحلیل قرار می گیرد:

1. اجرای نمادین هدایت شده در فضای رویداد[[68]](#footnote-75)

هدف از این مرحله، تولید داده های ورودی برای اجرای برنامه به گونه ای است که مسیر های مختلف انتقال داده­ها پیمایش شوند. در ابزار APPINTENT از تحلیل آلایش ایستا استفاده شده است که با استفاده از آن تمامی انتقال داده های حساس و دنباله ی رویداد های مربوط به آن ها استخراج می شود. در ادامه با اجرای نمادین هدایت شده توسط اطلاعات به دست آمده از تحلیل آلایش ایستا ، ورودی های حساس برای برنامه تولید می شود. پوشش کد کافی نیز بنابر ماهیت ذاتی اجرای نمادین به دست می آید.

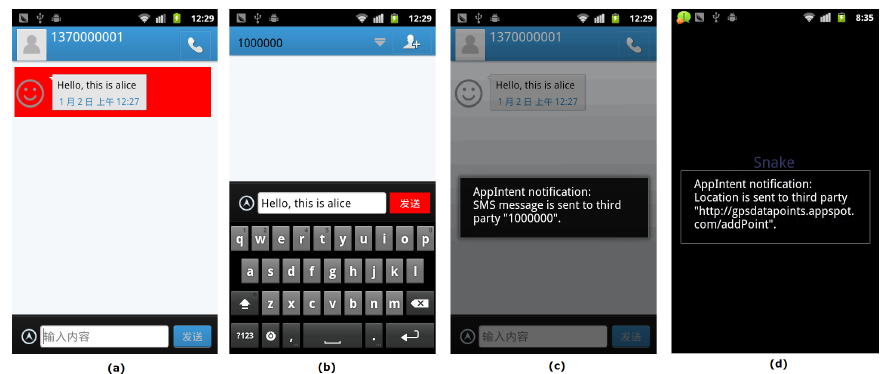
1. پلتفرم تحلیل پویای برنامه

با استفاده از داده های ورودی به دست آمده در مرحله ی 1 در یک پلتفرم تحلیل پویا به اجرای خودکار برنامه ی اندرویدی می پردازیم. مراحل مختلف انتقال داده های حساس به صورت گرافیکی ثبت می شوند تا تحلیل گر راحت تر بتواند در مرد خواسته یا ناخواسته بودن انتقال تصمیم گیری کند.



شکل 21-معماری کلی ابزار APPINTENT

نمونه ای از اجرای ابزار APPINTENT را برای تحلیل یک برنامه ی اندرویدی در شکل 1 مشاهده می نمایید.



شکل 22- اسکرین شات از مراحل مختلف یک انتقال داده

## 4-6-مقایسه ابزارهای اجرای پویانمادین برای تحلیل برنامه های اندرویدی

|  |  |  |  |
| --- | --- | --- | --- |
| ابزار | سال ارائه | هدف | ویژگی ها |
| ACTEV | **2012** | **آزمون  نرم افزار** | چگونه می توان یک رویداد را معتبر تولید کرد؟  چگونه می توان دنباله ی معتبری از رویداد ها را تولید کرد؟  محدود به رخداد ضربه  نیاز به بهینه‌سازی برای کاهش فضای حالت در ترتیب‌های مختلف از رخدادها  انفجار مسیر |
| APP-Intent | **2013** | **تشخیص نقض**  **حریم خصوصی** | انتقال داده بین برنامه های مختلف اندروید آیا با خواست کاربر بوده یاخیر؟  ارائه ی یک دنباله از تغییرات رابط گرافیکی کاربر  ایجاد داده های خاص ورودی که منجر به انتقال داد های حساس در برنامه های اندرویدی گردد |
| Sig-Droid | **2015** | **آزمون  نرم افزار** | استخراج گراف کنترل جریان(CFG) با MoDisco  ساخت mock و stub به صورت خودکار |
| Condroid | **2015** | **تحلیل  پویا هدفمند** | تحلیل ایستای مقدماتی– استخراج گراف کنترل جریان(CFG)  با هدف به دست آوردن entry point برای تست  اجرای پویانمادین برنامه  انفجار مسیر |

جدول 1-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل برنامه های اندرویدی

**فصل پنجم:**

**روش‌های تشخیص دژافزار**

# فصل پنجم: روش‌های تشخیص دژافزار

## 5-1- تحلیل ایستا

در یک محیط تحلیل دژافزار تحلیل ایستا، ساده ترین و ابتدایی ترین 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 دژافزارهای شناخته‌شده است که بخش عمده‌ای از حملات رایانه ای به سازمان‌ها را شامل می شود.

برخی از سامانه‌ها تحلیل ایستا از یادگیری ماشین برای تشخیص استفاده می کنند. 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 جست و جو قرار می­دهد. هرچه دژافزارهای بیشتری، به عنوان نمونه ی آموزشی مورد بررسی قرار گیرد و فهرست شود، توانایی سامانه برای تشخیص حملات در طول زمان رشد می‌یابد.

## 5-2-تحلیل پویا

فایل ،[[69]](#footnote-76)گردچیده زمانی کهژ بتواند تشخیص دهد که محیط اجرا یک محیط واقعی نیست بلکه یک است؛ طور خاصدژا

ه آن خانواده از دژ یکی از ایده ها آن است که برای تشخیص دژافزار ازپویا در کنار هم استفاده شود.

## 5-3-روش های تشخیص دژافزار های اندرویدی

کارهای جدی در حوزه ی تشخیص دژافزار های اندرویدی از سال 2009 با مقالات [12] و [13] شروع شد.  
مقاله ی [12] اولین کار روی اندروید برای تشخیص دژافزار بوده است.این کار تحلیل ایستا را برروی فایل های elf [[70]](#footnote-77) باینری انجام می دهد. در این فایل ها فراخوانی های تابعی[[71]](#footnote-78) و نام فایل های تغییر یافته ذخیره می شوند.صحت تشخیص اعلام شده در این روش تا 96 درصد و نرخ مثبت کاذب آن نیز 10 درصد بوده است. مشکل این پژوهش این بود دژافزار هایی که برروی آن ها تست انجام گرفت، دژافزار هایی بودند که توسط خود نویسندگان مقاله نوشته شده بود به همین علت داده ی کمی برای آزمون روش پیشنهادی شان داشتند. از سویی دیگر آزمون برروی دستگاه های اندرویدی انجام نشده بود.

کار بعدی [13]در همان سال با استفاده از رویکرد تحلیل ایستا با استفاده از قوانین ترکیبی از مجوزها، انجام شده است.در این روش برنامه های ناخواسته بلوکه می شوند.البته هدف اصلی این روش تشخیص آسیب پذیری است.

در سال بعد مقاله ی [14] ارائه شد که روش ترکیبی ایستا و پویا را برای تشخیص ارائه داده بود. در این کار در مرحله ی اول فایل apk را از حالت کامپایل خارج کرده و به کد جاوا تبدیل می کنند. سپس در این کد به دنبال الگوی مخرب می گردند و آن ها را علامت گذاری می کنند. سپس تعداد فراخوانی های سیستمی در زمان اجرا شمرده می شود و از روی این دو ویژگی تشخیص را انجام می دهد. مشکل این مقاله درصد صحت پایین و هم چنین نبود کاربرد واقعی در آزمایش است.

در همان سال 2010 مقاله ی دیگری[15] در زمینه ی تحلیل ایستا ارائه شد که همین مشکل یعنی عدم وجود کاربرد واقعی و شبیه سازی بودن را داشت. در این مقاله ویژگی ها و رخداد های موبایل مانند کارکرد پردازشگر، تعدا بسته های ارسالی از wifi، تعداد پردازه ها، تعداد فشرده شدن کلید یا لمس صفحه نظارت می شوند.سپس از روش یادگیری ماشین استفاده می کنند تا داده ها را به دو دسته ی عادی و خطرناک تقسیم کند.

مقاله ی مهم بعدی در سال 2011 ارائه شد[16]. روش کار این مقاله تحلیل پویا است. به این صورت که با اجرای برنامه، ،Trace های آن ها( برای مثال فراخوانی های سیتمی برای مجموعه ای از کاربران در زمان اجرا) را به دست آورده و. در فایل log ذخیره می کنند. این فایل برای سرور فرستاده می شود. 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[17] 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 می کنند، استفاده می شود.نرخ منفی کاذب این روش به دلیل استفاده از روش اکتشافی بالاست.

در مقاله ی[18] تشخیص دژافزار به صورت خودکار از روی رفتار شبکه انجام می شود.پس از تشخیص اقدام به از بین بردن دژافزار و برگرداندن دستگاه به حالت عادی می کند. اصل عملیات تشخیص برروی شبکه توسط سیستم تشخیص نفوذ انجام می شود به همین علت حافظه ی زیادی از موبایل اشغال نمی گردد. زمانی که مانیتورینگ شبکه ترافیک غیرطبیعی را شناسایی کرد به برنامه اطلاع می دهد. عامل داخل دستگاه فرآیندی را که منجر به تولید آن ترافیک شده استن را یافته و عملیات بازگردانی را انجام می دهد. این کار می تواند شامل فیلتر کردن ترافیک مشکوک، sand box کردن یا حذف کردن برنامه و یا بازیابی تنظیمات کارخانه باشد.

بهترین مقاله ی ارائه شده در این زمینه با رویکرد تحلیل ایستا در سال 2014 اراده شد[19]. در این مقاله ابزاری ارائه شده که با تحلیل و استفاده از ویژگی های ایستا، تشخیص می دهد برنامه دژافزار است یا خیر. این تشخیص در خود موبایل صورت می گیرد. از جمله ویژگی هایی که می توانیم از آن ها برای تحلیل ایستا برنامه استفاده کنیم، می توان به مولفه های سخت افزاری که برنامه از آن ها استفاده می کند، مجوزهای برنامه،مولفه های برنامه،رابط بین مولفه ها[[72]](#footnote-79)، فراخوانی های API و آدرس های IP مورد استفاده توسط برنامه اشاره کرد. در ادامه در این مقاله با استفاده از روش های یادگیری ماشین(SVM خطی) برنامه را به یکی از دوحالت آسیب رسان و عادی تقسیم می کند و تشخیص را انجام می دهد. این پژوهش هم از نظر سرعت و هم از نظر دقت عملکرد بسیار مطلوبی را داشته است. علاوه بر این مجموعه ی ویژگی های مورد استفاده آن نیز به آسانی از هر فایل APK حتی مبهم شده نیز قابل دسترسی است.

مقالات بسیار دیگری در سال های بعد نظیر [20]و[21] مطرح شدند که بر اساس مجوز های برنامه کاوش انجام می دادند. کار [22]که یک sand box برای تحلیل دژافزار ها ارائه دااد و کار [23]که یک چهارچوب برای تشخیص و بهبود گوشی های هوشمند ارائه داده است و بسیار از کار های مشابه دیگر در سال ها ی اخیر مطرح شدند اما به دلیل مشابهت از شرح آن ها صرف نظر می کنیم.

**فصل ششم:**

**کارهای پیشین در اجرای پویانمادین با هدف   
تشخیص یا تحلیل دژافزار**

# فصل ششم:کارهای پیشین اجرای پویانمادین با هدف تشخیص یا تحلیل دژافزار

## 6-1- Automatically Identifying Trigger-based Behavior in Malware

### 6-1-1-مقدمه

برخی از دژافزار ها شامل رفتارهای موذیانه ی پنهانی هستند که تنها در شرایط خاصی این رفتار ها را به اجرا می گذارند. چند نمونه مشهور از این دژافزار ها عبارتند از:

1. کرم واره[[73]](#footnote-80) MyDoom که حملات منع سریس توزیع شده[[74]](#footnote-81) را در زمان های مشخصی صورت می دهد.
2. دژافزار هایkeylogger خاص که تنها گزارشات صفحه­کلید برای وب سایت های معین نظیر وب سایت های بانکی را ثبت می کنند.
3. دژافزار های حملات منع سریس توزیع شده برروی دستگاه های زامبی[[75]](#footnote-82) که فقط با توجه به دستور خاص از جانب کنترل کننده ی بات فعال می شوند.

ما رفتار موذیانه ی مخفی در این دژافزار ها را که فقط در شرایط خاصی فعال می شود رفتار مبتنی بر ماشه[[76]](#footnote-83) می نامیم. این رفتار مبتنی بر ماشه می تواند بر اساس یک ماشه[[77]](#footnote-84) از نوع زمان،رویداد های سیستمی[[78]](#footnote-85)و یا ورودی شبکه باشد. در حال حاضر، تجزیه و تحلیل رفتار مبتنی بر ماشه اغلب به صورت دستی و با روش های قدیمی و مبتنی بر صرف زمان توسط تحلیلگر انسانی انجام می شود؛ از این رو حتی کمک اندک در راستای خودکارسازی و تسریع تحلیل این گونه از دژافزار ها می تواند به میزان قابل توجهی موثر باشد.

در این مقاله[24]، نویسندگان تجزیه و تحلیل خودکار مبتنی بر ماشه را پیشنهاد می دهند. به طور 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 طراحی شده است. رویکرد این پژوهش نشان می دهد که در بسیاری از موارد  
ما می توانیم:

1. وجود رفتار مبتنی بر ماشه را تشخیص دهیم
2. شرایط خاصی را که باعث فعال شدن چنین رفتار پنهانی می شود بیابیم.
3. ورودی هایی را برای برآورده کردن این شرایط ارائه دهیم که با استفاده از آن ها بتوان رفتار موذیانه ی دژافزار را در یک محیط کنترل شده مشاهده کرد.

در این پژهش ابزارMinSweeper پیاده سازی شده است. این ابزار در آزمایشات صورت گرفته نشان داده است که می تواند رفتار مبتنی بر ماشه را در دژافزار های واقعی تشخیص دهد. اگر چه چالش های بسیاری برای تشخیص کاملا خودکار رفتار های مبتنی بر ماشه وجود دارد ولی وجود ابزار MineSweeper نشان می دهد که چنین تجزیه و تحلیل خودکاری امکان پذیر است و درآینده می توان آن را انتظار داشت.

### 6-1-2-تحلیل دستی کد برای تشخیص رفتار مبتنی بر ماشه

نویسندگان مقاله ادعا می کنند که تا پیش از سال 2007 و ارائه این پژهش که منجر به ابزارMinSweeper شد ، پژوهش شاخص دیگری با هدف تشخیص رفتار مبتنی بر ماشه صورت نگرفته است. درواقع در دنیای واقعی تحلیل دستی کد برای تشخیص رفتار مبتنی بر ماشه به صورت زیر بوده است:

1. تحلیل گر انسانی دژافزار را در یک ماشین مجازی اجرا می کرد، در حقیقت ممکن بود با هیچ رفتار موذیانه ای مواجه نشود زیرا احتمال برآورده شدن شرایط ماشه بسیار کم بود.
2. تحلیل گر انسانی سعی می کرد تا کد باینری را به کد اسمبلی برگرداند[[79]](#footnote-86)، تا بتواند با استفاده از کد اسمبلی یک مدل مفهومی نادقیق از اجرای برنامه داشته باشد.
3. تحلیل گرانسانی در این مرحله سعی می کرد تا حدس بزند که تغییر کدام بخش ورودی و یا تنظیم پیکربندی برنامه می تواند منجر به فعال سازی اجرای ماشه گردد.

این فرآیند ادامه می یافت تا زمان درنظر گرفته شده برای تحلیل دژافزار پایان یابد و یا تحلیل گر انسانی ناامید شود و یا در بهترین حالت تحلیل گر موفق شود تا با خوش شانسی رفتار مبتنی بر ماشه را اجرا کند و رفتار موذیانه ی دژافزار را مشاهده نماید.

این پژوهش با تمرکز برروی بات نت ها انجام شده است.درحقیقت سعی شده تا ابزاری برای تحلیل خودکار کد باینری بات نت ها باهدف یافتن رفتار مبتنی بر ماشه توسعه یابد.

### 6-1-3-رویکرد ابزارMinSweeper

در این زیربخش رویکرد کلی و ساختار ابزارMinSweeper برای تحلیل خودکار کد باینری را ارائه می دهیم.   
با درنظر گرفتن یک انتزاع کلی می توان بیان کرد که رفتار های مبتنی بر ماشه با استفاده از پرش های شرطی[[80]](#footnote-87) در کد باینری نمایان می شوند. این پرش های شرطی بر اساس ورودی های ماشه که از نوع زمان ، رویداد های سیستمی، ورودی شبکه یا فشردن یک دکمه ی خاص صفحه کلید هستند اجرا می شوند. درواقع رفتار موذیانه زمانی که یکی از این پرش های شرطی برآورده شود و مسیر اجرایی برنامه به مسیر موذیانه تغییر یابد، اجرا می گردد. ما در این پژوهش شرط هایی را که منجر به اجرای رفتار مبتنی بر ماشه می شوند، شرط ماشه[[81]](#footnote-88) و مقدار ورودی لازم برای برآورده شدن شرط ماشه را ورودی ماشه[[82]](#footnote-89) می نامیم. از آن جا که رفتار مبتنی بر ماشه   
می­تواند در جایگاه های مختلف برنامه قرار بگیرد بنابراین ما نیاز داریم تا برای تحلیل خودکار کد مسیر های متفاوتی را اجرا نماییم.

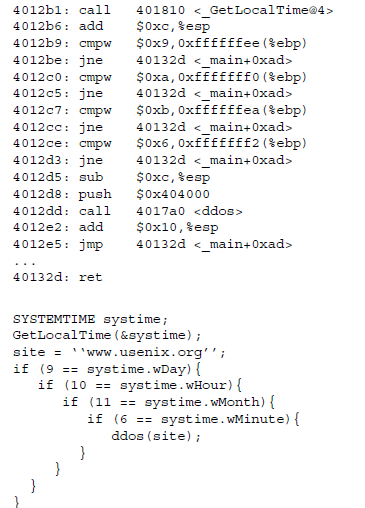
در رویکرد ابزارMinSweeper کد باینری دژ افزار و یک مجموعه از انواع ماشه‌های احتمالی به‌عنوان ورودی به ابزار داده می‌شود. در این رویکرد برای تحلیل خودکار از ترکیب اجرای نمادین و اجرای عددی استفاده می‌شود، این ترکیب درواقع بهبودیافته‌ی اجرای نمادین است که اجرای پویانمادین نام دارد. در اجرای پویا نمادین هر یک از ورودی‌های ماشه به عنوان یک نماد در نظر گرفته می شود و دستورالعمل های باینری[[83]](#footnote-90) که براساس این ورودی ها اجرا می گردد به صورت نمادین[[84]](#footnote-91) اجرا می شود. بنابراین با اجرای یک مسیر از برنامه یک فرمول نمادین برروی 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 می شود تا با حل کردن آن ورودی های متناسب با مسیر های اجرایی جدیدی را ایجاد کند.

تا پیش از این پژوهش، ابزار های 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 با هدف تشخیص دژافزار انجام می دهیم بنابراین فرض دسترسی به کد منبع دژافزار فرضی غیرقابل قبول است.

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[[85]](#footnote-92) یا بسته بندی[[86]](#footnote-93) شده است؛ از این رو استفاده از اجرای پویانمادین مبتنی بر باینری کاملا ضروری است.

### 6-1-4-شرح مساله تشخیص رفتار مبتنی بر ماشه

در شکل6 یک نمونه از کد اسمبلی و کد منبع یک دژافزار مشابه کرم واره Mydoom نمایش داده شده است.



شکل 23- نمونه ی کد اسمبلی و کد منبع یک دژافزار مشابه واره ی Mydoom

در این مثال، حمله ی ممانعت از سرویس توزیع شده تنها زمانی انجام خواهد شد که فراخوانی تابع GetLocalTime مقدار 10:06 9/11 را خروجی دهد. بنابراین در این جا حمله ی ممانعت از سرویس یک رفتار مبتنی بر ماشه تلقی می شود که تنها در یک شرایط خاص زمانی اجرا خواهد شد.

لازم به ذکر است که ما در این مثال صرفا با هدف 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.

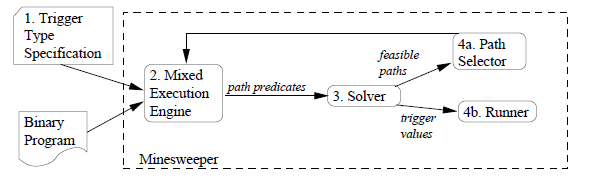
در این مثال نوع ماشه را GetLocalTime فرض می کنیم. بنابراین خروجی تابع systime، همان ورودی ماشه[[87]](#footnote-94) موردنظر می باشد. در واقع رفتار مبتنی بر ماشه تحت مسیراجرایی قرار دارد که با فقط با ورودی زیر   
می توان آن را اجرا کرد:



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 از آن جا که چنین رویکردی درعمل ناکارآمد است ، این پژوهش یک رویکرد مبتنی بر تکرار را با اجرای پویانمادین مورد نظر قرار داده است.

همانطور که در شکل7 مشاهده می شود، ابزارMinSweeper چهار مرحله ی زیر را برای تشخیص خودکار دژافزار طی می کند:

* مرحله 1: 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، رویداد های سیستمی، ورودی های شبکه،هرگونه کتابخانه یا فراخوانی سیستمی باشد.
* مرحله ی2: 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[[88]](#footnote-95) متفاوت ایجاد می­شود. یکی از این مسند ها، مسند مسیری است که به ازای زیرشاخه ی True دستورالعمل شرطی ایجاد می شود. مسند مسیر دوم نیز به ازای زیرشاخه ی False دستورالعمل شرطی ایجاد می گردد.
* مرحله ی 3:در مرحله ی سوم دو مسند مسیر ایجاد شده در مرحله ی قبل به حل کننده ی قید داده می شود تا با حل کردن آن امکان پذیر بودن آن زیرشاخه ی مربوطه را ارزیابی کند. با امکان پذیر بودن هریک از زیرشاخه ها یک مسیر اجرایی جدید از برنامه کشف می شود.
* مرحله ی 4-الف: در این مرحله، جز انتخاب کننده ی مسیر در ابزار یکی از مسیر ها را از مجموعه ی مسیر های امکان پذیر 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 4-ب: جز اجرا کننده[[89]](#footnote-96) در این بخش از مرحله ی نهایی برنامه را به صورت عددی اجرا می کند. این اجرای عددی با استفاده از مقادیر عددی تولید شده در مرحله ی3 توسط حل کننده ی قید انجام می گیرد.



شکل 24- مراحل مختلف ابزارMinSweeper

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

1. موتور اجرای پویانمادین[[90]](#footnote-97)
2. حل کننده ی قید
3. انتخاب کننده ی مسیر
4. 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 می شود تا پس از بررسی امکان پذیر بودن آن، به مجموعه ی مسیر های امکان پذیر 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 برمی­گزیند. 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 موذیانه ی مبتنی بر ماشه ی دژافزار آشکار گردد.

تفاوت جدی این کار با کار های قبل از آن در اجرای پویانمادین برروی کد باینری می باشد. درواقع در تمامی کارهای پیشین که برروی کد منبع صورت گرفته است به طور ایستا کد برنامه بازنویسی می شود تا بتوانند اجرای پویانمادین را اجرا کنند. اما برای آن که بتوان اجرای پویا نمادین را برروی 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.

## 6-2- Exploring Multiple Execution Paths for Malware Analysis

عنوان پژوهش[25]: جست و جوی مسیر های اجرایی مختلف برنامه برای تحلیل دژافزار

### 6-2-1-مقدمه

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

روش های ابتدایی تحلیل دژافزار روش های مبتنی بر تحلیل گر انسانی بودند که از نظر زمان و هزینه بسیار ناکارآمد تلقی می شدند. پس از آن ابزار های تحلیل پویا با هدف 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 روش های 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 موذیانه ی خود را از دید ابزار تحلیل پنهان نماید بسیار بالا بود.

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

### 6-2-2-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 می شود. 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 می شود.

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.

### 6-2-3-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.

## 6-3-مقایسه ی ابزار های اجرای پویانمادین برای تحلیل یا تشخیص دژافزار

در این بخش ابزار های مطرح شده در همین فصل را با یکدیگر مقایسه می نماییم:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ابزار** | **سال ارائه** | **زبان** | **هدف** | **ویژگی ها** |
| **Temporal search [27]**  **(TimeBom detection)** | **2006** | **Binary** | **تشخیص**  **بمب زمانی** | یک سیستم تحلیل مبتنی بر ماشین مجازی  اجرای نمادین محدود برای پیش بینی مسند های زمانی  جریان کنترل دژافزار را بررسی نمی کند.  بخش های زیادی از کار مبتنی بر کمک تحلیل گر انسانی |
| **[26]Bitscope** | **2007** | **Binary** | **تحلیل دژافزار** | یه سیستم تحلیل، پنج جز برای تحلیل دژافزار برروی Bitscope  کنترل جریان(مثلا حلقه)-رفتار دژافزار-ورودی های برنامه-وابستگی ها  اجرای برنامه با مقادیر نمادین  تعریف مقدار بازگشت فراخوانی های سیستمی به صورت نمادین |
| **Mine**  **Sweeper**  **(Botnet Detection)** | **2008** | **Binary** | **تشخیص دژافزار** | ورودی فایل باینری+ مجموعه ی انواع ماشه ی دلخواه  ترجمه ی کد اسمبلی به یک زیان میانی قراردادی (IR)  استفاده از حل کننده ی قید STP |

جدول 2-مقایسه ی ابزار های اجرای پویانمادین برای تشخیص یا تحلیل دژافزار

**فصل هفتم:**

**بحث و نتیجه گیری**

# فصل هفتم: بحث و نتیجه گیری

## 7-1-مقدمه

در این فصل ابتدا بر اساس چالش‌های انفجار مسیر، حل قیدها، مدل سازی حافظه، همروندی و چارچوبه‌های 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 عنوان شده و در نهایت پروژه کارشناسی ارشد بیان خواهد شد.

## 7-2-مقایسه کارهای پیشین و آینده بحث

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

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

**2-حل قیدها**: یکی از نقاط چالش برانگیز 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 از نظر زمانی کارا نیستند و لازم است بهبودهایی در پیاده‌سازی آنها صورت پذیرد

**3-مدل‌سازی حافظه**: نحوه برخورد با حافظه و مدل کردن آن، دیگر چالش این حوزه است. به طور مثال یک متغیر int به شکل یک خانه حافظه در نظر گرفته شود یا اینکه به صورتِ 4 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 به عنوان 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.

**4-همروندی**: برنامه‌های امروزی به صورت توزیع‌شده هستند و معمولا کاربرهای مختلف به صورت همروند و چندنخی اجرا می‌شوند. نحوه آزمون این چنین برنامه‌ها از دیگر چالش‌های این حوزه است

**5-چارچوبه‌های کاری مختلف**: یکی از چالش‌های امروز در مورد آزمون برنامه‌ها، توسعه برنامه برای چارچوبه‌های کاری جدید مثل چارچوب اندروید یا برنامه‌های تحت وب هستند. در هر یک از این چارچوبه‌های کاری چالش‌های جدیدی وجود دارد. مثلا در مقالهACTEV عنوان شد که در این گونه برنامه‌ها علاوه بر داده‌های عادی، رخدادها هم باید به صورت خودکار تولید شوند تا بتوان تمام مسیرهای موجود در برنامه را پوشش داد.

## 7-3-مسائل باز

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

1. بهبود مسئله انفجار مسیر در اجرای پویانمادین با ارائه هیوریستیک کارا

(روش های جستجوی هوشمند، هرس کردن مسیر، کَش کردن پرس‌و‌جوهای قبلی، ترکیب تحلیل‌های ایستا و پویا، استفاده از روش‌های متن‌کاوی[[91]](#footnote-99) برای انتخاب کارای مسیر‌های برنامه)

1. بهبود حل کننده های قید برای ارتقای توان تحلیلی

(به عنوان مثال پشتیبانی کردن از محاسبات ممیز شناور)

1. اجرای پویانمادین بروی پلتفرم اندروید
2. اجرای پویانمادین برای تشخیص دژافزار

## 7-4-پروژه کارشناسی ارشد

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

## 7-5-عنوان پروژه

بهبود تشخيص بمب منطقي در دژافزارهاي اندرويد با اجراي پويا نمادين

## 7-6-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 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[[92]](#footnote-101)و یا تهدیدات پیشرفته مداوم[[93]](#footnote-102) 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 نظیرFlowdroid ,[28]DroidAPIMiner ,[13]Kirin [29] مطرح‌شده‌اند که دقت تشخیص قابل قبولی را برای دژ افزارهای سنتی[[94]](#footnote-103) اندروید ارائه می­کنند اما این ابزارها در تشخیص  
 بمب منطقی ناکارآمد هستند.[30] در سال2016 میلادی ابزارTriggerScope [30]به‌عنوان یک‌قدم اولیه در  
زمینه‌ی 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[[95]](#footnote-104) نظیر بارگذاری پویای کد[[96]](#footnote-105)و کد بومی[[97]](#footnote-106) 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[[98]](#footnote-107) موجود در کد بسنده می‌شود و بررسی کد مخرب[[99]](#footnote-108) 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   
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[[100]](#footnote-109)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 سال2015 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 استفاده شده است. فاز ایستا باهدف یافتن نقطه ورود به برنامه صورت گرفته است پس‌ازآن اجرای پویانمادین در فاز پویا انجام شده است. در مقاله Condroid به‌عنوان ارزیابی،تنها یک مثال ساده از بمب منطقی توسط نویسندگان مقاله مطرح شده 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 می‌باشد. نقطه‌ضعف اصلی آن ابزار محدودیت‌های مربوط به اجرای پویانمادین خالص، مانند انفجار مسیر است که در خود مقاله موردبررسی قرارنگرفته 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 قرار دادیم ازاین‌رو هدف‌گذاری ما استفاده از اجرای پویانمادین برای این کار بوده است. پژوهش فعلی انجام شده 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.

**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.

## 7-7-مراحل اجرای پروژه

مراحل اجرای پروژه در زیر آمده است:

1. مطالعه روش اجرایی پویانمادین و ابزار های موجود در این حوزه
2. مطالعه ساختار و معماری اندروید، برنامه های اندرویدی و دژافزار های اندرویدی
3. مطالعه و ارائه روشی برای پوشش تمام مسیرهای برنامه‌های اندروید
4. توسعه ی یک ابزار آزمون برای برنامه های اندرویدی با پوشش کد مناسب
5. مطالعه ی روش های فعلی تشخیص پویای دژافزار اندرویدی
6. ارائه ی یک طرح جامع برای تشخیص دژافزار اندروید مبتنی بر اجرای پویانمادین برنامه های اندرویدی
7. توسعه ی یک ابزار جامع برای تشخیص دژافزار اندروید مبتنی بر اجرای پویانمادین برنامه­های اندرویدی
8. آزمون روش و ابزار پیشنهادی با استفاده از دژافزار های اندرویدی و بمب منطقی در اندروید
9. نگارش پایان‌نامه

**فصل هشتم:**

**مراجع**

# فصل هشتم: مراجع

1. Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." *Communications of the ACM* 56.2 (2013): 82-90.
2. King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.
3. Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.
4. 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), 2010 IEEE symposium on*. IEEE, 2010.
5. Cha, Sang Kil, et al. "Unleashing mayhem on binary code." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.
6. Stephens, Nick, et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." NDSS. Vol. 16. 2016.
7. Choudhary, Shauvik Roy, Alessandra Gorla, and Alessandro Orso. "Automated test input generation for android: Are we there yet?(e)." *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015.
8. Anand, Saswat, et al. "Automated concolic testing of smartphone apps." *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.
9. Schütte, Julian, Rafael Fedler, and Dennis Titze. "Condroid: Targeted dynamic analysis of android applications." *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*. IEEE, 2015.
10. Mirzaei, Nariman, et al. "Sig-droid: Automated system input generation for android applications." *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015.
11. M Yang, Zhemin, et al. "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.
12. Schmidt, A-D., et al. "Static analysis of executables for collaborative malware detection on android." Communications, 2009. ICC'09. IEEE International Conference on. IEEE, 2009.Arp.
13. Enck, William, Machigar Ongtang, and Patrick McDaniel. "On lightweight mobile phone application certification." Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.
14. Bläsing, Thomas, et al. "An android application sandbox system for suspicious software detection." Malicious and unwanted software (MALWARE), 2010 5th international conference on. IEEE, 2010.
15. Shabtai, Asaf, and Yuval Elovici. "Applying behavioral detection on android-based devices." Mobile Wireless Middleware, Operating Systems, and Applications (2010): 235-249.
16. Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for android." *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011.
17. Zhou, Yajin, et al. "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets." *NDSS*. Vol. 25. No. 4. 2012.
18. Nadji, Yacin, Jonathon Giffin, and Patrick Traynor. "Automated remote repair for mobile malware." *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011.
19. Daniel, et al. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." *NDSS*. 2014.
20. Aswini, A. M., and P. Vinod. "Droid permission miner: Mining prominent permissions for Android malware analysis." *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*. IEEE, 2014.
21. Wu, Dong-Jie, et al. "Droidmat: Android malware detection through manifest and api calls tracing." *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012.CM, 2014.
22. Spreitzenbarth, Michael, et al. "Mobile-sandbox: having a deeper look into android applications." *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013.
23. Roshandel, Roshanak, Payman Arabshahi, and Radha Poovendran. "LIDAR: a layered intrusion detection and remediationframework for smartphones." *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems*. ACM, 2013.
24. Brumley, David, et al. "Automatically identifying trigger-based behavior in malware." *Botnet Detection* (2008): 65-88.
25. Moser, Andreas, Christopher Kruegel, and Engin Kirda. "Exploring multiple execution paths for malware analysis." *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007.
26. Brumley, David, et al. *Bitscope: Automatically dissecting malicious binaries*. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, 2007.
27. Crandall, Jedidiah R., et al. "Temporal search: Detecting hidden malware timebombs with virtual machines." *ACM Sigplan Notices*. Vol. 41. No. 11. ACM, 2006.
28. 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, 2013.
29. Arzt, Steven, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." *Acm Sigplan Notices* 49.6 (2014): 259-269.
30. Fratantonio, Yanick, et al. "Triggerscope: Towards detecting logic bombs in android applications." *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.
31. Wong, Michelle Y., and David Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." *NDSS*. 2016.
32. Rasthofer, Siegfried, et al. "Making malory behave maliciously: Targeted fuzzing of android execution environments." *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017.



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 2017

1. Unit Testing [↑](#footnote-ref-1)
2. Integration Testing [↑](#footnote-ref-2)
3. Quality Assurance Testing [↑](#footnote-ref-3)
4. User Acceptance Testing [↑](#footnote-ref-4)
5. Symbolic Execution [↑](#footnote-ref-5)
6. Symbolic Execution [↑](#footnote-ref-6)
7. Test case [↑](#footnote-ref-7)
8. Code coverage [↑](#footnote-ref-8)
9. Constraint satisfiability [↑](#footnote-ref-9)
10. Scaleability [↑](#footnote-ref-10)
11. Exception [↑](#footnote-ref-11)
12. Memory corruption [↑](#footnote-ref-12)
13. Symbolic state [↑](#footnote-ref-13)
14. Symbolic path constraint [↑](#footnote-ref-14)
15. Constraint solver [↑](#footnote-ref-15)
16. instance [↑](#footnote-ref-16)
17. Concrete [↑](#footnote-ref-17)
18. Concolic [↑](#footnote-ref-18)
19. Interface [↑](#footnote-ref-19)
20. Driver [↑](#footnote-ref-20)
21. Unit testing [↑](#footnote-ref-21)
22. Black box [↑](#footnote-ref-22)
23. Bug [↑](#footnote-ref-23)
24. program crashe [↑](#footnote-ref-24)
25. assertion violation [↑](#footnote-ref-25)
26. Fuzzing [↑](#footnote-ref-26)
27. Code coverage [↑](#footnote-ref-27)
28. Path Constraint [↑](#footnote-ref-28)
29. Soundness [↑](#footnote-ref-29)
30. Completness [↑](#footnote-ref-30)
31. f(x,y) [↑](#footnote-ref-31)
32. Concrete Executor Client [↑](#footnote-ref-32)
33. Symbolic Execution Server [↑](#footnote-ref-33)
34. Exploit [↑](#footnote-ref-35)
35. Taint Analysis [↑](#footnote-ref-38)
36. Path Formula [↑](#footnote-ref-39)
37. Exploitability Formula [↑](#footnote-ref-40)
38. instruction pointer: اشاره گری که نشنان دهنده ی خط بعدی برنامه برای اجرا است [↑](#footnote-ref-41)
39. Checkpoint Restoration [↑](#footnote-ref-43)
40. State [↑](#footnote-ref-44)
41. Preconditioned Symbolic Execution [↑](#footnote-ref-45)
42. symbolic instruction pointer [↑](#footnote-ref-46)
43. Refinement Cache [↑](#footnote-ref-47)
44. canonical representation [↑](#footnote-ref-48)
45. Index Search Trees [↑](#footnote-ref-49)
46. Indexed Search Tree [↑](#footnote-ref-50)
47. Symbolic region [↑](#footnote-ref-51)
48. selective concolic [↑](#footnote-ref-52)
49. Software Development Kit [↑](#footnote-ref-56)
50. Inter-process communication [↑](#footnote-ref-57)
51. GUI [↑](#footnote-ref-58)
52. Event [↑](#footnote-ref-59)
53. Tap event [↑](#footnote-ref-60)
54. Event-Driven Programs [↑](#footnote-ref-61)
55. Tap Event [↑](#footnote-ref-62)
56. Wiget [↑](#footnote-ref-63)
57. TextView [↑](#footnote-ref-64)
58. Manual [↑](#footnote-ref-65)
59. State Transition Diagram [↑](#footnote-ref-66)
60. Read Only [↑](#footnote-ref-67)
61. Action [↑](#footnote-ref-68)
62. Incoming Call [↑](#footnote-ref-69)
63. dynamically loaded [↑](#footnote-ref-70)
64. static call path analysis [↑](#footnote-ref-71)
65. تا سال 2015 میلادی این امکان وجود داشت زیرا ابزار Google Bouncer که مسئول ارزیابی امنیتی برنامه های موجود درGoogle play می باشد صرفا از تحلیل ایستا بهره می برد. ممکن است نسخه های جدید Google Bouncer بهبود یافته باشند که نیازمند بررسی بیش تری است. [↑](#footnote-ref-72)
66. Goldfish kernel [↑](#footnote-ref-73)
67. Java Path Finder [↑](#footnote-ref-74)
68. Event-space Constraint Guided Symbolic Execution [↑](#footnote-ref-75)
69. Sand­box [↑](#footnote-ref-76)
70. Executable and Linkable Format [↑](#footnote-ref-77)
71. Function Call [↑](#footnote-ref-78)
72. Intent [↑](#footnote-ref-79)
73. Worm [↑](#footnote-ref-80)
74. Distributed Denial of Service(DDoS) [↑](#footnote-ref-81)
75. Zombie [↑](#footnote-ref-82)
76. Trigger-based behavior [↑](#footnote-ref-83)
77. Trigger [↑](#footnote-ref-84)
78. System events [↑](#footnote-ref-85)
79. Disassembeling [↑](#footnote-ref-86)
80. Conditional jump [↑](#footnote-ref-87)
81. Trigger condition [↑](#footnote-ref-88)
82. Trige [↑](#footnote-ref-89)
83. Binary instructions [↑](#footnote-ref-90)
84. Symbolically [↑](#footnote-ref-91)
85. Obfuscated [↑](#footnote-ref-92)
86. Packed [↑](#footnote-ref-93)
87. Trigger input [↑](#footnote-ref-94)
88. برای هر مسیر اجرایی از یک برنامه مجموعه ای از شروط به دست می آید که نشان دهنده ی آن مسیر مشخص از برنامه می باشد، به این مجموعه از شروط مسند مسیر یا Path predicateگویند. [↑](#footnote-ref-95)
89. Runner [↑](#footnote-ref-96)
90. اجرای پویانمادین در این نوشتگان این مقاله با واژه ی Mixed execution ذکر شده است. [↑](#footnote-ref-97)
91. Text Mining [↑](#footnote-ref-99)
92. Specific targeted attacks [↑](#footnote-ref-101)
93. Advanced Persistent Threat [↑](#footnote-ref-102)
94. Traditional malware [↑](#footnote-ref-103)
95. obfuscation [↑](#footnote-ref-104)
96. Dynamic code loading [↑](#footnote-ref-105)
97. Native code [↑](#footnote-ref-106)
98. Trigger [↑](#footnote-ref-107)
99. Payload [↑](#footnote-ref-108)
100. emulator [↑](#footnote-ref-109)