دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر و فن‌آوری اطلاعات

گزارش سمینار کارشناسی ارشد
گرایش امنیت اطلاعات

عنوان

تشخیص آسیب پذیری‌های نرم‌افزاری با اجرای Concolic

نگارش
احسان عدالت
۹۴۱۳۱۰۹۰

استاد راهنما
دکتر بابک صادقیان

تابستان ۱۳۹۵

چکیده

در چرخه تولید نرم‌افزار، یکی از مهم‌ترین فرآیندها، آزمون نرم‌افزار است. آزمون می‌تواند به صورت دستی یا خودکار صورت پذیرد. هر کدام از این روش‌ها مزیت‌ها و عیب‌های خاص خود را دارد. در آزمون دستی نرم‌افزار می‌توان انواع خطاها و آسیب‌پذیری‌ها از جمله آسیب‌پذیری‌های منطقی را یافت. ولی با این حال این فرآیند زمان‌بر و گران است. علاوه بر آزمون‌گر، نیاز است تا متخصص امنیت هم به منظور کشف آسیب‌پذیری‌ها حضور داشته باشد. همچنین ممکن است که نیروی انسانی نتواند تمام مسیرهای برنامه را بیابد و بررسی کند.

از آن طرف در آزمون خودکار نرم‌افزارها، می‌توان در زمانی کم، هزاران خط از یک برنامه را آزمود و تمام مسیرهای برنامه و حالت‌های ممکن برای پردازش روی داده‌های ورودی را بررسی کرد. با این حال این روش دارای محدودیت‌هایی است و توانایی یافتن تمام آسیب‌پذیری‌ها از جمله آسیب‌پذیری‌های منطقی را ندارد.

سازمان‌ها با توجه به این که از نظر زمان و هزینه محدودیت دارند، علاقه‌مندند تا فرآیند کشف آسیب‌پذیری‌ها را خودکار کنند. از این رو از گذشته تا کنون روش‌های مختلفی برای این کار ارائه شده است. تحلیل ایستا، تحلیل پویا، آزمون نفوذ و روش فاز از جمله راه‌کارهای این حوزه است.

در این گزارش تاکید بر روی تشخیص آسیب‌پذیری‌ها به کمک روش اجرای Concolic است که به صورت پویا برنامه را تحلیل می‌کند. برای آشنایی با این روش و یافتن راه حل‌های ارائه شده، ابتدا این روش و مفاهیم مرتبط با آن یعنی اجرای نمادین بیان شده است. همچنین سایر روش‌های آزمون نرم‌افزار هم به صورت اجمالی معرفی شده است. چون تاکید بر روی تشخیص آسیب‌پذیری نرم‌افزاری است، تعدادی از آسیب‌پذیری‌ها از جمله آسیب‌پذیری سرریز بافر و تزریق نیز معرفی می‌شود. در بحثی مجزا، کارهای پیشین و ابزارهای ارائه شده در حوزه اجرای Concolic نیز معرفی شده‌اند تا ایده‌ها و نوآوری‌های هر یک روشن شود و بتوان مسائل باز موجود در این حوزه را ارائه کرد. در نهایت پروژه کارشناسی ارشد تعریف شده در این حوزه به طور مختصر معرفی می‌شود.

واژه‌های کلیدی:

آزمون نرم‌افزار، اجرای نمادین، اجرای Concolic، تحلیل آلایش، کشف آسیب‌پذیری نرم‌افزاری، آسیب‌پذیری تزریق، آسیب‌پذیری سرریز بافر.

فهرست مطالب

فهرست شکل‌ها

فهرست جدول‌ها

۱ مقدمه

دنیای امروز دنیای کامپیوتر و ابزارهایی است که از محاسبات کامپیوتری استفاده می‌کنند. تعداد زیاد این گونه ابزارها از PCها، گوشی‌های هوشمند همراه و تبلت‌ها گرفته تا کامپیوترهای جاسازی شده[۱] در وسایل خانه مثل تلویزیون، یخچال و غیره نشان‌دهنده گسترش عمیق آنهاست. علاوه بر مصارف خانگی، در صنایع نظامی، هوافضا، سدها و نیروگاه‌ها هم به طور گسترده از کامپیوترها و نرم‌افزارها استفاده می‌شود.

درستی اجرای نرم‌افزارها و همچنین امن بودن آنها به معنای نبود راه نفوذ به آنها، از جمله مسائل مهم در صنعت و البته محیط دانشگاهی بوده است. در این فصل ابتدا در قسمت ۱-۱ پیشینه موضوع آزمون نرم‌افزارها بیان می‌شود. سپس در قسمت ۱-۲ روش‌های مختلف آزمون نرم‌افزار مورد بررسی قرار می‌گیرد. در پایان هم ساختار گزارش و موضوع فصل‌های مختلف گفته خواهند شد.

۱-۱ پیشینه موضوع

در مقاله [۲۱] روند پیشرفت آزمون نرم‌افزار آمده است. این مقاله، ۵ دوره مختلف برای این موضوع در نظر گرفته است که در جدول ۱ آمده است. در اینجا به طور مختصر دوره‌های مختلف بیان می‌شوند.

دوره	تا	از سال
مبتنی بر اشکال‌زدایی[۲]	۱۹۵۶	-
مبتنی بر نشان دادن[۳]	۱۹۷۸	۱۹۵۷
مبتنی بر خرابی[۴]	۱۹۸۲	۱۹۷۹
مبتنی بر تکامل[۵]	۱۹۸۷	۱۹۸۳
مبتنی بر ممانعت[۶]	-	۱۹۸۸

جدول ۱: دوره‌های مختلف در آزمون نرم‌افزار

[۱] Embedded Systems
[۲] Debugging Oriented
[۳] Demonstration Oriented
[۴] Destruction Oriented
[۵] Evaluation Oriented
[۶] Prevention Oriented

۱-۱-۱ دوره مبتنی بر اشکال‌زدایی

در سال‌های ابتدایی گسترش سیستم‌های کامپیوتری، مفهوم آزمون نرم‌افزار بیشتر مربوط به سخت‌افزار سیستم می‌شد. در آن زمان مفاهیم «آزمون»، «بررسی[۷]» و «اشکال‌زدایی» یکی در نظر گرفته می‌شد. اولین مقاله‌هایی که در مورد آزمون نرم‌افزار ارائه شد مربوط به آلن تورینگ در سال ۱۹۴۹ و ۱۹۵۰ بود. در مقاله اول در مورد اثبات درستی صحبت شده بود. در مقاله دوم در رابطه با این که کد، نیازمندی‌های نرم‌افزار را در خود دارد، صحبت شده بود.

۱-۱-۲ دوره مبتنی بر نشان دادن

در سال ۱۹۵۷ چارز بِیکر مفهوم اشکال‌زدایی را از آزمون نرم‌افزار جدا کرد. او بیان کرد که منظور از اشکال‌زدایی «اطمینان از اجرای برنامه» و منظور از آزمون «اطمینان از حل مسئله» است. هر دو مفهوم شامل تلاش‌هایی برای کشف[۸]، محل‌یابی[۹]، شناسایی[۱۰] و درست کردن[۱۱] خرابی در برنامه، البته با دو هدف مختلف اجرا و یا حل مسئله است.

۱-۱-۳ دوره مبتنی بر خرابی

در سال ۱۹۷۹، مایِر آزمون را «فرایند اجرای برنامه به منظور کشف خطا» تعریف کرد. نگرانی مایر نشان دادن این موضوع بود که برنامه خطایی ندارد. تغییر تاکید از نشان دادن درستی به کشف خطا، به طور طبیعی آزمون نرم‌افزار را همراه با روش‌ها و ابزارهای کشف خطا کرد. مقالات زیادی نیز در رابطه با کشف خطا منتشر شده است.

۱-۱-۴ دوره مبتنی بر تکامل

در سال ۱۹۸۳ متدولوژی جدید برای آزمون نرم‌افزار ارائه شد. در این متدولوژی مجموعه‌ای از راه‌کارها بیان شد که آزمون نرم‌افزار را همراه با بازبینی[۱۲] و تحلیل آن می‌کرد. در چرخه حیات نرم‌افزار مجموعه‌ای از

---

[۷] Check out
[۸] Detect
[۹] Locate
[۱۰] Identify
[۱۱] Correct
[۱۲] Review

فعالیت‌ها اضافه می‌شد که در طول تولید و تکامل نرم‌افزار، مسئله درستی و عدم وجود خطا هم در آن بررسی می‌شد.

۱-۱-۵ دوره مبتنی بر ممانعت

در سال ۱۹۸۵ متدولوژی کامل به نام «آزمون قاعده‌مند و فرایند تکامل[۱۳]» ارائه شد. در این متدولوژی به صورت موازی با چرخه تکامل نرم‌افزار، مجموعه فعالیت‌هایی دیده می‌شود که شامل برنامه‌ریزی، تحلیل، طراحی، پیاده‌سازی، اجرا و نگهداری است.

- در مورد روش‌های جدید آزمون نرم‌افزار در بخش بعدی صحبت خواهد شد.

۱-۲ روش‌های آزمون نرم‌افزار

در این قسمت، راه‌کارهای آزمون نرم‌افزارها بررسی می‌شوند. مطالبی که در این قسمت آورده می‌شوند از فصل هشتم کتاب [۲۲] است. یکی از مسائل مهمی که باید در مورد آزمون نرم‌افزار در نظر گرفت این است که هزینه‌های آزمون نرم‌افزار قبل از انتشار، به مراتب از هزینه‌های آزمون بعد از انتشار، کمتر است. پس بهتر است در متدولوژی توسعه نرم‌افزار، مسئله آزمون آن نیز در نظر گرفته شود. برای آزمون نرم‌افزار مراحل مختلف زیر در نظر گرفته می‌شوند:

- آزمون واحد[۱۴]
- آزمون تجمیع[۱۵]
- آزمون اطمینان از کیفیت[۱۶]
- آزمون تایید کاربر[۱۷]

آزمون واحد یکی مهمترین قسمت‌های آزمون نرم‌افزار است. از آنجایی که برنامه‌نویس بهتر از هر کس دیگری با کدی که نوشته است آشنایی دارد، بهترین فرد برای آزمون واحد خواهد بود. آزمون واحد می‌تواند به صورت دستی[۱۸] یا خودکار صورت پذیرد.

[۱۳] Systematic Test and Evaluation Process (STEP)
[۱۴] Unit Testing
[۱۵] Integration Testing
[۱۶] Quality Assurance Testing
[۱۷] User Acceptance Testing

بررسی دستی کد تنها به منظور کشف خطا در کد اتفاق نمی‌افتد. بلکه در این فرایند کدهای اضافی، کدهای پیچیده و بدون استفاده هم تصحیح می‌شوند. بررسی دستی کدها معمولا فرایندی هزینه‌بر خواهد بود. چون علاوه بر برنامه‌نویس معمولا متخصص امنیت نیز باید در این بررسی همکاری کند. این روش معمولا برای شناسایی آسیب‌پذیری‌های منطقی بهتر است چون روش‌های خودکار معمولا قادر به تشخیص اینگونه خطاها نیستند.

در بررسی نرم‌افزارها اگر کد منبع، مستندسازی‌ها و مدل تهدید در اختیار باشد، به آن آزمون جعبه سفید[۱۹] می‌گویند. آزمون نرم‌افزار از بیرون و بدون دسترسی به مستندسازی و مشخصات آن، آزمون جعبه سیاه[۲۰] گویند و آزمون جعبه خاکستری [۲۱] چیزی میان دوتای قبلی است.

در سازمان‌ها به دلیل اهمیت زمان و در عین حال بالا بودن کیفیت محصول نهایی، لازم است که بدون حذف کردن آزمون دستی، آزمون خودکار را هم انجام داد. چون می‌توان هزاران خط از برنامه را در زمان و هزینه کم بررسی کرد. البته این نوع ابزارها مشکلاتی هم دارند. از جمله وجود مثبت کاذب[۲۲] که باعث می‌شود زمان زیادی از سازمان صرف بررسی آنها گردد. همچنین این ابزارها در کشف بعضی از آسیب‌پذیری‌ها مثل آسیب‌پذیری‌های منطقی ناتوانند.

تعدادی از این ابزارها که برای چندین زبان برنامه‌نویسی هم قابل استفاده‌اند عبارتند از:

- O2
- RATS
- YASCA

این ابزارها به منظور آزمون جعبه سفید مورد استفاده قرار می‌گیرند و بهتر است در زمان توسعه نرم‌افزار مورد استفاده قرار گیرند. بعد از پیاده‌سازی نرم‌افزار می‌توان از ابزارهای مربوط به آزمون جعبه سیاه استفاده کرد یا اگر در نرم‌افزار از کتابخانه‌ای استفاده شده است که کد آن در اختیار نیست، می‌توان از این آزمون استفاده کرد تا از درستی برنامه اطمینان حاصل شود. تعدادی از ابزارهایی که از آزمون جعبه سیاه استفاده می‌کنند، در زیر آمده‌اند:

---

[۱۸] Manual
[۱۹] White Box Testing
[۲۰] Black Box Testing
[۲۱] Gray Box Texting
[۲۲] False positive

- Cenzic Hailstorm
- HP WebInspect
- IBM AppScan

اگر آزمون جعبه سیاه و سفید به صورت ترکیبی استفاده شوند، آزمون جعبه خاکستری خواهیم داشت. در مورد نرم‌افزارهای مختلف، متخصصان یک بار از دید توسعه‌دهندگان و از داخل به بیرون کدها و برنامه را می‌آزمایند. سپس یک بار هم از دید مهاجم و از بیرون به داخل لایه به لایه کدها و برنامه را بررسی می‌کنند.

در این قسمت روش‌های جدید آزمون نرم‌افزار و مراحل و انواع آن عنوان شد. همچنین اهمیت استفاده از ابزارهای آزمون‌های مختلف و خودکارسازی آزمون نیز توضیح داده شد. در فصل ۲، مفاهیم و روش های پایه برای بیان یکی از روش‌های آزمون نرم‌افزارها به صورت خودکار یعنی اجرای نمادین[۲۳]، مورد بررسی قرار می‌گیرد.

۱-۳ ساختار گزارش

در فصل ۲، مفاهیم و روش های پایه برای آشنایی با موضوع آزمون از طریق اجرای نمادین، گفته خواهد شد. همچنین بیان می‌شود که دو روش نوین در اجرای نمادین وجود دارند: ۱) روش EGT ۲) روش اجرای Concolic.

در فصل ۳، روش‌های تشخیص آسیب‌پذیری‌های نرم‌افزاری بیان می‌شوند. روش‌های تحلیل ایستا و پویا، آزمون نفوذ و روش فاز نیز در این مورد معرفی خواهند شد. در پایان فصل هم مزیت‌های تحلیل فایل دودویی، نسبت به کد منبع بررسی خواهند شد.

در فصل ۴، مطالعه بر روی آسیب‌پذیری‌ها صورت می‌گیرد. در این فصل انواع آسیب‌پذیری‌ها، کدهای نمونه، چگونگی آگاهی از آلودگی و راه‌های مقابله با آنها بیان می‌شوند. آسیب‌پذیری‌هایی که بررسی می‌شوند عبارتند از: سرریز بافر یعنی سرریز پشته، هیپ، قالب رشته، اعدا int و همچنین آسیب‌پذیری تزریق مثل تزریق SQL یا دستورات سیستم عامل و یا تزریق به webView.

در فصل ۵، کارهای پیشین و ابزارهای موجود بررسی می‌شوند. ابزارهای بیان شده عبارتند از: DART، CUTE، EXE، KLEE، jCUTE، jFUZZ، LCT، Jalangi. همچنین در این فصل راه‌کارهای بهبود روش

[۲۳] Symbolic Execution

اجرای Concolic گفته خواهد شد. همچنین تحلیل آلایش و ابزار Avalanche عنوان می‌شود که برای کشف آسیب‌پذیری مفید خواهد بود. علاوه بر آن نمونه‌کار اجرای روش Concolic برای برنامه‌های گوشی‌های هوشمند همراه نیز بررسی خواهد شد.

در پایان در فصل ۶، به بحث و نتیجه‌گیری 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 مراجع استفاده شده معرفی می‌شوند.

۲ مفاهیم و روش های پایه

در فصل گذشته اهمیت آزمون نرم‌افزار و روش‌های آزمون بررسی شدند. گفته شد که یکی از مهمترین موضوع‌ها در این حوزه، آزمون خودکار برنامه‌هاست که روش‌های مختلفی برای آن وجود دارد. در این فصل یکی از روش‌های خودکار یعنی اجرای نمادین و زیر مجموعه‌های آن مورد بررسی قرار می‌گیرند.

۲-۱ روش اجرای نمادین

در این قسمت اجرای نمادین معرفی می‌شود که برای یافتن خطاهای موجود در برنامه و یا تولید مورد آزمون[۲۴] استفاده می‌شود. توضیحات مربوط به اجرای نمادین از مقاله [۱] بیان می‌شود. با توجه به دسته‌بندی موجود در این مقاله ابتدا روش سنتی و اصلی اجرای نمادین بحث و بررسی می‌شود. بعد از آن دو روش نوین در این حوزه بیان خواهند شد که اجرای Concolic و EGT[۲۵] هستند.

قبل از شروع بحث لازم است دو مفهوم بیان شوند. در حوزه آزمون نرم‌افزارها، ورودی‌های نرم‌افزار بر دو نوع هستند. ورودی‌های Concrete که در این نوشته به عنوان ورودی‌های عددی مطرح می‌شوند و دیگری ورودی‌های Symbolic که به عنوان مقدار نمادین بیان خواهد شد. منظور از مقدار عددی، ورودی از برنامه‌ها هستند که مقدار می‌گیرند و می‌توان بر روی آنها عملیات، مثل عملیات حسابی انجام داد. برای مثال در تابع زیر اگر برای دو متغیر ورودی، مقادیر عددی $x=1$ و $y=2$ اختصاص دهیم در پایان مقدار بازگشتی تابع ۳ خواهد بود. ولی در مقدار نمادین، متغیرهای ورودی به شکل نمادین مقداردهی می‌شوند مثلا $x=x_0$ و $y=2$. در این حالت بعد از اجرای تابع مقدار خروجی x_0+2 خواهد بود.

```
int addFunction(x, y){
   return x+y;
}
```

۲-۱-۱ روش سنتی اجرای نمادین

هدف از اجرای نمادین در آزمون نرم‌افزارها این است که تا حد امکان مسیرهای مختلفی از برنامه در زمان معقول جست‌وجو شوند تا برای هر مسیر دو هدف زیر برآورده شوند:

[۲۴] Test Case

[۲۵] Execution-Generated Testing

- مقدار عددی استخراج شود که به وسیله آن بتوان، آن مسیر از برنامه را طی کرد.
- بررسی وجود انواع خطاها در آن مسیر، مثل آسیب‌پذیری نرم‌افزاری، خرابی حافظه یا استثنا[۲۶]های بررسی نشده.

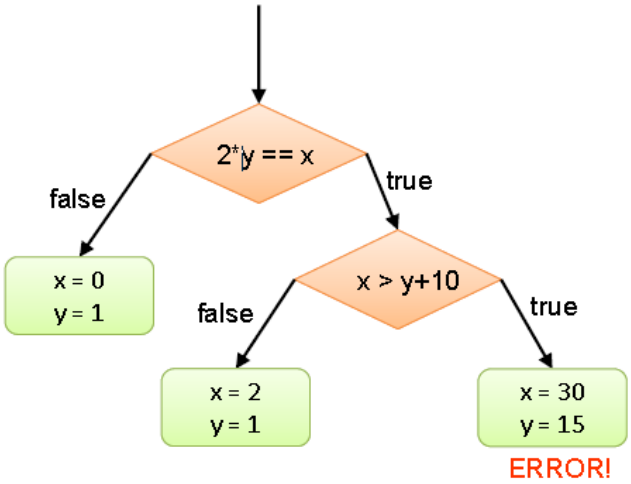
ایده اصلی در اجرای نمادین استفاده از مقادیر نمادین به جای مقادیر عددی در اجرای برنامه است. در این اجرا متغیرهای برنامه به شکل عبارت نمادین[۲۷] نشان داده می‌شوند. در نتیجه خروجی برنامه به شکل تابعی از مقادیر ورودی محاسبه می‌شود. در آزمون نرم‌افزارها از اجرای نمادین برای تولید موردآزمون برای پیمایش مسیرهای اجرایی[۲۸] مختلفی از برنامه استفاده می‌شود. منظور از مسیر اجرایی ترتیبی از مقادیر True و False است. هر برنامه شامل تعدادی بیان[۲۹] و تعدادی بیان شرطی[۳۰] است. مقدار True(False) در i-امین محل از ترتیب یعنی؛ اجرا در برخورد با i-امین بیان شرطی شاخه then (else) را انتخاب کرده است. تمام مسیرهای اجرای برنامه را می‌توان توسط درخت اجرا[۳۱] نمایش داد. شکل ۱ قطعه کدی ساده آمده است. در شکل ۲ هم

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

شکل ۱: قطعه کدی ساده برای بیان اجرای نمادین

[۲۶] Exception
[۲۷] Symbolic Expression
[۲۸] Execution Path
[۲۹] Statement
[۳۰] Conditional Statement
[۳۱] Execution Tree

درخت اجرای کد موجود در شکل ۱ ارائه شده است.



شکل ۲: درخت اجرا برای کد شکل ۱

در اجرای نمادین، یک حالت نمادین[۳۲] σ وجود دارد که نگاشت متغیرها به عبارت نمادین در آن نگهداری می‌شود. همچنین در شرط مسیر[۳۳] مجموعه قیدهای نمادین مسیر[۳۴] نگهداری می‌شود. هنگام شروع، مجموعه σ تهی و مقدار شرط مسیر برابر True است. در حین اجرا این دو مقدار به‌روز می‌شوند. در نهایت مقدار شرط مسیر به یک حل‌کننده قید[۳۵] داده می‌شود تا مقدار عددی که در شرط مسیر صدق می‌کند را تولید کند. اگر به برنامه این مقادیر عددی داده شود، دقیقا همان مسیری طی می‌شود که شرط مسیر در آن ایجاد شده است.

برای مثال برای قطعه کد شکل ۱ در خط ۱۶ عبارت $x \mapsto x_0$ به σ اضافه می‌شود. در خط ۶ عبارت $z \mapsto 2y_0$ به σ اضافه می‌شود. در خط ۷ که یک بیان شرطی وجود دارد، شرط مسیر به $pc = pc \wedge (2y_0 = x_0)$ برای شاخه then و برای شاخه else عبارت $pc = pc \wedge (2y_0 \neq x_0)$ به‌روز می‌شود. در نهایت اگر اجرا به یک خطا یا دستور exit() ختم شود، شرط مسیر به یک حل‌کننده قید داده می‌شود تا مقدار عددی تولید شود که شرط مسیر را ارضا کند. مثلا در شکل ۱ با مقادیر {x=0, y=1} و {x=2, y=1} و {x=30, y=15} تمام مسیرها پیمایش می‌شوند.

---

[۳۲] Symbolic State
[۳۳] Path Condition
[۳۴] Symbolic Path Constraint
[۳۵] Constraint Solver

در برنامه‌هایی که حلقه وجود دارند یا توابع بازگشتی وجود دارند ممکن است که اجرای نمادین در حلقه بی‌نهایت بیفتد. برای مثال در شکل ۳ نمونه کدی آورده شده است که می‌تواند بی‌نهایت مسیر اجرایی داشته باشد. چون که مقدار N که پایان دهنده حلقه هست، هر بار به عنوان ورودی از کاربر گرفته می‌شود.

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

شکل ۳: نمونه کد برای نشان دادن بینهایت مسیر اجرایی

در اجرای نمادین اگر در برنامه تابع بازگشتی وجود داشته باشد یا اینکه تابعی از یک کتابخانه خارجی فراخوانی شود یا اینکه عبارتی که به عنوان قید به حل‌کننده قید داده می‌شود، خطی نباشد (با فرض اینکه حل‌کننده قید توانایی حل توابع غیرخطی را نداشته باشد)، حل‌کننده قید نمی‌تواند شرط مسیر را حل کند و اجرای نمادین با شکست مواجه می‌شود. برای مقابله با این موضوع روش‌های نوین Concolic و EGT ارائه شده‌اند که به اختصار در ادامه بررسی می‌شوند. بررسی کامل این روش‌ها در فصل‌های آینده با بیان ابزارهای پیاده‌سازی شده برای هر یک و ویژگی‌های آنها و روند تکاملشان، خواهند آمد.

۲-۲ روش‌های اجرای نمادین نوین

در قسمت قبل محدودیت‌های اجرای نمادین سنتی به اختصار بیان شده‌اند. برای مقابله با این محدودیت‌ها روش‌های جدید ارائه شدند که ایده اصلی آنها ترکیب اجرای عددی با نمادین است.

۲-۲-۱اجرای Concolic

واژه Concolic از ترکیب دو واژه Concrete و Symbolic به وجود آمده است. در اجرای Concolic 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 این اجرا به صورت همزمان برنامه به صورت عددی و نمادین اجرا می‌شود. برای اجرای عددی لازم است تا ورودی‌های برنامه مقداردهی اولیه شوند. این مقادیر می‌تواند به صورت دلخواه[۳۶] انتخاب شود. در حین اجرای بیان‌های شرطی، قیدهای نمادین نگهداری می‌شوند. در نهایت این قیدها به حل‌کننده قید داده می‌شوند تا ورودی عددی برای اجرای بعدی فراهم شوند. این اجراها تا زمانی ادامه پیدا می‌کنند که همه مسیرها پیمایش شوند یا اینکه زمان مقرر به پایان برسد.

[۳۶] Random

برای مثال در کد شکل ۱ مقدار دلخواه {x=22, y=7} انتخاب می‌شود و برنامه هم به صورت نمادین و هم عددی اجرا می‌شود. این اجرا در خط ۷ برنامه شاخه else را انتخاب می‌کند. اجرای نمادین، قید $(2y_0 \neq x_0)$ را ایجاد می‌کند. در حل‌کننده قید، عبارت نهایی مکمل می‌شود و شرط جدید $(2y_0 = x_0)$ ایجاد می‌شود. حل‌کننده قید آن را حل می‌کند و {x=2, y=1} به عنوان ورودی جدید برای اجرای بعدی تولید می‌شود. اجرا، با این ورودی در خط ۷، شاخه then و در خط ۸، شاخه else را انتخاب می‌کند. به این ترتیب مسیر جدیدی از برنامه ایجاد و اجرا می‌شود. در این حالت $(2y_0 = x_0) \wedge (x_0 > y_0 + 10)$ به عنوان شرط مسیر جدید به حل‌کننده قید داده می‌شود و در نهایت ورودی {x=30, y=15} بدست می‌آید که به وسیله آن مسیر منتهی به ERROR اجرا می‌شود. از جمله ابزارهایی که از اجرای Concolic استفاده می‌کنند DART، CUTE و CREST هستند که در فصل‌های آینده به تفصیل بررسی خواهند شد.

۲-۲-۲ روش EGT

در این روش به جای اینکه تمام متغیرها به صورت نمادین محاسبه شوند، تنها متغیرهایی که کاربر مشخص می‌کنند، به صورت نمادین خواهند بود. همچنین به صورت پویا در هر عبارت این بررسی صورت می‌گیرد که آیا تمام متغیرهای موجود در برنامه عددی هستند یا نه. اگر همه عددی باشند عبارت به صورت عادی و به صورت عددی اجرا می‌شود. ولی اگر حتی یکی از متغیرها نمادین باشند عبارت به صورت نمادین اجرا می‌شود و قید مربوط به شرط مسیر اضافه می‌شود. اگر در خط ۱۷ عبارت به شکل y=10 تغییر کند، خط ۶ به صورت عددی اجرا می‌شود چون ورودی تابع twice مقدار عددی ۱۰ خواهد بود. در خط ۷ عبارت شرطی به شکل if(20==x) 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 ۸ عبارت به شکل if(x>20) در می‌آید. با توجه به اینکه x برابر مقدار ۲۰ است شاخه then قابل اجرا نخواهد بود و اجرا بدون اینکه به شاخه ERROR برسد پایان می‌پذیرد.

۲-۲-۳ بحث

با استفاده از روش‌های نوین می‌توان محدودیت‌های مربوط به حل‌کننده قید و کدهای کتابخانه‌های خارجی را تعدیل کرد. برای مثال در برنامه‌های کاربردی واقعی اگر ورودی یک تابع کتابخانه خارجی، نمادین باشد، برای ادامه اجرا لازم است تا ورودی عددی شود. برای این منظور در روش EGT قید موجود در آن عبارت توسط حل‌کننده قید حل می‌شود یا در روش Concolic از حالت عددی موجود مقدار عددی آن ورودی محاسبه می‌شود. به این ترتیب اجرا می‌تواند ادامه پیدا کند.

مثلا در همان کد نمونه شکل ۱ اگر تابع twice از یک کتابخانه خارجی دریافت شود یا اینکه این تابع به شکل (v*v)%50 که غیرخطی است درآید، هنگامی که اجرا به خط ۶ از برنامه می‌رسد اجرا متوقف می‌شود. چون حل‌کننده قید نمی‌تواند قید مربوطه را حل کند. در این حالت در اجرای Concolic مقدار نمادین برای متغیر y با مقدار عددی آن مثلا ۷ جایگزین می‌شود و اجرا ادامه پیدا می‌کند. در این حالت قید مربوط به خط ۷ به شکل if(x==49) در می‌آید و اجرا ادامه پیدا می‌کند. در اجرای نمادین سنتی چون حل‌کننده قید قادر به حل این گونه قیدها نبود اجرا متوقف می‌شد.

۳ روش‌های تشخیص آسیب‌پذیری‌های نرم‌افزاری

یکی از مشکلاتی که در امنیت نرم‌افزارها مطرح است مسئله آسیب‌پذیری‌های موجود در نرم‌افزارهاست. نحوه کشف این آسیب‌پذیری‌ها، از جمله مطالب مورد علاقه محققان بوده است. مطالبی که در قسمت ۳-۱ مورد بحث قرار می‌گیرند از مقاله [۵] آمده است. در این مقاله مطالعه‌ای بر روی آسیب‌پذیری‌های نرم‌افزاری و نحوه کشف آنها به همراه دسته‌بندی روش‌های تشخیص آمده است. همچنین در هر مورد تعدادی ابزار معرفی شده است. در قسمت ۳-۲ شرحی از مقاله [۶] آمده است. در این مقاله نویسندگان بیان می‌کنند که آنچه که در کد برنامه دیده می‌شود با آنچه که در فایل باینری اجرا می‌شود یکی نیست! آنها با بیان تعدادی دلیل و مثال، تاکید می‌کنند که لازم است فایل باینری برنامه مورد تحلیل قرار بگیرد.

۳-۱ مطالعه‌ای بر روش‌های تشخیص آسیب‌پذیری در نرم‌افزارها

نویسندگان مقاله بیان می‌کنند که تعاریف متفاوتی در مورد آسیب‌پذیری موجود در نرم‌افزارها وجود دارد:

تعریف ۱: ضعف یا خطای موجود در طراحی، پیاده‌سازی و یا اجرای سامانه که یک کاربر بدخواه می‌تواند از آنها در دور زدن خط مشی‌های امنیتی استفاده کند.

تعریف ۲: اشتباه در تعریف[۳۷]، توسعه یا تنظیمات نرم‌افزار که اجرای آن توسط مهاجم به طور صریح یا ضمنی، یکی از خط مشی‌های امنیتی را نقض می‌کند.

تحقیقات در این حوزه در دو دسته طبقه‌بندی می‌شود:

۱. تحلیل آسیب‌پذیری: در این حوزه بر روی ویژگی‌های آسیب‌پذیری‌های موجود تحقیقات انجام می‌شود. مثل دلیل، محل و یا ویژگی‌های پیاده‌سازی. هدف، شناسایی آسیب‌پذیری‌های شناخته نشده است.
۲. کشف آسیب‌پذیری: هدف، کشف آسیب‌پذیری‌های شناخته شده‌ای است که به صورت ناخواسته در نرم‌افزارها وجود دارند.

[۳۷] Specification

در این نوشته تاکید بر روی کشف آسیب‌پذیری است. با توجه به دسته‌بندی که در مقاله [۵] وجود دارد، تکنیک‌های این حوزه به تحلیل ایستا، تحلیل پویا، روش فاز و آزمون نفوذ در نرم‌افزارها تقسیم می‌شود.

۳-۱-۱ تحلیل ایستا

تعریف ۳: تحلیل ایستا فرایند ارزیابی یک سیستم بر اساس شکل، ساختار، محتوا یا مستندات آن است و نیازی به اجرای برنامه در آن نیست.

بعضی از آسیب‌پذیری‌ها توسط این روش قابل تشخیص نیست و این یعنی این تحلیل کامل نیست. علاوه بر آن، تحلیل ایستا می‌تواند تخمینی از رفتار برنامه را داشته باشد و این یعنی مثبت کاذب و منفی کاذب[۳۸] در آن بالا است. منفی کاذب خطرناک‌تر از مثبت کاذب است. برای تحلیل مثبت کاذب هم لازم به دخالت انسان است.

یکی از ساده‌ترین ابزارهای موجود در این حوزه، grep در سیستم یونیکس[۳۹] است. با این ابزار رشته‌های موجود در کد برنامه بررسی شده و با لیستی از رشته‌هایی از آسیب‌پذیری‌ها مطابقت داده می‌شود. ابزار دیگر IST4 و RAT است. این دو ابزار از تحلیل کلمات[۴۰] استفاده می‌کنند. در این ابزارها ابتدا یک پیش‌پردازش انجام می‌شود و از کد برنامه کلمات استخراج می‌شوند سپس این کلمات با کلمات موجود در کتابخانه‌ای از کلمات آسیب‌پذیری‌ها مطابقت داده می‌شود. ابزار دیگر SWORD4J است. این ابزار با پیاده‌سازی الگوریتم MARCO (تحلیل SBAC[۴۱]) یا الگوریتم ESPE (تحلیل RBAC[۴۲]) برنامه‌های به زبان جاوا را تحلیل می‌کند.

۳-۱-۲روش فاز

تعاریف مختلفی از آزمون فاز در مقاله‌های مختلف بیان شده است. از جمله: «روش آزمون با داده‌های دلخواه»، «آزمون جعبه سیاه خودکار» یا «روش آزمون خودکار با داده‌های ورودی مختلف به منظور کشف آسیب‌پذری برنامه». به طور کلی می‌توان این روش را بر اساس مراحل اجرای آن تعریف کرد. ابتدا تولید ورودی دلخواه سپس اجرای برنامه با این ورودی‌ها و در آخر بررسی این که آیا برنامه با این وروی‌ها متوقف می‌شود یا نه.

---

[۳۸] False Negative
[۳۹] Unix
[۴۰] Lexical Analysis
[۴۱] Stack Based Access Control
[۴۲] Role Based Access Control

ابتدا برنامه با داده‌های ورودی کاملا دلخواه اجرا می‌شد که پوشش کاملی از تمام مسیرهای برنامه نداشت. بعد از آن محققان دو روش برای تولید ورودی برنامه‌ها پیشنهاد دادند: ۱) روش تولید-داده ۲) روش جهش-داده. در روش اول، داده‌های ورودی بر اساس تعریف ورودی برنامه ایجاد می‌شود. مثلا اگر ورودی به شکل فایل باشد، ورودی‌ها بر اساس قالب فایل ورودی ایجاد می‌شود. برای این کار نیاز به اطلاعات زیادی در مورد قالب فایل یا پروتکل ورودی است و نیاز به تعامل زیاد انسان است. اگر تعریف داده ورودی پیچیده باشد و تهیه داده نمونه آسان باشد، راه‌کار دوم بهتر است. در این مورد با توجه به ورودی مسیر خاصی از برنامه اجرا می‌شود. ابزارهای Autodafe و APIKE Proxy از روش دوم استفاده می‌کنند. ابزار Peach ترکیب هر دو روش است.

۳-۱-۳ تحلیل پویا

تعریف: خطایابی بر اساس اجرای برنامه.

ویژگی‌های تحلیل پویا

- نیاز به ورودی برای تحلیل برنامه
- تنها خطاهایی که در مسیری که با آن ورودی خاص طی می‌شود قابل شناسایی است
- مثبت کاذب ندارد چون برنامه اجرا می‌شود.

در این مقاله روش‌های پویای آزمون نرم‌افزارها مثل روش Concolic به عنوان آزمون جعبه سفید هم جزء روش فاز به حساب آمده است. این روش به تفصیل در قسمت ۲-۲-۱ و ابزارهای آن در فصل ۵ آمده است.

۴-۱-۳ آزمون نفوذ

آزمون نفوذ، امنیت یک سیستم را با شبیه‌سازی حمله افراد بدخواه به آن و میزان موفقیت در حمله را ارزیابی می‌کند. آزمون نفوذ توسط یک تیم خاص که به استخدام شرکت ارائه دهنده سیستم در می‌آیند اجرا می‌شود و سه دسته کلی دارد:

۱. جعبه سیاه: آزمون‌گر هیچ اطلاعاتی در مورد سیستم ندارد.
۲. جعبه سفید: آزمون‌گر اطلاعاتی از قبیل نحوه پیاده‌سازی، کد برنامه، تعدادی از کلمه عبورها و غیره را می‌داند.
۳. جعبه خاکستری: که چیزی میان دو مورد قبلی است.

در آزمون نفوذ دو مرحله وجود دارد. اول آزمون‌گر سیستم را تحلیل کرده و تهدیدات و میزان اثر و اهمیت هر یک را تهیه می‌کند(مثلا درخت تهدید را ایجاد می‌کند). سپس بر اساس اهمیت تهدیدات حملاتی علیه امنیت سیستم اجرا می‌کند. در نهایت در گزارش نهایی علاوه بر اینکه سیستم نسبت به تهدیدی خاص آسیب‌پذیر است یا نه، سناریوهای حمله، نمونه کد حمله و میزان اهمیت آن نیز ذکر خواهد شد.

۳-۲چرا اجرای فایل اجرایی بهتر است؟

در تحلیل ایستا کد منبع مورد تحلیل قرار می‌گیرد. ولی آن چیزی که در کد منبع هست اجرا نمی‌شود! تبدیل زبان سطح بالا به زبان ماشین توسط کامپایلر باعث به وجود آمدن یک سری آسیب‌پذیری‌هایی می‌شود که توسط تحلیل ایستا قابل تشخیص نیست. یک مثال از این پدیده کد زیر است. خط اول گذرواژه را در حافظه با نویسه[۴۳] '\0' جایگزین می‌کند و سپس در خط بعد، حافظه آن را آزاد می‌کند تا اطلاعات حساس از حافظه موقت پاک شود. متاسفانه کامپایلر در بهینه‌سازی‌های خود (یعنی حذف کدهای بدون استفاده) خط اول را حذف می‌کند. و این موضوع باعث می‌شود گذرواژه در حافظه باقی بماند.

```
memset(password,'0', len);
free(password);
```

دلایل مختلفی وجود دارد که نشان می‌دهد که تحلیل کد اجرایی ضرورت دارد:

- علاوه بر مثال فراخوانی توابع در بالا، بهینه‌سازی‌های کامپایلرها، باگ‌های مربوط به پلتفرم، ساختار ذخیره‌سازی مثل Activation Record و غیره هم می‌توانند موجب ایجاد آسیب‌پذیری شوند.
- تحلیل‌های ایستا، فرضیات و محاسبات مجاز کامپایلرها را در نظر نمی‌گیرند. مثل عملیات حسابی روی اشاره‌گرها به توابع که به صورت غیرمستقیم فراخوانی می‌شوند.
- فراخوانی پویای کد یا کتابخانه‌ها
- بهینه‌سازی کامپایلرها یا کدهای اضافه شده به کد منبع به منظور تجهیز آن برای کارهای خاص منظوره
- چند زبانی کد منبع
- وجود کد اسمبلی در میان کدهای زبان بالا
- در دسترس نبودن کد منبع
- اگر کد منبع هم در دسترس باشد به دلایل بالا بهتر است که ابتدا عملیات کامپایل انجام شود سپس کد تولید شده تحلیل شود که این خود سربار زیادی دارد و بهتر است کد اجرایی تحلیل شود.

در ادامه تعدادی مثال دیگر برای زبان C به منظور روشن شدن اهمیت این موضوع بیان می‌شوند.

[۴۳] Character

زبان C از چند نخی پشتیبانی نمی‌کند. برای این که از امکانات چندنخی استفاده شود کتابخانه‌ای به این منظور پیاده‌سازی شده است که البته نحوه رفتار یک کد در اجراهای متفاوت غیرمنتظره و غیرقابل پیش‌بینی است و این موضوع ضرورت تحلیل کد اجرایی نشان می‌دهد.

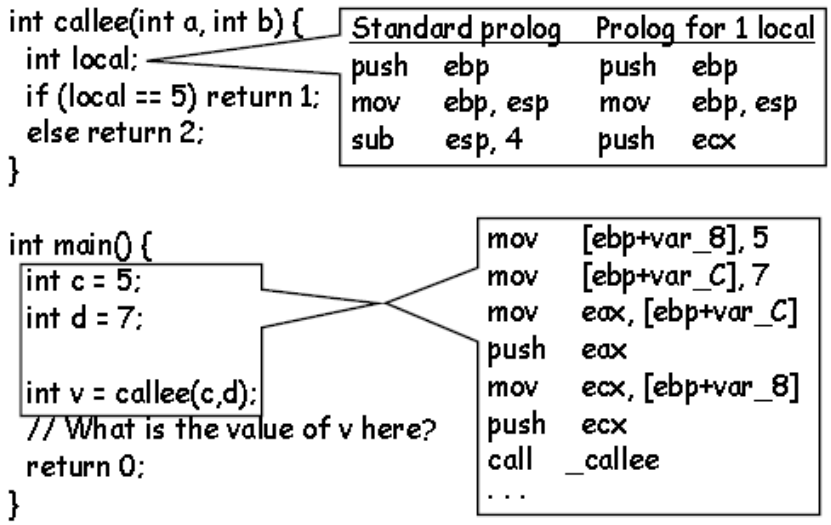
در زبان سطح بالا ممکن است که رفتار و نحو کد مشخص نباشد و کامپایلر تعیین کند که یک کد چه رفتاری داشته باشد. مثلا محاسبه یک عبارت از چپ به راست یا برعکس.

در C امکان فراخوانی تابع به وسیله اشاره‌گرها و به طور غیرمستقیم وجود دارد. نمونه‌ای از چنین کدی در شکل ۴ آمده است. مشکلی که وجود دارد این است که در تحلیل ایستا نمی‌توان محاسبات حسابی اشاره‌گرها را انجام داد تا بتوان تشخیص داد که که آیا نتیجه مجموعه محاسبات، به حافظه مربوط به تابع خاص اشاره می‌کند یا خیر.

```
int (*f)(void);
int diff = (char*)&f2 - (char*)&f1; // The offset between f1 and f2
f = &f1;
f = (int (*)())((char*)f + diff); // f now points to f2
(*f)(); // indirect call;
```

شکل ۴: نمونه کد مربوط به محاسبات اشاره‌گرهای به یک تابع

بهینه‌سازی‌هایی که در کامپایلرهای مختلف وجود دارد باعث می‌شود که یک کد مشخص بعد از کامپایل رفتار متفاوتی از خود نشان دهد. مثلا کد شکل ۵ با کامپایلر مایکروسافت همیشه خروجی یک دارد ولی با کامپایلرهای دیگر می‌تواند خروجی یک یا دو داشته باشد.

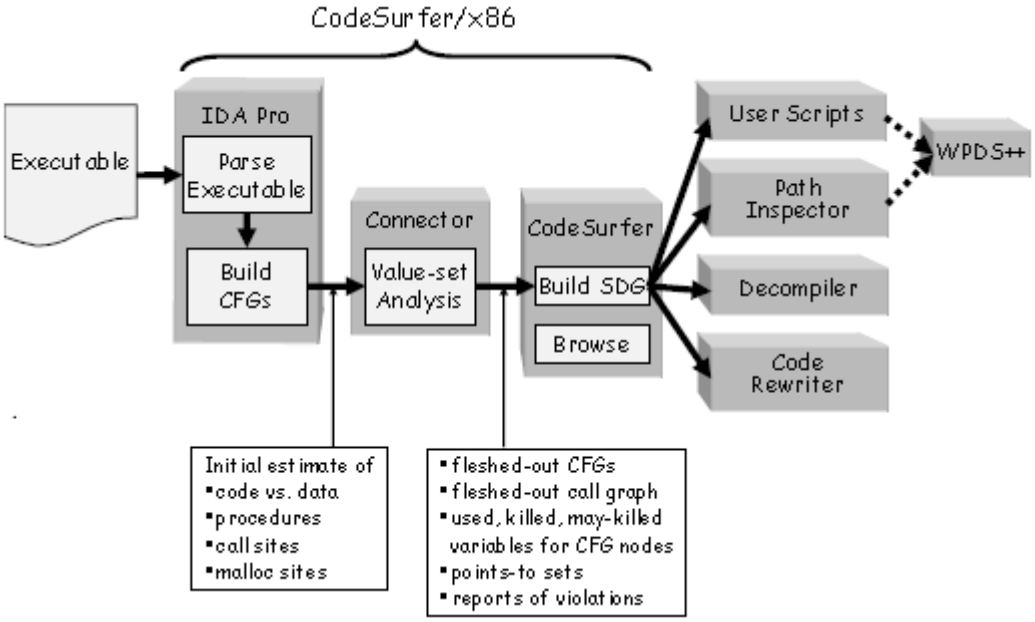


شکل ۵: مثالی از رفتار غیرمنتظره یک کد در اثر بهینه‌سازی‌های کامپایلر

اگر تابعی کمتر از تعداد مورد انتظار ورودی دریافت کند، بقیه آرگومان‌ها را به صورت

فراخوانی از مرجع[۴۴] از متغیرهای محلی می‌خواند. اگر یک انتساب[۴۵] در تابع وجود داشته باشد می‌تواند مقدار آن متغیر را عوض کند. تحلیل کد باینری می‌تواند مشخص کند که کدام متغیرها تغییر خواهند کرد.

ابزار ارائه شده در این مقاله، برنامه‌های اجرایی را در نبود کد منبع تحلیل می‌کند. مولفه‌های اصلی این ابزار عبارتند از CodeSurfer/x86، WPDS++ و Path Inspector که در شکل ۶ به طور کامل همراه با روابط



شکل ۶: معماری ابزار ارائه شده در مقاله آنچه می‌بینید اجرا نمی‌شود.

بین آنها دیده می‌شود.

IDA Pro یک Desassebler است. علاوه بر تبدیل باینری به اسمبلی، امکان ایجاد افزونه و استخراج اطلاعات بیشتری در مورد کد باینری را فراهم می‌آورد. برای مثال مولفه Connector همین افزونه است. اطلاعاتی از قبیل محدوده توابع، آدرس‌های ایستای استفاده شده و فراخوانی توابع کتابخانه‌ای در ساختمان داده‌هایی در این افزونه پیاده‌سازی شده‌اند. به وسیله همین اطلاعات CFG و SDG استخراج می‌شوند. علاوه بر آن در این افزونه بررسی‌های لازم صورت می‌گیرد. برای این منظور که کامپایلر عملیات خود را درست انجام می‌دهد یا نه. مثلا قطعه کدی وجود نداشته باشد که آدرس بازگشت یک تابع را از داخل خودش تغییر دهد. این موضوع باعث می‌شود CFG درست نباشد. WPDS هم یک ابزار Model Checker است که رفتار برنامه را مدل

[۴۴] Call by Reference
[۴۵] Assignment

می‌کند. Path Inspector هم یک رابط کاربری است که به وسیله آن می‌توان پرس‌وجوهای مربوط به امن بودن برنامه را در آن ایجاد و اجرا کرد.

۴ مطالعه بر روی آسیب‌پذیری‌ها

اطلاعات مربوط به انواع آسیب‌پذیری‌ها از سایت OWASP که سایت معتبر و رسمی بررسی آسیب‌پذیری‌هاست و کتاب جان آیکاک [۱۵] استخراج شده است. در این فصل آسیب‌پذیری‌های سرریز بافر و تزریق و زیر مجموعه‌های آنها به همراه نمونه کد، راه‌های شناسایی وجود آسیب‌پذیری‌ها و راه‌های مقابله با آنها عنوان می‌شوند.

۴-۱ آسیب‌پذیری سرریز بافر

در [۱۶] آمده است که تقریبا تمام پلتفرم‌ها نسبت به سرریز بافر آسیب‌پذیرند. به غیر از Java\J2EE، NET. از آنجایی که فراخوانی‌های سیستمی و توابع سطح پایین[۴۶] و ناامن در آنها قابل اجرا نیست و PHP، Python و Perl از آنجایی که برنامه‌های خارجی و افزونه‌های آسیب‌پذیر در آنها استفاده نمی شود.

حملات سرریز بافر می‌تواند هم در سرور وب و هم سرور برنامه کاربردی اتفاق بیفتد که ماهیت پویا و ایستا دارند. نقص‌های سرریز بافر ممکن است در کتابخانه‌ها و سرویس‌های موجود یا در برنامه‌هایی که به صورت سفارشی نوشته شده‌اند، موجود باشد. نقص‌های مورد اول یا کشف شده‌اند که جزئیات آنها روزانه منتشر می‌شوند یا هنوز پنهان مانده‌اند. در مورد دسته دوم هم کمبود اطلاعات امنیتی توسعه دهنده موجب چنین نقص‌هایی در کد می‌شود.

حملات سرریز بافر معمولا با همراهی دو مورد زیر اتفاق می‌افتند:

- نوشتن اطلاعات در آدرس‌هایی خاص از برنامه
- سیستم عامل در کنترل کردن نوع[۴۷]ها دچار اشتباه شود.

این موضوع نشان می‌دهد که زبان‌های مبتنی بر نوعِ قوی که اجازه دسترسی مستقیم به حافظه را نمی‌دهند در معرض چنین حملاتی قرار نمی‌گیرند. در جدول ۲ مقایسه‌ای میان زبان‌های مختلف دیده می‌شود.

[۴۶] Native
[۴۷] Type

Language/Environment	Compiled or Interpreted	Strongly Typed	Direct Memory Access	Safe or Unsafe
Java, Java Virtual Machine (JVM)	Both	Yes	No	Safe
.NET	Both	Yes	No	Safe
Perl	Both	Yes	No	Safe
Python - interpreted	Intepreted	Yes	No	Safe
Ruby	Interpreted	Yes	No	Safe
C/C++	Compiled	No	Yes	Unsafe
Assembly	Compiled	No	Yes	Unsafe
COBOL	Compiled	Yes	No	Safe

جدول ۲: مقایسه زبان‌های برنامه نویسی مختلف

۴-۱-۱ راه‌های کلی مقابله

- بازرسی کد به صورت دستی یا خودکار
- آموزش برنامه‌نویسان
- غیراجرایی کردن پشته[۴۸]
- ابزارهای کامپایلر
- استفاده از توابع امن مثلا استفاده از strncat به جای strcat
- استفاه در از وصله[۴۹]های نرم‌افزاری و به‌روز نگه‌داشتن برنامه

۴-۱-۲سرریز پشته

آسیب‌پذیری سرریزپشته از جمله آسیب‌پذیری‌های شناخته شده‌ی سرریز بافر است. مراحل این حمله به طور ساده شده در زیر آمده است:

[۴۸] Stack
[۴۹] Patch

- در کد دو بافر وجود دارد که اندازه آنها برابر نیست و بافر کوچکتر در پشته در همسایگی آدرس بازگشت تابع قرار گرفته است.
- کد معیوب مقادیر بافر بزرگتر را در کوچکتر کپی می‌کند، بدون اینکه اندازه بافرها را بررسی کند. اطلاعات در بافر کوچکتر و تعدادی آدرس حافظه بعد از آن دوباره نویسی می‌شود. آدرس بازگشت تابع نیز بازنویسی می‌شود.
- حال وقتی سیستم عامل به آدرس بازگشت تابع می‌رسد، به جای دستور بعدی در کد برنامه به دستوری از کد مهاجم که در پشته نوشته است برمی‌گردد و دستورات مهاجم اجرا می‌شود.

نمونه کد این حمله در زیر به زبان C آمده است:

```
#include <string.h>

void f(char* s) {
    char buffer[10];
    strcpy(buffer, s);
}

void main(void) {
    f("01234567890123456789");
}

[root /tmp]# ./stacktest

Segmentation fault
```

۴-۱-۲-۱ چطور از آسیب‌پذیر بودن برنامه آگاه شویم

اگر برنامه:

- به زبان ناامن مطابق آنچه در جدول بالا آمده است نوشته شده باشد و
- از یک بافر به بافر دیگر در پشته بدون بررسی اندازه آنها اطلاعات کپی شود و
- از روش‌های مقابله مثل روش قناری در کامپایلر یا غیراجرایی کردن پشته استفاده نشده باشد آنگاه

می‌توان گفت که برنامه نسبت به این حمله آسیب‌پذیر است.

۴-۱-۲-۲ راه‌های مقابله

- استفاده از سیستم‌هایی که از غیراجرایی بون پشته حمایت می‌کنند مثل ویندوز XP SP2.
- استفاده از زبان‌های برنامه نویسی سطح بالا
- اعتبارسنجی داده ورودی و بررسی معتبر بودن نویسه‌های ورودی
- اگر بر توابع سیستم عامل که به زبان ناامن نوشته شده‌اند متکی است باید مطمئن شد که:
  - استفاده از اصل حداقل مجوز[۵۰]
  - استفاده از کامپایلرهایی که در برابر نقص‌های سرریز بافر مقاوم هستند.
  - استفاده از وصله‌های امنیتی به‌روز

۴-۱-۳ سرریز هیپ[۵۱]

هیپ حافظه‌ای است که توسط برنامه در زمان اجرا اختصاص می‌یابد. در زیر نمونه کد این آسیب‌پذیری به زبان C آمده است:

```
#include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <string.h>

 #define BSIZE 16
 #define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */

 void main(void) {
    u_long b_diff;
    char *buf0 = (char*)malloc(BSIZE);            // create two buffers
    char *buf1 = (char*)malloc(BSIZE);

    b_diff = (u_long)buf1 - (u_long)buf0;// difference between locations
    printf("Initial values:  ");
    printf("buf0=%p, buf1=%p, b_diff=0x%x bytes\n", buf0, buf1, b_diff);

    memset(buf1, 'A', BUFSIZE-1), buf1[BUFSIZE-1] = '\0';
    printf("Before overflow: buf1=%s\n", buf1);
```

[۵۰] Least Privilage Principle

[۵۱] Heap overflow

```
    memset(buf0, 'B', (u_int)(diff + OVERSIZE));
    printf("After overflow:  buf1=%s\n", buf1);
}

[root /tmp]# ./heaptest

Initial values:  buf0=0x9322008, buf1=0x9322020, diff=0xff0 bytes
Before overflow: buf1=AAAAAAAAAAAAAAA
After overflow:  buf1=BBBBBBBBAAAAAAA
```

در کد بالا دو بافر در هیپ ایجاد شده‌اند که بافر اول توسط محتویات بافر دوم سرریز شده است.

۴-۱-۳-۱ چطور از آسیب‌پذیر بودن برنامه آگاه شویم

اگر برنامه:

- به زبان ناامن مطابق آنچه در جدول بالا آمده است نوشته شده باشد و
- از یک بافر در پشته به بافر دیگر بدون بررسی اندازه آنها اطلاعات کپی شود و
- از روش‌های مقابله مثل روش قناری در کامپایلر یا غیراجرایی کردن پشته استفاده نشده باشد آنگاه

می‌توان گفت که برنامه نسبت به این حمله آسیب‌پذیر است.

۴-۱-۳-۲ راه‌های مقابله

- استفاده از سیستم‌هایی که از غیراجرایی بون پشته حمایت می‌کنند مثل ویندوز XP SP2.
- استفاده از زبان‌های برنامه نویسی سطح بالا
- اعتبارسنجی داده ورودی و بررسی معتبر بودن نویسه‌های ورودی
- اگر بر توابع سیستم عامل که به زبان ناامن نوشته شده‌اند متکی است باید مطمئن شد که:
    - استفاده از اصل کمترین مجوز
    - استفاده از کامپایلرهایی که در برابر نقص‌های سرریز بافر مقاوم هستند.
    - استفاده از وصله‌های امنیتی به‌روز

۴-۱-۴آسیب‌پذیری قالب رشته[۵۲]

از این آسیب‌پذیری برای کمک به آسیب‌پذیری‌های گذشته نیز استفاده می‌شود. کد زیر به زبان C را در نظر بگیرید:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

void main(void) {
    char str[100] = scanf("%s");
    printf("%s", str);
}
```

این کد آسیب‌پذیر نیست و تنها تعدادی نویسه از ورودی می‌خواند و در نوع رشته چاپ می‌کند. اگر خط آخر به شکل printf(str); در بیاید کد آسیب‌پذیر می‌شود. اگر کاربر ورودی به شکل *%08x.%08x.%08x.%08x.%08x* وارد کند، ۵ خانه اول پشته در خروجی چاپ می‌شود. از این روش برای بدست آوردن محل آدرس بازگشت در توابع یا دور زدن روش قناری نیز استفاده می‌شود.

۱-۴-۱-۴ چطور از آسیب‌پذیر بودن برنامه آگاه شویم

اگر برنامه:

- از توابعی مثل printf یا sprintf به طور مستقیم یا از طریق سرویس‌های سیستمی مثل syslog به طور غیر مستقیم استفاده کند و
- استفاده از این توابع به کاربر اجازه می‌دهد اطلاعات کنترلی وارد کند.

می‌توان گفت که برنامه نسبت به این حمله آسیب‌پذیر است.

۲-۴-۱-۴ راه‌های مقابله

- استفاده از زبان‌های برنامه نویسی سطح بالا
- اعتبارسنجی داده ورودی و بررسی معتبر بودن نویسه‌های ورودی (نبودن '%' در ورودی)

[۵۲] Format String

- منع کردن برنامه‌نویس از استفاده از توابعی مثل printf
- اگر بر توابع سیستم عامل که به زبان ناامن نوشته شده‌اند متکی است باید مطمئن شد که:
  - استفاده از اصل کمترین مجوز
  - استفاده از کامپایلرهایی که در برابر نقص‌های سرریز بافر مقاوم هستند.
  - استفاده از وصله‌های امنیتی به‌روز

۴-۱-۵سرریز اعداد int

وقتی در برنامه‌ای دو مقدار با اندازه ثابت با هم جمع می‌شوند، ممکن است حاصل جمع بزرگتر از حافظه اختصاص داده شده باشد. مثلا در جمع دو عدد ۱۹۲ و ۲۰۸ که هر دو ۲ بایت است، جواب ۴۰۰ است که برای نمایش دودویی آن نیاز به ۳ بایت حافظه است. در نتیجه جواب در ۲ بایت جا نمی‌شود و حاصل ۱۴۴ ذخیره می‌شود:

1100 0000

+ 1101 0000

= 0001 1001 0000

کد زیر نمونه‌ای از چنین آسیب‌پذیری به زبان C را نشان می‌دهد:

```
#include <stdio.h>
#include <string.h>

void main(int argc, char *argv[]) {
    int i = atoi(argv[1]);          // input from user
    unsigned short s = i;           // truncate to a short
    char buf[50];                   // large buffer

    if (s > 10) {                   // check we're not greater than 10
        return;
    }

    memcpy(buf, argv[2], i);        // copy i bytes to the buffer
    buf[i] = '\0';                  // add a null byte to the buffer
    printf("%s\n", buf);            // output the buffer contents

    return;
```

```
}

[root /tmp]# ./inttest 65580 foobar
Segmentation fault
```

این برنامه نسبت به این حمله آسیب‌پذیر است چون ورودی برنامه ۶۵۵۸۰ اعتبارسنجی نمی‌شود. این آسیب‌پذیری در تمام زبان‌ها امکان وقوع دارد.

۴-۱-۵-۱ چطور از آسیب‌پذیر بودن برنامه آگاه شویم

- آیا برنامه از نوع‌های signed int، short یا byte استفاده می‌کند؟
- آیا مقادیر بالا بعد از محاسبات ریاضی به عنوان اشاره‌گر به خانه‌های یک آرایه استفاده می‌شود؟
- برنامه در مواجه با مقادیر صفر یا منفی چطور عمل می‌کند مخصوصا در هنگام جست‌وجو در آرایه؟

۴-۱-۵-۲ راه‌های مقابله

- اگر از .NET. استفاده می‌کنید از «David LeBlanc’s SafeInt class » یا کلاس‌های مشابه استفاده کنید و یا اگر فرصت اعتبارسنجی ورودی‌ها را ندارید از «BigInteger» یا «BigDecimal» استفاده کنید.
- در صورت امکان کامپایلر را این طور تنظیم کنید که از Unsigned به صورت پیش فرض استفاده کند.
- از بررسی درستی محدوده اعداد[۵۳] حتما استفاده کنید.
- بررسی کردن استثناها اگر زبان آن را پشتیبانی می‌کند.

۴-۲ آسیب‌پذیری تزریق[۵۴]

در [۱۷] آمده است که وقتی برنامه کاربردی به نوعی باشد که از طرق یک برنامه واسط، یک دستور به سیستم عامل یا پایگاه‌داده ارسال شود، امکان وجود این آسیب‌پذیری وجود خواهد داشت. هرگاه برنامه‌ای از یک مفسر[۵۵] از هر نوعی استفاده کند، خطر اینگونه حملات وجود خواهد داشت.

[۵۳] Range checking
[۵۴] Injection
[۵۵] Interpreter

تعدادی زیادی از برنامه‌های تحت وب از برنامه‌های خارجی یا فراخوانی‌های سیستمی برای ارائه کاربردهای خود استفاده می‌کنند. برنامه Sendmail از پرکاربردترین این برنامه‌هاست. وقتی اطلاعات از طریق یک درخواست HTTP ارسال می‌شود، داده‌های ارسال شده باید کاملا مورد بررسی قرار گیرند. در غیر این صورت مهاجم در میان داده‌ها یک سری دستورات خرابکارانه یا نویسه‌های خاص را ارسال می‌کند. برنامه تحت وب هم کورکورانه این داده‌ها را برای اجرا ارسال می‌کند.

تزریق SQL از جمله انواع شایع حملات تزریق است. در این جا کافی است که مهاجم محلی از برنامه را پیدا کند که داده‌های ورودی را برای اجرای یک پرس‌وجو به پایگاه‌داده ارسال می‌کند. با این کار و جاسازی کردن یک پرس‌وجوی بدخواه در میان داده‌های ارسالی، مهاجم می‌تواند اطلاعات پایگاه‌داده را استخراج کند.

اجرای اینگونه از حملات ساده اما بسیار خطرناک هستند و می‌توانند باعث شوند کل سیستم به مخاطره بیفتد.

برای مثال در دستوراتی که به سیستم عامل ارسال می‌شوند می‌توان در ادامه دستورات دیگری نیز اضافه کرد مثلا «; rm –r *» برای پاک کردن تمام فایل‌ها به صورت بازگشتی. یا با اضافه کردن دستوراتی در قسمت «where» در پرس‌وجوهای به پایگاه‌داده می‌توان پرس‌وجوی مورد نظر خود را ارسال و اجرا کرد. مثلا اضافه کردن «” or 1=1» به انتهای پرس‌وجو باعث می‌شود پرس‌وجو به ازای هر مقدار «where» درست باشد.

۴-۲-۱چطور از آسیب‌پذیر بودن برنامه آگاه شویم

برنامه‌نویس باید کد خود را جست و جو کند و تمام محل‌هایی که به صورت مستقیم (مثلا system, exec, fork, Runtime.exec یا پرس‌وجوهای SQL) یا غیرمستقیم (تمام درخواست‌های HTTP) یک فراخوانی به سیستم یا پایگاه‌داده ارسال می‌شود را بررسی کند.

۴-۲-۲ راه‌های مقابله

- استفاده از توابع کتابخانه‌ای موجود به جای فراخوانی‌های سیستمی
- اعتبارسنجی ورودی توابع خاص و بررسی عملکرد آنها که بدخواه نباشد.
- رعایت اصل حداقل مجوز. یعنی برنامه‌های تحت وب به طور مثال توانایی اجرای یک دستور با مجوز روت[۵۶] را روی سیستم عامل نداشته باشد.

[۵۶] Root

نمونه کد ناامن به زبان جاوا برای آسیب‌پذیری تزریق کد SQL:

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
   + request.getParameter("customerName");

 try {
       Statement statement = connection.createStatement( … );
       ResultSet results = statement.executeQuery( query );
 }
```

نمونه کد امن به زبان جاوا برای آسیب‌پذیری تزریق کد SQL:

```
String custname = request.getParameter("customerName"); // This should
REALLY be validated too
 // perform input validation to detect attacks
 String query = "SELECT account_balance FROM user_data WHERE user_name = ?
";

 PreparedStatement pstmt = connection.prepareStatement( query );
 pstmt.setString( 1, custname);
 ResultSet results = pstmt.executeQuery( );
```

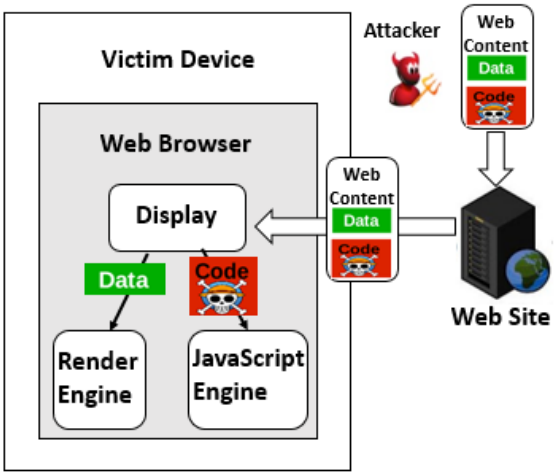
۳-۲-۴ بررسی آسیب‌پذیری تزریق به webView

در [۱۹] آمده است که متاسفانه بر خلاف مزایای بیان شده در رابطه با تکنولوژی وب برای پیاده‌سازی برنامه‌های گوشی‌های همراه، این نحوه پیاده‌سازی دارای یک سری ویژگی‌هایی است که می‌تواند موجب ایجاد آسیب‌پذیری در نرم‌افزار شود. در پیاده‌سازی تحت وب، کد و داده درهم آمیخته هستند و کدهای جاوا اسکریبت می‌توانند به

```
1   <img src=x onerror=
2   navigator.geolocation.watchPosition(
3   function(loc){
4   m='Latitude:'+loc.coords.latitude+
5   '\n'+'Longitude:'+loc.coords.longitude;
6   alert(m);
7   b=document.createElement('img');
8   b.src='http://128.***.213.66:5556?c='+m })>
```

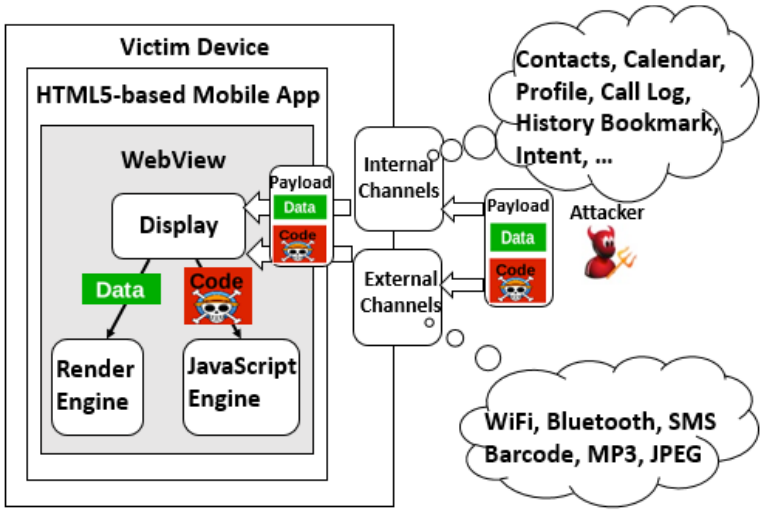
شکل ۷: کد نمونه جاوا اسکریپت برای حمله به webView

آسانی در میان کدهای HTML قرار گیرند و در نهایت پردازش شوند. همین موضوع موجب ایجاد حمله‌های تزریق کد[۵۷] می‌شود. در مرورگرها یک مکانیزم sandbox وجود دارد که کدهای دریافتی در آنها اجرا می‌شوند و اجازه دسترسی به منابع سیستم به آنها داده نمی‌شود. در webView نیز چنین مکانیزمی وجود دارد و مجوزهای درون آن با مجوزهای برنامه متفاوت است. ولی از آنجایی که نیاز است که از طریق کدهای جاوا اسکریبت به منابع سیستم مثل دوربین دسترسی پیدا شود، یک سری پل[۵۸] به زبان سیستم عامل ایجاد شده‌اند که با استفاده از آنها می‌توان pluginهای لازم را فراخوانی کرد. این موضوع باعث می‌شود کدهای جاوا اسکریبت هم مجوزهای داده شده به برنامه را بدست بیاورند که پیش از این، مکانیزم sandbox آنها را محدود کرده بود.



شکل ۹: نمایی از حمله XSS در برنامه‌های کاربردی تحت وب

در برنامه‌های تحت وب یکی از حملات مشهور، تزریق کد XSS است. در برنامه‌های تحت وب کدها تنها از طریق سرور قابل تزریق هستند. (شکل ۹) ولی در برنامه‌های گوشی‌های هوشمند کانال‌های تزریق کد متفاوت و فراوانند چون راه‌های انتقال اطلاعات متنوع هستند. این کانال‌ها به دو دسته کانالهای داخلی و



شکل ۸: نمایی از حمله تزریق کد به برنامه‌های کاربردی گوشی‌های هوشمند مبتنی بر وب

[۵۷] Code injection attack

[۵۸] Bridge

خارجی تقسیم می‌شوند. از جمله این کانال‌های خارجی می‌توان به بارکد، پیامک، WiFi، NFC، و کانال‌های داخلی می‌توان به سیستم مدیریت فایل اشاره کرد (شکل ۷).

برای اینکه حملات تزریق کد در گوشی‌های هوشمند موفق باشند، دو شرط در برنامه آسیب‌پذیر باید وجود داشته باشد:

- برنامه باید از طریق یکی از کانال‌ها داده دریافت کند.
- برنامه باید داده دریافتی را در صفحه نمایش نشان دهد.

با توجه به موارد گفته شده یک سناریو حمله به برنامه pic2shop بیان می‌شود که از طریق دوربین بارکد محصول را می‌خواند و اطلاعات آن را نشان می‌دهد. این برنامه در هر سه پلتفرم ویندوز، iOS و اندروید وجود دارد که نسخه اندرویدی آن ۵۰۰،۰۰۰ بار دانلود شده‌است.

یک بارکد که شامل کد شکل ۷ است تولید می‌شود. همان طور که دیده می‌شود این کد موقعیت جغرافیایی گوشی را در صفحه نمایش نشان می‌دهد و به یک آدرس IP خاص ارسال می‌کند. واضح است که این کد از طریق بارکد و دوربین وارد برنامه می‌شود. برای اینکه این کد اجرا شود لازم است که داده ورودی به یک تابع خاص داده شود. این تابع باید دارای این ویژگی باشد که در صورت وجود کد جاوا اسکریبت آن را به موتور اجرای اینگونه کدها ارسال کند. در جدول ۳ تابع‌های امن و ناامن برای این موضوع آورده شده است. در مقاله آمده است که این برنامه از یکی از این توابع ناامن استفاده کرده است و کد در آن تزریق می‌شود.

DOM	امن یا ناامن
document.write()	X
document.writeln()	X
innerHTML	X
outerHTML	X
innerText	Y
outerText	Y
textContent	Y
JQuery	
html()	X
append()	X
prepend()	X
before()	X
after()	X
replaceAll()	X
replaceWith()	X
text()	Y
val()	Y

جدول ۳: توابع امن و ناامن در DOM و JQuery. (X یعنی ناامن و Y یعنی امن)

در بررسی که در مقاله روی فراوانی این توابع انجام شده است، نشان می‌دهد که ۵۳ درصد از برنامه‌ها به طور متوسط از توابع ناامن استفاده می‌کنند و در مورد خاص، تابع innerHTML در ۹۱ درصد از برنامه‌ها استفاده می‌شود. این آمار اهمیت این آسیب‌پذیری را نشان می‌دهد. در [۲۰](CVE-2013-4710) این آسیب‌پذیری مورد بررسی قرار گرفته است.

در مقاله برای مقابله با حمله تزریق ابزاری ارائه شده است که از روش تحلیل ایستا استفاده می‌کند. نویسندگان در پایان به محدودیت‌های این روش اشاره کرده‌اند. برای اجرای این روش لازم بوده است که مدلی از چارچوبه کاری مورد استفاده ایجاد شود که به صورت دستی ایجاد شده است. علاوه بر آن همان طور که در CVE آمده است برای ایجاد پل لازم است تا از کتابخانه addJavascriptInterface استفاده شود تا بتوان از کدهای جاوای pluginها در کدهای جاوا اسکریبت استفاده کرد. همچنین ممکن است که کلاس‌هایی در زمان اجرا در سیستم بارگذاری شوند که تحلیل ایستا قادر به بررسی آنها نیست. علاوه بر آن با استفاده از تابع ()eval می‌توان یک رشته را به عنوان کد جاوا اسکریبت اجرا کرد. علاوه بر آن به وسیله object reflection می‌توان یک شی را در زمان اجرا بارگذاری کرد. برای حل این مشکل‌ها، ما در این پروژه از تحلیل پویا و به طور خاص روش Concolic استفاده می‌کنیم.

۵ کارهای پیشین و ابزارهای موجود

در این فصل ابزارهای مختلفی بررسی شده‌اند که از روش اجرای نمادین و یا Concolic برای آزمون یا کشف آسیب‌پذیری استفاده می‌کنند. برای زبان‌های برنامه نویسی مختلف مثل جاوا و C مثال‌هایی از ابزارهای مختلف آورده شده است. چارچوبه کاری Jalangi برای زبان جاوا اسکریپت تحلیل شده است که توانایی اجرای تحلیل‌های پویای مختلفی بر روی کدها را دارد. علاوه بر آن چالش‌های تحلیل کدهای برنامه‌های گوشی‌های هوشمند نیز بحث شده‌اند. همچنین تحلیل آلایش و ابزار Avalanche نیز عنوان شده است.

۵-۱ ابزار DART

DART(Directed Automated Random Testing) جزء اولین کارهایی است که از روش اجرای Concolic برای آزمون نرم‌افزارها استفاده می‌کند. P.Godefroid و دیگران در این مقاله [۲] در سال ۲۰۰۵ ابزار DART را برای آزمون نرم‌افزارهای به زبان C ارائه کرده‌اند. نویسندگان مقاله سه تکنیک که در ابزار موجود است را این گونه بیان می‌کنند:

- ابزار با استفاده از روش تحلیل ایستای کد به صورت خودکار رابط[۵۹] برنامه را استخراج می‌کند.
- ایجاد خودکار گرداننده[۶۰] آزمون به منظور آزمون دلخواه[۶۱] رابط، برای شبیه‌سازی محیط اجرای برنامه
- تحلیل پویای برنامه برای آزمون دلخواه آن و ایجاد خودکار مورد آزمون برای آزمون مسیرهای مختلف از برنامه

نگارندگان کار ارائه شده خود را با کارهای گذشته در این حوزه این گونه مقایسه می‌کنند. آزمون دلخواه نرم‌افزار یکی از تکنیک‌های ساده در این حوزه است. که معمولا توانایی پوشش تمام مسیرهای موجود در برنامه را ندارد. نوآوری DART نسبت به آزمون دلخواه در خودکار شدن فرایند و همچنین موثرتر شدن روند آزمون در پوشش تمام مسیرها و شاخه‌های برنامه است. روش دیگر تحلیل ایستای کد برنامه است. این روش دارای مثبت کاذب است همچنین در تحلیل کدهایی که در آنها از فراخوانی توابع کتابخانه‌ای استفاده می‌کنند یا فراخوانی

[۵۹] Interface
[۶۰] Driver
[۶۱] Random Testing

بین تابعی دارند، مناسب نیستند. در حالی که DART توانایی تحلیل این گونه از کدها را دارد. همچنین DART دارای ویژگی صحت[۶۲] است.

در ادامه بحث، ابزار DART، روش Concolic و مدل اجرایی ابزار مورد بررسی قرار می‌گیرند.

۵-۱-۱ بررسی ابزار DART

DART از روش Concolic که در ۲-۲-۱ توضیح داده شد استفاده می‌کند. در این قسمت مدل صوری که برای این ابزار ارائه شده و نحوه عملکرد ابزار بررسی می‌شود.

۵-۱-۱-۱مدل اجرایی

در این قسمت ابتدا تعدادی از نشانه‌های استفاده شده در مدل بیان می‌شوند. جدول ۴: نشانه‌های تعریف شده در مدل اجرایی DART را به طور خلاصه نشان می‌دهد.

مفهوم	نشانه	مفهوم	نشانه
خطا	$abort$	حافظه	M
پایان	$halt$	آدرس حافظه	m
اجرای e با مقادیر موجود در M	$evaluate - concrete(e, M)$	به‌روزرسانی	$+$
معادل goto l	$statement - at(l, M)$	مقدار ثابت	c
آدرس اولیه پارامترها	$\overrightarrow{M_0}$	ضرب	$*\ (e1, e2)$
مقادیر اولیه $\overrightarrow{M_0}$	$\vec{I}$	اشاره‌گر	$*\ e1$
حافظه نمادین؛ نگاشت آدرس‌های حافظه به عبارت‌ها. مقدار اولیه؛ نگاشت هر خانه به خودش	S	مقایسه	$\leq (e1, e2)$
تخصیص[۶۳]	$m \leftarrow e$	برچسب دستور	l

جدول ۴: نشانه‌های تعریف شده در مدل اجرایی DART

[۶۲] Soundness

[۶۳] Assignment

اگر C مجموعه بیان‌های شرطی و A مجموعه تخصیص‌ها باشد آنگاه عبارت $Execs(p) = (A \cup C)^*(abort|halt)$ نشان‌دهنده اجراهای متناهی است. اگر Execs(p) را به عنوان تمام اجراهای تولید شده با ورودی I در نظر بگیریم، اگر هر بیان را به عنوان یک گره در نظر بگیریم، Execs یک درخت اجرا را شکل می‌دهد که گره‌های تخصیص یک فرزند و گره‌های شرطی می‌توانند یک یا دو فرزند داشته باشند و برگ‌های درهت هم abort یا halt هستند.

۵-۱-۱-۲ گرداننده آزمون و برنامه تجهیزشده

در این قسمت تعدادی شبه کد بیان می‌شوند که نحوه عملکرد DART با آنها توصیف خواهد شد. شکل ۱۱ قطعه کد گرداننده آزمون آورده شده است. در این کد دو حلقه تودرتو وجود دارد. حلقه repeat برای آزمون دلخواه برنامه و حلقه while برای پیمایش هدفمند مسیرهای موجود در برنامه است. اگر در خلال اجرای یک مسیر از برنامه استثنایی رخ دهد یعنی خطایی در برنامه وجود دارد. همچنین حلقه بیرونی با دو متغیر all_linear و all_locs_definite کنترل می‌شوند. اولی به منظور بررسی اینکه آیا تمام متغیرهای موجود در یک عبارت خطی هستند

```
run_DART () =
  all_linear, all_locs_definite, forcing_ok = 1, 1, 1
  repeat
    stack = ⟨⟩; I⃗ = [] ; directed = 1
    while (directed) do
      try (directed, stack, I⃗) =
        instrumented_program(stack, I⃗)
      catch any exception →
        if (forcing_ok)
          print "Bug found"
          exit()
        else forcing_ok = 1
  until all_linear ∧ all_locs_definite
```

شکل ۱۱: گرداننده آزمون

```
instrumented_program(stack, I) =
  // Random initialization of uninitialized input parameters in M⃗0
  for each input x with I⃗[x] undefined do
    I⃗[x] = random()
  Initialize memory M from M⃗0 and I⃗
  // Set up symbolic memory and prepare execution
  S = [m ↦ m | m ∈ M⃗0].
  ℓ = ℓ0 // Initial program counter in P
  k = 0 // Number of conditionals executed
  // Now invoke P intertwined with symbolic calculations
  s = statement_at(ℓ, M)
  while (s ∉ {abort, halt}) do
    match (s)
      case (m ← e):
        S= S + [m ↦ evaluate_symbolic(e, M, S)]
        v = evaluate_concrete(e, M)
        M = M + [m ↦ v]; ℓ = ℓ + 1
      case (if (e) then goto ℓ'):
        b = evaluate_concrete(e, M)
        c = evaluate_symbolic(e, M, S)
        if b then
          path_constraint = path_constraint ^ ⟨c⟩
          stack = compare_and_update_stack(1, k,stack)
          ℓ = ℓ'
        else
          path_constraint = path_constraint ^ ⟨neg(c)⟩
          stack = compare_and_update_stack(0, k,stack)
          ℓ = ℓ + 1
        k= k + 1
    s =statement_at(ℓ,M)    // End of while loop
  if (s==abort) then
    raise an exception
  else // s==halt
    return solve_path_constraint(k,path_constraint,stack)
```

شکل ۱۰: تابع instrumented_program

یا نه استفاده می‌شود و دومی هم در مورد اینکه آیا مقدار یک اشاره‌گر عددی است یا نه نگهداری می‌شود. قبل از اجرای حلقه داخلی مقادیر پشته و $\vec{I}$ (مقادیری که به ازای آن مسیری خاص از برنامه اجرا می‌شود در آن نگهداری می‌شود) مقداردهی اولیه می‌شوند. پشته محل نگهداری اطلاعات مربوط به بیان‌های شرطی است که به شکل $stack[i] = (stack[i].branch, stack[i].done)$ مقداردهی و نگهداری می‌شود.

در حلقه اصلی، تابع instrumented_program اجرا می‌شود که در شکل ۱۰ آمده است. در این کد، $\vec{I}$ به صورت دلخواه مقداردهی اولیه می‌شود. M، S، l و k هم مقداردهی می‌شوند. در حلقه دو حالت از بیان‌ها بررسی می‌شوند که می‌توانند تخصیص یا شرطی باشند. در هر دو حالت عبارت e هم به صورت نمادین و هم به صورت عددی محاسبه می‌شوند. در بیان تخصیص آدرس موجود در M به روز می‌شود و یکی به l اضافه می‌شود یعنی به بیان بعدی در کد می‌پرد. در اینجا برای اجرای نمادین تابع evaluate-symbolic فراخوانی می‌شود که در شکل ۱۲ توضیح آن آمده است. در بیان شرطی با توجه به شاخه انتخاب شده شرط مناسب به قیود مسیر اضافه می‌شود. و با تابع compare_and_update_stack مقدار پشته به‌روز می‌شود که توضیح آن در شکل ۱۳ خواهد آمد. در نهایت اگر بیان بعدی abort باشد، یک استثنا برگردانده می‌شود و اگر halt باشد تابع solve_path_constraint فراخوانی می‌شود تا ورودی که به وسیله آن مسیر جدیدی از برنامه اجرا شود را تولید کند. توضیح این تابع هم در شکل ۱۴ آمده است.

```
evaluate_symbolic (e, M, S) =
  match e:
    case m:    //the symbolic variable named m
      if m ∈ domainS then return S(m)
      else return M(m)
    case *(e', e''):    //multiplication
      let f'= evaluate_symbolic(e', M, S);
      let f''= evaluate_symbolic(e'', M, S);
      if not one of f' or f'' is a constant c then
        all_linear = 0
        return evaluate_concrete(e, M)
      if both f' and f'' are constants then
        return evaluate_concrete(e, M)
      if f' is a constant c then
        return *(f', c)
      else return *(c, f'')
    case *e':    //pointer dereference
      let f'= evaluate_symbolic(e', M, S);
      if f' is a constant c then
        if *c ∈ domainS then return S(*c)
        else return M(*c)
      else all_locs_definite = 0
        return evaluate_concrete(e, M)
    etc.
```

شکل ۱۲: تابع evaluate_symbolic

در شکل ۱۲ عبارت e به شکل نمادین ارزیابی می‌شود. اگر عبارت به شکل ضرب باشد، بعد از این که هر کدام از ورودی‌های ضرب به همین تابع به شکل بازگشتی داده شدند، بررسی می‌شود که آیا هیچ کدام از آنها مقدار عددی دارند یا نه. اگر هیچ کدام عدد نباشد پس حاصل عبارت ضرب مقدار خطی نخواهد بود. به همین دلیل متغیر

```
compare_and_update_stack(branch,k,stack) =
  if k < |stack| then
    if stack[k].branch ≠ branch then
      forcing_ok = 0
      raise an exception
    else if k = |stack| − 1 then
      stack[k].branch = branch
      stack[k].done = 1
  else stack = stack ^ ⟨(branch, 0)⟩
  return stack
```

شکل ۱۳: تابع compare_and_update_stack

```
solve_path_constraint(k_try, path_constraint, stack) =
  let j be the smallest number such that
      for all h with -1 ≤ j < h < k_try, stack[h].done = 1
  if j = -1 then
    return (0, _, _) // This directed search is over
  else
    path_constraint[j] = neg(path_constraint[j])
    stack[j].branch = ¬stack[j].branch
    if (path_constraint[0, ..., j] has a solution I') then
      return (1, stack[0..j], I + I')
    else
      solve_path_constraint(j, path_constraint, stack)
```

شکل ۱۴: تابع solve_path_constraint

all_linear صفر می‌شود. در عبارت‌های به شکل اشاره‌گر هم اگر عبارت بعد از بررسی، عددی نباشد متغیر all_locs_definite صفر می‌شود. حال اگر حلقه داخلی در شکل ۱۱ پایان یابد در صورتی که یکی از این متغیرها صفر شده باشند، DART در حلقه باقی می‌ماند در غیر این صورت برنامه به سلامت پایان می‌یابد. با توجه به این توضیحات، شرط در قسمت until باید not شود تا توضیحات مقاله درست باشند.

در شکل ۱۳ با استفاده از مقدار k شرط مربوطه بررسی می‌شود. اگر شاخه پیش‌بینی شده در مرحله قبل با شاخه کنونی برابر نباشد، خطا رخ داده است و استثنا برگردانده می‌شود. ولی اگر k، شماره شرط آخر در پشته باشد مقدار شاخه به‌روز می‌شود و مقدار متغیر done هم یک می‌شود. در غیر این صورت شرط k، شرط جدیدی است که به پشته اضافه می‌شود.

در شکل ۱۴ تابع تعداد شرط‌های موجود یعنی k، پشته و شروط مسیر را دریافت می‌کند و در صورت امکان سعی می‌کند با مکمل کردن آخرین قید در مسیر مقادیر جدیدی برای $\vec{I}$ بدست آید. با این شرایط مسیر جدیدی از برنامه اجرا خواهد شد و به این ترتیب ابزار سعی می‌کند تمام مسیرهای موجود در برنامه را پوشش دهد.

نتیجه‌گیری: برنامه P را در نظر بگیرید، اگر DART با اجرا روی P عبارت «Bug found» را چاپ کند، P به ازای یک ورودی خاص دارای خطا است. اگر DART بدون چاپ این عبارت پایان یابد، یعنی ورودی که به ازای آن خطا ایجاد شود وجود ندارد و DART تمام مسیرهای برنامه را طی کرده است در غیر این صورت DART حلقه بی‌نهایت باقی خواهد ماند.

```
1  foobar(int x, int y){
2    if (x*x*x > 0){
3       if (x>0 && y==10)
4          abort();
5    } else {
6       if (x>0 && y==20)
7          abort();
8    }
9  }
```

شکل ۱۵: مثال برای برتری DART بر روش‌های نمادین

مزیت‌های DART: نسبت به روش‌های ایستا که برای بررسی اشاره‌گرها لازم به تحلیل نام مستعار[۶۴]های اشاره‌گرها است، DART تنها مقدار حافظه برای اشاره‌گرها را بررسی می‌کند پس لازم به انجام تحلیل

[۶۴] Alias

نیست. همچنین DART دارای ویژگی صحت است؛ یعنی اگر بگوید کدی خطادار است پس حتما خطا دارد. ویژگی دیگر DART این است که می‌تواند محدودیت‌های حل‌کننده قیدها را تعدیل کند. در شکل ۱۵ در خط ۲ شرط موجود خطی نیست. طبق ادعای نویسندگان مقاله حل‌کننده قیدها در حل این گونه از شرط‌ها مشکل دارند و دلیل محدودیت روش‌های نمادین هم همین موضوع است. در DART عبارت عددی معادل برای این شرط تولید می‌شودو در نهایت با بررسی شرط خط ۳ با احتمال ½ می‌تواند خطای خط ۴ را بیابد.

۵-۱-۲DART برای زبان C

۵-۱-۲-۱ استخراج رابط

در برنامه‌های به زبان C رابط‌های خارجی این طور تعریف می‌شوند:

- مجموعه متغیرها و توابع خارجی (در صورت کامپایل کد به تنهایی به صورت «undefined» خطا می‌دهد.
- پارامترهایی که در تابع سطح بالا توسط خود کاربر مقداردهی می‌شوند.

برای استخراج این رابط‌ها یک تحلیل ایستای کد انجام می‌شود. ورودی‌ها به عنوان خانه‌های حافظه که مقداردهی اولیه نشده‌اند در نظر گرفته می‌شوند. با این تعریف لیست‌های پیوندی و داده ساختارها هم شامل می‌شوند که به عنوان ورودی‌های پویا در نظر گرفته می‌شوند. علاوه بر نگاشت یک مقدار به یک آدرس، نگاشت چند ورودی به یک آدرس و نگاشت یک ورودی به چند آدرس (malloc) هم پیاده‌سازی شده‌اند. برای هر ورودی یک نوع[۶۵] هم در نظر گرفته می‌شود. در C یک نوع پایه مثل int، نوع داده ساختار[۶۶] که مجموعه‌ای از نوع‌های پایه است، آرایه‌ها([]int) و اشاره‌گرها(* int) وجود دارند. همچنین انواع توابع به شکل توابع درون برنامه، توابع خارجی(تعامل با دیگر برنامه‌ها) و توابع کتابخانه‌ای(به عنوان توابع داخل برنامه در نظر گرفته می‌شود ولی تحلیل نمی‌شوند.) است.

۵-۱-۲-۲ ایجاد گرداننده آزمون

شکل ۱۶ نمونه کد ایجاد یک گرداننده آزمون برای تابع فرضی ac_controller آمده است. متغیر depth تعداد دفعاتی که تابع هدف

```
void main() {
    for (i=0; i < depth ; i++) {
        int tmp;
        random_init(&tmp,int);
        ac_controller(tmp);
    }
}
```

شکل ۱۶: گرداننده آزمون ایجاد شده برای تابع هدف ac_controller

[۶۵] Type
[۶۶] Struct

اجرا می‌شود را کنترل می‌کند. تابع random_init هم ورودی اولیه به تابع هدف را به شکل دلخواه تولید می‌کند.

```
random_init(m,type) {
  if (type == pointer to type2) {
     if (fair coin toss == head) {
        *m = NULL;
     } else {
        *m = malloc(sizeof(type));
        random_init(*m,type2);
     }
  } else if (type == struct) {
            for all fields f in struct
                random_init(&(m->f),typeof(f));
  } else if (type == array[n] of type3){
            for (int i=0;i<n;i++)
                random_init((m+i),type3);
  } else if (type == basic type) {
            *m = random_bits(sizeof(type));
  }
}
```

شکل ۱۷: تابع random_init

شکل ۱۷ تابع random_init را توصیف می‌کند. اگر نوع ورودی به تابع اشاره‌گر باشد، با احتمال ½ با null مقداردهی اولیه می‌شود. در غیر این صورت به آن مقدار داده می‌شود. اگر نوع ورودی آرایه یا داده ساختار باشد به ازای تمام ورودی‌ها همین تابع به صورت بازگشتی فراخوانی می‌شود. اگر نوع ورودی نوع پایه باشد هم با توجه به اندازه آن نوع، بیت‌های دلخواه به آن تخصیص پیدا می‌کند.

اگر درون تابع هدف، تابع خارجی دیگری فراخوانی شود، برای اینکه اجرا ادامه پیدا کند، توسط تابع random_init مقدار خروجی آن تابع تولید می‌شود.

برای ایجاد شکل ۱۰: تابع instrumented_program از CIL که یک تحلیل‌گر زبان C است استفاده می‌شود. برای حل‌کننده قید هم از lp_solve که قیدهای خطی طبیعی[۶۷] و حقیقی[۶۸] را می‌تواند حل کند، استفاده شده است.

در پیاده‌سازی ابزار هم تعدادی فرض در نظر گرفته شده است. اول؛ توابع خارجی اثر جانبی[۶۹] نباید داشته باشند. دوم؛ تمام ورودی‌ها مقداردهی اولیه می‌شوند. که شناسایی آنها توسط ابزارهای ایستا و پویا یا هر دو انجام می‌شوند.

در نهایت تحلیل ایستا به منظور استخراج روابط می‌تواند به شکل پویا انجام گیرد که لازمه آن بررسی نحوه دسترسی به حافظه است. علاوه بر آن نحوه برخورد و محاسبه توابع خارجی هم به عنوان مسائل چالشی در این حوزه مطرح هستند.

---

[۶۷] Integer

[۶۸] Real

[۶۹] Side effect

۵-۲ابزار CUTE

در سال ۲۰۰۵ Koushik Sen و دیگران ابزار CUTE(Concolic Unit Testing Engine) را برای آزمون برنامه‌های به زبان C ارائه کردند. در مقاله [۳] نویسندگان آزمون واحد[۷۰] را روشی برای آزمون قطعه‌ای[۷۱] رفتار نرم‌افزارها بیان کرده‌اند. در رابطه با روش‌های آزمون در این مقاله آمده است که روش آزمون دلخواه به دو دلیل افزونگی[۷۲] در انتخاب تکراری مقادیر و احتمال پایین انتخاب مقداری از مجموعه مقادیر فضای حالت که موجب ایجاد خطا می‌شود، کارایی لازم را ندارد. روش دیگر روش اجرای نمادین معرفی شده است. که در قطعه کدهای پیچیده یا بزرگ از نظر محاسباتی و همچنین حل قیدهای موجود در مسیر این روش، ناکارآمد معرفی شده است.

نویسندگان نزدیک‌ترین کار موجود به CUTE را کار [۲] ابزار DART معرفی می‌کنند که از اجرای Concolic بهره می‌برد. آنها معتقدند که DART در برنامه‌های که داده ساختارهای پویا دارند، در محاسبات اشاره‌گرها با مشکل مواجه می‌شود. DART دارای سه قسمت است:۱) تولید هدفمند[۷۳] ورودی های آزمون ۲) استخراج خودکار رابط‌ها از کد برنامه ۳) تولید دلخواه ورودی‌های آزمون. CUTE استخراج رابط‌ها را به صورت خودکار انجام نمی دهد و آن را به کاربر واگذار می‌کند تا مشخص کند کدام تابع با چه پیش شرط‌هایی مرتبط است. DART هر تابع را به شکل تنها و ایزوله مورد آزمون قرار می‌دهد ولی CUTE توابع مرتبط را هم در نظر می‌گیرد مثلا پیاده‌سازی داده ساختارها. DART قیدهای مربوط به مقادیر طبیعی(int) را فقط محاسبه می‌کند و قیدهای مرتبط با اشاره‌گرها و داده ساختارها را نمی تواند محاسبه کند. DART در مورد گراف حافظه از روشی ساده استفاده می‌کند، هر اشاره‌گر یا null است یا به خانه دیگری اشاره می‌کند که به صورت بازگشتی آن خانه مقداردهی اولیه می‌شود. این روش یکسری مشکلات دارد:

- تولید دلخواه ممکن است هیچ‌گاه پایان نیابد.
- تولید دلخواه فقط می‌تواند درخت تولید کند؛ توانایی تولید حلقه یا DAG را ندارد.
- تولید هدفمند قیدهای مربوط به اشاره‌گرها را دنبال نمی‌کند.

---

[۷۰] Unit Testing
[۷۱] Modular
[۷۲] Redundancy
[۷۳] Directed

- تولید هدفمند هیچ‌گاه گراف حافظه را تغییر نمی‌دهد. فقط می‌تواند مقادیر پایه در هر گره از گراف را تغییر دهد.

در این مقاله روشی برای نمایش و حل قیدهای مربوط به اشاره‌گرها ارائه شده است. در این روش از مفهوم نگاشت منطقی ورودی‌ها استفاده می‌شود که می‌توان گراف‌های حافظه به عنوان مجموعه‌ای از متغیرهای نمادین را نشان داد و سپس قیدهای مربوط به آنها که در اجرای نمادین تولید می‌شوند را حل کرد. در این روش تعدادی تابع به کد برنامه اضافه می‌شوند تا اجرای نمادین ممکن شود. از نگاشت منطقی ورودی‌ها (I) برای تولید ورودی‌های عددی دلخواه استفاده می‌شود. همچنین همزمان دو حالت نمادین یکی برای برای اشاره‌گرها و دیگری برای مقادیر عددی عادی نگهداری می‌شود. سپس مانند DART اجرا صورت می‌گیرد. مهمترین نوآوری این کار جداسازی اشاره‌گرها از مقادیر عادی است. حل‌کننده قید در CUTE هم برای مقادیر عادی و هم اشاره‌گرها استفاده می‌شود.

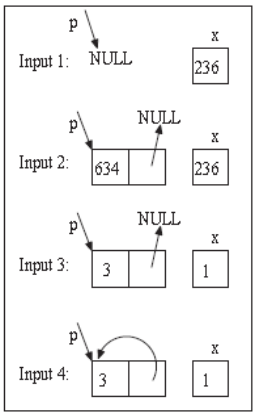
۵-۲-۱کلیات طرح

قبل از ورود به توضیح ابزار، مثالی از چگونگی آزمون برنامه ارائه می‌شود. در شکل ۱۸: نمونه‌ای از کدی به زبان C به همراه ورودی‌هایی که CUTE برای آزمون تولید می‌کند، آمده است. تابع testme یک اشاره‌گر به داده ساختار cell و یک مقدار طبیعی به عنوان ورودی می‌گیرد. ابتدا مقدار اشاره‌گر p با null و x با ۲۳۶ مقداردهی اولیه می‌شود. اگر مقدار نمادین p با p0 و مقدار نمادین x با x0 نشان داده شود، قید مسیر تولید شده تا دومین شرط به شکل (x0>0) and (p==null) در می‌آید. برای تولید ورودی که شاخه then از شرط دوم را اجرا کند عبارت (x0>0) and (p!=null) به حل‌کننده قید داده می‌شود. CUTE باید p را به یک cell اشاره دهد که دو مقدار p->v و p->next خواهد داشت. CUTE برای اشاره‌گر p->next مقدار null و برای p->v مقدار دلخواه ۲۳۴ اختصاص می‌دهد. پس مقدار ورودی جدید به شکل p->v=234 و x=236 خواهد بود. به همین ترتیب اجرا ادامه پیدا می‌کند تا با ورودی p->v=3 و x=1 و p->next=p دستور ERROR کشف خواهد شد.

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int
f(int v) {
  return 2*v + i;
}

int
testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
  return 0;
}
```



شکل ۱۸: نمونه‌ای از کدی به زبان C به همراه ورودی‌هایی که CUTE برای آزمون تولید می‌کند

برای اینکه CUTE مسیرهای مختلف از برنامه را اجرا کند، ابتدا کد برنامه مورد آزمون را مجهز به یکسری تابع می‌کند. سپس نگاشت منطقی از ورودی‌ها (I) را تولید می‌کند که می‌تواند گراف حافظه را به شکل نمادین نمایش دهد. سپس CUTE کد تجهیزشده را مانند زیر اجرا می‌کند:

- ابزار از I برای تولید ورودی‌های عددی به گراف حافظه برنامه و دو حالت نمادین استفاده می‌کند که یکی برای مقادیر اشاره‌گرها و دیگری برای مقادیر پایه است.
- ابزار سپس کد را با مقادیر ورودی اجرا کرده و قیدهای موجود در مسیر را جمع‌آوری می‌کند.
- ابزار یکی از قیدهای موجود در مسیر را مکمل کرده و آن را حل می‌کند تا نگاشت منطقی ورودی‌های I' را که مشابه I است ولی مسیر جدیدی را طی می‌کند، تولید شود. سپس I=I' قرار می‌دهد و فرایند را تکرار می‌کند.

در ادامه ابتدا نگاشت منطقی ورودی‌ها توضیح داده می‌شود سپس مدل برنامه و واحدهای آن، فرایند تجهیزسازی کد، اجرای Concolic، حل‌کننده قید و داده ساختارها مورد بررسی قرار می‌گیرند.

۵-۲-۱-۱ نگاشت منطقی ورودی‌ها

CUTE برای دنبال کردن گراف حافظه از نگاشت منطقی ورودی‌ها (I) استفاده می‌کند. علت استفاده از I این است که آدرس‌های حافظه ممکن است در هر اجرا تغییر کرده و مقدار جدیدی به خود بگیرند. اگر N مجموعه اعداد طبیعی و V مجموعه مقادیر پایه باشد: $I: N \to N \cup V$. هر آدرس منطقی مثل l دارای نوع است. یک نوع می‌تواند * T، اشاره‌گر به نوع T باشد (خود T می‌تواند یک نوع پایه یا داده‌ساختار باشد.) یا یک نوع پایه باشد. typeOf(l) نوع l را بر می‌گرداند و sizeOf(T) اندازه آن نوع را برمی‌گرداند. اگر $typeOf(l) = T *$ $and\ I(l) \neq null$ باشد، آنگاه ترتیب $I(v), \dots, I(v+n-1)$ مقادیری که l به آنها اشاره می‌کند را نگهداری می‌کند که I(l)=v وsizeOf(T)=n است. برای مثال ورودی شماره ۳ در شکل ۱۸ به شکل <3,1,3,0> و ورودی شماره ۴ به شکل <3,1,3,3> در می‌آید.

۵-۲-۱-۲ واحدها و مدل برنامه

یک قطعه کد ممکن است از تعدادی تابع تشکیل شده باشد. CUTE از کاربر می‌خواهد تا یکی از تابع‌ها را به عنوان تابع ورود «entry function» انتخاب کند. CUTE تابع main را ایجاد می‌کند که در آن ابتدا

$P ::= Stmt^*$ $\quad Stmt ::= [l:]\ S$
$S ::= lhs \leftarrow e \mid \texttt{if } p \texttt{ goto } l' \mid \texttt{START} \mid \texttt{HALT} \mid \texttt{ERROR}$
$lhs ::= v \mid *v$
$e ::= v \mid \&v \mid *v \mid c \mid v\ op\ v \mid input()$
where $op \in \{+, -, /, *, \%, \dots\}$,
v is a variable, c is a constant
$p ::= v = v \mid v \neq v \mid v < v \mid v \leq v \mid v \geq v \mid v > v$

شکل ۱۹: گرامر زبان ساده شده C

ورودی‌ها مقداردهی اولیه می‌شوند سپس تابع ورود فراخوانی می‌شود. CUTE از CIL برای تبدیل کد برنامه به کد ساده شده C استفاده می کند که در شکل ۱۹ گرامر آن آمده است.

۵-۲-۱-۳ تجهیز کد

CUTE برای تجهیز کد ورودی از تبدیلات موجود در شکل ۲۰ استفاده می‌کند. کد ابتدا توسط روابط بیان شده تجهیز می‌شود سپس I مقداردهی اولیه شده و حلقه اصلی اجرای کد اجرا می‌شود که در شکل ۲۱ نمای کلی از آن آمده است.

Before Instrumentation	After Instrumentation
// program start START	$global\ vars\ \mathcal{A} = \mathcal{P} = path\_c = M = [\,]$; $global\ vars\ i = inputNumber = 0$; START
// inputs $v \leftarrow input()$;	$inputNumber = inputNumber+1$; $initInput(\&v, inputNumber)$;
// inputs $*v \leftarrow input()$;	$inputNumber = inputNumber+1$; $initInput(v, inputNumber)$;
// assignment $v \leftarrow e$;	$execute\_symbolic(\&v, \text{“}e\text{”})$; $v \leftarrow e$;
// assignment $*v \leftarrow e$;	$execute\_symbolic(v, \text{“}e\text{”})$; $*v \leftarrow e$;
// conditional if (p) goto l	$evaluate\_predicate(\text{“}p\text{”}, p)$; if (p) goto l
// normal termination HALT	$solve\_constraint()$; HALT;
// program error ERROR	**print** “Found Error” ERROR;

شکل ۲۰: کدهایی که CUTE برای تجهیز برنامه ورودی اضافه می‌کند

```
// input: P is the instrumented program to test
//        depth is the depth of bounded DFS
run_CUTE(P,depth)
  I = [ ]; h = (number of arguments in P) + 1;
  completed=false; branch_hist=[ ];
  while not completed
    execute P
```

شکل ۲۱: شبه کد اجرای آزمون روی کد تجهیز شده

۵-۲-۱-۴ اجرای Concolic

در اجرای Concolic دو حالت نمادین A وP وجود دارند. که A مکان‌های حافظه را به عبارت‌های نمادین حسابی و P به عبارات اشاره‌گر نمادین نگاشت می‌کند. قیدهای حسابی به شکل $a_1x_1 + \cdots + a_nx_n + c \bowtie 0$

است که a_iها و c مقادیر عددی و x_iها مقادیر نمادین هستند که $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$ است. قیدهای مربوط به اشاره‌گرها هم به شکل $x \cong y$ است که $\cong \in \{\neq, =\}$ است. برای هر نگاشت M عبارت $M' = M[m \to v]$ یعنی همان M که M'(m)=v است. $M' = [M - m]$ یعنی همان M که M'(m) در آن تعریف نشده است و $m \in domain(M)$ یعنی M(m) تعریف شده است.

۵-۲-۱-۴-۱ مقداردهی ورودی به وسیله نگاشت منطقی ورودی

در شکل ۲۲ تابع initInput(m,l) که از I برای مقداردهی اولیه مکان حافظه m استفاده می‌کند تا حالات A و P و مقادیر موجود در I را به‌روز کند. در این تابع h آدرس منطقی در دسترس برای متغیر جدید را نگهداری می‌کند. M نگاشت آدرس منطقی به فیزیکی، m آدرس فیزیکی که مقداردهی اولیه می‌شود و l آدرس منطقی مورد نظر برای m است.

```
// input: m is the physical address to initialize
//        l is the corresponding logical address
// modifies h, I, A, P
initInput(m, l)
  if l ∉ domain(I)
    if (typeOf(*m) ==pointer to T) *m =NULL;
    else *m =random();
    I = I[l ↦ *m];
  else
    v' = v = I(l);
    if (typeOf(v) ==pointer to T)
      if (v ∈ domain(M))
        *m = M(v);
      else
        n = sizeOf(T);
        {m1, ..., mn} =malloc(n);
        if (v ==non-NULL)
          v' = h; h = h + n; // h is the next logical address
        *m = m1; I = I[l ↦ v']; M = M[v ↦ m1];
        for j = 1 to n
          initInput(mj, v' + j - 1);
    else
      *m = v; I = I[l ↦ v];
  // xl is a symbolic variable for logical address l
  if (typeOf(m) ==pointer to T) P = P[m ↦ xl];
  else A = A[m ↦ xl];
```

شکل ۲۲: تابع initInput

اگر m دارای نوع اشاره‌گر باشد با null مقداردهی می‌شود در غیر این صورت به صورت دلخواه مقداردهی می‌شود.

اگر m پیش از این در I تعریف نشده باشد، اگر نوعش اشاره‌گر باشد و آدرس فیزیکی برای آن تعریف شده باشد، مقدار موجود در M(v) به m اختصاص داده می‌شود. اگر نه به اندازه طول نوع آن آدرس فیزیکی به آن اختصاص داده می‌شود، مقادیر I و M به‌روز می‌شوند و برای هر خانه جدید به صورت بازگشتی تابع فراخوانی می‌شود. در نهایت m به P و A اضافه می‌شود.

۵-۲-۱-۴-۲ اجرای نمادین

برای اجرای نمادین تابع شکل ۲۶ اجرا می‌شود. E عبارتی است که قرار است به صورت نمادین اجرا شود. نکته مهم در این تابع این است که در عبارت ضرب (خط L) یکی عوامل با مقدار حسابی جایگزین می‌شود

تا عبارت غیرخطی به وجود نیاید. اگر بیارت برای مثال به شکل y''=y/y' باشد و هر دوی y و y' نمادین باشند، در خط D این آدرس هم از P و هم از A حذف می‌شود تا این عبارت تعریف نشده بماند. شکل ۲۶ هم نحوه ارزیابی عبارات شرطی را بیان می‌کند.

```
// inputs: p is a predicate to evaluate
//         b is the concrete value of the predicate in S
// modifies path_c, i
evaluate_predicate(p, b)
 if (i ≤ depth)
    match p:
      case "v1 ⋈ v2": // where ⋈ ∈ {<, ≤, ≥, >}
        m1 = &v1; m2 = &v2;
        if (m1 ∈ domain(A) and m2 ∈ domain(A))
          c = "A(m1) − A(m2) ⋈ 0";
        else if (m1 ∈ domain(A))
          c = "A(m1) − v2 ⋈ 0";
        else if (m2 ∈ domain(A))
          c = "v1 − A(m2) ⋈ 0";
        else c = b;
      case "v1 ≅ v2": // where ≅ ∈ {=, ≠}
        m1 = &v1; m2 = &v2;
        if (m1 ∈ domain(P) and m2 ∈ domain(P))
          c = "P(m1) ≅ P(m2)";
        else if (m1 ∈ domain(P) and v2 ==NULL)
          c = "P(m1) ≅ NULL";
        else if (m2 ∈ domain(P) and v1 ==NULL)
          c = "P(m2) ≅ NULL";
        else if (m1 ∈ domain(A) and m2 ∈ domain(A))
          c = "A(m1) − A(m2) ≅ 0";
        else if (m1 ∈ domain(A)) c = "A(m1) − v2 ≅ 0";
        else if (m2 ∈ domain(A)) c = "v1 − A(m2) ≅ 0";
        else c = b;
    if (b) path_c[i] = c;
    else path_c[i] =neg(c);
 cmp_n_set_branch_hist(b);
 i = i + 1;
```

شکل ۲۶: تابع evaluate_predicate

```
// inputs: m is a memory location
//         e is an expression to evaluate
// modifies A and P by symbolically executing *m ← e
execute_symbolic(m, e)
 if (i ≤ depth)
  match e:
    case "v1":
      m1 = &v1;
      if (m1 ∈ domain(P))
        A = A − m; P = P[m ↦ P(m1)]; // remove if A contains m
      else if (m1 ∈ domain(A))
        A = A[m ↦ A(m1)]; P = P − m;
      else P = P − m; A = A − m;
    case "v1 ± v2": // where ± ∈ {+, −}
      m1 = &v1; m2 = &v2;
      if (m1 ∈ domain(A) and m2 ∈ domain(A))
        v = "A(m1) ± A(m2)"; // symbolic addition or subtraction
      else if (m1 ∈ domain(A))
        v = "A(m1) ± v2"; // symbolic addition or subtraction
      else if (m2 ∈ domain(A))
        v = "v1 ± A(m2)"; // symbolic addition or subtraction
      else A = A − m; P = P − m; return;
      A = A[m ↦ v]; P = P − m;
    case "v1 * v2":
      m1 = &v1; m2 = &v2;
      if (m1 ∈ domain(A) and m2 ∈ domain(A))
L:      v = "v1 * A(m2)"; // replace one with concrete value
      else if (m1 ∈ domain(A))
        v = "A(m1) * v2"; // symbolic multiplication
      else if (m2 ∈ domain(A))
        v = "v1 * A(m2)"; // symbolic multiplication
      else A = A − m; P = P − m; return;
      A = A[m ↦ v]; P = P − m;
    case "*v1":
      m2 = v1;
      if (m2 ∈ domain(P)) A = A − m; P = P[m ↦ P(m2)];
      else if (m2 ∈ domain(A)) A = A[m ↦ A(m2)]; P = P − m;
          else A = A − m; P = P − m;
    default:
D:    A = A − m; P = P − m;
```

شکل ۲۶: تابع execute_symbolic

```
// modifies branch_hist, I, completed
solve_constraint() =
  j = i − 1;
  while (j ≥ 0)
    if (branch_hist[j].done == false)
      branch_hist[j].branch= ¬branch_hist[j].branch;
      if (∃I' that satisfies neg_last(path_c[0 . . . j]))
        branch_hist=branch_hist[0 . . . j];
        I = I';
        return;
      else j = j − 1;
    else j = j − 1;
  if (j < 0) completed=true;
```

شکل ۲۶: تابع solve_constraint

```
// modifies branch_hist
cmp_n_set_branch_hist(branch)
    if (i < |branch_hist|)
        if (branch_hist[i].branch≠branch)
            print "Prediction Failed";
            raise an exception; // restart run_CUTE
        else if (i == |branch_hist| − 1)
            branch_hist[i].done =true;
    else branch_hist[i].branch = branch;
        branch_hist[i].done = false;
```

شکل ۲۶: تابع cmp_n_set_branch_hist

۵-۲-۱-۴-۳ حل‌کننده قیود

در این مقاله روی lp_solve که خود یک حل کننده قیود خطی است، یک حل‌کننده قید ارائه شده است. حل‌کننده قید ارائه شده از سه جنبه بهبود پیدا کرده است:

- بررسی سریع ارضاناپذیری: اگر قید آخر مکمل یکی از قیدهای قبلی است، بررسی برای یافتن حل صورت نمی‌پذیرد.
- شناسایی و حذف زیر قیدهای معمول
- حل افزایشی: دو قید p و p' به هم وابسته‌اند اگر یکی از دو شرط زیر برقرار باشد:
  - $vars(p) \cap vars(p') \neq \emptyset$
  - قیدی مانند p'' وجود داشته باشد که p و p'' به هم وابسته باشند و p' و p'' هم به هم وابسته باشند.

حال C' قیود مسیر جدید باشد که آخرین قید به شکل $\neg path\_c[i]$ است. مجموعه D مجموعه تمام قیود وابسته به آخرین قید در C' یعنی $\neg path\_c[i]$ است. ابتدا حل I'' را برای مجموعه D پیدا می‌کند. سپس I'=I[I''] را به عنوان حل برای C' در نظر می‌گیرد. این عبارت یعنی اگر متغیری در I'' تعریف شده است حل از I'' و در غیر این صورت از I دریافت می‌شود. طول D معمولا یک‌هشتم C' است و این موضوع حل را سرعت می‌بخشد.

برای قیدهای مربوط به اشاره‌گرها که تنها برابری یا عدم برابری در آن مطرح است، این گونه عمل می‌کند: مجموعه تمام مقادیری که با x برابرند را با $[x]_=$ نشان داده می‌شود. گرافی از روابط میان اشاره‌گرها ایجاد می‌شود که در آن گره‌ها کلاس‌ها (مثل $[x]_=$) وجود دارند. یال میان گره‌ها هم نشان دهنده عدم برابری دو کلاس است. حال $\neg path\_c[i]$ قابل ارضاست اگر برای قید به شکل $x = y$ هیچ یالی میان دو کلاس $[x]_=$ و $[y]_=$ وجود نداشته باشد یا اینکه برای قید به شکل $x \neq y$، $[x]_= \neq [y]_=$ باشد. حال در صورت قابلیت ارضای قید آخر، از تابع شکل ۲۷ قید دارای اشاره‌گر حل می‌شود.

۵-۲-۱-۴-۴ آزمون ساختمان داده‌ها

دو رویکرد در این مورد وجود دارد: ۱) ترکیبی از فراخوانی‌ها به توابع موجود در آن ساختمان داده (مثل add، remove و ...) ایجاد می‌شود. ۲ شرط در این زمینه وجود دارد: اول اینکه همه توابع در دسترس باشند و

دوم اینکه از همه توابع در آزمون استفاده شود. ۲) در این حالت از توابعی که برای بررسی درستی ساختمان داده‌ها استفاده می‌شود که در بعضی از کتابخانه‌های زبان C مثل SGLIB وجود دارند. مزیت این روش این است که توابع را به تنهایی هم می‌توان مورد آزمون قرار داد.

```
// inputs: p is a symbolic pointer predicate
//         I is the previous solution
// returns: a new solution I''
solve_pointer(p, I)
  match p:
    case "x ≠NULL": I'' = {y ↦non-NULL| y ∈ [x]=};
    case "x =NULL": I'' = {y ↦NULL| y ∈ [x]=};
    case "x = y": I'' = {z ↦ v | z ∈ [y]= and I(x) = v};
    case "x ≠ y": I'' = {z ↦ non-NULL| z ∈ [y]=};
  return I'';
```

شکل ۲۷: تابع solve_pointer

۵-۴-۱-۲-۵ بحث

CUTE در مورد اشاره‌گرها از یک ساده‌سازی استفاده کرده است و آن این بود که تنها رابطه تساوی و نابرابری برای اشاره‌گرها در قیدها مطرح می‌شود. برای نشان دادن آرایه‌ها هم از خانه‌های حافظه مجزا و پشت سر هم استفاده کرده است. همین موضوع باعث شده است در کدهایی مثل $*p = 0; *q = 1; if(*p = 1)Error;$ (فرض: p و q برابر با null نیستند.) نتواند قید p==q را تولید کند یا در مثال $a[i] = 0;\ a[j] = 1; if(a[i] = 1)Error;$ قید i==j را تولید کند.

۵-۳ابزار EXE

در سال ۲۰۰۶ Cristian Cadar و دیگران ابزار EXE(EXecution generated Execution) را برای آزمون برنامه‌های به زبان C ارائه کرده‌اند. ابزار ارائه شده از روش EGT برای آزمون استفاده می‌کند. از آنجایی که روش EGT مانند روش Concolic از روش‌های نوین در اجرای نمادین در آزمون نرم‌افزارها است، در ادامه ابزار EXE و بعد از آن KLEE مورد مطالعه قرار می‌گیرند تا از ایده‌ها و روش‌های آنها در پژوهش آتی استفاده شود.

EXE در اجرای یک مسیر از برنامه موثر عمل می‌کند. چون این ابزار از STP (یک حل‌کننده قید) استفاده می‌کند، علاوه بر مقادیر عددی ورودی، تمام مقادیر ممکن که به ازای آن ممکن است آن مسیر اجرا شود را در نظر می‌گیرد. برای مثال ابزاری مثل Purify زمانی خطای خارج از محدوده[۷۴] را اعلام می‌کند که مقدار ورودی آرایه از محدوده تعریف شده بیشتر شود ولی EXE هر مقدار ممکنی که به ازای آن این خطا ایجاد شود را گزارش می‌دهد. برای یک عبارت حسابی اگر عمل‌گر تقسیم یا باقی‌مانده وجود داشته باشد، EXE مقداری از متغیر نمادین که به ازای آن خطای تقسیم بر صفر ایجاد شود را گزارش می‌دهد. همچنین EXE دارای توانایی بکارگیری اشاره‌گرها، اجتماع، متغیرهای بیتی، عملیات بیتی مثل شیف و تبدیل نوع‌ها[۷۵] است.

نویسندگان مقاله در رابطه با مزیت استفاده از روش‌هایی که به صورت پویا موردآزمون[۷۶] تولید می‌کنند، این موارد را بیان می‌کنند: ۱) هر کد ورودی را می‌توان آزمود. ۲) ایجاد حملات واقعی ۳) عدم وجود مثبت کاذب. همچنین در مورد استفاده از STP می‌گویند که: ۱) از تمام نوع‌های موجود در C به جز float پشتیبانی می‌کند. ۲) سرعت و دقت بالا همچنین می‌تواند در مورد اشاره‌گرهایی که با یک متغیر همراه هستند هم نظر بدهد مثل a[i]=0.

۵-۳-۱ کلیات طرح

در این قسمت قدم به قدم ویژگی‌های EXE بیان می‌شوند. برای این منظور از شکل ۲۹ به عنوان کد ورودی استفاده می‌شود که خروجی تولید شده توسط EXE در شکل ۲۹ آمده است. EXE تمام مسیرهای

[۷۴] Out of bound
[۷۵] Casts
[۷۶] Test Case

ممکن را پیمایش می‌کند و دو خطا یکی در خط ۱۲، خطای خارج زدن از محدوده و دیگری در خط ۱۶، خطای تقسیم بر صفر را پیدا می‌کند.

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p - 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

شکل ۲۹: کد ورودی به EXE

```
% exe-cc simple.c
% ./a.out
% ls exe-last
  test1.forks test2.out     test3.forks  test4.out
  test1.out   test2.ptr.err test3.out    test5.forks
  test2.forks test3.div.err test4.forks  test5.out
% cat exe-last/test3.div.err
  ERROR: simple.c:16 Division/modulo by zero!
% cat exe-last/test3.out
  # concrete byte values:
  0 # i[0]
  0 # i[1]
  0 # i[2]
  0 # i[3]
% cat exe-last/test3.forks
  # take these choices to follow path
  0 # false branch (line 5)
  0 # false (implicit: pointer overflow check on line 9)
  1 # true (implicit: div-by-0 check on line 16)
% cat exe-last/test2.out
  # concrete byte values:
  2 # i[0]
  0 # i[1]
  0 # i[2]
  0 # i[3]
```

شکل ۲۹: خروجی EXE برای کد شکل ۲۹

برای آزمون یک قطعه کد مراحل زیر باید طی شود:

۱. آن دسته از متغیرهایی که قرار است به صورت نمادین با آنها برخورد شود که معمولا ورودی‌ها هستند، باید مشخص شوند. برای این منظور از تابع make_symbolic(&i) استفاده می‌شود.
۲. کد باید توسط کامپایلر EXE کامپایل شود. کامپایلر EXE ابتدا به وسیله CIL کد ورودی را ساده می‌کند سپس توسط یک کامپایلر عادی مثل gcc کد را کامپایل می‌کند.
۳. وقتی برنامه اجرا می‌شود تمام مسیرهای ممکن پیمایش می‌شوند و همه قیود مسیر استخراج می‌شوند. وقتی مسیر پایان یافت قیدها به STP داده می‌شود تا مقادیر عددی برای آن مسیر تولید شود.

یک مسیر زمانی پایان می‌یابد که ۱) ()exit فراخوانی شود. ۲) خراب[۷۷] شود. ۳) یک assertion شکست بخورد. ۴) EXE خطایی بیابد.

کامپایلر EXE سه کار اصلی دارد. اول، در اطراف هر عبارت یکسری کد اضافه می‌کند تا معین کند که عملوندهای آن عددی هستند یا نمادین. یک متغیر عددی است اگر تمام بیت‌هایش عددی باشند. اگر تمام متغیرهای یک عبارت عددی باشد آن عبارت عددی خواهد بود. اگر یکی از متغیرهای عبارت نمادین باشد کل عبارت به عنوان یک قید برای مسیر جاری به STP داده می‌شود. همچنین EXE با حافظه به صورت خانه‌های بدون نوع برخورد می‌کند. برای مثال خط ۸ عبارت $p = (char *)a + i * 4$ وجود دارد. EXE بررسی می‌کند که متغیرهای a و i عددی هستند یا نه. در صورت عددی بودن مقدار عبارت محاسبه شده و در p ریخته می‌شود. اگر i نمادین باشد قید $(p = (char *)a + i * 4)$ اضافه می‌شود.

دوم، اضافه کرد دستور ()fork در عبارت‌های شرطی تا هر دو مسیر از برنامه اجرا شوند و در نهایت تمام مسیرهای موجود پیمایش شوند. هرگاه قید جدیدی اضافه می‌شود ابتدا پرس‌وجویی به STP فرستاده می‌شود تا مشخص شود که آیا برای آن قید راه‌حلی وجود دارد یا نه. در صورت عدم وجود راه‌حل EXE اجرا را متوقف می‌کند.

سوم، کامپایلر کدهایی به برنامه اضافه می‌کند تا در برخورد با عبارت‌های نمادین بررسی کنند که آیا مقادیری ممکن وجود دارد که باعث ایجاد خطا شود یا نه. خطاهایی که بررسی می‌شوند ۱) خطای ارجاع به آدرس حافظه null یا خارج از محدوده ۲) خطای تقسیم بر صفر.

اگر EXE همه قیدها را داشته باشد و STP هم توانایی حل آنها را داشته باشد، اگر به ازای ورودی خاصی خطایی امکان وقوع داشته باشد، EXE آن را می‌تواند پیدا کند.اگر هم به ازای هیچ ورودی خطا رخ ندهد مسیر امن خواهد بود.

همان طور که گفته شد EXE دو خطا در کد می‌یابد اول، از آنجایی که مقدار p به ازای i=2 می‌تواند ۴ شود در نتیجه در خط ۱۲ امکان خطای خارج از محدوده وجود دارد. همچنین به ازای i=0 مقدار مخرج در عبارت خط ۱۶ می‌تواند صفر شود و در اینجا خطای تقسیم بر صفر گزارش می‌شود.

[۷۷] Crash

۵-۳-۱-۱ ویژگی‌های STP

EXE برای بررسی و حل قیدها از STP استفاده می‌کند. به طور دقیق‌تر STP یک رویه تصمیم[۷۸] است. رویه تصمیم برنامه‌ای است که ارضاپذیری فرمولهای منطقی را مشخص می‌کند. برای طراحی STP از حل‌کننده‌های SAT[۷۹] استفاده شده است. STP ورودی را با استفاده از مفاهیم ریاضی و منطق با پیش‌پردازش تبدیل به منطق گزاره‌ای می‌کند سپس این عبارت‌ها را با miniSAT حل می‌کند.

STP حافظه را به شکل بایت‌های بدون نوع در نظر می‌گیرد. از ۳ نوع بول، بردار بیتی و آرایه‌ای از بردارهای بیتی پشتیبانی می‌کند. مثلا 0010 یک بردار بیتی ۴تایی است. STP از نوع float در C پشتیبانی نمی‌کند. از تمام عملیات ریاضی مثل جمع و تفریق و حتی عملیات غیرخطی مثل تقسیم پشتیبانی می‌کند. در STP دو نوع عبارت وجود دارد یکی مثل x+y که Term گفته می‌شود و دیگری x<y که فرمول است. در پیاده‌سازی، Termها به عملیات بولی بیتی تبدیل می‌شوند. مثلا جمع با جمع‌کننده نقلی آبشاری پیاده‌سازی شده است. فرمول‌ها هم به عملیات DAG بیتی تبدیل می‌شوند.

۵-۳-۱-۲ تبدیل کد C به قیدهای STP

EXE هر داده نمادین را به شکل آرایه‌ای از بردارهای بیتی ۸تایی نشان می‌دهد. مزیت این کار این است که بلوک‌های حافظه در C هم به همین روش نشان داده می‌شوند و این امر تعریف قیدها را آسان می‌کند.

EXE برای حل قیدها از STP استفاده می‌کند، اول در حافظه، آدرس‌های حافظه به متغیرهای نمادین را می‌یابد سپس عبارت‌ها را به شکل بردارهای بیتی تبدیل می‌کند.

ابتدا در کد هیچ مقدار نمادینی وجود ندارد. هنگامی که کاربر یک متغیر را به عنوان نمادین مشخص می‌کند. مثلا برای b، EXE با فراخوانی STP، b_{sym} را که دقیقا مانند b است را ایجاد می‌کند. سپس در جدول نگاشت‌ها این دو را به هم نگاشت می‌دهد. مثلا برای i در کد شکل ۲۹ i_{sym} به اندازه ۳۲ بیت (۴ بایت) ایجاد می‌شود.

با اجرای برنامه جدول نگاشت‌ها تغییر می‌کند:

[۷۸] Decision Procedure
[۷۹] SAT Solver

۱. v=e جایی که e عبارت نمادین است یعنی حداقل یک مقدار نمادین در آن وجود دارد. EXE، e_{sym} را ایجاد می‌کند و &v را به آن نگاشت می‌دهد. برای v مقدار جدیدی در STP ایجاد نمی‌شود. بلکه هر جا v بود با e_{sym} جایگزین می‌شود. اگر v دوبار نوشته شود یا از حافظه حذف شود، نگاشت در جدول حذف می‌شود.

۲. b[e] که e عبارت نمادین و b عددی است. از آنجایی که STP باید در مورد مقادیر مختلف موجود در b تصمیم بگیرد، b_{sym} درست به اندازه و b و مقادیر موجود در b ایجاد می‌شود. (مثل خط ۱۲). a[*p] برای a داریم: $a_{sym} = \{1, 0, 0, 0, 3, 0, 0, 0, 5, 0, 0, 0, 2, 0, 0, 0\}$

برای خواندن l به طول n از حافظه ابتدا بررسی می‌شود که آیا b عددی است یا نمادین. اگر عددی بود مقدار عددی l به خروجی داده می‌شود. اگر نمادین بود، l به اجزای تشکیل دهنده‌اش که n تا بردار ۸ بیتی است تقسیم می‌شود. برای هر قسمت اگر عددی بود با مقدار خودش و اگر نمادین بود با $(b_i)_{sym}$ جایگذاری می‌شود. مثلا برای عبارت (char *)a+4*i، ش عددی است ولی i نمادین است. i یک int است پس از ۴ بایت تشکیل شده است که با $i_{sym}[0]$ تا $i_{sym}[3]$ نشان داده می‌شود. ۴ هم عددی است و به شکل باینری نشان داده می‌شود. داریم: (@ نشانه اتصال[۸۰] دو عبارت است.)

$$a + (i_{sym}[3]@i_{sym}[2]@i_{sym}[1]@i_{sym}[0]) * 0...00000100$$

محدودیت STP در این است که از اشاره‌گرها پشتیبانی نمی‌کند برای محاسبه p* ابتدا بلوک b که p به آن اشاره می‌کند را می‌یابد، مقدار b_{sym} که مربوط به b است را از STP استخراج می‌کند و سپس مکان p را در آن بلوک پیدا می‌کند. مشکل اصلی در محاسبه p** است چون باید ابتدا p* را محاسبه و ثابت نگه دارد و سپس p** را محاسبه کند.

۲-۳-۵ تعدادی از بهینه‌سازی‌های EXE

۱. استفاده از روش کش[۸۱] برای جلوگیری از محاسبه عبارت‌های تکراری توسط STP.

۲. در ارزیابی ارضاپذیری قیدها، بررسی می‌شود که آیا در یک عبارت از قیدها، زیر قیدهایی موجود هستند که از هم مستقل باشند یا نه. منظور از زیر قیدهای مستقل آن زیرقیدهایی است که هیچ متغیر مشترکی نداشته باشند. EXE زیر قیدهای مستقل را یافته و آنها را جداگانه بررسی می‌کند. اول آن زیر

[۸۰] Concatanation

[۸۱] Cache

قیدهایی که ارتباطی در حل مسئله ندارند را می‌تواند حذف کند. دوم اینکه از کش در محاسبه زیرقیدها بهتر و بیشتر می‌تواند استفاده کند چون معمولا زیرقیدها تکراری می‌شوند.

۳. هیوریستیک‌های جست‌وجو: وقتی EXE دستور fork را فراخوانی می‌کند، پردازه[۸۲] جدیدی به برنامه اضافه می‌شود. EXE باید تصمیم بگیرد که کدام شاخه از برنامه را اجرا کند. در حالت عادی و بدون بهینه‌سازی EXE از جست‌وجوی عمق اول استفاده می‌کند و یک شاخه را تا انتها اجرا می‌کند. مشکل این روش حلقه‌های بی‌نهایت است. برای مقابله با این مشکل در EXE از یک هیوریستیک استفاده می‌شود که ترکیب جست‌وجوی عمق اول و بهترین-اولین (BFS) است. هنگامی که fork فراخوانی می‌شود، اطاعات هر پردازه مثل وضعیت کنونی(خط فعلی فایل اجرایی و غیره) و بلوک‌های آن به یک سرور ارسال می‌شود. سرور از میان پردازه‌های بلوک شده آن که کمتر اجرا شده است را انتخاب می‌کند و به صورت عمق-اول برای مدت زمان مشخصی آن را اجرا می‌کند. این کار تا پوشش کامل کد ادامه پیدا می‌کند. استفاده از این هیوریستیک باعث می‌شود که EXE سریعتر بتواند به پوشش کامل کد برسد. در پایان نویسندگان ادعا می‌کنند که EXE نقص‌های ابزار DART و CUTE که در قسمت‌هایی پیشین توضیح داده شده‌اند را برطرف می‌کند.

[۸۲] Process

۵-۴ابزار KLEE

در سال ۲۰۰۸ C. Cader و دیگران ابزار KLEE را ارائه کردند.[۷] نویسندگان مقاله ادعا می‌کنند که KLEE را برای آزمون برنامه‌های واقعی و در محیط واقعی ارائه کرده‌اند. نگرانی که در مورد برنامه‌های واقعی وجود دارد یکی تعداد مسیرهای فوق‌العاده زیاد برنامه و دیگری چالش‌های مربوط به کنترل کد هست که در تعامل با محیط بیرون مثل کاربر است.

این مقاله دو سهم اصلی در پیشرفت علم در این حوزه دارد. اول، ارائه ابزار KLEE که بر اساس اجرای نمادین کار می‌کند که برای بررسی محدوده زیادی از برنامه‌ها به کمک تجربیات EXE پیاده‌سازی شده است. دوم، KLEE به طور خودکار موردآزمون[۸۳] برای آزمودن پوشش بالای مسیرهای برنامه‌های واقعی و پیچیده تولید می‌کند. برای آزمون ابزار از GNU COREUTILS استفاده شده است که یافتن باگ در آن بسیار دشوار است. چون یکی از پرکابردترین کتابخانه‌ها است.

۵-۴-۱کلیات طرح

کد شکل ۳۰ در نظر بگیرید. این کد از دو جنبه قابل بررسی است. ۱) پیچیدگی ۲) وابستگی به محیط (مثل دریافت ورودی از فایل). هدف KLEE از اجرای این کد اول، اجرای تمام دستورات و پوشش کامل برنامه است و دوم، در رابطه با هر دستور خطرناک، پاسخ به این سوال است که آیا ورودی است که باعث ایجاد خطا در این دستورات شود یا نه؟

KLEE به صورت نمادین کد را اجرا می‌کند و

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10
4 :     if (*arg == '\\') {                                  11
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15
19:         *buffer++ = '[';
20:         arg -= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;      /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

شکل ۳۰: نمونه کد اجرا شده توسط KLEE

[۸۳] Test Case

موردآزمون‌های مختلف برای رسیدن به اهداف بالا را ایجاد می‌کند. برای اجرای KLEE دستورات زیر اجرا می‌شود. ابتدا با دستور اول برنامه به بایت‌کد به وسیله llvm کامپایل می‌شود بدون اینکه تغییری در برنامه ایجاد شود. سپس KLEE روی بایت‌کد ایجاد شده اجرا می‌شود.(دستور دوم) max-time بیشینه زمان اجرا به دقیقه، sym-args اندازه آرگومان‌های ورودی به نمادین برنامه و sym-files اندازه فایل با مقادیر نمادین ورودی را مشخص می‌کند.

```
llvm-gcc --emit-llvm -c tr.c -o tr.bc

klee --max-time 2 --sym-args 1 10 10 --sym-files 2 2000 --max-fail 1 tr.bc
```

اجرای نمادین شکل ۳۰: این کد دارای خطای سرریز بافر است. KLEE از main شروع می‌کند. وقتی به خط ۳۳ می‌رسد که یک عبارت شرطی است، دستور fork اجرا می‌شود تا هر دو مسیر به ازای درستی با نادرستی شرط خط ۳۳ اجرا شود. همین قاعده برای خط‌های ۳، ۴ و ۵ هم اتفاق می‌افتد. وقتی مسیر جدید اجرا می‌شود، ابزار باید تصمیم بگیرد که کدام را اجرا کند. در ادامه نحوه انتخاب مسیر اجرا بیان می‌شود. فرض کنیم ابزار همان مسیر یافتن خطا را انتخاب کند. ابزار در هر دستور خطرناک مثل اشاره‌گرها بررسی می‌کند که آیا احتمال خطا هست یا نه و قید مربوطه را اضافه می‌کند. وقتی به خط ۱۸ از برنامه می‌رسد، خطا پیدا می‌شود. در این جا دوبار اشاره‌گر به رشته ورودی، بدون بررسی طول آن افزایش می‌یابد و اگر برای مثال طول رشته صفر باشد، اشاره‌گر نویسه ’\0‘ را در انتهای رشته رد می‌کند و خطای سر ریز بافر رخ می‌دهد. در اینجا KLEE موردآزمون (”“ ”“ ’]‘)tr را تولید می‌کند. سپس این قید را برطرف می‌کند و اجرا را ادامه می‌دهد تا خطاهای دیگر را هم بیابد.

۲-۴-۵ معماری KLEE

KLEE بازطرحی شده EXE است. هر پردازه نمادین دارای فایل، ثبات[۸۴]، پشته، شمارنده برنامه، هیپ و شرط مسیر است. برای اینکه با پردازه سیستم عامل اشتباه نشود، به پردازه نمادین حالت[۸۵] گفته می‌شود. برنامه‌ها با کامپایلر llvm کامپایل می‌شوند و KLEE با کامپایل شده برنامه به زبان اسمبلی[۸۶] کار می‌کند.

[۸۴] Register
[۸۵] State
[۸۶] Assembly

۵-۴-۲-۱ معماری پایه

در KLEE در هر زمان می‌تواند تعدادی حالت وجود داشته باشد. حلقه‌ای وجود دارد که تا زمانی که تمام حالت‌ها اجرا نشوند یا زمان مقرر پایان نیابد، اجرا می‌شود.

در KLEE بر خلاف پردازه‌های عادی برای اشاره به حافظه از یک درخت استفاده می‌کند که برگ‌های آن عملوندها (نمادین یا عددی) و گره‌های میانی عملگرها هستند. یعنی ابزار به عبارت‌ها نگاه می‌کند.

در مواجه با عبارت‌های شرطی، عبارت بولی به STP داده می‌شود. اگر STP بتواند آن را حل کند، شاخه‌ای که ابزار انتخاب می‌کند، مشخص می‌شود. در غیر این صورت دستور fork اجرا شده و یک نمونه از حالت کنونی ایجاد می‌شود تا هر دو مسیر با هم اجرا شوند.

برای دستورات خطرناک یک سری قید تولید می‌شود. مثلا برای تقسیم، قید صفر بودن مقسوم‌علیه اضافه می‌شود. اگر مقسوم‌علیه با توجه به خروجی STP بتواند صفر شود، یک موردآزمون برای آن تولید می‌شود در غیر این صورت شرط صفر نبودن در نظر گرفته می‌شود.

برای دستورات load و store قیدهای بررسی در محدوده بودن آدرس‌ها باید اضافه شوند. بهترین روش برای نشان دادن حافظه یک آرایه یک بعدی از بایت‌ها است ولی STP قادر به حل قیدهای مربوطه نخواهد بود. به همین دلیل مانند EXE یک آرایه منحصربه‌فرد در STP در نظر گرفته می‌شود.

ممکن است یک اشاره‌گر بتواند به چندین شی اشاره کند. (هر عملیات نیازمند یک شی خاص است) در این صورت به ازای وضعیت‌های ممکن حالت ایجاد می‌شود.

۵-۴-۲-۲ بهینه‌سازی پرس‌وجوها

مولفه‌ای که در زمان اجرای ابزار تاثیر مستقیم دارد STP است. KLEE برای بهبود زمان و سربار اجرا از بهینه‌سازی‌های زیر استفاده می‌کند:

۱. دوباره نویسی و کاهش عبارت‌ها: مثلا $2x-x=x$ یا $x+0=x$.
۲. در اجرای مسیری از برنامه قیدهای مختلفی در رابطه با یک متغیر ایجاد می‌شود که می‌توان بعضی از آنها را حذف کرد. مثلا $x<5$ و $x=5$ در این صورت در ادامه مسیر مقدار x برابر با 5 قرار داده می‌شود.
۳. در مورد تساوی‌ها مثلا $x+1=10$، در مسیری که این قید حضور دارد، مقدار x برابر ۹ و عددی در نظر گرفته می‌شود.

۴. بحث زیرقیدهای مستقل: همانند EXE زیرقیدهای مستقل از هم در یک مجموعه از قیدها را یافته و پرس‌وجو را همراه با زیر قید مرتبط به STP ارسال می‌کند. برای مجموعه قیدهای {i<j, j<10, k>8} و پرس‌وجوی i=10 فقط {i<j, j<10 } کافی است.

۵. استفاده از کش[۸۷]: در KLEE برای پرس‌وجو در مورد قیدها از کش با ساختمان داده Hoffmann and Hoehler استفاده می‌شود. این ساختمان داده امکان جست‌وجوی سریع در مجموعه قیدها و زیر قیدها را فراهم می‌آورد. استفاده از این مکانیزم ۳ روش جدید برای بهبود پرس‌وجوها را فراهم می‌آورد: (مثلا کش دارای {i<10, i=10}(بدون راه حل) و {i<10, j=8} (دارای راه‌حل i=5 و j=8 است.)

a. اگر یک زیرمجموعه از مجموعه قیدها راه‌حل نداشته باشد، مجموعه اصلی هم راه حل ندارد. مثلا {i<10, i=10, j=12}.

b. اگر مجموعه‌ای زیرمجموعه‌ای از قیدهای دارای راه حل باشد، آن مجموعه هم راه حل دارد.

c. اگر زیرمجموعه‌ای از یک مجموعه دارای راه حل باشد، راه حل آن می‌تواند جزئی از راه حل مجموعه بزرگتر باشد. مثلا راه حل $\{i < 10, j = 8, i \neq 3\}$ همان راه حل {i<10, j=8} است.

۵-۴-۲-۳ برنامه‌ریزی حالت‌ها

KLEE برای انتخاب حالت‌هایی که که اجرا می‌شوند از دو استراتژی به صورت راندرابین استفاده می‌کند:

۱) انتخاب دلخواه[۸۸]: در این حالت درختی از حالت‌ها ایجاد می‌شود که برگ‌ها حالت‌ها و گره‌های میانی محل fork را نشان می‌دهد. برای انتخاب حالت بعدی به محل شاخه مربوطه می‌رود. در اینجا مجموعه‌ای از حالت‌ها هستند که همگی با احتمال یکسان احتمال انتخاب شدن دارند. این استراتژی دو مزیت دارد: اول، از گیرکردن در حلقه بی‌نهایت جلوگیری می‌کند. دوم، حالت‌هایی که در درخت بالاترند، زودتر انتخاب می‌شوند. (تعداد قیدهای کمتری دارند.)

۲) انتخاب برای پوشش بیشترین مسیرها: بر اساس یک سری هیوریستیک به حالت‌ها وزن اختصاص داده می‌شود و سپس به صورت دلخواه یکی از این حالت‌ها انتخاب می‌شوند. این هیوریستیک‌ها بر اساس کمترین فاصله تا دستور پوشش داده نشده، بیشینه فراخوانی حالت و یا اینکه یک حالت اخیرا دستور جدیدی را پوشش داده است یا نه، محاسبه می‌شود.

---

[۸۷] Cache

[۸۸] Random

ترکیب دو استراتژی بالا باعث می‌شود هم پوشش تمامی دستورات فراهم شود و هم از گیر کردن در حلقه جلوگیری به عمل آید.

۳-۴-۵ مدل کردن محیط

منظور از محیط، فایل‌ها، شبکه و غیره است. هدف از این کار این است که اگر برنامه از محیط داده دریافت کرد، تمامی حالات در دریافت درنظر گرفته شود، نه‌تنها مقدار عددی خاص و همچنین هنگام نوشتن در محیط اثر این عمل در خواندن‌های بعدی نمایان باشد. برای این منظور محیط اطراف توسط یک کد یه زبان C مدل شده است و تمام فراخوانی‌ها به آن ارسال می‌گردد. در مقاله مثالی از مدل کردن سیستم فایل آورده شده است که نحوه خواندن از فایل را در شکل ۳۱ توضیح می‌دهد. خطوط ۷ تا ۱۱ برای خواندن از فایل‌های عددی و واقعی است و خطوط ۱۳ تا ۱۹ برای خواندن از فایل نمادین. یک بافر با طول ثابت نقش فایل نمادین را بازی می‌کند. تعداد فایل‌های نمادین و اندازه آنها قابل تنظیم است. نکته‌ای که در این رابطه قابل ذکر هست این است که به جای مدل کردن فراخوانی‌های سطح پایین Open، Read و غیره می‌توان فراخوانی‌های سطح بالا از کتابخانه‌های C را بازنویسی کرد. مثل fopen، fread و غیره. دلیل استفاده از تابع‌های سطح پایین این است که

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

شکل ۳۱: طرحی از مدل کردن read از سیستم فایل

تابع‌ها ساده‌تر و کمتر هستند در حالی که تابع‌های سطح بالا بیشترند و بازنویسی آنها امکان بروز خطا را بالا می‌برد.

فراخوانی‌های سیستمی ممکن است شکست بخورند مثل نوشتن روی دیسک پر. در این حالت ممکن است برنامه خراب شود. برای بازسازی اینگونه خطاها هم KLEE روشی پیاده‌سازی کرده است. اینگونه خطاها به صورت اختیاری بازسازی می‌شوند چون ممکن است در تمام برنامه‌ها بسته به کاربرد، بررسی اینگونه خطاها لازم نباشد.

۵-۵ابزار jCUTE

این مقاله [۸] سعی دارد تا از روش Concolic برای آزمون برنامه‌های چندنخی استفاده کند. زبان جاوا زبانی است که از چند نخی پشتیبانی می‌کند. نتیجه این پژوهش ابزار jCUTE است که در سال ۲۰۰۶ K.Sen و دیگران آن را پیاده‌سازی کرده‌اند. در ادامه ایده‌های مطرح شده بیان می‌شوند.

۵-۵-۱ کلیات طرح

در روش اجرای Concolic هدف تولید ورودی برای اجرای مسیرهای مختلف از برنامه است. در برنامه‌های تک‌نخی تنها قیدهای مسیر باهم در نظر گرفته می‌شوند و شرط مسیر را به وجود می‌آوردند. در برنامه‌های چندنخی علاوه بر شرط مسیر، شرط مسابقه[۸۹] هم میان رخداد[۹۰]های مختلف در نظر گرفته شود. یک رخداد نمایش اجرای یک نخ است. دو رخداد در شرایط مسابقه با هم هستند اگر از دو نخ مجزا از هم باشند که به یک فضای مشترک از حافظه بدون استفاده از قفل مشترک دسترسی داشته باشند. ترتیب اجرای رخدادها با تغییر زمانبندی نخ‌ها قابل تغییر است.

```
x is a shared variable
z = input();

t1:                t2:
1: x = 3;          1: x = 2;
                   2: if (2 * z + 1 == x)
                   3:     ERROR;
```

شکل ۳۲: کد مثال از برنامه‌ای با دو نخ

ابتدا الگوریتم به صورت دلخواه یک مجموعه از ورودی‌ها و زمانبندی برای اجرای نخ‌ها را تولید می‌کند. سپس در یک حلقه این کارها را انجام می‌دهد تا زمانی که تمامی مسیرهای برنامه با استراتژی عمق‌اول اجرا شوند. برنامه را با ورودی‌ها و زمانبندی موجود اجرا می‌کند. هم زمان شرط‌های مسیر و شرط‌های مسابقه را هم جمع‌آوری می‌کند. سپس ورودی‌ها و زمانبندی جدید برای اجرای مسیری جدید از برنامه را با استفاده از این شرط‌ها تولید می‌کند. برای تولید ورودی جدید از مکمل کردن آخرین شرط مسیر و حل قیدها به وسیله حل‌کننده قید[۹۱] استفاده می‌کند. الگوریتم برای ایجاد زمانبندی جدید، دو رخداد را برمی‌دارد که با هم در مسابقه هستند. یکی را به عنوان نخ اول زمانبندی می‌کند و اجرای دومی را به اندازه کافی به تاخیر می‌اندازد.

---

[۸۹] Race Condition

[۹۰] Event

[۹۱] Constraint Solver

برای مثال کد شکل ۳۲ را در نظر بگیرید. این کد شامل دو نخ و یک قید در نخ T2 است. بدون از بین رفتن کلیت معمولا نخ با شماره کمتر اجرا می‌شود. پس اجرای اول به شکل (t2,2) (t2,1) (t1,1) خواهد بود. شماره اول، شماره نخ و شماره دوم، شماره برچسب دستور در کد است. مقدار اولیه و دلخواه نخ دوم هم z=z0 در نظر گرفته می‌شود. ابتدا الگوریتم قید $2 * z0 + 1 \neq 2$ را بر می‌دارد و مکمل می‌کند و سعی می‌کند که آن را حل کند که جواب ندارد. ابزار یک زمانبندی جدید تولید می‌کند و به صورت (t1,1) (t2,2) (t2,1) کد را اجرا می‌کند. مکمل شده قید باز به شکل $2 * z0 + 1 = 2$ است که قابل حل نیست. در آخرین حالت، زمانبندی جدید برای نخ‌ها به شکل (t2,2) (t1,1) (t2,1) خواهد بود. مکمل شده قید به شکل $2 * z0 + 1 = 3$ در می‌آید که قابل حل است. مقدار z0=1 خواهد بود. این بار برنامه با مقدار z0=1 و زمانبندی نهایی اجرا می‌شود که به خط ERROR می‌رسد.

در این مقاله مدل صوری برای طرح نیز ارائه شده است. که مفاهیم اجرای پشت سر هم عبارت‌ها و عبارت‌های در حال مسابقه و غیره به شکل صوری بیان شده‌اند. از بیان این مطالب در این نوشته می‌گذریم.

پیش از این نمایی کلی از الگوریتم بیان شد. حال با استفاده از شبه‌کدهای ارائه شده الگوریتم بررسی می‌شود. در پیاده‌سازی صورت گرفته تعدادی ساختمان داده استفاده شده‌اند که لازم است برای فهم شبه‌کدها، توضیح داده شوند. I همان نگاشت منطقی ورودی[۹۲]ها است که در کار CUTE توضیح داده شد. path_c مجموعه‌ای شامل انتخاب‌های زمانبند و قیدهای نمادین محاسبه شده در طول اجرا برنامه را نگه می‌دارد. branch_hist دنباله‌ای از شاخه‌هایی که در اجرا انتخاب شده‌اند را نگهداری می‌کند که فقط اجرای Concolic از آن استفاده می‌کند. در path_c هر مولفه نقطه‌ای از اجرای فعلی را نشان می‌دهد که شامل اجزای زیر است:

- Constraint: قید تولید شده در عبارت شرطی
- Event: رخدادی که در آن نقطه از اجرا اتفاق می‌افتد مثلا x=3 یا fork(l) و غیره.
- Postponed: مجموعه‌ای از نخ‌ها که در اجرای بعدی اجرا نخواهند شد.
- hasRace: مقدار آن True می‌شود اگر event ذخیره شده در این نقطه با eventای در مسیر اجرای برنامه در آینده با هم مسابقه داشته باشد.

[۹۲] Logical Input Map

برای هر نقطه از اجرا یا جز constraint مقدار دارد یا اجزای event, postponed, hasRace.

```
// input: P is the program to test
run_jCUTE(P)
  completed=false; I = path_c = branch_hist=[ ];
  while not completed
    (I,path_c,branch_hist,completed)
          = scheduler(I,path_c,branch_hist);
```

شکل ۳۳: شبه‌کد الگوریتم

شکل ۳۳ شبه‌کد الگوریتم را نشان می‌دهد. شکل ۳۴ شبه‌کد زمانبند است. ابتدا $T_{enabled}$ با اطلاعات نخ اصلی و اولیه برنامه مقداردهی می‌شود. تا زمانی که مجموعه $T_{enabled}$ دارای نخ فعال است حلقه اجرا می‌شود. هر بار نخ، رخداد و عبارت بعدی که باید اجرا شوند، انتخاب شده و به صورت Concolic اجرا می‌شوند. اگر با خروج از حلقه هنوز نخ فعالی وجود داشته باشد، نشانه وجود deadlock در کد برنامه ورودی است.

```
scheduler(I,path_c,branch_hist)
  T_enabled = {t_0}; t_current = null; i = 0;
  pc_t0 = the label of the first statement
  initialize input variables using I;
  while (T_enabled ≠ ∅)
    t_current = choose_next_thread(T_enabled, i);
    path_c[i].event = nextEvent(t_current);
    path_c[i].enabled = T_enabled;
    i = i + 1;
    s =statement_at(pc_t_current);
    (i,path_c,branch_hist)
          = execute_concolic(t_current, s, i,path_c,branch_hist);
    T_enabled = set of enabled threads;
  if there is an active thread
    print "Found deadlock";
  return compute_next_input_and_schedule(i,I,path_c,branch_hist);

choose_next_thread(T_enabled, i)
  if i < |path_c|
    // schedule the thread as in the previous execution
    (t, l, m, a) = path_c[i].event;
    return t;
  else
    return smallest indexed thread from T_enabled;
```

شکل ۳۴: شبه‌کد زمانبند

در نهایت زمانبندی و ورودی جدید برای اجرای بعدی، تولید می‌شود که کد آن در شکل ۳۶ آمده است. در این قطعه کد اگر رخداد کنونی قید شرطی نداشته باشد و هنوز نخ‌هایی در آن نقطه وجود داشته باشند، نخ کنونی postponed می‌شود و رخداد کنونی با نخ جدید اجرا می‌شود. اگر رخداد قید شرطی باشد، قید به حل‌کننده قید داده می‌شود تا ورودی جدید را تولید کند. تمامی این کارها برای مولفه‌های موجود در path_c از آخر به اول اجرا می‌شود. اجرای Concolic هم در شبه‌کد شکل ۳۵ آمده است.

```
compute_next_input_and_schedule(i,I,path_c,branch_hist)
  for (j = i − 1 ; j ≥ 0 ; j = j − 1)
    if path_c[j].event has proper value
      // compute a new schedule
      if |path_c[j].enabled | > |path_c[j].postponed| + 1
        (t, l, m, a) = path_c[j].event;
        path_c[j].postponed = path_c[j].postponed ∪{t};
        t = smallest indexed thread in
              path_c[j].enabled \ path_c[j].postponed;
        path_c[j].event = (t, l, m, a) ;
        return (I,path_c[0 . . . j],branch_hist[0 . . . j],false);
    else
      // compute a new input
      if (not branch_hist[j].done)
        branch_hist[j].branch=¬branch_hist[j].branch;
        if (∃I′ that satisfies neg_last(path_c[0 . . . j]))
          return (I′,path_c[0 . . . j],branch_hist[0 . . . j],false);
    return ([ ],[ ],[ ],true);
```

شکل ۳۶: شبه‌کد تولید ورودی برای اجرای بعدی

الگوریتمی که تا به حال توضیح داده شد، الگوریتمی ساده است که همه فضای حالت را بررسی می‌کند. برای اینکه این فضا را کوچک و محدود به اجراهای در حال مسابقه باهم کنیم، بهینه‌سازی زیر را انجام می‌دهیم.

در مواجهه با هر رخداد e_j بررسی می‌شود که آیا آن رخداد با هیچ یک از رخدادهای گذشته (e_i) در حال مسابقه هست یا نه. $P = e_1, e_2, \dots, e_i, e_j$ در این صورت جز hasRace را برای رخداد e_i، True می‌کنیم. برای اجرای بعدی e_i را آن قدر به تاخیر می‌اندازیم که ترتیب اجرای آنها معکوس شود. یعنی ابتدا e_j و بعدا e_i اجرا شود تا تاثیر اجرای معکوس روی حافظه مشترک هم دیده شود.

$P = e_1, e_2, \dots, e_{i-1}, e_{i+1}, \dots, e_j, e'_{j+1}, \dots, e_i, \dots$

```
execute_concolic(t, s, i,path_c,branch_hist)
  match(s)
    case (lv ← e):
      m =evaluate_concrete(&lv); val =evaluate_concrete(e);
      execute_symbolic(m, e);
      S = S[m ↦ val]; pc_t = pc_t + 1;
    .
    .
    // skipping other cases
    .
    .
    case (if p goto l′):
      b =evaluate_concrete(p);
      c =evaluate_symbolic_predicate(p, b);
      if (b)
        path_c[i].hasConstraint = true;
        path_c[i].constraint= c; pc_t = l′;
        cmp_n_set_branch_hist(true,i,branch_hist);
      else
        path_c[i].constraint=neg(c); pc_t = pc_t + 1;
        cmp_n_set_branch_hist(false,i,branch_hist);
      i = i + 1;
  if (s == ERROR) print "Found Error";
  return (i,path_c,branch_hist);
```

شکل ۳۵: شبه‌کد اجرای Concolic

برای بررسی اینکه رخدادهای قبلی با فعلی در حال مسابقه هستند تابع check-and-set-race پیاده‌سازی شده است. نکته دیگر این است که اگر e_i و e_j با فاصله از هم در اجرای جدید قرار گیرند ممکن است ابزار دوباره این دو را در حالت مسابقه ببیند. برای حل این مشکل هم در رخدادهای e_{i+1} تا e_j در جز postponed قرار می‌گیرند و از تکنیک sleep استفاده می‌شود. کدهای بهینه شده در شکل‌هایشکل ۳۷،شکل ۳۹ وشکل ۳۸ آمده است. موارد جدید با برچسب M مشخص شده‌اند.

```
scheduler(I, path_c, branch_hist)
    T_enabled = {t_0}; t_current = null; i = 0;
    pc_{t_0} = the label of the first statement
    initialize input variables using I;
    while (T_enabled ≠ ∅)
      t_current = choose_next_thread(T_enabled, i);
      path_c[i].event = nextEvent(t_current);
      path_c[i].enabled = T_enabled;
M:    path_c = check_and_set_race(i, path_c);
      i = i + 1;
      s = statement_at(pc_{t_current});
      (i, path_c, branch_hist)
            = execute_concolic(t_current, s, i, path_c, branch_hist);
      T_enabled = set of enabled threads;
    if there is an active thread
      print "Found deadlock";
    return compute_next_input_and_schedule(i, I, path_c, branch_hist);

choose_next_thread(T_enabled, i)
    if i < |path_c|
      // schedule the thread as in the previous execution
      (t, l, m, a) = path_c[i].event;
      return t;
    else
M:    if t_current is enabled
M:      return t_current;
M:    else
        return smallest indexed thread from T_enabled;

check_and_set_race(i, path_c)
    ∀j ∈ [0, i) such that path_c[j].event ⋖ path_c[i].event
      if t not in path_c[j].postponed
        if e is a read or write event
          print "Warning: data race found"
          path_c[j].hasRace = true;
        else
          // path_c[j].event is an unlock event
          let j' be such that path_c[j'].event is the lock event
          matching the unlock event path_c[j].event
            path_c[j'].hasRace = true;
    return path_c;
```

شکل ۳۷:شبه‌کد زمانبند بهینه

```
compute_next_input_and_schedule(i,I,path_c,branch_hist)
    for (j = i − 1 ; j ≥ 0 ; j = j − 1)
      if path_c[j].event has proper value
        // compute a new schedule
        if |path_c[j].enabled | > |path_c[j].postponed| + 1
M:        if path_c.isRace
M:          path_c[j].isRace = false;
            (t, l, m, a) = path_c[j].event;
            path_c[j].postponed = path_c[j].postponed ∪{t};
            t = smallest indexed thread in
                  path_c[j].enabled \ path_c[j].postponed;
            path_c[j].event = (t, l, m, a) ;
            return (I, path_c[0 . . . j], branch_hist[0 . . . j ],false);
      else
        // compute a new input
        if (not branch_hist[j].done)
          branch_hist[j].branch=¬branch_hist[j].branch;
          if (∃I′ that satisfies neg_last(path_c[0 . . . j]))
            return (I′,path_c[0 . . . j], branch_hist[0 . . . j ],false);
      return ([ ],[ ],[ ],true);
```

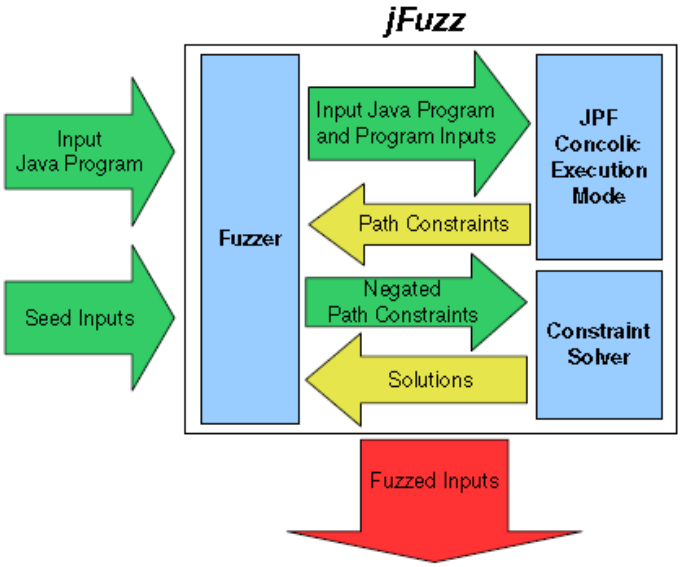
شکل ۳۹: شبه‌کد بهینه تولید ورودی برای اجرای بعدی

```
scheduler(I,path_c,branch_hist)
    T_enabled = {t0}; t_current = null; i = 0;
    pc_0 = the label of the first statement
M: sleep = {}; postponed = {};
    initialize input variables using I;
    while (T_enabled ≠ ∅)
M:    postponed = postponed ∪ path_c[i].postponed;
M:    sleep = {nextEvent(t) | t ∈ postponed};
M:    t_current = smallest indexed thread from T_enabled \ postponed;
      path_c[i].event = nextEvent(t_current);
      path_c[i].enabled = T_enabled;
      path_c = check_and_set_race(i,path_c);
M:    path_c[i].postponed =postponed;
M:    ∀e ∈ sleep if e⋘ path_c[i].event
M:      let (t, l, m, a) = e in postponed =postponed \t;
      i = i + 1;
      s =statement_at(pc_t_current);
      (i, path_c, branch_hist)
            = execute_concolic(t_current, s, i,path_c,branch_hist);
      T_enabled = set of enabled threads;
    if there is an active thread
      print "Found deadlock";
    return compute_next_input_and_schedule(i,I, path_c, branch_hist);
```

شکل ۳۸: بهینه‌سازی نهایی زمانبند

۵-۶ابزار jFuzz

jFuzz برخلاف jCUTE یک ابزار متن‌باز برای اجرای Concolic برنامه‌های به زبان جاوا است. این ابزار بر روی پروژه متن‌باز Java Path Finder(JPF) پیاده‌سازی شده است. معماری این ابزار در شکل ۴۱ آمده است. نویسندگان مقاله ادعا می‌کنند برای کمتر کردن هزینه فراخوانی حل‌کننده قید از بهینه‌سازی‌های موجود در کارهای EXE، KLEE، CUTE، jCUTE و غیره نیز استفاده کرده‌اند. همچنین ایده جدید این مقاله در این مورد این است که قیدهایی که رفتار یکسانی دارند ولی نام متغیرهایشان متفاوت است، تشخیص داده می‌شوند. شکل ۴۰ نمونه‌ای از برابری قیدها را نشان می‌دهد. با استفاده از روش کش و روش‌های بهینه‌سازی تعدادی از پرس‌وجوهای به حل‌کننده قید را جواب می‌دهند.[۹]



شکل ۴۱: معماری jFuzz

Path Condition 1:	Path Condition 2:
[1] a + b < 10	[1] x + y < 10
[2] b > 6	[2] y > 6
[3] a < 3	[3] x < 3
[4] a != 2	[4] x != 2

شکل ۴۰: نمونه‌ای از قیدهای با رفتار یکسان ولی ظاهری متفاوت

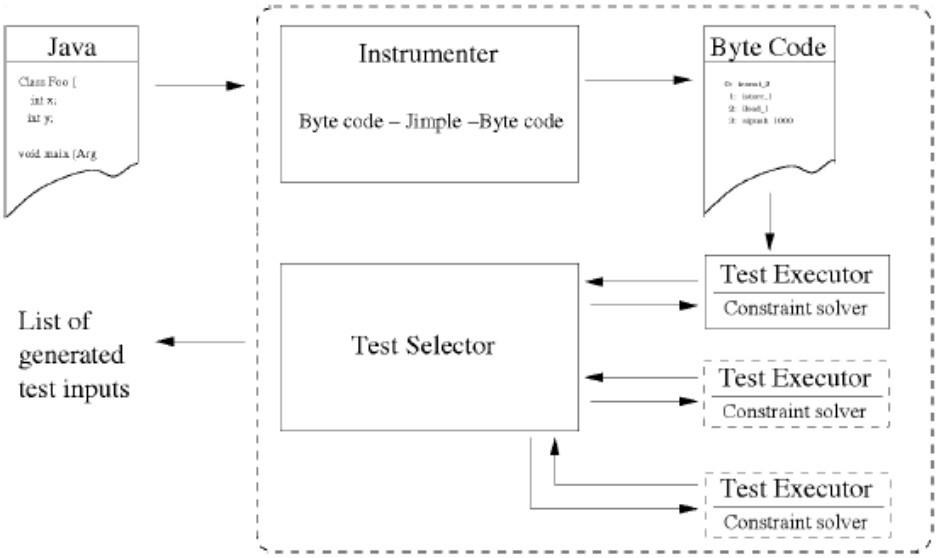
۵-۷ابزار LCT

LCT(LIME Concolic Tester) یک ابزار متن‌باز برای آزمون برنامه‌هایی به زبان جاوا است. این ابزار نوآوری‌های زیر را داشته است.[۱۰]

۱. استفاده از SMT Solver که باعث می‌شود اجرای نمادین را دقیق‌تر کند.
۲. پشتیبانی از معماری توزیع‌شده
۳. سلسه مراتب کلاس‌های دوقلو[۹۳]: برای تجهیز کلاس‌های هسته جاوا

۵-۷-۱ معماری LCT

معماری LCT در شکل ۴۲ آمده است. مولفه‌های اصلی این ابزار Instumenrter, Test Executer و Test Selector هستند. بعد از مشخص شدن ورودی‌های برنامه و مقداردهی آنها (int x= LCT.getInt()) به وسیله ابزار Soot برنامه تجهیز می‌شود تا اجرای نمادین روی آن صورت پذیرد. Test Selector درختی نمادین از برنامه با استفاده از قیدهای بدست آمده از Test Executer برای اجرای مسیرهای مختلف ایجاد می‌کند. این مولفه مشخص می‌کند کدام مسیر باید در اجرای بعدی طی شود. ارتباط میان Test Executer و Test Selector توسط TCP Socket پیاده‌سازی می‌شود. این امر توزیع پردازش آزمون را راحت‌تر می‌کند. LCT می‌تواند از



شکل ۴۲: معماری LCT

[۹۳] Twin Class Hierarchy

حل‌کننده قیدهای Yices و یا Boolector استفاده کند.

۵-۷-۲ محدودیت‌ها

۱. عدم پشتیبانی از چندنخی
۲. مشکل عدم پوششِ کامل کد برنامه در حالتی که فراخوانی به کتابخانه تجهیز شده صورت گیرد.
۳. مشکل برخورد با آرایه‌ها مثل a[i]=0; a[j]=1; if(a[i]!=0) ERROR; نمی‌تواند قید i=j را تشخیص دهد.

۵-۸ ابزار Jalangi

Jalangi [۱۱] یک چارچوبه کاری[۹۴] است که از آن برای تحلیل برنامه‌های به زبان جاوا اسکریپت استفاده می‌شود. از زبان جاوا اسکریپت برای برنامه‌نویسی سمت کاربر برنامه‌های تحت وب و همچنین برنامه‌نویسی برنامه‌های تحت گوشی‌های هوشمند استفاده می‌شود. مهمترین دلیل محبوبیت جاوا اسکریپت جابه‌جاپذیری[۹۵] برنامه‌های به این زبان است. Jalangi یک چارچوبه کاری برای تحلیل پویای این زبان است. این چارچوبه کاری دو ویژگی مهم دارد:

- ثبت-بازاجرای انتخابی[۹۶]: برنامه‌های به زبان جاوا اسکریپت ممکن هست از کتابخانه‌های مختلفی مثل jQuery استفاده کند. Jalangi این ویژگی را دارد که کاربر می‌تواند انتخاب کند که رفتار کتابخانه‌ای خاص، تنها بررسی و تحلیل شود.
- مقادیر سایه[۹۷]: این مقادیر اطلاعاتی اضافی (مثل آلوده بودن یا نمایش نمادین) را در مورد داده‌های اصلی در خود نگهداری می‌کنند. Jalangi از اجرا روی مقادیر سایه پشتیبانی می‌کند که مقادیر سایه را به‌روز می‌کند. مثل اجرای نمادین یا تحلیل آلایش.

۵-۸-۱ نوآوری‌های طرح

۱. استقلال از مرورگر[۹۸]: با استفاده از ویژگی ثبت-بازاجرای انتخابی حاصل می‌شود.
۲. اجرا و تحلیل برنامه‌های تحت یک پلتفرم خاص روی پلتفرمی دیگر (مثلا برنامه تحت گوشی روی مرورگر تحت PC تحلیل شود.) به دلیل کمبود منابع برای تحلیل در پلتفرم اصلی.
۳. پشتیبانی از مقادیر سایه برای اجرای تحلیل‌های پویا مثل اجرای Concolic یا تحلیل آلایش.

۵-۸-۲ تحلیل‌های پویایی که در Jalangi وجود دارند

۱. اجرای Concolic
۲. دنبال کردن منبع مقدار Null یا تعریف نشده
۳. تحلیل پویای آلایش

---

[۹۴] framework
[۹۵] Portability
[۹۶] Selective Record-Replay
[۹۷] Shadow Values
[۹۸] Browser

۴. تشخیص ناسازگاری نوع‌ها: این تحلیل به دنبال تشخیص اشیائی است که در یک برنامه بتوان به آن به طور هم زمان دو نوع ناسازگار اختصاص داد.
۵. تخصیص حافظه اشیا[۹۹]: در این تحلیل مشخص می‌شود که تعداد اشیائی که تولید شده‌اند و تعداد دفعات ارجاع به آنها چقدر است. چون تعداد زیاد اشیا می‌تواند موجب مشکلات در کارایی برنامه در مورد حافظه شود.

۵-۸-۳ جزئیات فنی طرح

در شکل ۴۳ نحو زبان جاوا اسکریپت به شکل ساده شده آن آمده است. با استفاده از یک کامپایلر می‌توان دستورات پیچیده را به شکل ساده شده در آورد. برای مثال دستورات for و while به دستور if b goto l تبدیل می‌شوند. برای فراخوانی تابع از v1=call(v2,v3,v4,…) استفاده می‌شود که v2 نام تابع، v3 متغیر this و بقیه، آرگومان‌های تابع هستند. برای دسترسی به اجزای آرایه یا شی هم از v1[v2] استفاده می‌شود.

```
v, v1, v2,   ... are variable identifiers
f, f1, f2,   ... are field identifiers
p, p1, p2,   ... are function parameter identifiers
      op     are operators such as +, -, *, ...
Pgrm ::=     (ℓ: Stmt)*
Stmt ::=     var v
             v = c
             v1 = v2 op v3
             v1 = op v2
             v1 = call(v2, v3, v4, ...)
             if v goto ℓ
             return v
             v1 = v2[v3]
             v1[v2] = v3
             function v1(p1, ...){(ℓ: Stmt)*}        function definition
   c ::=     number
             string
             undefined
             null
             true
             false
             {f1: v1, ...}                           object literal
             [v1, ...]                               array literal
             function v1(p1, ...){(ℓ: Stmt)*}        function literal
```

شکل ۴۳: نحو زبان ساده شده جاوا اسکریپت

[۹۹] Simple object allocation profiler

۵-۸-۳-۱ ثبت-بازاجرای انتخابی

فرض می‌شود کاربران Jalangi زیر مجموعه‌ای از کدهای جاوا اسکریپت را برای ثبت-بازاجرا انتخاب می‌کنند. در فاز اول کدهای انتخابی توسط ابزار تجهیز می‌شوند. سپس تمام کدهای تجهیزشده و تجهیز نشده باهم اجرا می‌شوند. در فاز دوم کدهای تجهیز شده تنها اجرا می‌شوند. این موضوع ۲ مزیت دارد: اول اینکه فاز اول (ثبت) می‌تواند روی پلتفرم اصلی اجرا شود و فاز دوم به منظور تحلیل روی PC اجرا شود. دوم اینکه در فاز دوم، چون فقط کدهای تجهیزشده اجرا می‌شوند، می‌توان تحلیل‌های پویای متفاوتی را پیاده‌سازی کرد.

راه حل بدیهی این است که در فاز ثبت تمام بارگذاری‌های حافظه‌ای ثبت شوند تا در فاز بازاجرا از مقادیر این حافظه‌ها بتوان استفاده کرد. این موضوع با دو چالش روبه‌رو است:

۱. مقدار شی‌ها و توابع چگونه ثبت شوند؟
۲. اگر توابع تجهیزشده توسط توابع تجهیز شده فراخوانی شوند(مثل JavaScript event dispatcher) چگونه ثبت و بازاجرا خواهند شد؟

برای چالش اول، به ازای هر شی و یا تابع، یک شناسه منحصر به فرد و عددی نگهداری می‌شود و برای چالش دوم، به صورت ضمنی توابع تجهیزشده ثبت می‌شوند که توسط توابع دیگر فراخوانی شده‌اند و در بازاجرا این توابع دوباره اجرا می‌شوند.

همچنین تمام مقادیر حافظه ثبت نمی‌شوند. آن دسته از مقادیری که در بازاجرا توان محاسبه آنها را به وسیله کدهای مجهزشده به تنهایی داریم ثبت نمی‌شوند. برای مشخص شدن این موضوع، ابزار در کنار حافظه معمول یک حافظه سایه نگهداری می‌کند. حافظه سایه هم مثل حافظه اصلی در طول اجرا به‌روز می‌شود. با این تفاوت که تنها تغییرات توابع تجهیزشده در حافظه سایه اعمال می‌شود. در ادامه Jalangi در طول اجرا اگر تفاوتی در مقدار سایه و اصلی ببیند مقدار آن حافظه را ثبت می‌کند.

شکل ۴۵ نحوه تجهیز کدها را نشان می‌دهد و شکل ۴۵ کد توابع استفاده شده در شکل ۴۵ را نشان می‌دهد. در شکل ۴۵ هر متغیری که با «’» نشان داده شده است مثل v’ مربوط به حافظه سایه است. در بیان‌های مختلف v1=op v2 ابتدا v2’=v2=sync(v2, v2’) اجرا می‌شود. تابع sync بررسی می‌کند که آیا آرگومان‌های ورودی آن با هم برابر هستند یا نه. در فاز ثبت اگر این دو مقدار با هم متفاوت بودند ابزار مقدار

$$
\begin{array}{lcl}
\texttt{var } v & \Longrightarrow & \texttt{var } v' \\
 & & \texttt{var } v \\
v = e & \Longrightarrow & v' = v = \mathrm{sync}(e) \\
v1 = v2 \texttt{ op } v3 & \Longrightarrow & v2' = v2 = \mathrm{sync}(v2, v2') \\
 & & v3' = v3 = \mathrm{sync}(v3, v3') \\
 & & v1' = v1 = v2 \texttt{ op } v3 \\
v1 = \texttt{op } v2 & \Longrightarrow & v2' = v2 = \mathrm{sync}(v2, v2') \\
 & & v1' = v1 = \texttt{op } v2 \\
\texttt{if } v \texttt{ goto } \ell & \Longrightarrow & v' = v = \mathrm{sync}(v, v') \\
 & & \texttt{if } v \texttt{ goto } \ell \\
\texttt{return } v & \Longrightarrow & v' = v = \mathrm{sync}(v, v') \\
 & & \texttt{return } v \\
v1 = v2[v3] & \Longrightarrow & v2' = v2 = \mathrm{sync}(v2, v2') \\
 & & v3' = v3 = \mathrm{sync}(v3, v3') \\
 & & v2[v3 + \text{"}_{m}\text{"}] = v2[v3] = \\
 & & \quad \mathrm{sync}(v2[v3], v2[v3 + \text{"}_{m}\text{"}]) \\
 & & v1' = v1 = v2[v3] \\
v1[v2] = v3 & \Longrightarrow & v1' = v1 = \mathrm{sync}(v1, v1') \\
 & & v2' = v2 = \mathrm{sync}(v2, v2') \\
 & & v3' = v3 = \mathrm{sync}(v3, v3') \\
 & & v1[v2 + \text{"}_{m}\text{"}] = v1[v2] = v3 \\
v1 = \mathrm{call}(v2, v3, v4, \ldots) & \Longrightarrow & v2' = v2 = \mathrm{sync}(v2, v2') \\
 & & v3' = v3 = \mathrm{sync}(v3, v3') \\
 & & v4' = v4 = \mathrm{sync}(v4, v4') \\
 & & \vdots \\
 & & v1' = v1 = \mathrm{sync}(\\
 & & \quad \mathrm{instrCall}(v2, v3, v4, \ldots)) \\
\{f1 : v1, \ldots\} & \Longrightarrow & \{f1 : v1' = v1 = \\
 & & \quad \mathrm{sync}(v1, v1'), \ldots\} \\
[v1, \ldots] & \Longrightarrow & [v1' = v1 = \mathrm{sync}(v1, v1'), \ldots] \\
\texttt{function } v1(p1, \ldots)\{ & \Longrightarrow & \texttt{function } v1(p1, \ldots)\{ \\
\quad (\ell : \mathrm{Stmt})^* & & \quad \mathrm{enter}(v1) \\
\} & & \quad \texttt{var } p1' \\
 & & \quad \vdots \\
 & & \quad (\ell : \mathrm{Stmt})^* \\
 & & \} \\
\end{array}
$$

شکل ۴۵: نحوه تجهیز کردن کد ورودی

```
// persist trace after recording
// during replay initialize trace
// from persisted trace
var trace = [];
var i = 0, id = 0, objectMap = [];

function getRecord(v) {
  if (v !== null && (typeof v ==='object'||
      typeof v === 'function')){
    if (!v["*id*"]) v["*id*"] = ++id;
    return {type:typeof v, val:v["*id*"]}
  } else {
    return {type:typeof v, val:v};
  }
}

function syncRecord(rec, v) {
  var result = rec.val
  if(rec.val !==null && (rec.type==='object'||
      rec.type === 'function')){
     if (objectMap[rec.val])
       result = objectMap[rec.val];
     else {
      if(typeof v !== rec.type||v["*id*"])
       v = (rec.type==='object')?{}:function(){}
      v["*id*"] = rec.val;
      objectMap[rec.val] = v;
      result = v;
     }
  }
  return result
}

function sync(v1, v2) {
  i = i + 1;
  if (recording) {
    if (v1 !== v2)
      trace[i] = getRecord(v1);
    return v1;
  } else {
    if (trace[i])
      return syncRecord(trace[i], v1);
    else
      return v1;
  }
}

function enter(v) {
  i = i + 1;
  if (recording) {
    trace[i] = getRecord(v)
    trace[i].isFunCall = true
  }
}

function instrCall(f, o, a1, ..., an) {
  if (recording || isInstrumented(f))
     return call(f, o, a1, ..., an)
  else
     return replay()
}

function replay() {
  while (trace[i+1].isFunCall) {
     var f = syncRecord(trace[i+1], undefined)
     f()
  }
  return undefined
}
```

شکل ۴۵: توابع استفاده شده در شکل ۴۵

موجود در حافظه را ثبت می‌کند. در فاز باز اجرا مقدار آرگومان اول را باز می‌گرداند. این باعث می‌شود در فاز بازاجرا توابع با همان مقادیر ثبت شده در فاز اجرا ثبت شوند.

در فراخوانی توابع، Call(v2,v3,v4, …)، با sync(instrCall(v2,v3,v4, ….)) جایگزین می‌شود. instrCall در فاز ثبت یا در فاز بازاجرا برای توابع تجهیزشده خود v2 را فراخوانی می‌کند. در غیر این صورت تابع ()replay اجرا می‌شود. خط اول تعریف توابع تجهیزشده، تابع enter قرار می‌گیرد. این تابع مشخصات تابع تجهیزشده را در فاز ثبت نگه می‌دارد. برای حالتی که توابع تجهیزنشده توسط instrCall در فاز بازاجرا فراخوانی می‌شوند، تابع replay به کمک تابع enter که در فاز ثبت اجرا شده است، تمامی توابع تجهیزشده‌ای را فراخوانی می‌کند که توسط توابع تجهیزنشده فراخوانی شده‌اند.

آرایه trace به منظور نگهداری و ثبت استفاده می‌شود. برای مواردی که نیاز به ثبت است، مقدار متغیر و نوعش ثبت می‌شوند. اگر متغیر تابع f یا شی o باشد، شناسه آن که به صورت پنهان در f[*id*] وجود دارد به عنوان مقدار با آن ذخیره می‌شود. بعد از اجرای فاز اول، trace در قالب JSON در یک فایل ذخیره می‌شود.

```
var v                          ⟹  var v'
                                   var v
                                   if(anlys && anlys.literal)
                                     v = anlys.literal(undefined)

v = c                          ⟹  v' = v = sync(c)
                                   if(anlys && anlys.literal)
                                     v = anlys.literal(c)

v1 = v2 op v3                  ⟹  v2' = v2 = sync(v2, v2')
                                   v3' = v3 = sync(v3, v3')
                                   v1' = v1 = a(v2) op a(v3)
                                   if(anlys && anlys.binary)
                                     v1 = anlys.binary(op, v2, v3, v1)

v1 = op v2                     ⟹  v2' = v2 = sync(v2, v2')
                                   v1' = v1 = op a(v2)
                                   if(anlys && anlys.unary)
                                     v1 = anlys.unary(op, v2, v1)

if v goto ℓ                    ⟹  v' = v = sync(v, v')
                                   if(anlys && anlys.conditional)
                                     anlys.conditional(v)
                                   if a(v) goto ℓ

return v                       ⟹  v' = v = sync(v, v')
                                   return v

v1 = v2[v3]                    ⟹  v2' = v2 = sync(v2, v2')
                                   v3' = v3 = sync(v3, v3')
                                   a(v2)[a(v3) + "'"] = a(v2)[a(v3)] =
                                     sync(a(v2)[a(v3)], a(v2)[a(v3) + "'"])
                                   v1' = v1 = a(v2)[a(v3)]
                                   if(anlys && anlys.getField)
                                     v1 = anlys.getField(v2, v3, v1)

v1[v2] = v3                    ⟹  v1' = v1 = sync(v1, v1')
                                   v2' = v2 = sync(v2, v2')
                                   v3' = v3 = sync(v3, v3')
                                   a(v1)[a(v2) + "'"] = a(v1)[a(v2)] = v3
                                   if(anlys && anlys.putField)
                                     a(v1)[a(v2)] =
                                       anlys.putField(v1, v2, v3)

v1 =                           ⟹  v2' = v2 = sync(v2, v2')
  call(v2, v3, v4, ...)            v3' = v3 = sync(v3, v3')
                                   v4' = v4 = sync(v4, v4')
                                   ⋮
                                   v1' = v1 = sync(
                                     instrCall(a(v2), v3, v4, ...))
                                   if(anlys && anlys.call)
                                     v1 = anlys.call(v2, v3, v4, ..., v1)

{f1: v1, ...}                  ⟹  {f1: v1' = v1 =
                                     sync(v1, v1'), ...}

[v1, ...]                      ⟹  [v1' = v1 = sync(v1, v1'), ...]

function v1(p1, ...){          ⟹  function v1(p1, ...){
  (ℓ: Stmt)*                       enter(v1)
}                                  var p1'
                                   ⋮
                                   (ℓ: Stmt)*
                                 }
```

شکل ۴۶: نحوه تجهیز کد برای محاسبات سایه

۵-۸-۳-۲ مقادیر سایه و اجرای سایه

Jalangi امکانی را فراهم کرده است تا بتوان تحلیل‌های پویا را راحت‌تر آنجام داد. برای این منظور هر مقدار مانند ۳۱ دارای یک مقدار سایه هم هست. این مقدار سایه وضعیت ۳۱ را در آن تحلیل خاص نشان می‌دهد و مقدار اصلی و سایه با هم تشکیل یک شی می‌دهند مثلا در تحلیل آلایش {a="31" , s="tainted"} برای ۳۱ نگهداری می‌شود. همچنین

Jalangi امکان اجرای سایه روی مقادیر بالا را دارد. که با دوباره نویسی یک سری تابع از پیش تعریف شده مثل binary, unary و غیره می‌توان تحلیل مورد نظر مثل تحلیل آلایش یا اجرای نمادین را پیاده‌سازی و تعریف کرد. برای این منظور کد باید با توجه به شکل ۴۶ تجهیز شود. برای مثال در تحلیل آلایش در عبارت v1=v2+v3 اگر {a=”31” , s=”tainted”} v2= و {a=”5” , s=”undefined”} v3= باشد، با محاسبه v1=anlys.binary(+, v2, v3, v1) آنگاه {a=”36” , s=”tainted”} v1= خواهد شد. این تابع نشان می‌دهد که اگر حداقل یکی از متغیرها آلوده باشد، حاصل هم آلوده خواهد بود. برای اینکه محاسبات عادی دچار مشکل نشود به جای خود متغیر v1 از $\underline{a}(v1)$ استفاده می‌شود که مقدار اصلی در آن وجود دارد.

۵-۹اجرای Concolic برای برنامه‌های گوشی همراه

[۱۲] قصد دارد روش Concolic را برای برنامه‌های کاربردی گوشی‌های هوشمند همراه که از این به بعد با عنوان App بیان می‌شوند، به کار ببرد که سیستم عامل آن اندروید است. Appها دارای ویژگی‌هایی هستند که تحلیل ایستا در آنها دچار چالش می‌شود. مثل وجود SDK، ناهمگامی، تعامل میان‌پردازه‌ای[۱۰۰]، پایگاه‌داده‌ها و رابط کاربری گرافیکی.

در تحلیل پویا سعی در تولید پویای ورودی‌ها است. Appها علاوه بر ورودی‌های عادی که در تمام برنامه‌ها وجود دارند مثل اعداد، رشته‌ها و غیره، رخداد[۱۰۱]ها هم به عنوان ورودی وجود دارند. [۱۰۲]یک ضربه روی صفحه نمایش، فشردن یک کلید از دستگاه و پیامک‌ها از جمله رخدادها هستند.

Appها زیرمجموعه‌ای از برنامه‌های مبتنی بر رخداد[۱۰۳] هستند. این گونه برنامه‌ها شامل محاسباتی هستند که در تعامل با ترتیب نامحدودی از رخدادها و پاسخ به آنها هستند. در آزمون این گونه از برنامه‌ها با دو چالش روبه‌رو هستیم:

۱. چطور یک رخداد ایجاد شود؟
۲. چطور ترتیبی تز رخدادها ایجاد شود؟

این مقاله از روش اجرای Concolic برای تولید رخدادها استفاده می‌کند. رخدادها را از جای که تولید می‌شوند تا جایی که کنترل می‌شوند، دنبال می‌کند. در مورد چالش دوم اگر از روش Concolic مرسوم استفاده شود، در مورد برنامه‌های واقعی با مشکل انفجار مسیر روبه‌رو می‌شود. برای بهبود این موضوع هم سعی شده ورودی‌هایی که تولید می‌شوند، (مجموعه‌ای از رخدادها) بررسی شود که آیا در مجموعه‌ای از ورودی‌های دیگر حضور دارند یا نه. در این صورت این ورودی‌ها اجرا نمی‌شوند.

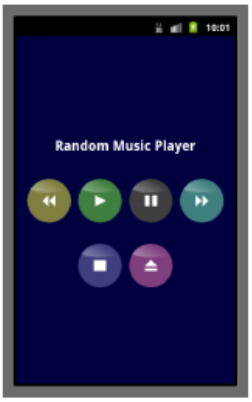
۵-۹-۱کلیات طرح

[۱۰۰] Inter-process communication
[۱۰۱] Event
[۱۰۲] Tap event
[۱۰۳] Event-Driven Programs

برای توضیح طرح از یک برنامه ساده به زبان اندروید استفاده می‌شود که در شکل ۴۸ آمده است. برنامه‌های اندرویدی خود به تنهایی تابع main ندارند. بلکه این برنامه‌ها کامل کننده SDK اندروید هستند. هر برنامه از تعدادی Activity تشکیل شده است. هر Activity هم دارای چرخه حیات[۱۰۴] است. (()onCreate، ()onStart، onResume و غیره) در شکل ۴۸ تابع onCreate آمده است. تعدادی دکمه تعریف شده است و برای هر کدام تابع onClickListener برای آنها تعریف شده است. کلاس MusicService هم برای اجرای سرویس خاص هر دکمه پیاده‌سازی شده است. شکل ۴۸ نمایی از برنامه را نشان می‌دهد.

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

شکل ۴۸: کد نمونه به زبان اندروید



(a) Main screen.

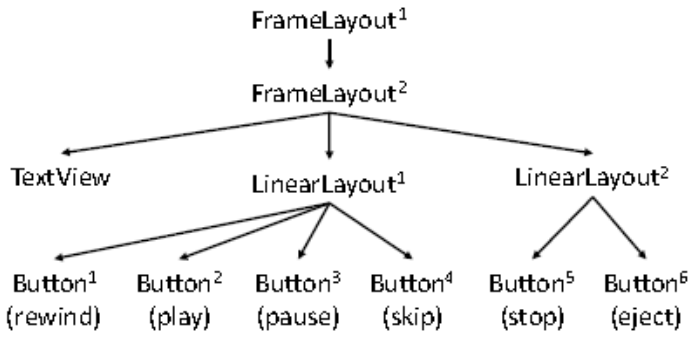
(b) Eject dialog.

شکل ۴۸: نمایی از برنامه

۵-۹-۲ تولید یک رخداد

[۱۰۴] Lifecycle

برای این برنامه رخداد ضربه[۱۰۵] در نظر گرفته شده است. شکل ۴۹ نمایی از سلسله مراتب ویجت[۱۰۶]های موجود در صفحه اصلی برنامه را نشان می‌دهد. برگ‌های درخت در نهایت دکمه‌های روی صفحه و نمایشگر متن[۱۰۷] هستند. هدف، تولید رخدادهایی است که برای همه این ۱۱ ویجت رخداد ضربه تولید شود.



شکل ۴۹: سلسله مراتب ویجت‌ها در صفحه اصلی برنامه

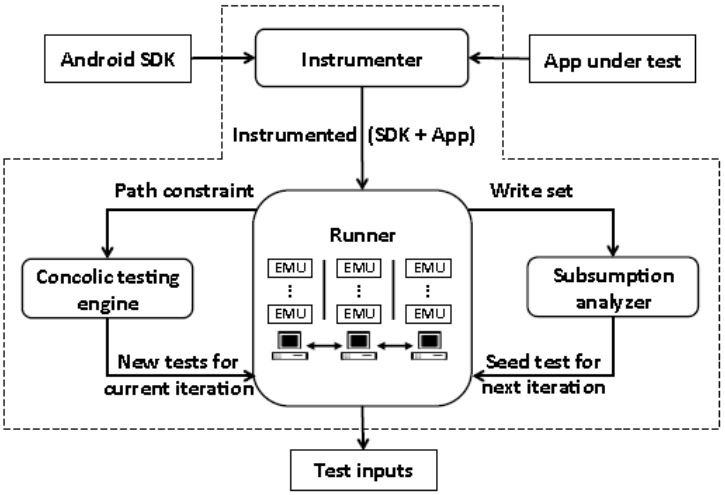
در روش‌های گذشته این درخت یا به صورت دستی[۱۰۸](Model-based) یا به صورت خودکار(Capture-Replay) تولید می‌شده است. در روش خودکار طرح به صورت موردی بوده است و همه ویجت‌ها، چه ویجت‌های SDK چه ویجت‌های سفارشی، پشتیبانی نمی‌کند. در این مقاله از روش Concolic برای تولید رخدادها استفاده می‌شود. برای این منظور SDK و برنامه تحت آزمون باید تجهیز شوند. سپس در حین اجرای یک رخداد عددی، یک رخداد به صورت نمادین هم تولید می‌شود که تمام قیدهای مسیر را در خود نگهداری می‌کند. با این روش می‌توان رخداد ضربه را برای هر ۱۱ ویجت ایجاد کرد.

۳-۹-۵ تولید ترتیبی از رخدادها

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

---

[۱۰۵] Tap Event
[۱۰۶] Wiget
[۱۰۷] TextView
[۱۰۸] Manual



شکل ۵۰: معماری ابزار ارائه شده

در این مقاله مجموعه رخدادهای تولید شده مورد بررسی قرار گرفته شده است. بررسی‌ها نشان می‌دهد که تعداد زیادی از مجموعه رخدادهای تولید شده تکراری هستند. برنامه‌ها را می‌توان به شکل یک نمودار حالت[۱۰۹] در آورد که با هر رخداد حالت برنامه تغییر می‌کند. تعدادی از رخدادهای تولید شده توسط روش Concolic حالت برنامه را تغییر نمی‌دهند و از آنجایی که در حافظه چیزی نمی‌نویسند به آنها رخدادهای فقط خواندنی[۱۱۰] گفته می‌شود. برای جلوگیری از تولید این گونه رخدادها موارد زیر در نظر گرفته شده است:

۱. تعدادی از ویجت‌ها مثل linearLayout و FrameLayout برخلاف دکمه‌ها هنگام رخداد دارای کنش[۱۱۱] نیستند. پس تنها ۶ ویجت از ۱۱ ویجت موجود در صفحه امکان اجرا دارند.
۲. تعدادی از ویجت‌هایی که فعالیت دارند ممکن است به هر دلیلی که برنامه‌نویس صلاح دیده است، غیر فعال باشند.
۳. 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.
۴. 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[۱۱۲].

[۱۰۹] State Transition Diagram
[۱۱۰] Read Only
[۱۱۱] Action
[۱۱۲] Incoming Call

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 ۲۱ هزار ترتیب رخداد ۴تایی را به حدود ۳هزار ترتیب رخداد ۴تایی کاهش می‌دهد. معماری ابزار ارائه شده در شکل ۵۰ نشان داده شده است.

۵-۱۰ بهبودهایی بر روش اجرای Concolic

در روش Concolic از جست‌وجوی عمق‌اول[۱۱۳] برای پوشش تمام مسیرهای برنامه استفاده می‌شود. [۱۳] ادعا می‌کند که این روش برای برنامه‌های واقعی و بزرگ کند است. به همین دلیل ۳ روش جدید برای جست‌وجوی و انتخاب مسیرهای اجرایی در این مقاله ارائه شده است. ابزار CREST هم به صورت متن‌باز ارائه شده است که این ویژگی‌ها را در خود دارد. برای توضیح روش‌های مختلف از کد شکل ۵۱ استفاده شده است. دستورات شرطی موجب ایجاد شاخه‌های مختلف در برنامه می‌شود. این شاخه‌ها را به شکل یک دوتایی نشان می‌دهیم مثل (l1,l2) (l6,l7) (l10,l13) (l11,l12).

```
    main(x, y) {
l0:     if (x > y)
l1:       z = f(y);
        else
l2:       z = y;
l3:     g(x, z);
l4:     return;
    }

    int f(a) {
l5:     if (a > 0)
l6:       ABORT;
        else
l7:       ;
l8:     return -a;
    }

    g(a, b) {
l9:     if (a == 4)
l10:      if (2*b > 9)
l11:        ABORT;
          else
l12:        ;
        else
l13:      print b-a;
l14:    return;
    }

    not_called(a) {
l15:    b = f(a)
l16:    print 2*b;
l17:    return;
    }
```

شکل ۵۱: کد نمونه

۵-۱۰-۱ جست‌وجوی عمق‌اول محدود

در این روش مانند روش کلی و مرسوم Concolic جست‌وجو صورت می‌گیرد با این تفاوت که تعداد شاخه‌هایی که هر دو مسیر آن بررسی خواهند شد، محدود و به اندازه d خواهد بود. یعنی در هر مسیر حداکثر به عمق d جست‌وجو صورت می‌گیرد.

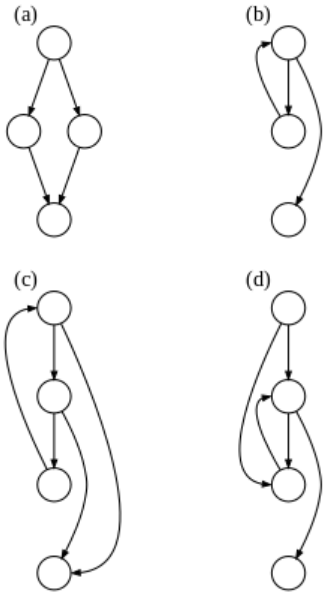
نکته قابل توجه در این مقاله این است که برای اجرای Concolic و جست‌وجوی عمق‌اول بر خلاف مقاله‌ها و ابزارهای پیشین به جای مکمل کردن آخرین قید درمسیر هربار اولین قید در آن مسیر بررسی می‌شود. همین موضوع باعث می‌شود که بتوان عمق جست‌وجو را محدود کرد.

[۱۱۳] Depth First Search (DFS)

۵-۱۰-۲ جست‌وجو بر اساس جریان کنترلی

در این روش به صورت ایستا گراف جریان کنترلی (CFG) برنامه استخراج می‌شود. سپس با توجه به آن و وزن یال‌های آن در گراف مسیر جدید از میان شاخه‌های مختلف برنامه انتخاب می‌شود.

ابتدا تعریفی کلی از CFG: در این گراف هر گره نشان‌دهنده مجموعه دستوراتی است که در آن هیچ پرشی وجود نداشته باشد. یال‌های جهت‌دار هم برای نمایش پرش‌ها هستند. در شکل ۵۲ نمونه‌ای از این CFGها دیده می‌شود.



شکل ۵۲: شکل a ساختار if-then-else
شکل b ساختار حلقه while
شکل c ساختار حلقه با دو خروجی
شکل d ساختار یک حلقه با دو ورودی و غیرقابل کاهش [۱۴]

برای اجرای Concolic ابتدا تحلیل ایستا صورت می‌گیرد و CFG برنامه استخراج می‌گردد. اگر بخواهیم خط l11 که پوشش داده نشده است را پوشش دهیم، در CFG برای l11 صفر، l10 یک، برای l1 وl2 دو و برای l6,l7,l12 و l13 مقدار بی‌نهایت قرار می‌دهیم. هیوریستیک سعی دارد مسیر با کمترین فاصله را پیدا کند.برای مثال دراجرای Concolic برنامه باx=1 وy=0 شروع به اجراکند. مسیر p0=l0,l1,l5,l7,l8,l3,l9,l13,l14,l4 با قید $x > y \wedge y \leq 0 \wedge x \neq 4$ اجرا می‌شود. برای انتخاب مسیر جدید از آنجایی که میان شاخه‌های ممکن l2,l6 و l10 شاخه l10 کمینه فاصله را دارد، انتخاب می‌شود و با مقدار جدید y=0 وx=4 برنامه دوباره اجرا می‌شود. دوباره کمینه فاصله را در نظر گرفته می‌شود و این بار مقدار جدید y=-5 وx=4 قرار می‌گیرد و نهایتا l11 اجرا می‌شود. از آنجایی که مقداردهی به هر شاخه در CFG به صورت ایستا صورت می‌گیرد و ممکن است دسترسی به یک شاخه اصلا امکان‌پذیر نباشد، بعد از یک اجرا از برنامه مقادیر فاصله‌های CFG باید دوباره به‌روز شوند.

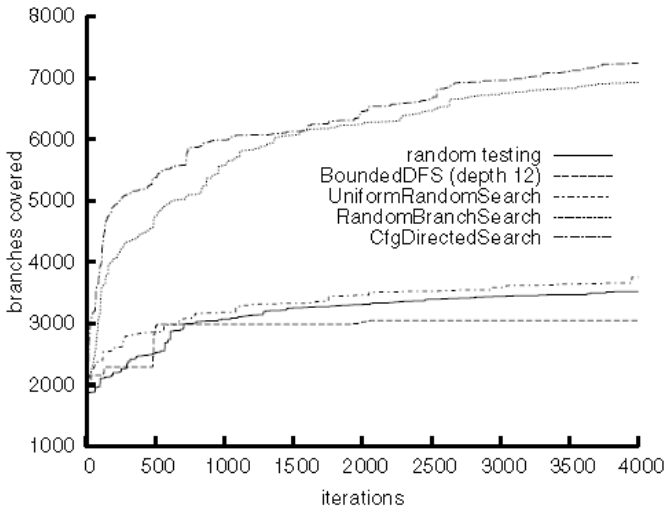
۵-۱۰-۳ جست‌وجوی دلخواه یکنواخت

در این حالت یکی از مسیرهای برنامه با انتخاب ورودی دلخواه (روش عادی Concolic) اجرا می‌شود. برای انتخاب مسیر بعدی در هر شاخه سکه اندازی صورت می‌گیرد و با احتمال ½ آن شاخه برای مسیر جدید انتخاب می‌شود و با حل قیدهای مسیر ورودی جدید تولید می‌شود.

۵-۱۰-۴ جست‌وجوی شاخه دلخواه

در این روش به صورت دلخواه از میان شاخه‌ها یکی انتخا می‌شود و برای تولید ورودی برای مسیر، این شاخه کنار گذاشته می‌شود. این کار بارها و بارها تکرار می‌شود تا نهایتا یک شاخه باقی بماند.

در نهایت با پیاده‌سازی هر چهار روش و اجرای آنها روی محک[۱۱۴]های مختلف که در مقاله توضیح داده شده‌اند، می‌توان نتیجه گرفت که ترکیب روش ایستا و پویا با هم یعنی استفاده از جست‌وجوی به وسیله CFG، می‌توان در زمان کمتر به پوشش بالاتری از دستورات برنامه دست یافت. نمونه‌ای از این مقایسه در نمودار شکل ۵۳ آمده است.



شکل ۵۳: نمودار مقایسه ۴ روش پیشنهادی روی محک Vim

[۱۱۴] Benchmark

۵-۱۱ تحلیل آلایش به کمک ابزار Avalanche

تحلیل آلایش از جمله روش‌های پویا برای کشف آسیب‌پذیری‌های برنامه است. در [۱۸] ابتدا تعریفی از این تحلیل آمده است. سپس ابزار Avalanche به عنوان ابزاری که از این تحلیل استفاده می‌کند معرفی می‌شود. از آنجایی که ایده استفاده از تحلیل آلایش در کشف آسیب‌پذیری‌ها مفید خواهد بود این مقاله بیان خواهد شد.

۵-۱۱-۱ تعریف تحلیل آلایش

داده آلوده به هر داده‌ای گفته می‌شود که از یک منبع خارجی دریافت شود. مثل فایل، شبکه و غیره. همه اطلاعات در مورد داده‌های آلوده به صورت یک سری قید بولی استخراج می‌شود. حل مجموعه این قیدها به وسیله حل کننده قید STP داده‌هایی برای ورودی تولید می‌کند که با اجرای برنامه با آن مجموعه ورودی‌ها، همان مسیر جریان داده‌های آلوده طی می‌شود.

۵-۱۱-۲ ابزار Avalanche

این ابزار برای تحلیل پویا با استفاده از چارچوبه کاری Valgrind و STP به عنوان حل کننده قید پیاده سازی شده است. اهداف تولید این ابزار عبارتند از:

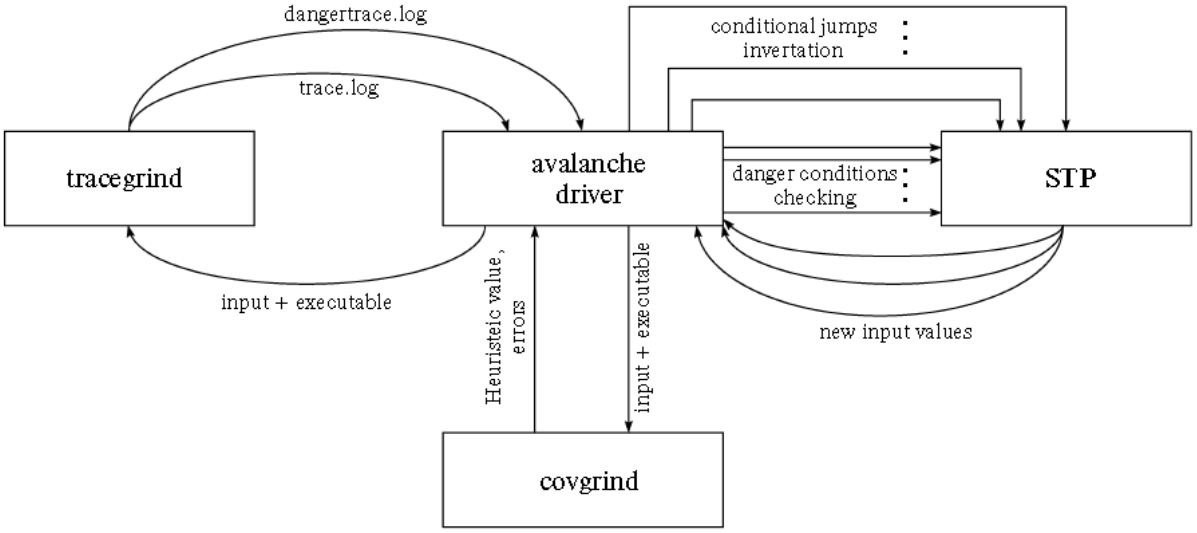
۱. نبود مثبت کاذب در نتایج
۲. تولید ورودی برای موردآزمون
۳. روش باید کامل باشد و همه شرایط و حالت‌ها را در نظر بگیرد. البته این ابزار فقط از یک منبع آلوده یعنی فایل استفاده می‌کند که این خود نقص برای ابزار است.

ابزار ۴ قسمت اصلی دارد که به شرح زیر هستند:

۱. Tracegrind: این مولفه جریان داده‌های آلوده در برنامه را دنبال می‌کند. همچنین قیدهای مسیر را جمع‌آوری می‌کند.
۲. Driver: این مولفه قیدهای جمع‌آوری شده توسط Tracegrind را به STP می‌دهد.
۳. STP: یک حل کننده قید است که ارضاپذیری[۱۱۵] قیدها را بررسی می‌کند.

[۱۱۵] Satisfiability

۴. Covgrind: این مولفه اگر قیدهای ارضا شوند تعدادی فایل به عنوان ورودی برنامه اصلی ایجاد می‌شود. برای اینکه فایلی که ما را به خطا نزدیکتر کند انتخاب شود، از یک هیوریستیک استفاده می‌شود که در Covgrind پیاده‌سازی شده است.



شکل ۵۴: معماری Avalanche

۵-۱۱-۲-۱ Tracegrind

مولفه Tracegrind دو کار اساسی انجام می‌دهد:

۱. دنبال کردن جریان داده آلوده
۲. ترکیب شرط‌ها: این شرط‌ها یا حاصل کد پرش هستند یا حاصل تحلیل دستورات خطرناک.

دستورات خطرناک با توجه به نوع خطاهایی که به دنبال آنها هستیم تعریف می‌شوند. مثلا اگر به دنبال خطای تقسیم بر صفر هستیم، همه دستورات تقسیم خطرناک خواهند بود.

Avalanche به دنبال تقسیم بر صفر و اشاره‌به فضای null (یعنی همه دستورات read یا write خطرناک خواهند بود.) است.

```
int len, fd;
…
read(fd, &len; sizeof(int));
int i = 0;
for (int j = 0; j < len; j++) {
  i++;
}
```

شکل ۵۵:کد نمونه‌ای که به اشتباه داده آلوده توسط ابزار شناسایی نمی‌شود.

۵-۱۱-۲-۲ تحلیل آلایش

برای این تحلیل فایل مشخصی به عنوان فایل

ورودی آلوده به ابزار داده می‌شود. در این ابزار فقط یک فایل به عنوان ورودی آلوده می‌توان تعریف کرد. داده‌های درون فایل در آدرس‌هایی از حافظه ذخیره می‌شوند. فضایی از حافظه که محتویات فایل آلوده در آن ذخیره شده است به عنوان آدرس‌های آلوده در نظر گرفته می‌شود. پس تمام دستورات حافظه‌ای مثل load و یا store که از این آدرس‌ها داده دریافت می‌کنند، دستورات خطرناک خواهند بود. تمامی متغیرهای موقتی که از این آدرس‌ها داده در آنها ریخته می‌شود هم آلوده در نظر گرفته می‌شوند. همچنین تمام دستوراتی که حداقل یکی از آرگومان‌های ورودی آنها آلوده باشد، خروجی آلوده خواهد داشت. اگر متغیر یا آدرسی از حافظه که آلوده بوده با داده‌های معمولی بازنویسی شوند، از حالت آلوده خارج می‌شوند. قواعد بالا تمام حالت‌هایی است که به صورت صریح داده‌های آلوده را مشخص می‌کنند. اما حالاتی مثل کد شکل ۵۵ وجود دارد که i آلوده است ولی برنامه به خطا آن را تشخیص نمی‌دهد. کلا دستورات و عملیاتی که مربوط به فراداده[۱۱۶]های فایل باشد توسط ابزار تشخیص داده نمی‌شود. این کد مربوط به اندازه فایل است.

۵-۱۱-۲-۳ تولید قیدها برای STP

در حین اجرای Tracegrind دو نوع دنباله از قیدها[۱۱۷] در نظر گرفته می‌شود:

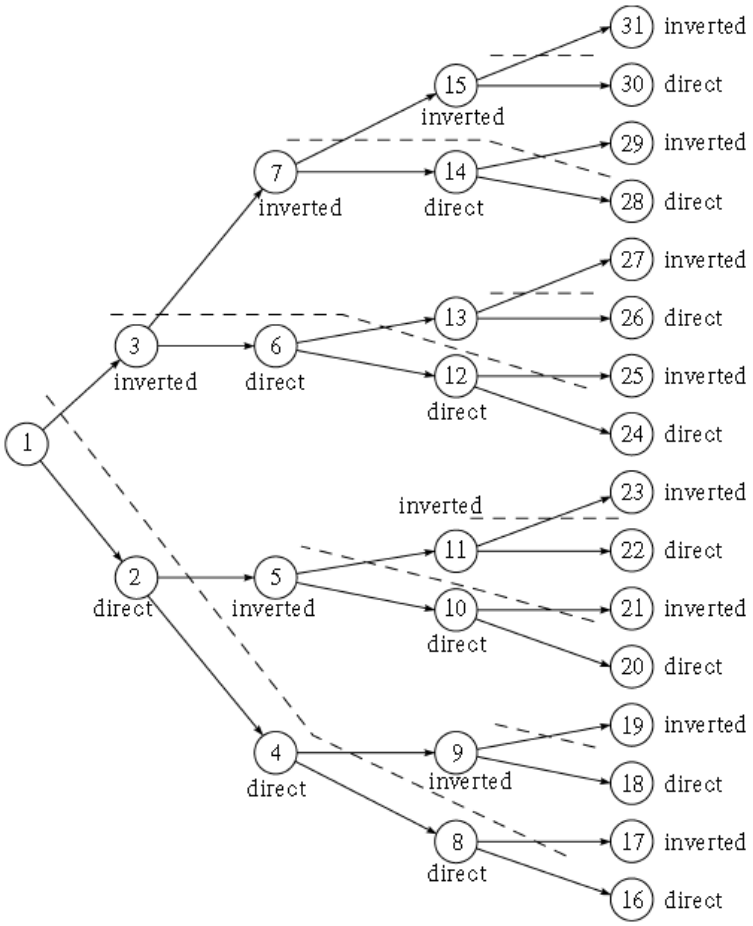
۱. دنباله‌ای از قیدهای مربوطه به پرش‌ها
۲. دنباله مربوط به دستورات خطرناک

برای مثال اگر در شرط یک عبارتِ پرش داده آلوده وجود داشته باشد، با توجه به اجرای پرش و انتخاب یک شاخه از برنامه، تشخیص داده می‌شود که داده آلوده درست یا غلط بوده است. در مورد عبارت‌های خطرناک برای مثال اگر به یک عبارت تقسیم برسد، در دنباله قیدها یک قید که نشان‌دهنده صفر بودن داده آلوده - اگر در مقسوم‌علیه باشد- است اضافه می‌شود. تمام این موارد برای تمام دستورات موجود بررسی و اجرا می‌شود.

۵-۱۱-۲-۴ Driver

این مولفه از ابزار وظیفه برقراری ارتباط میان مولفه‌های مختلف و کنترل پیمایش دنباله پرش‌ها را برعهده دارد. Driver دنباله‌های قیدهای خطرناک و پرش‌های شرطی را تجزیه و تحلیل می‌کند. در مورد دنباله قیدهای خطرناک، این قیدها به STP داده می‌شود. اگر STP بتواند آنها را حل کند یک ورودی به ازای آن دنباله

[۱۱۶] Meta Data
[۱۱۷] Trace



شکل ۵۶: نمونه‌ای از درخت تولید شده توسط Driver

ایجاد می‌شود که می‌تواند باعث ایجاد خطا در برنامه شود. Driver سپس برنامه را با ورودی‌های جدید اجرا و آزمایش می‌کند. اگر خطا رخ داد، آن ورودی‌ها به عنوان ایجاد کننده خطا ذخیره می‌شوند.

در مورد دنباله پرش‌های شرطی، Driver با توجه به شرط‌ها درختی از تمام حالت‌هایی که این شرط‌ها درست یا غلط باشند، ایجاد می‌کند و هر مسیر در درخت ایجاد شده را به STP می‌دهد تا حل کند. اگر راه حلی وجود داشته باشدف مقادیر متغیرها به عنوان ورودی ذخیره می‌شوند. هدف این است که تمام مسیرهای برنامه که داده‌های آلوده در آنها حضور دارند، پیمایش شوند.

۵-۲-۱۱-۵ Covgrind

این مولفه عملکرد ساده‌ای دارد. هنگامی که Tracegind مسیری از برنامه را اجرا می‌کند، آدرس بلوک‌هایی که پیمایش می‌شوند در فایلی ذخیره می‌شود تا دوباره پیمایش نشود. همچنین یک زمان‌سنج وجود دارد که اگر

اجرای مسیری از برنامه بیشتر از حد مجاز به طول انجامید، آن را متوقف می‌کند تا برنامه در حلقه بی‌نهایت گیر نکند.

۶ بحث و نتیجه‌گیری

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.

۶-۱ مقایسه کارهای پیشین و آینده بحث

در این قسمت چالش‌هایی که در رابطه با این حوزه مطرح می‌شود را به طور خلاصه بیان می‌کنیم. بیان مفصل هر یک و راه‌حل‌های ارائه شده برای آن‌ها، همراه با بیان ابزارهای این حوزه و نوآوری‌های هر یک در فصل ۵، کارهای پیشین و ابزارهای موجود بیان شده است.

- **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 بهبودهایی بر روش اجرای Concolic بیان شدند و در ابزار CREST پیاده‌سازی شده‌اند.
- **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. در ابزار CUTE، ابزار EXE و ابزار KLEE نمونه‌هایی از راه‌کارها بیان شده‌اند که نیاز به بهبود دارند.
- **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 ۴ خانه یک بایتی. که مورد دوم خطاهای مثل سرریزها را می‌تواند بررسی کند. یا در رابطه با اشاره‌گرها در ابزار DART اشاره‌گر به عنوان یک مقدار عددی int در نظر گرفته می‌شود. ولی در ابزار CUTE با مدل خاص خود می‌توان برابری یا نابرابری دو اشاره‌گر را بررسی و قید مربوط به آن را حل کند یا در کار ابزار EXE از تئوری آرایه‌ها استفاده می‌شود که حل‌کننده قیدهایی مثل Z3 و STP می‌توانند قیدهای مربوط به آنها را حل کنند.

- **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. در ابزار jCUTE که برای برنامه‌های چندنخی به زبان جاوا پیاده‌سازی شده‌اند نمونه‌ای از راه‌کارهای این چالش عنوان شدند.
- **چارچوبه‌های کاری مختلف**: یکی از چالش‌های امروز در مورد آزمون برنامه‌ها، توسعه برنامه برای چارچوبه‌های کاری جدید مثل گوشی‌های هوشمند همراه یا برنامه‌های تحت وب هستند. در هر یک از این چارچوبه‌های کاری چالش‌های جدیدی وجود دارد. مثلا در مقاله اجرای Concolic برای برنامه‌های گوشی همراه 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. ابزار Jalangi هم نمونه‌ای از اجرای نمادین و Concolic برای زبان جاوا اسکریپت است که در برنامه‌های تحت وب استفاده می‌شود. در این کار نحوه تحلیلِ کارای برنامه‌های تحتِ پلتفرم‌های مختلف و همچنین اجرای کارای تحلیل‌های پویا، از جمله تحلیل آلایش، عنوان شدند.

۶-۲ مسائل باز

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

۱. بهبود مسئله انفجار مسیر در اجرای Concolic با ارائه هیوریستیک کارا (روش های جستجوی هوشمند، هرس کردن مسیر، کَش پرس‌وجوهای قبلی، ترکیب تحلیل‌های ایستا و پویا، استفاده از روش‌های متن‌کاوی[۱۱۸] برای انتخاب کارای مسیرهای برنامه)
۲. اجرای نمادین روی توابع خارجی یا پیچیده ریاضی
۳. ارائه مدل حافظه کارا به منظور پوشش تمامی مسیرهای برنامه در پلتفرم‌های مختلف
۴. بهبود تحلیل‌کننده‌های قید برای ارتقای توان محاسباتی و تحلیلی آنها (مثلا محاسبات ممیز شناور)
۵. بهبود و گسترش تحلیل آلایش برای داده‌های دریافتی از شبکه (برای تحلیل پروتکل‌های تحت شبکه)
۶. بهبود و گسترش تحلیل آلایش برای داده‌های دریافتی از رابط کاربری برنامه‌های مختلف در پلتفرم‌های گوناگون
۷. اجرای Concolic روی پلتفرم گوشی‌های هوشمند همراه

[۱۱۸] Text Mining

۸. اجرای Concolic روی پلتفرم‌های تحت سیستم عامل ویندوز
۹. استفاده از اجرای Concolic در تحلیل آسیب‌پذیری‌های مختلف مثل XSS، قالب رشته، آسیب‌پذیری‌های منطقی یا زمانی[۱۱۹]
۱۰. استفاده از اجرای Concolic در تحلیل آسیب‌پذیری‌های پلتفرم‌های گوناگون مثل آسیب‌پذیری تزریق در گوشی‌های هوشمند همراه

۶-۳پروژه کارشناسی ارشد

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

۶-۳-۱عنوان پروژه

اجرای Concolic برای تشخیص آسیب‌پذیری تزریق به برنامه‌های کاربردیِ گوشی‌های هوشمند

۶-۳-۲ توضیح اجمالی پروژه

مقدمه: امروزه تعداد برنامه‌های گوشی‌های هوشمند افزایش یافته است. یکی از دغدغه‌های اصلی شرکت‌های تولید کننده این ابزارها، پیاده‌سازی نرم‌افزار خود برای سیستم‌عامل‌های مختلف است. برای پیاده‌سازی نرم‌افزار برای هر کدام از سیستم عامل‌ها، از زبان خاصی استفاده می‌شود. مثلا از جاوا برای اندروید و Object C برای iOS. راه حلی که امروزه برای افزایش جابه‌جاپذیری[۱۲۰] نرم‌افزارها استفاده می‌شود، این است که توسعه‌دهندگان از تکنولوژی وب برای پیاده‌سازی نرم‌افزارها استفاده می‌کنند. از آنجایی که سیستم‌عامل‌های محبوب مثل اندروید ، iOS و ویندوز از قابلیت اجرای برنامه‌های مبتنی بر وب پشتیبانی می‌کنند، تعداد این نرم‌افزارها رو به افزایش است. در هر کدام از این سیستم عامل‌ها مولفه‌ای برای اجرای اینگونه برنامه‌ها وجود دارد که این مولفه در اندروید webView، در iOS، UIWebView و در ویندوز WebBrowser نام دارد.[۱۹] که در این نوشته بدون از بین رفتن کلیت به آن webView خواهیم گفت. webView وظیفه اجرای کدهای HTML، CSS و جاوا اسکریپت را بر عهده دارد.

مساله موجود: کارهایی که تاکنون برای تشخیص آسیب‌پذیری به روش Concolic انجام شده است مثل [۲۳]، محدود به برنامه‌هایی بوده است که تنها ورودی آنها داده‌های کاربر بوده است. در این مقاله، برنامه‌ای

[۱۱۹] Race Condition
[۱۲۰] Portability

به زبان C در نظر گرفته شده است که تنها یک بار داده از ورودی دریافت کرده و در یک بافر می‌ریزد و در ادامه پردازش روی آنها انجام می‌شود. تاکنون از روش Concolic برای تشخیص آسیب‌پذیری‌های برنامه‌های رخدادمحور[۱۲۱]، استفاده نشده است. یکی از این نوع برنامه‌ها، برنامه‌های گوشی‌های هوشمند است. تفاوتی که این نوع برنامه‌ها دارند این است که، رخدادها علاوه بر داده‌ها در ایجاد مسیرهای اجرای برنامه موثرند. پس مسئله موجود، استفاده از روش Concolic برای تشخیص آسیب‌پذیری برنامه‌های رخدادمحور است.

ایده حل: در این پروژه برنامه‌های گوشی‌های هوشمند به عنوان یکی از برنامه‌های رخدادمحور در نظر گرفته می‌شود. همچنین تاکید بر روی برنامه‌های پیاده‌سازی شده از طریق Apache Cordova [۲۴] است که از تکنولوژی وب برای تولید برنامه‌ها استفاده می‌کند. برای تشخیص آسیب‌پذیری به روش Concolic باید ابتدا رخدادها را به طور خودکار تولید کرد. سپس با در نظر گرفتن شرط‌های تولید مسیر، شرط‌های آسیب‌پذیریِ مدنظر نیز بررسی می‌شوند تا مسیرهای خاصی از برنامه پیمایش شوند. برای دقیق‌تر شدن جست‌وجو از تحلیل آلایش[۱۲۲] نیز استفاده خواهد شد. در این پروژه برای نمونه و به طور خاص به آسیب‌پذیری تزریق کدهای جاوا اسکریبت می‌پردازیم. برای اینکه این حمله صورت پذیرد، دو شرط لازم است: اول، دریافت داده از یکی از کانال‌های داخلی یا خارجی و دوم، نمایش داده‌های دریافتی. در [۱۹] آمده است که به طور میانگین ۵۳ درصد از برنامه‌های مبتنی بر وب از توابع ناامن برای نمایش محتوا استفاده می‌کنند که این باعث می‌شود حمله تزریق کد آسان‌تر گردد.

روش ارزیابی: برای آزمون ابزار پیاده‌سازی شده و روش مورد استفاده، تعدادی برنامه دارای آسیب‌پذیریِ بیان‌شده پیاده‌سازی می‌شوند و آزمون بر روی آنها انجام خواهد گرفت. علاوه بر آن از Benchmarkهای موجود در[۲۵] OWASP و [۲۶]NIST برای آزمون استفاده خواهد شد. در نهایت توسط تعدادی برنامه کاربردی موجود در Google play نیز آزمون صورت می‌گیرد تا میزان آسیب‌پذیری موجود در برنامه‌های واقعی نیز مورد سنجش قرار گیرند.

۶-۳-۳ مراحل اجرای پروژه

مراحل اجرای پروژه در زیر آمده است:

- مطالعه روش اجرایی concolic

[۱۲۱] Event Driven
[۱۲۲] Taint Analysis

- مطالعه ساختار و معماری برنامه‌های تحت Apache Cordova
- مطالعه و ارائه روشی برای پوشش تمام مسیرهای برنامه‌های گوشی‌های همراه
- مطالعه حملات تزریق کد روی webView و برنامه‌های ترکیبی و استخراج الگو برای کشف آنها
- تغییر روش ارائه شده و اضافه کردن شروط آسیب‌پذیری و اضافه کردن تحلیل آلایش به منظور یافتن آسیب‌پذیری تزریق کد
- آزمون روش و ابزار
- نگارش پایان‌نامه

مراجع

[۱] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.

[۲] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.

[۳] Sen, Koushik, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. Vol. 30. No. 5. ACM, 2005.

[۴] Cadar, Cristian, et al. "EXE: automatically generating inputs of death." ACM Transactions on Information and System Security (TISSEC) 12.2 (2008): 10.

[۵] Liu, Bingchang, et al. "Software vulnerability discovery techniques: A survey." Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on. IEEE, 2012.

[۶] Balakrishnan, Gogul, and Thomas Reps. "WYSINWYX: What you see is not what you eXecute." ACM Transactions on Programming Languages and Systems (TOPLAS) 32.6 (2010): 23.

[۷] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI. Vol. 8. 2008.

[۸] Sen, Koushik, and Gul A. Agha. "Concolic testing of multithreaded programs and its application to testing security protocols." (2006).

[۹] Jayaraman, Karthick, et al. "jFuzz: A Concolic Whitebox Fuzzer for Java."NASA Formal Methods. 2009.

[۱۰] Kähkönen, Kari, et al. "LCT: An open source concolic testing tool for Java programs." Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'2011). 2011.

[۱۱] Sen, Koushik, et al. "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript." Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013.

[۱۲] 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.

[۱۳] Burnim, Jacob, and Koushik Sen. "Heuristics for scalable dynamic test generation." Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering. IEEE Computer Society, 2008.

[۱۴] "Control Flow Graph". Wikipedia. N.p., 2016. Web. 8 Sept. 2016.

[۱۵] Aycock, John. Computer viruses and malware. Vol. 22. Springer Science & Business Media, 2006.

[۱۶] "Buffer Overflows - OWASP". Owasp.org. N.p., 2016. Web. 10 Sept. 2016.

[۱۷] "Injection Flaws - OWASP". Owasp.org. N.p., 2016. Web. 10 Sept. 2016.

[۱۸] Isaev, I. K., and D. V. Sidorov. "The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs."Programming and Computer Software 36.4 (2010): 225-236.

[۱۹] Jin, Xing, et al. "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.

[۲۰] "CVE -CVE-2013-4710". Cve.mitre.org. N.p., 2016. Web. 30 Apr. 2016.

[۲۱] Gelperin, David, and Bill Hetzel. "The growth of software testing." Communications of the ACM 31.6 (1988): 687-695.

[۲۲] Merkow, Mark S., and Lakshmikanth Raghavan. Secure and Resilient Software Development. CRC Press, 2010.

[۲۳] Mouzarani, Maryam, Babak Sadeghiyan, and Mohammad Zolfaghari. "A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes." Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on. IEEE, 2015.

[۲۴] "Architectural Overview Of Cordova Platform - Apache Cordova". Cordova.apache.org. N.p., 2016. Web. 9 Apr. 2016.

[۲۵] "Benchmark - OWASP". Owasp.org. N.p., 2016. Web. 30 Apr. 2016.

[۲۶] "National Checklist Program Repository". nvd.nist.gov. N.p., 2016. Web. 30 Apr. 2016.

Amirkabir University of Technology
(Tehran Polytechnic)

Computer and Information Technology Engineering Department

Seminar Report

Title

Discovering Software Vulnerabilities by Concolic Execution

By
Ehsan Edalat

Supervisor
Dr. Babak Sadeghiyan

September 2016