یادگیری منطق اجرایی برنامه از روی نمونه‌های اجرا شده برنامه و به‌کارگیری آن در پایش رفتار برنامه در زمان اجرا

سعید جلیلی[1*]، حسین بلندقامت آذر[2]

۱- استادیار گروه مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه تربیت مدرس

۲- کارشناس ارشد مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه تربیت مدرس

* تهران، صندوق پستی ۱۴۳- ۱۴۱۱۵

sjalili@modares.ac.ir

(دریافت مقاله: اسفند ۱۳۸۲، پذیرش مقاله: اسفند ۱۳۸۴)

چکیده- به منظور اطمینان از صحت اجرای برنامه، رفتار برنامه مورد بررسی قرار می‌گیرد. با در اختیار داشتن کد اصلی برنامه می‌توان منطق برنامه را استخراج کرد و رفتار برنامه را شناخت. در این مقاله، دیدگاه ما نسبت به برنامه‌ها به‌صورت جعبه‌سیاه است، بنابراین از منطق اجرایی برنامه، اطلاعی در دست نیست. با این حال سعی شده است منطق اجرایی برنامه از روی اجراهای متعدد و موفق برنامه و ثبت فراخوانیهای سیستمی تولید شده توسط برنامه، با به کارگیری رویکرد استقرایی یادگیری شود. بدین منظور سازوکارهایی برای کشف حلقه‌ها و نقاط انشعاب برنامه به‌کار برده و منطق اجرایی برنامه را به صورت انتزاعی در سطح فراخوانیهای سیستمی بازسازی کرده‌ایم. آنگاه با در اختیار داشتن منطق اجرایی برنامه، همانند روشهای با نگرش جعبه‌سفید، انحراف رفتار برنامه در زمان اجرا را از منطق اجرایی آن تشخیص داده‌ایم. برای روش پیشنهادی دو کاربرد پیشنهاد شده است: (۱) ساخت سیستم تشخیص نفوذ به برنامه، (۲) حفاظت از برنامه در برابر خطا. نتایج کارایی روش پیشنهادی، با انجام آزمایشهایی ارائه شده است.

کلید واژگان: منطق اجرایی برنامه، یادگیری استقرایی، فراخوانی سیستمی، انحراف اجرای برنامه، تشخیص نفوذ، حفاظت برنامه.

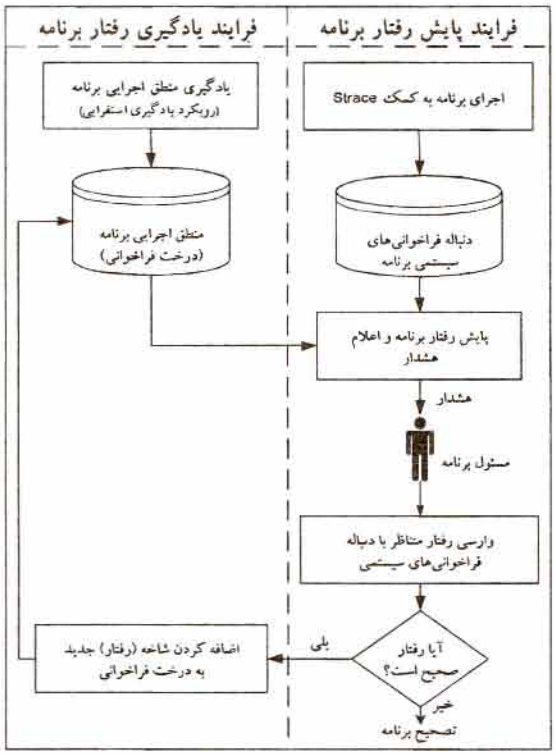
۱- مقدمه

نرم‌افزارهای مختلف امروز نقش مهمی را در زندگی پیشرفته بشریت ایفا می‌کنند. در واقع سرویسهای مختلفی بر مبنای نرم‌افزارها پیاده‌سازی شده‌اند که گاهی کنترل فعالیتهای بسیار مهمی را برعهده دارند. شکست برنامه‌های به‌کار رفته در سیستمهای حیاتی مانند سیستمهای کنترل پرواز، کنترل راکتور اتمی یا در سرویسهای اقتصادی مانند بانک الکترونیکی، ضایعات غیرقابل جبران یا هزینه‌های گزافی را تحمیل می‌کنند. حصول اطمینان از صحت کارکرد این نرم‌افزارها اهمیت فوق‌العاده‌ای دارد. بدین منظور نرم‌افزارها تحت آزمون قرار می‌گیرند. آزمون نرم‌افزار مرحلهٔ مهمی از فرایند مهندسی نرم‌افزار و به معنای اجرای برنامه است به‌طوری که خطاهای آن آشکار شود؛ اما در عمل همیشه شاهد بوده‌ایم که نرم‌افزارها با نقصهای مختلفی مواجه شده و حتی پس از گذراندن مراحل آزمون نیز اطمینان صد در صد از صحت اجرای برنامه حاصل نمی‌شود. علاوه بر این، حتی اگر آزمونهایی دقیقتر، از صحت اجرای برنامه

اطمینان کامل دهند؛ امکان آلوده شدن برنامه به قطعه کدهای مخرب را نمی‌توان نادیده گرفت.

به طور کلی دو دیدگاه مختلف برای بررسی رفتار برنامه ارائه شده است. در دیدگاه اول، نگرش ما نسبت به برنامه‌ها به صورت جعبه‌سفید[1] بوده و کد اصلی[2] برنامه در دست است. در این حالت امکان بررسی ساختارهای منطقی برنامه مانند حلقه، انشعاب و شرط وجود دارد و منطق اجرایی برنامه با تحلیل کد اصلی برنامه به صورت دستی یا خودکار بررسی می‌شود. در این نگرش، برای بررسی صحت اجرای برنامه، الزاماً نیازی به اجرای برنامه نیست [۱]. در دیدگاه دوم به برنامه‌ها به صورت جعبه‌سیاه[3] نگاه می‌شود و بنابراین اطلاعی از کد اصلی برنامه در دست نیست. در این نگرش، فقط وظیفه‌مندی[4] برنامه مشخص است و فقط با تغییر پارامترهای ورودی و اجرای برنامه و مشاهده خروجیها می‌توان صحت اجرای برنامه را بررسی کرد [۲].

در این مقاله سعی شده در هر لحظه، انطباق سیر اجرای برنامه با منطق اجرایی آن بررسی و در صورت بروز هر گونه انحراف، شکست در اجرای برنامه کشف شده و حتی در صورت امکان، قبل از وقوع شکست، اجرای برنامه به نقطه مطمئنی ارجاع داده یا با توقف اجرای برنامه، از خسارت دیدن احتمالی سایر بخشها جلوگیری شود. شکل ۱ معماری این روش را نشان می‌دهد. بخش چپ فرایند یادگیری منطق اجرایی برنامه را نشان می‌دهد. منطق اجرایی برنامه معمولاً از تجزیه و تحلیل کد اصلی برنامه به دست می‌آید؛ اما در این تحقیق نگاه ما به برنامه‌ها به صورت جعبه‌سیاه است و از منطق اجرایی آنها اطلاعی نداریم. بنابراین سعی می‌شود مانند روشهای با نگرش جعبه‌سیاه، از طریق اجراهای متعدد برنامه با ورودیها و شرایط سیستمی مختلف، منطق اجرایی برنامه را در سطح فراخوانیهای سیستمی[5] با استفاده از رویکرد استقرایی یادگیری و سپس مانند روشهای با دیدگاه جعبه‌سفید، سیر اجرای برنامه را بررسی کنیم. بخش سمت راست شکل ۱، فرایند پایش[6] رفتار برنامه را نشان می‌دهد. در حقیقت نوآوری اصلی این مقاله، یادگیری منطق اجرایی برنامه‌ها در سطحی از انتزاع (فراخوانی سیستمی) است درحالی‌که دسترسی به کد اصلی برنامه وجود ندارد. تلفیقی از دو نگرش جعبه‌سیاه و جعبه‌سفید همراه با روشی نوین در یادگیری منطق اجرایی برنامه به کار رفته است تا مسیرهای اجرایی مختلف برنامه، کشف شده و هر گونه انحراف اجرای برنامه از منطق آن، به عنوان شکستی در اجرای برنامه کشف شود.



شکل ۱ معماری بررسی انطباق سیر اجرای برنامه با منطق اجرایی برنامه

1. White-Box
2. Source Code
3. Black-Box
4. Functionality
5. System Calls
6. Monitoring

برای روش پیشنهادی دو کاربرد عمده توصیه شده است: (۱) تشخیص نفوذ به برنامه، (۲) حفاظت از برنامه در برابر خطا. در بخشهای ۲ و ۳، فراخوانیهای سیستمی و نحوه استفاده از آنها در مدل‌سازی رفتار برنامه بیان می‌شود. هدف از این تحقیق و ارائه روش پیشنهادی مبنی بر لزوم یادگیری منطق اجرایی برنامه، در بخش ۴ آورده شده است. در بخش ۵ فرایند یادگیری منطق اجرایی برنامه شامل مراحل کشف حلقه، کشف نقاط انشعاب و ایجاد منطق اجرایی برنامه، تشریح شده است. بعد از ایجاد منطق اجرایی برنامه، نحوه پایش رفتار برنامه و تشخیص انحراف اجرای برنامه، در بخش ۶ بیان می‌شود. در بخش ۷، با انجام آزمایشهایی بر روی دو برنامه lpr و stide روش پیشنهادی مورد ارزیابی و تحلیل قرار می‌گیرد. در بخش ۸، کاربردهای روش پیشنهادی ارائه شده و در بخش ۹ نتیجه‌گیری انجام می‌شود.

۲- دنباله فراخوانیهای سیستمی

یکی از ابزارهای موفق برای پایش رفتار برنامه در سیستم‌عاملهای مبتنی بر UNIX، استفاده از فراخوانیهای سیستمی است. پردازه‌ها بایستی از هسته سیستم‌عامل برای اجرای وظایفشان تقاضا کنند و فراخوانیهای سیستمی نیز واسط بین پردازه و هسته سیستم‌عامل هستند. درحقیقت فراخوانیهای سیستمی وسیله دستیابی پردازه‌ها به منابع سیستم است. پایش فراخوانیهای سیستمی پردازه، دنباله فراخوانیهای سیستمی را نتیجه می‌دهد. دنباله فراخوانیهای سیستمی، مجموعه مرتبی از فراخوانیهای سیستمی است که پردازه در طول اجرای خود از آنها استفاده کرده است. هر دنباله یکی از مسیرهای اجرایی مختلف برنامه را نشان می‌دهد. دنباله‌های فراخوانیهای سیستمی توسط برنامه‌های کاربردی مانند strace گردآوری می‌شوند. شکل ۲ نمونه بسیار کوتاه شده‌ای از خروجی برنامه strace است که برای استخراج فراخوانیهای سیستمی حاصل از اجرای برنامه‌ای مفروض به‌کار رفته است.

پارامترهای تقاضا شده در فراخوانیهای سیستمی مورد پردازش قرار نمی‌گیرند و فقط فراخوانیهای سیستمی اجرا شده، جزء با اهمیت خروجی تلقی می‌شوند. مطابق شکل ۲، دنباله فراخوانیهای سیستمی به صورت زیر خواهد بود:

munmap, open, fcntl64, fcntl64, fstat64, setresuid32, close.

```
munmap(0x40017000, 43532)                = 0
open("/etc/passwd", O_RDONLY)            = 3
fcntl64(0x3, 0x1, 0, 0x1)                = 0
fcntl64(0x3, 0x2, 0x1, 0x1)              = 0
fstat64(3, {st_mode=S_IFREG|0644,
        st_size=1186, ...})              = 0
setresuid32(0xffffffff, 0, 0xffffffff)   = 0
close(3)                                 = 0
```

شکل ۲ نمونه‌ای از خروجی strace

این دنباله، مسیر طی شده از بین مسیرهای اجرایی مختلف برنامه مولد دنباله را به صورت انتزاعی در سطح فراخوانی سیستمی نشان می‌دهد.

البته سطوح دیگری نیز برای پایش رفتار برنامه وجود دارد که در این مقاله مورد توجه نیستند. به‌عنوان مثال: Jones و همکاران در [۳] از فراخوانیهای کتابخانه‌ای استفاده کرده‌اند. Stillerman در [۴] از فراخوانیهای روش[1] CORBA برای بررسی رفتار برنامه‌های توزیع‌شده استفاده کرده است. Inoue در [۵] نشان داد که نمای رفتاری برنامه‌های نوشته شده به زبان جاوا را می‌توان از پایش فراخوانیهای روش جاوا در محیط ماشین مجازی جاوا به‌دست آورد.

۳- یادگیری و مدل‌سازی رفتار برنامه

ایده استفاده از فراخوانیهای سیستمی را از مجموعه تلاش‌هایی که برای تشخیص نفوذ[2] به برنامه شده است، به دست آورده‌ایم. در این تلاشها سعی شده نمای رفتاری

1. Method Invocations
2. Intrusion Detection

برنامه[1] بر اساس فراخوانیهای سیستمی برنامه، تهیه شده و انحراف اجرای برنامه از نمای رفتاری برنامه، تشخیص داده شود. در حقیقت تشخیص نفوذ به روش تشخیص رفتار غیرعادی[2] انجام می‌شود.

Ko و همکاران در [۶] با نگرش جعبه‌سفید به برنامه‌ها، رفتار برنامه‌های با اختیارات ریشه[3] را با سیاستهایی مدل کرده‌اند که در حقیقت توصیفی است از آنچه برنامه مورد نظر توانایی انجام آن را دارد. این سیاستها با تحلیل دستی کد اصلی برنامه به‌دست می‌آیند و بنابراین به دانش کافی در بارهٔ توابع عملیاتی برنامه نیاز است. گرچه زبان مورد استفاده برای توصیف سیاستها زبان ساده‌ای است و سیاستهای خلاصه شده‌ای را می‌پذیرد؛ اما نیاز به تحلیل دستی کد برنامه، نتایج این کار را سخت تحت تأثیر قرار داده است. بنابراین Ko در کار بعدی خود [۷] سعی کرد روش دستی و خودکار را برای تولید سیاست ترکیب کند. Wagner و همکاران نیز در [۸] توانستند با تحلیل خودکار کد اصلی برنامه، نمای رفتار عادی برنامه را در چند مدل مختلف ایجاد کنند. بر طبق این روش، هرگاه فراخوانی سیستمی تقاضا شده با کد اصلی برنامه تطابق نداشته باشد، انحراف در اجرای برنامه اعلام می‌شود.

بیشترین تلاشها، با نگرش جعبه‌سیاه انجام شده است. در این نگرش، نمای رفتاری برنامه با توجه به دنباله‌های فراخوانیهای سیستمی حاصل از اجراهای قبلی برنامه ایجاد می‌شود. در سال ۱۹۹۶، Forrest و همکاران نشان دادند که می‌توان با اعمال پنجره لغزان[4] بر دنباله فراخوانیهای سیستمی حاصل از اجراهای موفق و عادی برنامه، نمای رفتاری برنامه را مدل‌سازی کرد. Forrest و همکاران راهکاری را ارائه کردند که بر طبق آن، تشخیص انحراف رفتار برنامه به روش "تطابق کامل" انجام می‌شود [۹]. در این روش، پنجره‌ای با طول ثابت k بر روی دنباله‌های فراخوانیهای سیستمی حاصل از اجراهای عادی برنامه لغزیده و توالیهای به‌دست آمده در یک پایگاه داده ذخیره می‌شوند. توالیهای فراخوانیهای سیستمی منحصر به فرد موجود در پایگاه داده، نمای رفتاری برنامه را تشکیل می‌دهند. تشخیص رفتار غیرعادی با جستجوی توالیهای فراخوانیهای سیستمی حاصل از اجرای فعلی برنامه در پایگاه داده انجام می‌شود. اگر نسبت تعداد فقدان توالی‌ها به کل تعداد توالیهای استخراج شده از دنباله، از حد آستانهٔ پیش‌فرض بیشتر شود، دلیلی بر رفتار غیرعادی برنامه تلقی می‌شود. Hofmeyer و همکاران در ادامهٔ کار قبلی، سازوکار بهتری را برای تشخیص رفتار غیرعادی معرفی کردند [۱۰]؛ بدین ترتیب که سازوکار تهیه نمای رفتاری برنامه تغییری نکرده است اما اگر توالی جاری مطابق با نمای رفتاری برنامه نبود، کمترین اختلاف آن با توالیهای منحصر به فرد موجود در نمای رفتاری برنامه، محاسبه می‌شود. اگر این مقدار از حد آستانه‌ای بیشتر باشد، توالی مزبور غیرعادی شناخته می‌شود. Helman و همکاران [۱۱] راهکاری را مبتنی بر روش آماری ارائه می‌کنند که در آن نمای رفتاری برنامه، با توجه به بسامد تکرار توالیهای عادی و غیرعادی تعریف می‌شود. در این روش بسامد تکرار هر توالی منحصر به فرد محاسبه می‌شود. توالیهایی که بسامد کمتری در دنباله‌های عادی یا بسامد بیشتری در دنباله‌های غیرعادی داشته باشند، به احتمال زیاد بیانگر رفتار غیرعادی برنامه هستند. Lee و همکاران [۱۲] از روش داده‌کاوی[5] استفاده کرده و توانستند با استفاده از ابزار یادگیری قانون RIPPER از روی توالیهای فراخوانیهای سیستمی، نمای رفتاری برنامه را به صورت مجموعه قوانینی مدل‌سازی کنند. Ghosh و همکاران [۱۳] توانستند با استفاده از شبکه‌های عصبی انتشار به عقب[6] و Elman نمای رفتاری برنامه را مدل‌سازی کنند و رفتار غیرعادی آن را تشخیص دهند.

1. Program Behaviour Profile
2. Anomaly Detection
3. Privileged Programs
4. Sliding Window
5. Data Mining
6. Backpropagation

Warrender و همکاران [۱۴] توانستند نمای رفتاری برنامه را با استفاده از نوعی ماشین حالتهای متناهی بسیار قدرتمند به نام HMM[1] مدل‌سازی کنند و رفتار غیرعادی برنامه‌ها را تشخیص دهند. گرچه استفاده از HMM کارایی بالایی ارائه کرده اما نسبت به هزینه محاسباتی بسیار بالای آن، کارایی چندان قابل توجهی را ارائه نمی‌کند و روشهای ساده‌تر به سبب هزینه پایین‌تر و کارایی قابل قبول ترجیح داده می‌شوند. Eskin و همکاران [۱۵] سعی کردند با دقت بیشتری نمای رفتاری برنامه را تشکیل دهند. بدین ترتیب که نوعی روش مبتنی بر آنتروپی را برای تعیین اندازه بهینه برای پنجره لغزان ارائه کردند. آنان معتقدند که پنجره لغزان با اندازه ثابت، دقت کمتری در تهیه نمای رفتاری برنامه دارد. بنابراین با استفاده از پنجره‌ای با اندازه وابسته به متن توانستند به کارایی بالاتری دست یابند.

۴- مدل‌سازی منطق اجرایی برنامه

هدف این تحقیق، مراقبت از رفتار برنامه در زمان اجرای آن با اتکا به منطق برنامه است. بنابراین لازم است منطق برنامه به شکلی ارائه شود که در زمان اجرای برنامه بتوان رفتار آن را وارسی کرد. یادآوری می‌شود که در سیستم عامل UNIX می‌توان فراخوانیهای سیستمی برنامه‌ها را در زمان اجرای آنها ثبت و نگهداری کرد. در نتیجه چنانچه منطق برنامه‌ها براساس فراخوانیهای سیستمی آنها ارائه شود، در این‌صورت وارسی رفتار برنامه‌ها در زمان اجرا به راحتی امکان‌پذیر می‌شود. ما این گونهٔ خاص از منطق برنامه را- که در زمان اجرای برنامه قابل ردگیری است- منطق اجرایی[2] برنامه می‌نامیم.

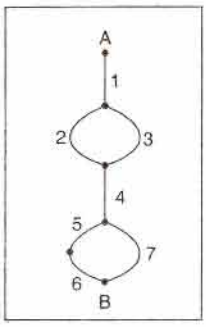
فراخوانیهای سیستمی موجود در هر دنباله، مسیر اجرایی طی شده توسط برنامه (به‌بیان دیگر بخشی از رفتار برنامه) را نشان می‌دهند، در نتیجه تمامی مسیرهای اجرایی ممکن برنامه (یعنی تمام رفتارهای برنامه) را می‌توان با گراف فراخوانی[3] نشان داد. گراف فراخوانی، گرافی است که هر مسیری از آن، یک مسیر اجرایی بالقوه برنامه است. به‌بیان دیگر، آن فراخوانی، ساختار رفتاری برنامه را مدل‌سازی و مسیرهای اجرایی ممکن را مشخص می‌کند. با در دست داشتن کد اصلی برنامه‌ها (نگرش جعبه‌سفید)، می‌توان گراف فراخوانی هر برنامه را با تحلیل کد اصلی برنامه به‌دست آورد، اما حتی با در اختیار داشتن کد اصلی برنامه، ایجاد گراف فراخوانی برنامه به‌صورت کامل و صحیح، غیر عملی است [۱۵]. زیرا مسیرهای اجرایی برنامه نه فقط به کد اصلی برنامه، بلکه به پیش‌فرضهای مترجم مورد استفاده و سیستم‌عامل وابسته است؛ این عوامل مشخص می‌کنند که در نقاط انشعاب، کدام مسیر اجرایی بایستی توسط برنامه طی شود. مهمتر اینکه، گراف فراخوانی ممکن است شامل مسیرهایی باشد که بر طبق کد اصلی برنامه، عادی شناخته می‌شوند اما در حقیقت عادی نیستند و بازتاب خطاهای منطقی برنامه محسوب می‌شوند. محدودیت دیگر ساخت گراف فراخوانی به شکل بالا، داشتن مسیرهای اجرایی غیرممکن است [۸]. به‌عنوان مثال، شبه کد مفروض در شکل ۳، دارای گراف فراخوانی مطابق شکل ۴ است.

```
System_Call(1)
If x then
        System_Call(2);
        y=true;
else
        System_Call(3)
        y=false;
end if
System_Call(4)
If y==true then
        System_Call(5)
        System_Call(6)
else
        System_Call(7)
end if
```

شکل ۳ شبه‌کد نمونه

1. Hidden Markov Model
2. Runtime Logic
3. Call Graph

برطبق مسیرهای موجود در گراف فراخوانی شکل ۴، مسیرهای 7«4«2«1 و 6«5«4«3«1 جزو مسیرهای عادی برنامه محسوب می‌شوند درحالی‌که مطابق کد اصلی برنامه، اجرای چنین مسیرهایی غیرممکن است.



شکل ۴ گراف فراخوانی حاصل از شبه کد شکل ۳

از بحثی که در بالا ارائه شد نتیجه‌گیری می‌شود که منطق اجرایی برنامه را نمی‌توان به روش ایستا به‌طور مستقیم از روی کد اصلی برنامه استخراج کرد. اما با استفاده از نوعی روش پویا (با اجرای برنامه به کمک strace) می‌توان فراخوانیهای سیستمی برنامه را ثبت کرده و سپس هر دنباله- متناظر با رفتار صحیحی از برنامه- را در گراف فراخوانی برنامه در مکان متناظر با منطق برنامه قرار داد و بدین‌صورت گراف فراخوانی معادل منطق اجرایی برنامه را ایجاد کرد.

در این‌صورت، تمام پارامترهایی که در تعیین مسیر اجرایی برنامه نقش دارند (پارامترهایی که ناشی از پیش‌فرضهای مترجم یا سیستم‌عامل هستند)، در منطق اجرایی برنامه ملحوظ خواهند شد.

بدین ترتیب نه فقط مسیرهای اجرایی غیرممکن، بلکه حتی مسیرهایی که بنابه شرایط واقعی سیستم، عبور از آنها ممکن نباشد، در منطق اجرایی برنامه جای نخواهند گرفت.

نکته مهمی که باید به آن توجه شود این است که در این تحقیق، فرض بر آن است که منطق برنامه و کد اصلی برنامه در دست نیست (نگرش جعبه‌سیاه)؛ مانند بسیاری از بسته‌ها[1]، قطعات نرم‌افزاری[2] خریداری شده یا توابع موجود در کتابخانه‌های نرم‌افزاری[3]. در نتیجه نمی‌توان مشابه روشی که در بالا ارائه شد (نگرش جعبه‌سفید) منطق اجرایی برنامه‌ها را تولید کرد. در این تحقیق، سعی شده با گردآوری دنباله‌های فراخوانیهای سیستمی حاصل از اجراهای مختلف و صحیح برنامه، مسیرهای اجرایی مختلف برنامه را (در قالب دنباله‌های فراخوانیهای سیستمی) به‌دست آورده و با استفاده از رویکرد کلی‌سازی[4] که در بخش ۵ تشریح می‌شود، منطق اجرایی برنامه را به روش استقرا، یادگیری کنیم. در مرحله بعد، تطابق سیر ایجاد فراخوانیهای سیستمی حاصل از اجرای برنامه را با منطق اجرایی آن برنامه وارسی می‌کنیم تا در صورت عدم تطابق فراخوانیهای سیستمی، هشدار لازم را اعلام کنیم.

۵- یادگیری منطق اجرایی برنامه

در نگرش جعبه‌سیاه به برنامه‌ها، تقریباً در تمامی راهکارهای ارائه شده قبلی، به علت عدم دسترسی به کد اصلی برنامه، سعی شده تا با جمع‌آوری دنباله فراخوانیهای سیستمی ناشی از اجراهای مختلف برنامه، منطق اجرایی برنامه به کمک توالیهای کوچک فراخوانیهای سیستمی ایجاد شود. این توالی‌ها از لغزاندن پنجره‌ای کوچک بر روی دنباله‌های فراخوانیهای سیستمی به‌دست می‌آیند. فلسفه استفاده از پنجره لغزان این است که ظهور فراخوانیهای سیستمی در هر برنامه، از ترتیب خاصی پیروی می‌کند که وابسته به کد اصلی برنامه است.

با اینکه نگرش ما به برنامه‌ها جعبه‌سیاه است؛ هدف آن است که مانند روشهای مبتنی بر نگرش جعبه‌سفید، مسیرهای اجرایی برنامه را از روی اجراهای مختلف و صحیح برنامه یادگیری کرده، منطق اجرایی برنامه را

1. Packages
2. Off-the-Shelf software components
3. Software Libraries
4. Generalization

به‌دست آوریم. در این راستا به کشف ساختارهای حلقه و انشعاب در برنامه‌ها توجه ویژه‌ای مبذول شده است.

شکل ۵ فرایند یادگیری استقرایی منطق اجرایی برنامه از روی نمونه‌های اجرا شده برنامه را نشان می‌دهد.

شکل ۵ فرایند کشف منطق اجرایی برنامه

در این فرایند، ابتدا برنامه صدها یا چند هزار بار برای ورودیهای مختلف و صحیح- تحت نظارت برنامه strace- اجرا و دنباله فراخوانیهای سیستمی متناظر با هر مرتبه اجرای برنامه در فایل جداگانه‌ای ذخیره می‌شود. در قدم دوم، به منظور فشرده‌سازی دنباله‌ها، تکرار توالیهایی از فراخوانیهای سیستمی در هر دنباله را- که بیانگر وجود حلقه‌ای در برنامه باشد- کشف کرده و با کلی‌سازی این توالیهای تکراری (که نوعی یادگیری استقرایی است) مجموعه فراخوانیهای سیستمی مربوط به حلقه را فشرده کرده، و در نتیجه منطق اجرایی برنامه را بهتر نمایان می‌کنیم. در قدم سوم، دنباله‌هایی را- که تفاوت آنها فقط در دفعات تکرار حلقه‌های یکسان است- شناسایی کرده و از روی هر مجموعه، یک دنباله را که دنباله منحصر به فرد می‌خوانیم به عنوان نماینده آن مجموعه ایجاد می‌کنیم. یادآوری می‌شود که هر دنباله منحصر به فرد، متناظر با یک مسیر خاص از منطق برنامه است. در قدم چهارم با مقایسه دنباله‌های منحصر به فرد با یکدیگر، نقاط انشعاب دنباله‌ها را کشف کرده و مسیرهای اجرایی مختلف برنامه را به صورت درخت نمایش می‌دهیم. درخت مسیرهای اجرایی برنامه را درخت فراخوانی[1] برنامه می‌نامیم و در قدم پنجم آن را در یک فایل با استفاده از تعدادی نشانه ذخیره می‌کنیم. در ادامه، روش کشف حلقه، روش کشف نقاط انشعاب و ایجاد منطق اجرایی برنامه و روش ذخیره منطق اجرایی برنامه در فایل توضیح داده می‌شود.

۵-۱- کشف حلقه

با بررسی دنباله‌های فراخوانیهای سیستمی می‌توان تکرار توالیهای خاصی از فراخوانیهای سیستمی را به‌وضوح دید. مثلاً در مجموعه داده‌های عادی برنامه lpr در محیط UNIX، به‌سادگی می‌توان تکرار یک حلقه شامل یک جفت فراخوانی سیستمی read-write را مشاهده کرد. در حقیقت تفاوت اصلی بسیاری از دنباله‌های lpr، در حجم فایل ارسالی برای چاپ است. با اینکه برنامه، مسیر اجرایی یکسانی را طی می‌کند اما تفاوت در میزان تکرار حلقه‌های برنامه، دنباله‌های متفاوتی را ایجاد می‌کند. با شناسایی حلقه‌های برنامه، نه‌تنها می‌توان دنباله‌های فراخوانیهای سیستمی را فشرده کرد، بلکه مسیرهای اجرایی برنامه را بدون توجه به تکرار متغیر حلقه‌ها می‌توان به دست آورد. کشف حلقه، به شناخت منطق اجرایی برنامه کمک شایانی می‌کند و با فشرده کردن دنباله‌ها، ایجاد نمای رفتاری برنامه را امکان‌پذیر می‌سازد.

به منظور کشف حلقه، تکرار توالیهای فراخوانیهای

1. Call Tree

سیستمی را در دنباله‌ها بررسی می‌کنیم. در زیر، پروسه‌ای بازگشتی[1] که به منظور کشف حلقه استفاده می‌شود با یک مثال تشریح شده است.

عبارت (۱) دنباله فراخوانیهای سیستمی حاصل از اجرای عادی و صحیح برنامه‌ای فرضی را نشان می‌دهد. در این دنباله، تکرار فراخوانیهای سیستمی 18 و 7 دیده می‌شود.

6,15,14,18,18,7,7,7,18,18,18,18,7,7,18,18,18,7
,7,7,7,7,13 (۱)

با قراردادن توالی فراخوانی سیستمی تکرار شونده درون {} و ذکر میزان تکرار توالی، دنباله به شکل عبارت (۲) در می‌آید.

$$6,15,14,\{18\}^2,\{7\}^3,\{18\}^4,\{7\}^2,\{18\}^3,\{7\}^5,13 \quad (۲)$$

بدون توجه به میزان تکرار حلقه‌ها و با گسترش دامنه تحت پوشش حلقه، می‌توان تکرار توالی {7},{18} را تشخیص داد. در این میان توالیهای تکرار شونده {18} و {7} تکرارهای متفاوتی دارند که کمترین و بیشترین میزان تکرار آنها را به صورت اندیس‌هایی در پایین و بالای نماد { نمایش می‌دهیم. بدین ترتیب دنباله به شکل عبارت (۳) در می‌آید.

$$6,15,14,\{\{18\}_2^4,\{7\}_2^5\}_3^3,13 \quad (۳)$$

در اینجا مرحله کشف حلقه به پایان می‌رسد. عبارت (۳) شکلی فشرده و قابل فهم از دنباله مفروض را ارائه می‌کند. توالیهای فراخوانیهای سیستمی که تکرار بیش از یک بار داشته‌اند، کشف حلقه‌های برنامه را ممکن می‌سازند. بدین ترتیب می‌توان تصور کرد که دنباله عبارت (۱) توسط شبه کد ارائه شده در شکل ۶ ایجاد شده است.

۵-۲-کشف نقاط انشعاب و ایجاد منطق اجرایی برنامه

نقاط انشعاب برنامه با مقایسه دنباله‌های فراخوانیهای سیستمی حاصل از اجراهای مختلف برنامه به‌دست می‌آید. یادآوری می‌شود که هر دنباله، متناظر با یک بار اجرای برنامه است. بدین منظور باید برنامه، صدها یا چند هزار بار به صورت عادی و با ورودیهای مختلف و صحیح، بدون شکست و در شرایط سیستمی واقعی، اجرا و دنباله‌های فراخوانیهای متناظر با آنها ثبت و ذخیره شود.

```
System_Call(6)
System_Call(15)
System_Call(14)
loop (min=3،max=3)
    loop (min=2،max=4)
        System_Call(18)
    end
    loop (min=2،max=5)
        System_Call(7)
    end
end
System_Call(13)
```

شکل ۶ شبه کدی که برای دنباله عبارت (۱) می‌توان در نظر گرفت

عبور از بیشترین مسیرهای اجرایی ممکن برنامه در طول اجراهای مختلف، بسیار مهم و کلیدی است. وجود حلقه‌های متعدد در برنامه، مقایسه دنباله‌های فراخوانیهای سیستمی حاصل از اجراهای متعدد برنامه را بسیار مشکل می‌سازد. بنابراین ابتدا مرحلهٔ کشف حلقه بر روی تک‌تک دنباله‌ها انجام می‌شود. پس از کشف حلقه، دنباله‌هایی که تفاوت آنها، فقط در تعداد دفعات اجرای حلقه(ها)ی یکسان است، به‌عنوان مصادیق اجرای یک مسیر خاص از برنامه در نظر گرفته می‌شوند، در نتیجه مجموعه دنباله‌های فراخوانیهای سیستمی برنامه، با توجه به معیار بالا دسته‌بندی و از هر دسته، یک دنباله به‌عنوان نمایندهٔ آن دسته- که یک دنباله منحصر به فرد خوانده می‌شود و متناظر با یک مسیر خاص از برنامه است- انتخاب می‌شود. تعداد دفعات تکرار هر دنبالهٔ منحصر به فرد نیز نگهداری می‌شود تا در محاسبه احتمال رخداد هر شاخه از مسیرهای اجرایی برنامه، استفاده شود. بدیهی است که شاخه‌هایی که احتمال رخداد آنها بیشتر از سایر شاخه‌ها

1 .Recursive

باشد، مسیرهای اصلی برنامه را نشان می‌دهند.

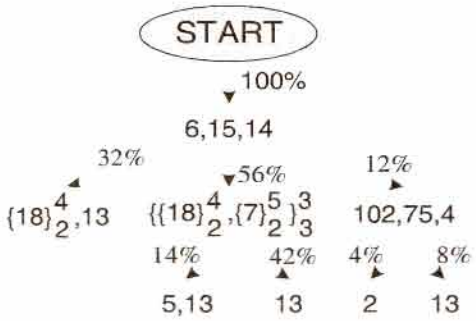
به منظور کشف نقاط انشعاب برنامه در مسیرهای اجرایی ممکن، فراخوانیهای سیستمی موجود در دنباله‌های منحصر به فرد، به‌ترتیب از ابتدای دنباله‌ها با یکدیگر مقایسه و نتیجه به صورت نموداری درختی ارائه می‌شود. اختلاف در فراخوانی سیستمی در هر دنباله، باعث ایجاد اشتقاق در درخت می‌شود. درحقیقت نمودار درختی به‌دست آمده، مسیرهای اجرایی مختلف برنامه را نشان می‌دهد و ما آن را درخت فراخوانی یا منطق اجرایی برنامه می‌نامیم. در زیر، پروسهٔ کشف نقاط انشعاب برنامه و ایجاد منطق اجرایی برنامه با یک مثال تشریح شده است.

مثال: برنامه‌ای ۱۰۰ بار به طور عادی و صحیح و بدون شکست اجرا شده و از دنباله‌های فراخوانیهای سیستمی حاصل، بعد از کشف حلقه، فقط پنج دنبالهٔ منحصر به فرد باقی مانده است. دنباله‌های منحصر به فرد و فراوانی هر یک در شکل ۷ نشان داده شده است. شکل ۸ درخت منطق اجرایی (درخت فراخوانی) برنامهٔ متناظر با این چهار دنباله را نمایش می‌دهد.

1:	6,15,14,$\{18\}_2^4$,13	%Iteration=32
2:	6,15,14,102,75,4,2	%Iteration=4
3:	6,15,14,$\{\{18\}_2^4,\{7\}_2^5\}_3^3$,13	%Iteration=42
4:	6,15,14,102,75,4,13	%Iteration=8
5:	6,15,14,$\{\{18\}_2^4,\{7\}_2^5\}_3^3$,5,13	%Iteration=14

شکل ۷ دنباله‌های منحصر به فرد حاصل از اجرای برنامه

منطق اجرایی برنامه (به بیان دیگر درخت فراخوانی شکل ۸)، با یک گره آغازین شروع می‌شود و گره‌های فرزند، توالیهای فراخوانیهای سیستمی هستند که در اجراهای مختلف مشترکند. اختلاف موجود در فراخوانیهای سیستمی در دنباله‌ها، باعث ایجاد اشتقاق در درخت شده و مسیرهای اجرایی مختلف را جدا می‌کند.



شکل ۸ منطق اجرایی برنامه متناظر با دنباله‌های منحصر به فرد شکل ۷

منطق اجرایی برنامه، مسیرهای اجرایی برنامه را به‌وضوح از یکدیگر جدا می‌کند و بر خلاف گراف فراخوانی، احتمال ایجاد مسیری غیرممکن را از بین می‌برد. نکتهٔ مهم آن است که: اگر در آینده مشخص شود که مسیری از برنامه در منطق اجرایی برنامه جای نگرفته (به این دلیل که آن شاخه از برنامه در فرایند یادگیری منطق اجرایی برنامه هرگز اجرا نشده)، با توجه به ساختار درختی منطق اجرایی برنامه، به‌سادگی می‌توان شاخه جدیدی را که متناظر با آن مسیر است به درخت اضافه کرد. بنابراین کشف منطق اجرایی برنامه می‌تواند در یک فرایند زمانی و طبق یک پروسهٔ تکمیل شونده[1] انجام شود.

۵-۳- ذخیره‌سازی منطق اجرایی برنامه

با پیمایش منطق اجرایی برنامه و ذخیرهٔ گره‌ها به کمک نشانه‌های تعریف شده، درخت فراخوانی برنامه را در یک فایل می‌توان ذخیره کرد. پیمایش درخت فراخوانی از گره آغازین شروع شده و سپس، زیردرختهای آن به‌ترتیب از چپ به راست به صورت بازگشتی پیمایش می‌شوند. نشانه‌های کمکی که بدین منظور استفاده می‌شود، چنین تعریف شده‌اند:

نشانه‌های] و [که به‌ترتیب نشان‌دهندهٔ شروع و خاتمه اشتقاق در درخت است.

1. Incremental

نشانه‌های) و (که به‌ترتیب نشان‌دهندهٔ شروع و خاتمهٔ یک شاخه در درخت است.

نشانه‌های } و { که به‌ترتیب نشان‌دهندهٔ شروع و خاتمه یک حلقه در شاخه‌ای از درخت است.

بعد از نشانه] تعداد شاخه‌های مشتق شده قرار می‌گیرد. بعد از نشانه { کمترین و بیشترین میزان تکرار حلقه قرار می‌گیرد و در نهایت بعد از نشانه) احتمال رخداد آن شاخه قرار می‌گیرد. با توجه به این تعاریف، درخت شکل ۸ در فایلی به صورت شکل ۹ ذخیره می‌شود.

۶- پایش رفتار برنامه

هدف اصلی این تحقیق، مراقبت از رفتار برنامه‌ها در زمان اجرای آنها به صورت بی‌درنگ[1] است به نحوی که هر گونه انحراف از رفتار برنامه، اعلام شود و در صورت نیاز، ادامهٔ اجرای برنامه متوقف گردد. به منظور عملیاتی کردن پایش رفتار برنامه‌ها، اولاً بایستی رفتار صحیح برنامه‌ها (به‌عبارت دیگر منطق اجرایی برنامه‌ها در قالب درخت فراخوانی آنها) یادگیری شود. ثانیاً برنامه‌ها با کمک strace اجرا شوند تا به موازات اجرای آنها، دنباله فراخوانیهای سیستمی آنها ثبت شود. در این صورت واحد تشخیص انحراف اجرای برنامه، دنباله فراخوانیهای سیستمی حاصل از اجرای فعلی برنامه را با درخت منطق اجرایی برنامه تطبیق می‌دهد و هر گونه مغایرت در تطابق را به‌عنوان انحراف رفتار برنامه تعبیر می‌کند.

واحد تشخیص انحراف اجرای برنامه، به موازات اجرای برنامه عملیات خود را انجام می‌دهد، در نتیجه به صورت بی‌درنگ رفتار برنامه را پایش می‌کند. در ادامه چگونگی بارگذاری منطق اجرایی برنامه در حافظه و تشخیص انحراف از رفتار شناخته شده برنامه شرح داده می‌شود.

```
(P=100% 6 15 14) [Branch=3 (P=32% {18}
Min=2 Max=4 13) (P=56% {{18} Min=2 Max=4
{7} Min=2 Max=5 } Min=3 Max=3) [Branch=2
(P=14% 5 13) (P=42% 13) ] (P=12% 102 75 4)
[Branch=2
(P=4% 2 ) (P=8% 13) ] ]
```

شکل ۹ ذخیرهٔ درخت فراخوانی (منطق اجرایی برنامه) در فایل

۶-۱- بازسازی منطق اجرایی برنامه در حافظه

ساختاری که قبلاً برای ذخیره منطق اجرایی برنامه به کار رفته ، بازسازی آن را در حافظه تسهیل می‌کند؛ زیرا ابتدا گره‌های پدر و سپس گره‌های فرزند ذخیره شده و تعداد گره‌های فرزند در گره پدر ثبت شده است. برای بازسازی منطق اجرایی برنامه در حافظه، ابتدا ساختار داده‌ای جامع باید برای گره‌های درخت فراخوانی تعریف شود به‌طوری‌که با یک قالب خاص بتوان هر نوع گرهی را بازسازی کرد. ساختار دادهٔ استفاده شده در شکل ۱۰ نشان داده شده است. بر طبق این ساختار، اندازه هر گره درخت برحسب بایت در Node Size و احتمال رخداد آن گره در قسمت Node Possibility نگهداری می‌شود. هر فراخوانی سیستمی در قسمت SC ذخیره می‌شود. فراخوانیهای سیستمی که داخل حلقه باشند، بین دو نشانه Loop Start و Loop End قرار می‌گیرند و Min و Max نشان‌دهندهٔ کمترین و بیشترین میزان تکرار حلقه است. برای مشخص کردن انتهای توالی موجود در هر گره، از نشانه End استفاده شده است. اولین فراخوانی سیستمی مربوط به گره‌های فرزند در قسمت Next Possible SC و اشاره‌گر به آن گره در بخش Pointer ذخیره می‌شود. طول نشانه‌های استفاده شده یک بایت است. این ساختار طوری طراحی شده که حجم کمی را در حافظه اشغال کرده و سرعت بالایی را در انتخاب گره بعدی برای پیمایش درخت ارائه می‌کند.

1. Real Time

شکل ۱۰ ساختار داده گره‌های درخت فراخوانی برنامه

۶-۲- تشخیص انحراف اجرای برنامه

با توجه به در دست داشتن منطق اجرایی برنامه می‌توان مانند روشهای با نگرش جعبه‌سفید، سیر اجرای برنامه را با منطق اجرایی آن مقایسه و در صورت مشاهدهٔ هر گونه تخطی در اجرای برنامه، انحراف اجرای برنامه از منطق اجرایی آن را متذکر شد. بدین منظور سازوکار بسیار ساده و کارایی برای تشخیص انحراف اجرای برنامه به کار رفته است. بدین صورت که واحد تشخیص انحراف اجرای برنامه، با دریافت درخت فراخوانی برنامه و پیگیری فراخوانیهای سیستمی حاصل از اجرای فعلی برنامه، روی درخت فراخوانی برنامه (مانند یک دیاگرام گذار حالت[1]) حرکت کرده و هر گونه انحراف از درخت فراخوانی برنامه را به عنوان انحراف از رفتار شناخته شده برنامه اعلام می‌کند. فقط در صورتی‌که انحراف برنامه به‌سبب اجرای حلقه‌ای به میزان تکرار بیشتر یا کمتر از حد مجاز باشد، به اعلام یک اخطار بسنده می‌شود و بررسی مابقی دنباله ادامه می‌یابد.

اگر انحراف در اجرای برنامه‌ای تشخیص داده شود، پیغام هشداری تهیه شده و به مسؤول برنامه داده می‌شود. مسؤول برنامه باید بررسی کند که آیا اجرای چنین مسیری از برنامه ناشی از سوء استفاده از برنامه بوده یا نقصی در شناخت کامل منطق اجرایی برنامه وجود داشته است. اگر اعلام هشدار به علت نقص منطق اجرایی برنامه باشد، بنابراین مسیرهای اجرایی برنامه به طور کامل شناسایی نشده‌اند. بنابراین با اضافه کردن شاخه‌ای به منطق اجرایی برنامه، شناخت منطق اجرایی برنامه کاملتر می‌شود. با توجه به ساختار کاملاً پویایی که برای درخت فراخوانی (منطق اجرایی) برنامه در نظر گرفته شده؛ به‌سادگی می‌توان شاخه‌ای جدید را به درخت اضافه کرد. بدین ترتیب مسیرهایی از برنامه که در طی دوران یادگیری در منطق اجرایی برنامه جای نگرفته‌اند، می‌توانند با هزینه‌ای بسیار پایین به منطق اجرایی برنامه اضافه شوند. بنابراین در یک فرایند زمانی و بر طبق پروسه‌ای تکمیل شونده، می‌توان منطق اجرایی برنامه را تکمیل کرد.

۷- ارزیابی روش پیشنهادی

به منظور ارزیابی رفتار برنامه‌ها در رویکرد مبتنی بر فراخوانیهای سیستمی ابتدا مطالعاتی در زمینهٔ مجموعه داده‌های مصنوعی (ساختگی) انجام شد. دنباله‌های مصنوعی با اجرای برنامه در محیط آزمایشگاهی و با گزینه‌های انتخابی و فقط به منظور آزمایش برنامه ایجاد می‌شوند و درحقیقت هیچ درخواست واقعی از طرف کاربر حقیقی صورت نمی‌گیرد. از آنجا که مجموعه داده‌های مصنوعی، براورد خوبی از نحوه عملکرد روشها ارائه نمی‌کنند [۱۰] بنابراین در این تحقیق از این نوع داده‌ها استفاده نشده است. با توجه به اینکه در روش پیشنهادی، شناخت مسیرهای اجرایی مختلف برنامه اهمیت بسیار زیادی دارد، در نتیجه از مجموعه وسیعتری از داده‌ها برای ارزیابی روش پیشنهادی استفاده شده است. این مجموعه داده شامل دنباله‌هایی از برنامه‌هایی است که توسط کاربران حقیقی در طول زمان و در طی استفاده عادی آنها از سیستم کامپیوتری ایجاد شده‌اند. برای دو برنامه lpr و stide، دانشگاه‌های MIT و UNM چنین مجموعه‌هایی را گردآوری کرده‌اند که ما نیز در ادامه با استفاده از این داده‌ها این دو برنامه را ارزیابی می‌کنیم.

1. State Transition Diagram

۷-۱- آزمایش برنامه lpr

برنامه lpr تحت سیستم‌عامل UNIX برای ارسال فایل به چاپگر استفاده می‌شود. دو مجموعه دادهٔ عادی برای این برنامه توسط دانشگاههای MIT و UNM [۱۶] تهیه شده است. مجموعه داده عادی MIT در طول دو هفته جمع‌آوری شده و شامل ۲۶۹۹ مرتبه اجرای عادی برنامه است؛ درحالی‌که مجموعه دادهٔ عادی UNM، از فعالیت ۱۵ ماهه شبکه جمع‌آوری شده و مشتمل بر ۴۲۹۸ دنباله عادی است. تعداد فراخوانیهای سیستمی مجموعه داده MIT برابر ۲،۹۲۶،۳۰۴ و برای مجموعه داده UNM این مقدار برابر ۲،۰۲۷،۴۶۸ است.

به منظور کشف منطق اجرایی برنامه lpr، ابتدا مجموعه داده MIT استفاده شد. در مرحله کشف حلقه، حجم دنباله‌ها پس از کشف حلقه و فشرده‌سازی به ۱۳/۷۲ درصد حجم اولیه رسید. سپس از میان دنباله‌های فشرده شده، دنباله‌های منحصر به فرد شناخته شد که جمعاً تعداد ۸۱ دنباله منحصر به فرد به‌دست آمد درحالی‌که حجم آنها ۲/۵۹ درصد حجم اولیه بود. با مقایسه دنباله‌های منحصر به فرد و تشخیص نقاط انشعاب برنامه، منطق اجرایی برنامه ایجاد و در فایلی ذخیره شد. نمایش درخت فراخوانی برنامه در حافظه فقط ۸ کیلوبایت از فضای حافظه را اشغال کرد.

در آزمایشی دیگر، مجموعه داده UNM بررسی شد. پس از مرحله کشف حلقه و فشرده‌سازی، حجم دنباله‌ها به ۲۷/۷۵ درصد حجم اولیه رسید. تعداد دنباله‌های منحصر به فرد نیز در حدود ۱۳۳ بود؛ درحالی‌که حجم دنباله‌های منحصر به فرد ۲/۰۱ درصد حجم اولیه بود. حافظه مورد نیاز برای نمایش درخت فراخوانی برنامه کمتر از ۹ کیلوبایت از فضای حافظه را اشغال کرد.

۷-۲- آزمایش برنامه stide

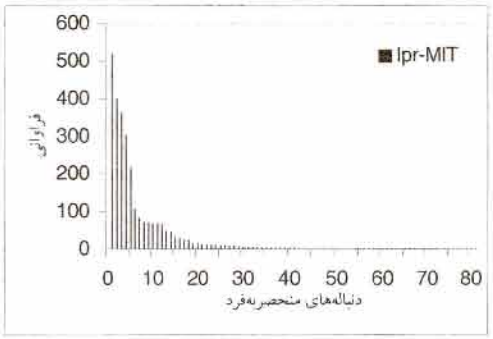
Stide یک برنامهٔ کاربردی است که توسط Forrest برای تشخیص نفوذ به برنامه به روش توالیهای مجاور نوشته شده است. مجموعه داده عادی برنامه stide به‌هنگام اجرای این برنامه برای تشخیص نفوذ به برنامه sendmail تهیه شده و شامل ۱۳،۷۲۶ دفعه از اجرای عادی و موفق برنامه است که جمعاً حاوی ۱۵،۶۱۸،۲۳۷ فراخوانی سیستمی می‌شود. پس از کشف حلقه، حجم دنباله‌ها به ۶/۰۴ درصد حجم اولیه رسید و فقط ۱۰۴ دنبالهٔ منحصر به فرد حاصل شد. حجم دنباله‌های منحصر به فرد ۰/۱۰ درصد حجم اولیه است. پس از ایجاد منطق اجرایی برنامه، بازسازی درخت فراخوانی در حافظه کمتر از ۶/۷ کیلوبایت از فضای حافظه را اشغال کرد. خلاصه‌ای از نتایج آزمایشهای انجام شده در جدول ۱ ارائه شده است.

جدول ۱ خلاصه‌ی نتایج آزمایشات بر روی برنامه‌های lpr و stide

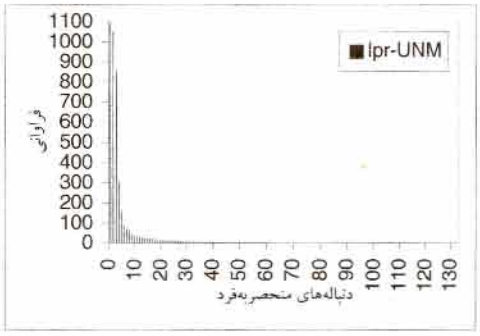
برنامه	تعداد دنباله‌های عادی	تعداد فراخوانیهای سیستمی در دنباله‌های عادی	حجم دنباله‌های عادی بعد از کشف حلقه نسبت به حجم اولیه	تعداد دنباله‌های منحصر به فرد	حجم دنباله‌های منحصر به فرد نسبت به حجم اولیه	حجم حافظه جهت بازسازی درخت منطق اجرای برنامه
Lpr (MIT)	۲۶۹۹	۲،۹۲۶،۳۰۴	۱۳/۷۲٪	۸۱	۲/۵۹٪	< 8 KB
Lpr (UNM)	۴۲۹۸	۲،۰۲۷،۴۶۸	۲۷/۷٪	۱۳۳	۲/۰۱٪	< 9 KB
stide	۱۳۷۲۶	۱۵،۶۱۸،۲۳۷	۶/۰۴٪	۱۰۴	۰/۱۰٪	< 6/6 KB

۷-۳- عمومیت‌پذیری داده‌ها

هر برنامه از آغاز اجرا تا پایان، با توجه به شرایط اجرا (مانند فایل یا پارامترهای ورودی، شرایط سیستم در لحظه اجرا و غیره) ممکن است مسیرهای اجرایی مختلفی را طی کند اما تعداد مسیرها محدود بوده و به کد اصلی برنامه وابسته است. در آزمایشهای ما، دنباله‌های منحصر به‌فرد به‌دست آمده، مسیرهای ممکن برای اجرای برنامه را نشان می‌دهند. برنامه در بیشتر موارد چند مسیر اصلی را طی می‌کند و دنباله‌هایی که نشان‌دهندهٔ اجرای چنین مسیرهایی باشند، دارای بیشترین میزان تکرار هستند. دنباله‌هایی که تکرار کمتری دارند، مربوط به حالات خاصی از برنامه هستند که در شرایط نادر اتفاق می‌افتند. در شکلهای ۱۱، ۱۲ و ۱۳ نرخ تکرار هر دنبالهٔ منحصر به فرد برای برنامه‌های lpr و stide نشان داده شده است.



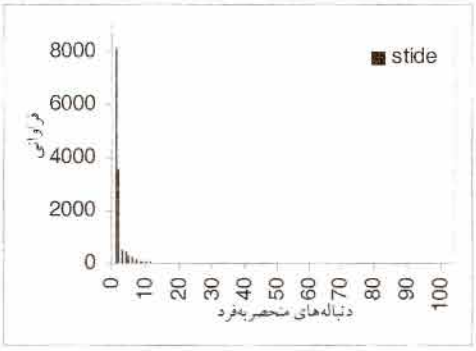
شکل ۱۱ فراوانی دنباله‌های منحصر به فرد برنامه lpr در MIT



شکل ۱۲ فراوانی دنباله‌های منحصر به فرد برنامه lpr در UNM

در این نمودارها دنباله‌های منحصر به فرد با توجه به میزان فراوانی‌هایشان به صورت نزولی مرتب شده‌اند تا تحلیل نمودارها ساده‌تر شود.

چند دنبالهٔ منحصر به فرد که بیشترین فراوانی را دارند، نشان‌دهندهٔ تنه اصلی برنامه هستند که بیشترین اجرا را داشته است. هر چه نمودار حاصل نزول سریعتری داشته باشد به همان میزان با تعداد دنباله‌های کمتری می‌توان مسیرهای اصلی برنامه را شناخت. درحقیقت میزان عمومیت‌پذیری جزئی از داده به کل داده‌ها را می‌توان توسط این عامل سنجید. نتایج حاصل حاکی از عمومیت‌پذیری بالای داده‌های برنامه stide نسبت به بقیه است.



شکل ۱۳ فراوانی دنباله‌های منحصر به فرد برنامه stide

۷-۴- ارزیابی کارایی

به منظور براورد کارایی روش پیشنهادی [۱۷]، میزان دو نوع خطای منفی کاذب[1] و مثبت کاذب[2] بررسی می‌شود. خطای منفی کاذب یعنی رفتار برنامه توسط واحد تشخیص انحراف، عادی گزارش می‌شود درحالی‌که فراخوانی سیستمی غیرعادی از سوی برنامه درخواست شده است. خطای مثبت کاذب یعنی رفتار برنامه غیرعادی گزارش می‌شود درحالی‌که برنامه رفتاری عادی داشته است.

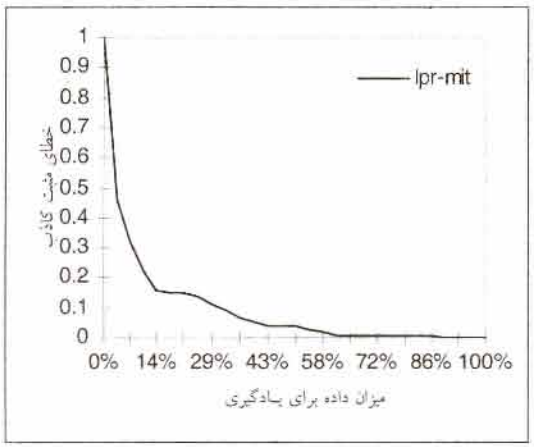
لازم است یادآوری شود که منطق اجرایی برنامه، فقط

1. False Negative
2. False Positive

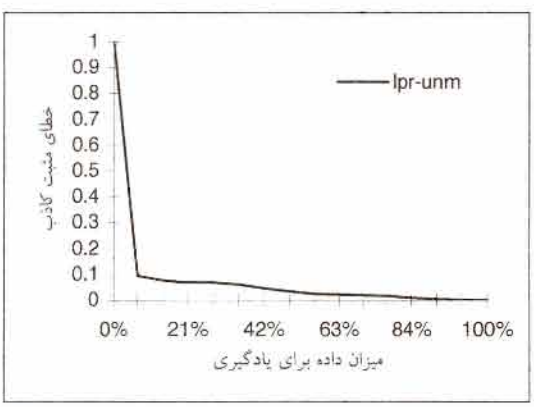
در سطحی از انتزاع یعنی در سطح فراخوانیهای سیستمی، توانایی توصیف رفتار برنامه را دارد. بنابراین اگر رفتاری از برنامه سر زند که تأثیری در فراخوانیهای سیستمی درخواستی نداشته باشد، این رفتار مشاهده نخواهد شد و فقط این حالت است که می‌تواند به بروز خطای منفی کاذب منجر شود. زیرا بر طبق سیاست بدبینانه‌ای که برای تشخیص انحراف در نظر گرفته شده، هر رفتاری در سطح فراخوانی سیستمی- که جزو رفتارهای مشاهده شدهٔ قبلی نباشد- رفتاری خارج از منطق اجرایی برنامه محسوب می‌شود. بنابراین فقط رفتاری که می‌تواند منجر به بروز خطای منفی کاذب شود، رفتاری است که آثار آن در سطح فراخوانی سیستمی مشاهده نشود.

میزان خطای مثبت کاذب، عامل به مراتب مهمتری برای سنجش کارایی روش پیشنهادی است. در روش پیشنهادی هرگونه نقصی در شناخت کامل منطق اجرایی برنامه، منجر به خطای مثبت کاذب خواهد شد. اگر مجموعه داده عادی برنامه، مشتمل بر تمام مسیرهای اجرایی ممکن برنامه نباشد، شناخت منطق اجرایی برنامه به طور کامل انجام نشده و به خطای مثبت کاذب منجر می‌شود. بنابراین خطای مثبت کاذب، رابطهٔ معکوس با میزان و تنوع داده‌های عادی برنامه دارد. با توجه به اینکه هیچ معیاری برای تعیین میزان داده عادی لازم برای شناخت کامل منطق اجرایی برنامه در دست نیست، با انجام آزمایشی تأثیر میزان داده عادی را بر میزان خطای مثبت کاذب، سنجش کردیم. در این آزمایش، در هر مرحله فقط کسری از دنباله‌های عادی را برای یادگیری منطق اجرایی برنامه به کار برده و مابقی را به منظور سنجش میزان خطای مثبت کاذب به‌کار بردیم. انتخاب دنباله‌های عادی برای یادگیری منطق اجرایی برنامه به صورت تصادفی است. نتایج این آزمایش بر روی برنامه‌های lpr (دو مجموعه داده MIT و UNM) و stide در شکلهای ۱۴، ۱۵ و ۱۶ نشان داده شده است.

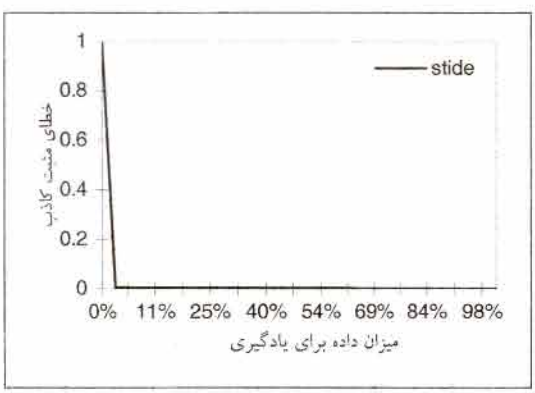
بر طبق این شکلها، همانطور که انتظار می‌رفت منحنی خطای مثبت کاذب با افزایش داده‌های یادگیری سقوط کرده و به صفر میل می‌کند. یعنی هر چه داده‌های یادگیری بیشتر و متنوعتری به کار رود؛ کشف منطق اجرایی برنامه کاملتر شده و از خطای مثبت کاذب کاسته می‌شود.



شکل ۱۴ میزان خطای مثبت کاذب برای برنامه lpr در MIT



شکل ۱۵ میزان خطای مثبت کاذب برای برنامه lpr در UNM



شکل ۱۶ میزان خطای مثبت کاذب برای برنامه stide

نتایج حاصل برای برنامه stide (شکل ۱۶) بهتر از بقیه است زیرا stide برنامهٔ ساده‌ای است و به شبکه نیز دسترسی ندارد. بنابراین منطق اجرایی برنامه با میزان داده کمتری کشف شده و خطای مثبت کاذب پایین‌تری دارد. از بین دو مجموعه داده عادی lpr، داده‌های MIT نتایج بهتری را نشان می‌دهد. با توجه به داده‌های عادی بیشتری که در مجموعه داده UNM در اختیار داشتیم و همچنین با توجه به شکل ۱۲ که نزول سریعتری دارد، انتظار داشتیم که نتایج حاصل از داده‌های UNM خطای مثبت کاذب کمتری را نشان دهد. شکل ۱۵ نیز نزول سریع منحنی را نشان می‌دهد که قابل پیش‌بینی بود اما در ادامه، تغییرات بسیار کمی را می‌بینیم که در مجموع خطای مثبت کاذب بیشتری را نسبت به داده‌های MIT نشان می‌دهد. در [۱۴] نیز نتایج مشابهی به‌دست آمده و علت آن چنین است که مجموعه داده UNM در مدت زمان بسیار طولانی (۱۵ ماه) جمع‌آوری شده است و بنابراین تغییرات پیکربندی شبکه در این داده‌ها تأثیر داشته است.

۷-۵- تحلیل

داده‌ای که برای یادگیری و درنتیجه کشف منطق اجرایی برنامه در اختیار است، فقط مجموعه‌ای از دنباله‌های حاصل از اجراهای موفق برنامه است. هر یک از این دنباله‌ها مسیری از برنامه را طی کرده‌اند. البته بیشتر مسیرهای اجرایی طی شده یکسان بوده و متعلق به زیرمجموعه نسبتاً کوچکی از مسیرهای اجرایی ممکن برنامه هستند. نتایج حاصل از تعداد فراوانی دنباله‌های منحصر به فرد نشان‌دهندهٔ این موضوع است. وقتی فقط بخشی از داده‌های موفق را برای یادگیری منطق اجرایی برنامه به‌کار می‌بریم، تعدادی از دنباله‌های متناظر با مسیرهایی از برنامه که به‌ندرت اجرا می‌شوند؛ در منطق اجرایی برنامه جای نمی‌گیرند و شناخت منطق اجرایی برنامه به طور کامل انجام نمی‌شود؛ این موجب بروز خطای مثبت کاذب می‌شود. از طرفی تعیین میزان داده‌ای که باید برای یادگیری استفاده شود تا خطای مثبت کاذب نداشته باشیم، تقریبا غیرممکن است؛ زیرا نگرش ما به برنامه‌ها به صورت جعبه‌سیاه است و بدون در اختیار داشتن کد اصلی برنامه نمی‌توان هیچ‌گونه براوردی را نسبت به کامل بودن منطق اجرایی برنامه به دست آورد. با این حال می‌توان با به‌کار بردن داده‌های یادگیری کافی، میزان خطای مثبت کاذب را تا حد زیادی کاهش داد. به‌عنوان مثال برای برنامه stide فقط با استفاده از ۶۱ درصد از داده‌های موجود، خطای مثبت کاذب محاسبه شده ۰/۰۰۱ است. برای برنامه lpr با مجموعه داده MIT، فقط به ۷۲ درصد داده‌ها نیاز است تا خطای مثبت کاذب به ۰/۰۰۳ کاهش یابد. در مجموعه داده UNM نیز به ۷۶ درصد از داده‌ها نیاز است تا این خطا به ۰/۰۱ برسد. بدیهی است با به‌کار بردن بخش بیشتری از داده‌ها برای یادگیری، خطا نیز کمتر خواهد شد. سیر نزولی منحنیها در شکلهای ۱۴، ۱۵ و ۱۶ مؤید این موضوع است.

همانطور که قبلاً نیز اشاره شد، یکی از خصوصیتهای مهم منطق اجرایی برنامه، امکان یادگیری و در نتیجه ساخت آن به صورت تکمیل‌شونده است. لذا پس از اعلام انحراف اجرای برنامه اگر مشخص شود که این رفتار صحیح بوده، آنگاه مسیر اجرایی جدید به منطق اجرایی برنامه اضافه می‌شود تا در دفعات بعدی چنین خطایی روی ندهد. با توجه به ساختار کاملاً پویای منطق اجرایی برنامه، این کار با اضافه کردن شاخه‌ای جدید در نقطه‌ای مناسب از درخت فراخوانی انجام می‌شود. در نتیجه با استفاده از این روش، منطق اجرایی برنامه می‌تواند به مرور زمان و با اضافه کردن شاخه‌های جدید کامل شود.

روش پیشنهادی در مرحله یادگیری به پردازش نسبتاً زیادی نیاز دارد زیرا کشف حلقه وقت‌گیر است و هر دنباله باید چندین بار پردازش شود. مقایسه کل دنباله‌های فشرده شده برای جداسازی دنباله‌های منحصر به فرد نیز به مدت زمانی متناسب با تعداد دنباله‌ها و طول هر دنباله نیاز دارد. ایجاد منطق اجرایی برنامه نیز به مدت زمانی

متناسب با تعداد دنباله‌های منحصر به فرد و طول هر یک نیاز دارد. روش پیشنهادی در مرحله تشخیص رفتار غیرعادی برنامه به پردازش کمی نیاز دارد؛ زیرا واحد تشخیص انحراف اجرای برنامه، صرفاً براساس فراخوانیهای سیستمی ایجاد شده توسط برنامه در حال اجرا، بر روی درخت فراخوانی برنامه (مانند یک دیاگرام گذار حالت) حرکت کرده و فقط بر اساس عدم تطابق یک فراخوانی سیستمی ایجاد شده با فراخوانی سیستمی مورد انتظار در درخت فراخوانی برنامه، انحراف از اجرای برنامه را تشخیص می‌دهد. این سازوکار ساده برای تشخیص انحراف باعث می‌شود که بررسی رفتار برنامه در حال اجرا، به صورت بی‌درنگ امکان‌پذیر شود. در مقایسه با رویکردهایی که در آنها از پنجره لغزان [۹،۱۰] برای ذخیره‌سازی الگوهای فراخوانیهای سیستمی استفاده می‌شود و حتی RIPPER [۱۲]، روش پیشنهادی در مرحلهٔ یادگیری به پردازش نسبتاً زیادی نیاز دارد (اما چون فرایند یادگیری به صورت دسته‌ای[1] انجام می‌شود اهمیت زیادی ندارد) اما در مرحله تشخیص انحراف اجرای برنامه، نسبت به سایر رویکردهای ارائه شده نیاز به پردازش اندکی دارد. یادآوری می‌شود که روش پنجرهٔ لغزان [۹،۱۰] برای تشخیص رفتار غیرعادی برنامه در ازای هر فراخوانی سیستمی ایجاد شده توسط برنامهٔ در حال اجرا، به مراجعات متعددی (وابسته به طول پنجره لغزان) به پایگاه الگوهای فراخوانیهای سیستمی برنامه نیاز دارد و روش RIPPER [۱۲] نیز به انجام تعدادی قدم استنتاجی نیاز دارد که بر حسب تعداد قوانین و طول زنجیره استنتاج متغیر است.

از بعد تکامل تدریجی شناخت رفتار عادی برنامه‌ها، هرگاه یک خطای مثبت کاذب تشخیص داده شود آن رفتار باید به مجموعه رفتار عادی برنامه اضافه شود. برای این کار در روش پیشنهادی به‌راحتی می‌توان به شاخه مربوط از درخت فراخوانی برنامه مراجعه و زیرشاخه جدید را در آن تعبیه کرد؛ بنابراین لازم نیست فرایند یادگیری منطق اجرایی برنامه تکرار شود (در نتیجه روش پیشنهادی یک روش تکمیل شونده است) اما در روش پنجره لغزان [۹،۱۰] لازم است فرایند ساخت پایگاه الگوهای فراخوانیهای سیستمی برنامه تجدید شود و در روش RIPPER [۱۲] نیز لازم است فرایند کلی‌سازی و تولید قوانین (معادل رفتار عادی) برنامه تکرار شود.

به طور خلاصه به خلاف برخی روشها، سازوکاری که برای کشف منطق اجرایی برنامه در روش پیشنهادی به کار رفته، برای تمامی‌برنامه‌ها یکسان است و نیازی به طراحی ساختاری مجزا برای هر برنامه ندارد. دانش فردی نیز دخالتی در این فرایند ندارد و تمامی پردازشها به‌صورت خودکار انجام می‌شود. به‌عنوان مثال در روشهای شبکه عصبی و گذار حالت، تعداد گره‌های حالت یا لایه برای هر برنامه متغیر است یا اینکه برای دستیابی به کارایی لازم باید وجود توالیهای خاصی مانند توالی Read-Write در lpr که تکرار بالایی دارد، مشخص شود [۱۳].

۸- کاربردهای روش پیشنهادی

در این تحقیق، از روش پیشنهادی برای حل مسائل در دو گروه متفاوت (۱) تشخیص نفوذ به برنامه و (۲) حفاظت از برنامه در برابر خطا استفاده شده که در ادامه معرفی می‌شود.

۸-۱- تشخیص نفوذ به برنامه

تشخیص نفوذ به برنامه بر اساس شناسایی رفتار غیرعادی برنامه، رویکرد مهمی است که در چند سال گذشته بر روی آن تأکید شده است. در این رویکرد سعی می‌شود نمای رفتار عادی برنامه تهیه شود و سپس با پایش رفتار برنامهٔ در حال اجرا، هرگونه انحراف از رفتار عادی برنامه به عنوان نفوذ به برنامه در نظر گرفته می‌شود. به‌طور کلی رفتار هر برنامه می‌تواند در سطح شبکه (یعنی وارسی

1. Batch / off-line

بسته‌های ارسالی و دریافتی برنامه) یا در سطح کاربران (یعنی وارسی نحوهٔ استفادهٔ کاربران از برنامه‌ها) و یا در سطح منطق اجرایی برنامه یادگیری شود. یادگیری رفتار برنامه در سطحی که پایداری بیشتری در مقابل تغییرات عادی در طول زمان داشته باشد و نیز تأثیر نفوذ در آن سطح متجلی شود، برای تهیه نمای رفتار عادی برنامه امری حیاتی است [۱۰]. با توجه به اینکه محققان دریافتند که نفوذیها به طور عمده از عیوب نرم‌افزاری[1] و یا طراحی برنامه‌ها سوء استفاده می‌کنند، بنابراین شناسایی رفتار برنامه‌ها در سطح منطق اجرایی آنها در مقایسه با محدوده رفتاری کاربران یا ترافیک شبکه، بسیار محدودتر و کارامدتر بوده و در طول زمان نیز نسبتاً ثابت است- به‌ویژه در مقایسه با رفتار کاربران. بنابراین سیستمهای تشخیص نفوذ، بررسی رفتار برنامه را در سطح منطق اجرایی آن، سطح مناسبی برای پایش رفتار برنامه یافتند [۶-۱۵].

یادآوری می‌شود که مهمترین داده برای پایش رفتار برنامه، فراخوانیهای سیستمی برنامه است. فراخوانی سیستمی ویژگی خاصی دارد که آن را انتخاب خوبی برای پایش رفتار برنامه- از نظر تخطی امنیتی- قرار داده است. بنابراین انحراف اجرای برنامه از منطق اجرایی آن می‌تواند نشانه‌ای از وقوع رفتاری غیرعادی یا نفوذ باشد، مانند حملاتی که از خطای سرریز بافر[2] استفاده می‌کنند. درحقیقت با استفاده از منطق اجرایی برنامه به‌عنوان نمای رفتار عادی برنامه در سیستم تشخیص نفوذ، می‌توان رفتارهای غیرعادی برنامه‌ها را تشخیص داد. در این تحقیق، بدین منظور از منطق اجرایی برنامه‌های lpr و stide (که در بخش ۷ مقاله تشریح شد) به‌عنوان نمای رفتار عادی این برنامه‌ها استفاده و سعی کردیم از آنها در تشخیص نفوذ به برنامه‌ها استفاده کنیم [۱۸]. ابتدا از منطق اجرایی برنامه lpr حاصل از داده‌های MIT به‌عنوان نمای رفتار عادی برنامه استفاده شد. سپس دنباله‌های مربوط به حمله lprcp پردازش شد. حمله lprcp از خطای سرریز بافر موجود در برنامه lpr بهره‌برداری می‌کند تا محتویات یک فایل اختیاری را با محتویات یک فایل دیگر جایگزین کند. با اجرای هر حمله lprcp، ۱،۰۰۱ دنباله ایجاد می‌شود. این دنباله‌ها توسط سیستم تشخیص نفوذ پیشنهادی پردازش شده و همگی غیرعادی تشخیص داده شدند. آزمایشهای انجام شده توسط دیگران نیز مؤید این نکته است که تمامی‌دنباله‌های مربوط به حمله lprcp غیرعادی است [۱۴]. آزمایش با استفاده از داده‌های UNM نیز نتیجه‌ای مشابه داشت. این بیانگر موفقیت سیستم تشخیص نفوذ در تشخیص حمله است.

دنباله‌های مربوط به حمله ازکاراندازی سرویس[3] نیز برای آزمایش برنامه stide به کار رفت. این نوع حمله بر روی هر برنامه‌ای که تقاضای حافظه می‌کند، تأثیر می‌گذارد [۱۴]. داده‌های حمله مشتمل بر ۱۰۵ دنباله است [۱۴] که سیستم تشخیص نفوذ پیشنهادی توانست در تمام ۱۰۵ مورد، حملهٔ انجام شده را با موفقیت تشخیص دهد.

با توجه به سازوکار تشخیص نفوذ، تمام نفوذهایی که از مسیری به‌جز مسیرهای شناخته‌شده و عادی برنامه عبور کنند، تشخیص داده خواهند شد. بنابراین سیستم تشخیص نفوذ پیشنهادی، نفوذهای جدید و ناشناخته را می‌تواند تشخیص دهد.

۸-۲- حفاظت از برنامه در برابر خطا

هدف این کاربرد، نظارت بر حسن اجرای برنامه است به طوری‌که اجرای برنامه از منطق تعریف شده قبلی منحرف نشده یا مسیرهای شناختهٔ شده قبلی که منجر به خطا می‌شوند، اجرا نشوند. یادآوری می‌شود که در فرایند ساخت و آزمون نرم‌افزارها، معمولاً حدود ۲٪ خطا در نرم‌افزارها باقی می‌ماند که ریشهٔ آنها به مراحل طراحی و کدنویسی برنامه‌ها باز می‌گردد. کاربران در دورهٔ

1. Bugs
2. Buffer Overflow
3. Denial of Service

بهره‌برداری از نرم‌افزار، معمولاً با خطاهایی روبرو می‌شوند که آنها را به سازندگان نرم‌افزار اعلام می‌کنند و پس از مدتی با دریافت نسخهٔ جدیدی از نرم‌افزار، خطاهای قبلی برطرف می‌شوند. اما نکته جالب توجه این است که در فاصله بین اعلام خطا و دریافت نسخهٔ جدید، کاربران نمی‌توانند کاری انجام دهند و فقط سعی می‌کنند به نحوی از نرم‌افزار بهره‌برداری کنند که خطاهای مشاهده شدهٔ قبلی روی ندهند. اما با استفاده از روشی که در بخشهای ۵ و ۶ این مقاله توضیح داده شد می‌توان دنبالهٔ فراخوانیهای سیستمی متناظر با آن اجرای برنامه را که منجر به خطا شده است به‌دست آورد و مسیر اجرایی آن دنباله را به شکل یک زیرشاخه جدید در منطق اجرایی (به‌بیان دیگر درخت فراخوانی) برنامه تعبیه کرد و یال اولین فراخوانی آن زیرشاخه را با "خطا" برچسب گذاری کرد. در این‌صورت در هنگام پایش رفتار برنامهٔ در حال اجرا، هرگاه اجرای برنامه به زیرشاخه‌ای با برچسب خطا برسد، اجرای برنامه را متوقف می‌کند تا بدین‌صورت از بروز مجدد خطاهای شناخته شدهٔ قبلی به صورت خودکار پیشگیری کند. خطا، به علت وجود عیوب برنامه‌نویسی یا اینکه کنترل اجرای برنامه به قطعه کدی غیرخودی انتقال یافته باشد، روی می‌دهد.

نظارت بر اجرای برنامه نیز در سطوح مختلف با کاراییهای متفاوت امکانپذیر است. به‌عنوان مثال Kiriansky و همکاران در [۱۹] با استفاده از سیستم بهینه‌سازی شفاف در زمان اجرا[1] [۲۰] توانستند با دیدگاهی امنیتی، مانع از انتقال اجرای برنامه به قطعه کدهای غیرخودی شوند. این کدها به صورت عمده با بهره‌گیری از خطای سرریز بافر به سیستم تزریق می‌شوند.

در این کاربرد، نظارت بر برنامه با استفاده از منطق اجرایی آن برنامه انجام شده است. در ساخت منطق اجرایی برنامه، فقط از داده‌هایی استفاده شده که حاصل اجرای صحیح و بدون خطای برنامه بوده است. بنابراین هر اجرایی از برنامه که با یکی از شاخه‌های درخت انطباق داشته باشد، مسیری موفق و بدون خطا را طی می‌کند. حال با تغییر اندکی در ساختار گره‌های درخت، مسیرهایی را نیز که منجر به خطا و شکست برنامه می‌شوند با مشخص کردن برچسب "خطا" و شاخصی که نشانگر نوع خطا و راهکار مقابله با آن باشد، به منطق اجرایی برنامه اضافه کردیم. در این حالت اگر مسیر اجرایی برنامه به شاخه‌ای که منجر به خطا خواهد شد انتقال یابد، قبل از وقوع خطا می‌توان تصمیم مقتضی را (مانند ایجاد وقفه در اجرای برنامه و اعلان هشدار و درخواست تأیید کاربر برای ادامه اجرای برنامه، توقف اجرای برنامه، ذخیره داده‌های حیاتی یا سایر موارد ممکن) در رابطه با نحوهٔ ادامه برنامه اتخاذ کرد.

به عنوان مثال به عملیاتی که بر روی برنامه lpr انجام شده است توجه کنید. ابتدا تمام یالهای منطق اجرایی برنامه lpr با برچسب "سالم" علامت‌گذاری شد. سپس دنباله‌های فراخوانیهای سیستمی حاصل از اجرای حملهٔ lprcp– که مشتمل بر ۱۰۰۱ دنباله باشد– به بخش کشف حلقه سپرده شد. پس از کشف حلقه و جداسازی دنباله‌های منحصر به فرد، فقط سه دنباله باقی ماند که این دنباله‌ها دقیقاً متناظر با سه مرحلهٔ حمله‌اند. در ادامه، این سه دنباله با برچسب "خطا" به درخت فراخوانی برنامه lpr اضافه شد. در اینجا راهکار خاصی برای مقابله با این خطا در صورت رویارویی در نظر گرفته نشده و فقط به اعلان هشدار و جلوگیری از ادامهٔ اجرای برنامه بسنده شده است. بنابراین هرگاه مسیری توسط برنامه lpr اجرا شود که قصد عبور از این شاخه‌ها را داشته باشد، در مراحل اولیه شناسایی شده و قبل از اینکه داده‌های دیگر را به خطر اندازد، اجرای برنامه متوقف می‌شود.

برای انواع خطاهای شناخته شده راهکارهای مختلفی برای مقابله وجود دارد و برای مواقعی که برنامه با خطای ناشناخته‌ای روبرو شود –یعنی برنامه قصد عبور از مسیری را داشته باشد که در درخت فراخوانی وجود

1. Transparent Runtime Optimization System

ندارد- می‌توان راهکار پیش‌فرضی مانند اعلان هشدار و درخواست تأیید کاربر برای ادامه برنامه در نظر گرفت. بدین ترتیب برنامه تحت نظارت و نیز حفاظت قرار می‌گیرد.

۹- نتیجه‌گیری

بررسی رفتار برنامه‌ها به دو روش جعبه‌سفید و جعبه‌سیاه انجام می‌شود. در نگرش جعبه‌سفید، با اطلاع از کد اصلی برنامه، منطق اجرایی برنامه با تحلیل کد اصلی به‌دست می‌آید ولی در نگرش جعبه‌سیاه، فقط وظیفه‌مندی برنامه مشخص است. در این مقاله با توجه به اینکه نگرش ما به برنامه‌ها به صورت جعبه‌سیاه است با این حال سعی کرده‌ایم با پردازش دنباله‌های فراخوانیهای سیستمی حاصل از اجراهای متعدد برنامه، مسیرهای اجرایی مختلف برنامه را یادگیری کرده و منطق اجرایی برنامه را در سطح فراخوانیهای سیستمی به‌دست آوریم. نتیجه حاصل به شکل درخت فراخوانی برنامه ساخته می‌شود. حال با توجه به شناختی که از منطق اجرایی برنامه به‌دست آمده، اجراهایی که از این منطق پیروی نکنند تشخیص داده می‌شوند. در حقیقت هدف از این کار اطمینان از صحت اجرای برنامه است. از این روش می‌توان به منظور ساخت سیستم تشخیص نفوذ به برنامه یا سیستم حفاظت از برنامه در برابر خطا استفاده کرد. نتایج آزمایشها بر روی دو برنامه lpr و stide نشان می‌دهد که اگر داده‌های موفق و عادی به میزان کافی در اختیار باشد این روش می‌تواند با خطای کم، انحراف اجرای برنامه را به صورت بی‌درنگ تشخیص دهد. شناخت منطق اجرایی برنامه به صورت تکمیل‌شونده، امکان پیاده‌سازی روش به صورت بی‌درنگ، و با هزینه پایین از مزایای روش پیشنهادی است.

۱۰- منابع

[1] I. I. Sommerville; "Software Engineering"; Addison-Wesley; 1996.

[2] B. Beizer; "Black-Box Testing: Techniques for Functional *Testing* of Software and Systems"; John Willey Press; 1996.

[3] A. Jones; Y. Lin; "Application Intrusion Detection Using Language Library Calls"; *Proceedings of the 17th Annual Computer Security Applications Conference*; New Orleans, Louisiana; 2001.

[4] M. Stillerman; C. Marceau; M. Stillman; "Intrusion Detection for Distributed Applications"; *Communications of the ACM* 42(7); July 1999; PP62-69.

[5] H. Inoue; S. Forrest; "Generic Application Intrusion Detection"; Technical Report TR-CS-2002-07, University of New Mexico; 2002.

[6] C. Ko; G. Fink; K. Levitt; "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring"; *Proceedings of the 10th Annual Computer Security Applications Conference*; Orlando; 1994; pp134-144.

[7] C. Ko; "Logic Induction of Valid Behavior Specifications for Intrusion Detection"; *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA; 2000; pp142-153

[8] D. Wagner; D. Dean; "Intrusion Detection via Static Analysis"; *Proceedings of IEEE Symposium on Security and Privacy*; 2001.

[9] S. Forrest; S.A. Hofmeyr; A.B. Somayaji; T.A. Longstaff; "A Sense of Self for UNIX Processes"; *Proceedings of IEEE Symposium on Security and Privacy*; 1996; pp120-128

[10] S.A. Hofmeyr; S. Forrest; A.B. Somayaji; "Intrusion Detection Using Sequences of System Calls"; *Journal of Computer Society* 6; 1998; pp151-180

[11] P. Helman; J. Bhangoo; "A Statistically-Based System for Prioritizing Information Exploration Under Uncertainty"; *IEEE Transactions on Systems*, Man and Cybernetics, Part A: Systems and Humans; July 1997; pp449-466

[12] W. Lee; S.J. Stolfo; "Data Mining Approaches for Intrusion Detection"; *Proceedings of the 7th USENIX Security Symposium*; 1998.

[13] A.K. Ghosh; A. Schwartzbard; M. Schatz; "Learning Program Behaviour Profiles for Intrusion Detection"; *Proceedings of the 1st Workshop on Intrusion Detection andNetwork Monitoring*; The USENIX Association, Santa Clara, CA; 1999.

[14] C. Warrender; S. Forrest; B. Pearlmutter; "Detecting Intrusion Using System Calls: Alternative Data Models"; *Proceedings of IEEE Symposium on Security and Privacy*; 1999; pp133-145

[15] E. Eskin; W. Lee; S.J. Stolfo; "Modeling System Calls for Intrusion Detection with Dynamic Window Sizes"; *Proceedings of DARPA Information Survivability Conference and Exposition II (DISCEX II)*, CA; 2001.

[16] Computer Immune systems - Data Sets and Software; http://www.cs.unm.edu/~immsec/data-sets.htm.

[۱۷] بلندقامت آذر، ح؛ "تشخیص نفوذ به برنامه با استفاده از فراخوانی سیستمی"، پایان نامه کارشناسی ارشد، گروه مهندسی کامپیوتر، دانشگاه تربیت مدرس، زمستان ۱۳۸۱

[۱۸] جلیلی، س؛ بلندقامت آذر، ح؛ "تشخیص نفوذ به برنامه از طریق ردگیری دنباله فراخوانیهای سیستمی"؛ هشتمین کنفرانس سالانه انجمن کامپیوتر ایران؛ مشهد؛ اسفند ۸۱.

[19] V. Kirinsky; D. Bruening; S. marasinghe; "Secure Execution via Program Shepherding"; Proceedings of the *11th USENIX Security Symposium*; 2002.

[20] V. Bala; E. Duesterwald; S. Banerjia; "Dynamo: A Transparent Runtime Optimization System"; *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*; 2000.