نشریه علمی «علوم و فناوری‌های پدافند نوین»
سال دهم، شماره ۳، پاییز ۱۳۹۸؛ ص ۳۲۲-۳۰۷

ارائه روشی هوشمند برای تولید داده آزمون به‌منظور کشف آسیب‌پذیری‌های نرم‌افزار

تقی تقوی[۱*]، مسعود باقری[۲]

۱- دانشجوی دکترای ۲- استادیار، دانشگاه جامع امام حسین(ع)

(دریافت: ۹۷/۴/۰۳، پذیرش: ۹۷/۱۱/۲۴)

چکیده

در این مقاله، یک فازر جعبه خاکستری، جهت کشف آسیب‌پذیری در کدهای باینری اجرایی ارائه شده است. بررسی‌ها نشان می‌دهد که آزمون فازینگ دارای سه مشکل اساسی است. در ابتدا، فضای ورودی فراهم‌شده جهت پوشش مسیرهای اجرایی برنامه، توسط فازرها می‌تواند بسیار بزرگ باشد. از سوی دیگر به‌واسطه بزرگی فضای ورودی، اغلب فازرها پوشش نامناسبی از مسیرهای اجرای کد را ارائه می‌دهند. درنهایت، به دلیل این پوشش نامناسب، تعداد زیادی از آسیب‌پذیری‌های ممکن در مسیرهای نادیده گرفته شده، آشکار نمی‌گردند. روش پیشنهادی این مقاله، مشکل بزرگی فضای ورودی را با محدود کردن تولید داده آزمون به پوشش مسیرهای مشکوکی که الگوهای آسیب پذیری در آن‌ها بیشتر مشاهده شده، در طی یک فرایند تکاملی، حل نموده است. در راهکار ارائه شده، زمانی که یک داده آزمون یک آسیب‌پذیری را در یک مسیر اجرایی آشکار می‌کند، داده آزمون بعدی به‌گونه‌ای تولید می‌شود که باعث رخداد آن آسیب‌پذیری شود. درنتیجه تعداد آسیب‌پذیری‌های شناسایی‌شده افزایش می‌یابد. ارزیابی‌های انجام‌شده، کارایی برتر روش پیشنهادی را در مقایسه با سایر روش‌های آزمون فازینگ نشان می‌دهند.

کلیدواژه‌ها: آزمون فازینگ، تولید داده آزمون، تحلیل رفتاری، پوشش مسیر

Presenting an Intelligence Test Data Generation Method to Discover Software Vulnerabilities

T. Taghavi[*], M. Bagheri
Imam Hossein Comprehensive University
(Received: 24/06/2018; Accepted: 13/02/2019)

Abstract

In this paper, a gray box fuzzer is presented to detect vulnerabilities in executable binary code. The literature surveys show that fuzz testing has three major problems. At first, the input space provided by the fuzzers to coverage execution paths in a binary program, can be very large. Secondly, most fuzzers can not support sufficient coverage of execution paths because of large input space. Finally, a large number of possible vulnerabilities can not reveal within an unseen execution path because of this unsufficient coverage. The proposed method, resolves the problem of large input space, in an evolutionary process, by conducting the test data generation towards suspicious paths in which one or more vulnerability patterns are observed. In the presented method, when a vulnerablility pattern is observed in an execution path by a test data, the next test data is generated revealing the vulnerablity. As a result, the number of detected vulnerabilities can increase. Our evaluations show better performance of presented method compared to other fuzz testing methods.

Keywords: Fuzz Testing, Test Data Generation, Dynamic Analysis, Path Coverage

*Corresponding Author E-mail: ttaghavi@ihu.ac.ir

۱. مقدمه

امروزه اغلب آسیب‌پذیری‌های روز صفر از طریق فازرها کشف شده و مورد استفاده قرار می‌گیرند، تا حدی که مشاهده می‌شود فازرها به‌عنوان سلاح‌های استراتژیک سایبری مطرح و خرید و فروش آن‌ها ممنوع شده است. مشکل عمده در فازرها فضای بزرگ جستجوی برنامه هدف برای یافتن آسیب‌پذیری و مشاهده نتایج حاصل از فازینگ بر آسیب‌پذیری‌ها است. این فضای بزرگ جستجو ناشی از مسیرهای اجرایی متعدد و بسیار زیاد ممکن در برنامه‌ها است [۱]. یکی از راهکارهای حل این مشکل استفاده از روش اجرای نمادین است[1]. روش اجرای نمادین مورد استفاده شرکت‌های بزرگی همانند مایکروسافت است [۲]. مشکل این نوع روش‌ها بزرگ شدن معادلاتی است که از جایگزینی متغیرها با توابع محاسبه کننده آن‌ها بر اساس ورودی‌های برنامه‌ها، به وجود می‌آید. برای حل این نوع توابع از حل‌کننده‌های محدودیت[2] استفاده می‌شود. این حل‌کننده‌ها در حالتی که توابع بزرگ و غیرخطی می‌شوند در عمل پاسخگو نیستند [۳].

روش‌های مبتنی بر جستجو سعی در یافتن مسیرهای جدید به‌صورت تصادفی دارند [۴]. مشکل روش‌های جستجوی کورکورانه عدم در نظر گرفتن مسیرهای پوشش داده شده توسط داده‌های قبلی است. به عبارت دیگر احتمال مؤثر بودن داده‌های ورودی جدید با قبلی‌ها در پوشش مسیرها یکسان است [۵]. برای رفع این مشکل روش‌های تطبیق‌پذیر مطرح شده‌اند. این روش‌ها داده‌های جدید را بر اساس میزان پراکندگی و فاصله آن‌ها با داده‌های قبلی به دست می‌آورند. در روش‌های مبتنی بر فرایند تکاملی هدف به دست آوردن مسیرها با حداکثر پراکندگی و فاصله از یکدیگر است [۶]. در این مقاله با توجه به اینکه ملاک اصلی برای تشخیص آسیب‌پذیری‌ها، وجود توابع آسیب‌پذیر است. در گام اول سعی در تولید داده‌های آزمونی است که بتوانند مسیرهایی را شناسایی کنند که دارای بیش‌ترین توابع متفاوت از لیست توابع موجود در مسیرهای قبلی باشند. بعد از شناسایی مسیرهای دارای توابع آسیب‌پذیری بیشینه، با انجام یک فرایند تحلیل آلودگی، داده‌های ورودی انتخاب می‌شوند که حداکثر حافظه مورد دسترسی توابع آسیب‌پذیر را پوشش داده و موجب آشکار شدن آسیب‌پذیری و خطای ناشی از آن گردند. ارزیابی‌های انجام‌شده با روش‌های موجود، برتری روش پیشنهادی را مشخص می‌کند.

در این مقاله، ابتدا روش‌های کشف آسیب‌پذیری نرم‌افزار بررسی می‌گردد. در ادامه روش‌های تولید خودکار داده آزمون به‌صورت خلاصه بررسی می‌شود. سپس روش پیشنهادی و الگوریتم‌های استفاده‌شده ارائه می‌گردد. همچنین مراحل انجام کار مورد بررسی قرار می‌گیرد. در انتهای مقاله ارزیابی و نتیجه‌گیری ارائه شده است.

۲. روش‌های کشف آسیب‌پذیری نرم‌افزار

روش‌های اصلی کشف آسیب‌پذیری نرم‌افزار شامل تحلیل ایستا[3]، تحلیل پویا[4]، آزمون فازینگ و آزمون نفوذ[5] است. از دیدگاه مفهومی روش‌های بیان‌شده، دارای همپوشانی هستند. برای مثال ماهیت آزمون فازینگ متعلق به تحلیل پویا است و در اینجا به‌عنوان نماینده تحلیل پویا در نظر گرفته می‌شود [۷].

۲-۱. تحلیل ایستا

تحلیل ایستا، فرایند ارزیابی یک سیستم یا مؤلفه بر اساس شکل، ساختار، محتوا یا مستندات آن است، که نیاز به اجرای برنامه ندارد. ایده‌ی اصلی تحلیل ایستا این است که متن برنامه به‌منظور شناسایی عیب یا ضعف در برنامه به‌صورت ایستا، کنترل می‌شود. بازرسی کردن کد به‌صورت دستی یک روش تحلیل ایستای قدیمی است، که باعث صرف زمان زیادی می‌شود. محققان از ابزارهای تحلیل ایستا، به‌منظور اجرای تحلیل ایستای کد با کمک کامپیوتر استفاده می‌کنند. اما ابزارهای تحلیل ایستا، فرایند تحلیل ایستا را به‌صورت کاملاً خودکار فراهم نمی‌کنند و خروجی‌ها هنوز به تائید یا تخمین انسان نیاز دارند [۸].

تحلیل ایستا یک مسئله تصمیم‌ناپذیر است. این موضوع نشان می‌دهد که برخی آسیب‌پذیری‌ها در برنامه توسط تحلیل ایستا شناسایی نمی‌شوند. به عبارت دیگر هنگامی‌که خروجی تحلیل ایستا، عاری از آسیب‌پذیری است نمی‌توان نتیجه گرفت که سیستم فاقد آسیب‌پذیری است بنابراین، خروجی تحلیل ایستا کامل نیست. علاوه بر این مشکل منفی کاذب[6] و مثبت کاذب[7] نیز در تحلیل ایستا وجود دارد. به علت اینکه تحلیل ایستا تصمیم ناپذیر است، مشکل منفی کاذب اکیداً غیرقابل‌اجتناب است. مثبت کاذب به تائید خروجی تحلیل ایستا توسط انسان نیاز دارد.

بر اساس نوع برنامه هدف، تحلیل ایستا می‌تواند در دو دسته طبقه‌بندی شود: تحلیل ایستا برای کد منبع[8] و تحلیل ایستا برای کد ماشین[9]. ماهیت و روش اجرایی هر دو تحلیل یکسان است، فقط تحلیل ایستای کد ماشین نیاز به رمزگشایی دستورات دارد،

[1] Concolic Execuion
[2] Constraint Solver

[3] Static Analysis
[4] Dynamic Analysis
[5] Penetration Test
[6] False Negative
[7] False Positive
[8] Source Code
[9] Machine Code

که باعث پیچیده‌تر شدن فرایند تحلیل می‌شود. بسیاری از روش‌های تحلیل ایستا از کد منبع استفاده می‌کنند [۸].

۲-۲. فازینگ

روش فازینگ در سال ۱۹۹۰ توسط میلر و همکارانش ارائه شد. فازینگ از تمایل برنامه‌های مودم به سمت خطا به علت ورودی تصادفی ناشی از اختلال در خطوط تلفن‌های فازی الهام گرفته شده است [۹]. امروزه، آزمون فازینگ اصلی‌ترین روش شناسایی آسیب‌پذیری در برنامه‌های بزرگ است. در واقع آزمون فازینگ ورودی‌های شبه معتبر[1] و تصادفی را به برنامه در حال اجرا تزریق می‌کند. در صورت شکست نرم‌افزار، یک آسیب‌پذیری پنهان شناسایی شده است. منظور از داده شبه‌معتبر، داده‌ای است که به اندازه کافی برای عبور از کنترل‌های ورودی برنامه معتبر است، اما می‌تواند در برنامه مشکلاتی ایجاد کند. هیچ روش دقیقی برای آزمون فازینگ وجود ندارد، این موضوع به برنامه هدف و قالب ورودی‌های آن بستگی دارد [۹].

تولید داده آزمون عنصر کلیدی آزمون فازینگ است. در اولین فازرها، با استفاده از یک روش کاملاً تصادفی داده آزمون تولید می‌شد، درنتیجه بیشتر داده‌های تولیدشده نامعتبر بودند. بعدها، محققان دو روش اصلی به‌منظور تولید داده آزمون پیشنهاد دادند. روش تولید داده[2] و روش جهش داده[3]. روش تولید داده معمولاً بر اساس ویژگی‌هایی مانند قالب فایل و پروتکل شبکه، است. این روش نیاز دارد که کاربران در مورد قالب فایل و پروتکل اطلاعات زیادی داشته باشند و احتیاج دارد که انسان بیشتر در فرایند تولید داده درگیر شود.

روش جهش داده، از طریق تغییر در برخی مقادیر ورودی معتبر، داده را تولید می‌کند. در این مورد فقط لازم است کاربران در مورد قالب فایل و پروتکل اطلاعات اندکی داشته باشند. زمانی که قالب داده‌های ورودی خیلی پیچیده باشند و داده‌های نمونه به سادگی جمع‌آوری شوند، جهش داده، مناسب‌تر از تولید داده است. اما روش جهش داده در موقعیت‌های بسیار حساس و به شدت وابسته به مقداردهی اولیه، به خوبی کار نمی‌کند. اتوداف[4] و اپیاک پراکسی[5] ابزارهایی هستند که با استفاده از جهش داده، فرایند تولید داده آزمون را انجام می‌دهند. پیچ[6] ابزاری است که هر دو روش را ترکیب می‌کند.

ابزاری که از آزمون فازینگ استفاده می‌کند، فازر نام دارد. هر فازر دارای سه مؤلفه اصلی است. تولیدکننده داده آزمون، مکانیسم تحویل و سیستم نظارت. هرکدام از این مؤلفه‌ها در زیر شرح داده شده‌اند [۹].

- تولیدکننده داده آزمون: این مؤلفه مسئول تولید داده‌هایی است که توسط سیستم تحت آزمون، مورد استفاده قرار می‌گیرد. خروجی تولیدکننده داده آزمون، داده‌هایی است که به‌صورت متوالی توسط مکانیسم تحویل به برنامه هدف ارسال می‌شوند.
- مکانیسم تحویل: این مؤلفه، ورودی‌ها را از تولیدکننده داده آزمون دریافت می‌کند و آن‌ها را به‌منظور استفاده در اختیار برنامه هدف قرار می‌دهد.
- سیستم نظارت: زمانی که برنامه هدف داده ورودی را مورد پردازش قرار می‌دهد، این مؤلفه برنامه را نظارت می‌کند و در صورت بروز خطا آن را گزارش می‌دهد.

آزمون فازینگ به سه دسته کلی تقسیم می‌شود.

- آزمون فازینگ جعبه سیاه[7]: در این نوع آزمون فازینگ هیچ اطلاعاتی در مورد برنامه هدف مورد نیاز نیست.
- آزمون فازینگ جعبه سفید[8]: در این نوع آزمون فازینگ کد منبع برنامه هدف در دسترس است و از آن استفاده می‌شود.
- فازینگ جعبه خاکستری[9]: از طریق فراهم کردن ورودی‌ها، مشاهده رفتار و تحلیل خروجی‌ها می‌توان دانش جزئی از برنامه هدف به دست آورد. این روش در نبود کد منبع، روش قابل قبولی است.

هنگام شناسایی آسیب‌پذیری با روش فازینگ، چالش‌های مختلفی مطرح می‌شود. یکی از مهم‌ترین چالش‌ها در آزمون فازینگ جعبه خاکستری، این است که کد باید در سطح دودویی تحلیل شود. به دلیل مفاهیم سطح پایین و تنوع و پیچیدگی دستورات زبان اسمبلی، این کار مشکل است. سایر چالش‌ها مانند شناسایی اجراهای خطرناک پنهان، شناسایی الگوهای آسیب‌پذیری، تولید ورودی‌های مرتبط با آسیب‌پذیری‌ها، محاسبه و بیشینه کردن معیار پوشش کد در فازرها نیز وجود دارد [۹].

۲-۳- آزمون نفوذ

آزمون نفوذ، از طریق شبیه‌سازی حمله توسط مهاجم و ارزیابی حمله‌های موفق، امنیت یک سیستم را مورد بررسی قرار می‌دهد. آزمون نفوذ توسط آزمونگرهای نفوذ انجام می‌شود. با توجه به

[1] Semi-valid
[2] Generation Based
[3] Mutation Based
[4] Autodafe
[5] APIKE Proxy
[6] Peach
[7] Black Box
[8] White Box
[9] Gray Box

اطلاعات در دسترس، این آزمون در سه دسته زیر طبقه‌بندی می‌شود [۱۰].

آزمون نفوذ جعبه سیاه: در این آزمون، هیچ اطلاعات قبلی در مورد ساختاری که آزمون می‌شود، در دسترس نیست. آزمون نفوذ جعبه سیاه، حمله شخصی که هیچ اطلاعاتی از سیستم ندارد را شبیه‌سازی می‌کند.

- آزمون نفوذ جعبه سفید: اطلاعات کاملی در مورد ساختاری که آزمون می‌شود، فراهم می‌کند. این اطلاعات اغلب شامل نمودارهای شبکه، کد منبع، اطلاعات آدرس‌دهی و غیره است.
- آزمون نفوذ جعبه خاکستری: در این نوع آزمون، دانش جزئی در مورد برنامه هدف در دسترس است.

به‌طورکلی آزمون نفوذ دارای دو مرحله است: تولیدکنندگان نرم‌افزار آزمون را طراحی می‌کنند و آزمونگرها آن را اجرا می‌کنند. آزمون نفوذ فقط یک ابزار برای حل مشکلات امنیتی است و اینکه مفید یا نامناسب باشد بستگی به اهداف تولیدکنندگان نرم‌افزار و آزمونگرها دارد [۱۰].

۳. تولید داده آزمون

در میان مراحل آزمون نرم‌افزار، تولید داده آزمون یک مرحله مهم و بحرانی است، زیرا تأثیر زیادی روی اثربخشی و کارایی فرایند آزمون دارد. در دهه‌های اخیر تعداد زیادی از محققان روی این موضوع پژوهش انجام داده‌اند. در نتیجه روش‌های زیادی به‌منظور تولید داده آزمون پیشنهاد شده است.

اگرچه روش‌های خودکار تولید داده آزمون، برای اعمال روی نرم‌افزار واقعی ایجاد شده‌اند اما هنوز فاصله زیادی بین سیستم‌های نرم‌افزاری واقعی و استفاده از روش پیشنهادی تولید داده آزمون وجود دارد. در این بخش کارهای انجام‌شده در زمینه تولید خودکار داده آزمون ارائه می‌گردد.

۳-۱. تولید خودکار داده آزمون با اجرای نمادین

اجرای نمادین، یک روش تحلیل برنامه است که به‌منظور تولید خودکار داده آزمون، کد برنامه را تحلیل می‌کند. مفید بودن اجرای نمادین در بسیاری از مسائل مهندسی نرم‌افزار مانند تولید داده آزمون، اثبات شده است. اما این روش حداقل سه مشکل کلی دارد که اثربخشی آن را روی نرم‌افزارهای واقعی محدود می‌کند. اجرای نمادین، مقادیر نمادین را به‌جای مقادیر واقعی به‌عنوان ورودی‌های برنامه استفاده می‌کند و مقادیر متغیرهای برنامه را به‌عنوان عبارت‌های نمادین این ورودی نمایش می‌دهد.

در هر نقطه از اجرای نمادین، حالت برنامه اجراشده به‌صورت نمادین شامل مقادیر نمادین متغیرهای برنامه در آن نقطه، محدودیت مسیر برای دست یافتن به آن نقطه و یک شمارنده برنامه است. محدودیت مسیر[1]، یک فرمول بولی[2] روی ورودی‌های نمادین است که شامل محدودیت‌هایی است که به‌منظور اجرای مسیر باید توسط ورودی‌ها محقق شود. در هر نقطه پرش در اجرای نمادین، محدودیت مسیر، توسط محدودیت‌های روی ورودی‌ها به‌روزرسانی می‌شود. اگر محدودیت مسیر محقق نشود، مسیر متناسب با آن اجرانشدنی است و اجرای نمادین در طی این مسیر ادامه پیدا نمی‌کند. در صورتی که محدودیت مسیر محقق شود، هر راه‌حل برای آن یک ورودی برنامه است که مسیر متناسب با آن را اجرا می‌کند. شمارنده برنامه جملات بعدی را نشان می‌دهد [۱۰].

برای درک موضوع به یک مثال توجه فرمایید. کد شکل (۱) هنگامی‌که مقداردهی اولیه x بزرگ‌تر از مقداردهی اولیه y باشد، دو مقدار متغیر x و y را باهم جابه‌جا می‌کند. شکل (۲) درخت اجرای نمادین متناسب با این کد را نشان می‌دهد. در درخت، گره‌ها حالت‌های برنامه را نشان می‌دهند و یال‌ها انتقال بین حالت‌های برنامه را نشان می‌دهند. اعداد موجود در گوشه سمت راست هر حالت، نمایانگر شمارنده برنامه در آن حالت است.

شکل (۳) محدودیت‌های برنامه و راه‌حل‌های آن‌ها (اگر موجود باشد) را برای سه مسیر برنامه نشان می‌دهد. به‌عنوان‌مثال محدودیت مسیر 1,2,3,4,5,8 برابر X>Y و Y-X<=0 است. بنابراین، یک ورودی از برنامه که این محدودیت را حل کند، مسیر متناسب را پیمایش می‌کند. ورودی برنامه مانند X=2 و Y=1 یک راه حل برای این محدودیت است.

با این‌که روش اجرای نمادین در اواسط دهه هفتاد میلادی پیشنهاد شد، این روش اخیراً به دو دلیل، بیشتر توجه محققان را جلب کرده است. اول اینکه، اجرای نمادین برنامه‌های بزرگ و واقعی نیاز به حل محدودیت‌های فراوان و پیچیده‌ای دارد. در طی دهه‌های اخیر، حل‌کننده‌های محدودیت قدرتمندی (مانند Z3، Yices و STP) توسعه یافته‌اند. استفاده از این حل‌کننده‌های محدودیت باعث اعمال راحت‌تر اجرای نمادین در برنامه‌های بزرگ شده است.

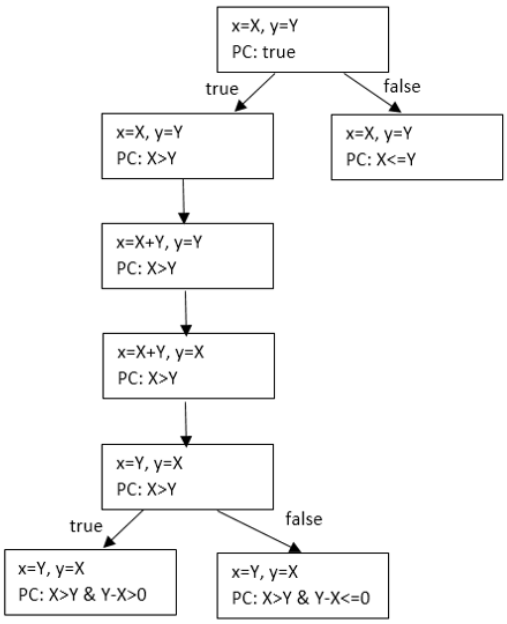
دوم اینکه، اجرای نمادین ازنظر محاسباتی از دیگر روش‌های تحلیل پرهزینه‌تر است. توانایی محاسباتی کامپیوترهای قدیمی محدود بود، بنابراین، اجرای نمادین برنامه‌های بزرگ غیرممکن بود. اما کامپیوترهای امروزی، بسیار قدرتمندتر هستند. بنابراین، امروزه موانع اجرای نمادین در برنامه‌های واقعی و بزرگ بسیار

[1] Path Constraint (PC)
[2] Bool Formula

کمتر از گذشته است. اما اثربخشی، اجرای نمادین روی برنامه‌های واقعی به دلیل وجود مشکلات اساسی در اجرای نمادین هنوز محدود است [۳].

```
int x,y;
if(x>y){
        x=x+y;
        y=x-y;
        x=x-y;
if(x-y>0)
        assert false
}
Print(x,y)
```

شکل ۱. کد جابجایی دو عدد



شکل ۲. درخت اجرای نمادین متناسب

Path	PC	Program Input
1, 8	X<=Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

شکل ۳. داده آزمون و محدودیت مسیر متناسب با مسیرهای مختلف برنامه

روش اجرای نمادین برای تولید داده آزمون با اهداف مختلفی استفاده شده است. اما شناخته‌شده‌ترین استفاده از این روش تولید داده آزمون برای پوشش کد و افشای خطاهای برنامه است. کاربردهای دیگر این روش شامل گزارش‌های خطای حفظ حریم خصوصی، تولید خودکار بهره‌برداری‌های امنیتی، آزمون بار، مکان‌یابی خطا، آزمون رگرسیون، داده ناشناخته برای آزمون برنامه‌های مبتنی بر پایگاه داده و آزمون طراح واسط کاربری است [۱۱].

اجرای نمادین به‌منظور تولید داده آزمون استفاده شده است [۱۲]. در این راهکار، به‌منظور انجام عملیات فازینگ، برنامه اجرا شده و برای مسیر پیمایش‌شده، محدودیت‌های مسیر و محدودیت‌های آسیب‌پذیری به‌صورت نمادین محاسبه می‌شوند. سپس با حل این محدودیت‌ها، آسیب‌پذیری‌های موجود در مسیر شناسایی می‌شوند. مزیت این راهکار این است که از ترکیب محدودیت‌های آسیب‌پذیری و محدودیت‌های مسیر استفاده شده است. بنابراین، داده آزمون برای شناسایی در یک مسیر اجرایی خاص مطابق با محدودیت‌های مسیر، تولید می‌شود. درنتیجه داده آزمون مسیر تعیین‌شده را طی می‌کند و باعث افشای آسیب‌پذیری می‌شود.

بر معیار پوشش کد در فرایند فازینگ تأکید شده است [۱۳]. بدین منظور معیار پوشش کد نسبت تعداد بلاک‌های اولیه در حالت پویا به تعداد بلاهای اولیه در حالت ایستا در نظر گرفته شده است. ابتدا آزمون فازینگ با ورودی‌های تصادفی شروع به کار می‌کند. زمانی که افزایش نرخ پوشش کد متوقف شد، اجرای نمادین شروع به کار می‌کند. پس از استخراج محدودیت‌های مسیر و حل آن‌ها، داده‌های آزمون تولید می‌شوند که مسیرهای مختلف و جدیدی را پوشش می‌دهند. سپس آزمون فازینگ با ورودی‌های جدید تکرار می‌شود. این فرایند تا به دست آوردن یک معیار پوشش ایده‌آل ادامه می‌یابد.

در یکی از روش‌های پیشنهادی [۱۴]، ابتدا با استفاده از تحلیل ایستا، گراف جریان کنترلی برنامه استخراج می‌شود. با کمک تحلیل آلودگی و اجرای نمادین، مسیرهایی از برنامه که از ورودی‌های برنامه استفاده می‌کنند، مشخص می‌شوند و گراف جریان کنترلی هرس می‌شود. درنهایت با حل محدودیت‌های تولیدشده توسط اجرای نمادین، داده آزمون‌ها تولید می‌شوند. مزیت این روش استفاده از تحلیل آلودگی است. زیرا این تحلیل باعث متمرکز شدن اجرای نمادین روی مسیرهایی است که از ورودی‌های برنامه استفاده می‌کنند. درنتیجه باعث کاهش تعداد داده آزمون‌های تولید شده می‌شوند.

۳-۲. تولید خودکار داده آزمون به‌صورت تصادفی

آزمون تصادفی، یک روش آزمون جعبه سیاه که به‌صورت گسترده در ابزارهای آزمونگر استفاده می‌شود. آزمون تصادفی، یک روش با مفهومی ساده است و به آسانی قابل پیاده‌سازی است [۱۵]. به‌طورکلی وجود آزمون کامل (آزمون یک برنامه با همه ورودی‌های ممکن) امکان‌پذیر نیست. با وجود اهمیت رویکرد

تولید داده آزمون به‌صورت تصادفی، این روش به‌صورت مجزا استفاده نمی‌شود، اما نقش حیاتی را در دیگر روش‌های آزمون ایفا می‌کند. آزمون تصادفی تا زمان برقراری یک شرط توقف مانند شناسایی یک آسیب‌پذیری، اجرای تعداد معینی داده آزمون و یا اتمام زمان معین، داده‌های آزمون را به‌صورت تصادفی تولید می‌کند. با وجود اینکه آزمون تصادفی دارای مزیت‌های ویژه‌ای است اما از اطلاعات موجود در برنامه تحت آزمون استفاده نمی‌کند [۱۶]. این موضوع واضح است که احتمال آشکار شدن آسیب‌پذیری با توزیع داده آزمون‌ها به‌طور یکسان افزایش می‌یابد. بر اساس این ایده ساده، تعداد زیادی روش آزمون تصادفی تطبیقی[1] پیشنهاد شد. در این روش‌ها، انتخاب داده آزمون‌ها به صورتی انجام می‌گیرد که به‌طور یکسانی توزیع شده باشند [۱۶].

بعضی از روش‌های آزمون تصادفی تطبیقی، مانند آزمون تصادفی تطبیقی مبتنی بر فاصله[2] یا آزمون تصادفی محدود[3]، سربار محاسباتی بالایی دارند. به‌منظور کاهش این سربار، روش‌های آزمون تصادفی تطبیقی متعددی مانند آزمون تصادفی تطبیقی دوبخشی یا آزمون تصادفی تطبیقی جزءبندی[4] ارائه شده است [۴]. در صورتی که خروجی متناظر عناصری از دامنه ورودی نادرست باشد، این عناصر به‌عنوان ورودی‌های مسبب شکست در نظر گرفته شده‌اند [۶]. نرخ شکست با تقسیم تعداد ورودی‌های مسبب شکست بر کل ورودی‌ها به دست می‌آید. F_measure به‌عنوان یک معیار مؤثر در این روش استفاده شده است. این معیار بیانگر تعداد داده آزمون‌های مورد نیاز برای شناسایی اولین شکست است. در بسیاری از مواقع زمانی که یک شکست شناسایی می‌شود، آزمون متوقف می‌شود و اشکال‌زدایی دوباره آغاز می‌شود.

آزمون تصادفی تطبیقی مبتنی بر فاصله، به‌صورت تصادفی یک مجموعه کاندید از داده آزمون‌ها را از دامنه ورودی انتخاب می‌کند. بر اساس معیار کمترین و یا بیش‌ترین فاصله بین داده آزمون‌های کاندید و داده آزمون‌هایی که قبلاً اجرا شده‌اند، فقط یکی از این داده آزمون‌ها انتخاب می‌شود. در آزمون تصادفی محدود یک ناحیه محدود سرتاسر هر داده آزمون اجرا شده، ایجاد می‌شود. داده آزمون از میان دامنه ورودی انتخاب می‌شود. اما درصورتی‌که داده آزمون‌ها در ناحیه محدود قرار داشته باشند، رد می‌شوند و سپس داده آزمون بعدی انتخاب می‌شود [۵].

آزمون تصادفی تطبیقی آینه‌ای[5]، به کاهش محاسبات سربار فاصله کمک می‌کند [۱۷]. در این آزمون، ابتدا همه دامنه ورودی در زیر دامنه‌های مجزایی جزءبندی می‌شود، سپس آزمون تصادفی تطبیقی فقط روی زیر دامنه‌ها اجرا می‌شود و توابع آینه‌ای، باید برای نگاشت داده آزمون به دیگر زیر دامنه‌ها استفاده شود. بنابراین، اگر دامنه ورودی را به m زیر دامنه مجزا تقسیم شود، سربار محاسبه فاصله کاهش می‌یابد. به‌منظور کاهش سربار محاسبات، دو روش آزمون تصادفی تطبیقی توسط تقسیم‌بندی تصادفی و روش آزمون تصادفی تطبیقی دوبخشی پیشنهاد شده است [۱۸]. به‌منظور حذف سربار محاسبات این روش از تقسیم‌بندی‌های مختلفی استفاده می‌کنند. در این روش‌ها، به‌جای انتخاب یک داده آزمون از مجموعه کاندید، یک ناحیه برای داده آزمون بعدی انتخاب می‌شود. در مرحله بعد داده آزمونی که از این ناحیه انتخاب می‌شود، اجرا می‌شود.

تولید تصادفی موارد آزمون مزایا و معایبی دارد. مزایای آن سادگی، کم‌هزینه، سرعت اجرا، عدم نیاز به نرم‌افزارهای جانبی فراوان و همچنین توزیع عادلانه موارد آزمون در بین مقادیر داده ممکن است. معایب این روش نیز شامل کورکورانه بودن، مقدار منفی کاذب بالا، تولید حجم بالای موارد آزمون تکراری یا بلااستفاده و عدم پوشش مناسب خطاهای برنامه به‌طور معمول است.

۳-۳. تولید خودکار داده آزمون مبتنی بر جستجو

آزمون نرم‌افزاری مبتنی بر جستجو، شاخه‌ای از مهندسی نرم‌افزار مبتنی بر جستجو است که از الگوریتم‌های بهینه‌سازی به‌منظور خودکارسازی تولید داده آزمون و بیشینه کردن معیارهای آزمون نرم‌افزار و کاهش هزینه‌های آزمون، استفاده می‌کند.

در آزمون نرم‌افزاری مبتنی بر جستجو، یک الگوریتم بهینه‌سازی، فضای ورودی‌های برنامه را برای دست یافتن به اهداف آزمون مورد جستجو قرار می‌دهد. مسئله اصلی تعریف یک تابع هدف (یا چندین تابع هدف) است که منجر به برآوردن اهداف آزمون می‌شود. تابع هدف برای هدایت الگوریتم بهینه‌سازی مورد استفاده قرار می‌گیرد. الگوریتم‌های بهینه‌سازی بسیاری وجود دارد که ازجمله می‌توان به الگوریتم ژنتیک، الگوریتم تپه‌نوردی، الگوریتم تبرید شبیه‌سازی‌شده و غیره اشاره کرد.

به دلیل کارایی الگوریتم ژنتیک، بسیاری از کارهای انجام‌شده در زمینه تولید داده آزمون مبتنی بر جستجو از این الگوریتم استفاده می‌کنند. در یکی از راه‌کارها [۱۹]، ابتدا فایل اجرایی نرم‌افزار مورد تحلیل ایستا قرار می‌گیرد. سپس بر طبق نتایج تحلیل ایستا اطلاعات خاصی از پوشش مسیر محاسبه می‌شود. این اطلاعات شامل تعداد مسیرهای برنامه، تعداد حلقه‌های موجود و زمان اجرا است. درنهایت تابع هدف الگوریتم ژنتیک بر

[1] Adaptive Random Testing (ART)
[2] Distance-Based Adaptive Random Test (D-ART)
[3] Restricted Random Testing (RRT)
[4] Partitioning Adaptive Random Testing
[5] Mirror Adaptive Random Testing

اساس این اطلاعات به‌منظور هدایت فرایند تولید داده آزمون ساخته می‌شود. الگوریتم ژنتیک تا زمان یافتن آسیب‌پذیری‌های پنهان در فایل اجرایی ادامه می‌یابد. راهکار دیگری [۲۰]، نیز از این روش استفاده می‌کند، با این تفاوت که با استفاده از تحلیل آلودگی فضای جستجوی الگوریتم ژنتیک را کاهش می‌دهد. بنابراین، کارایی عملیات تولید داده آزمون را افزایش می‌دهد [۲۱]. نیز با محاسبه معیار پوشش بلاک‌های اولیه به‌عنوان معیار پوشش روش مشابهی را ارائه داده است. بر اساس یکی از روش‌ها [۲۲]، به‌منظور جلوگیری از همگرایی قبل از موقع الگوریتم ژنتیک، ترکیب الگوریتم ژنتیک را با الگوریتم کلونی مورچه‌ها پیشنهاد می‌کند.

فیلتر کالمن[1] ازجمله فیلترهای کارآمد جهت تخمین حالت‌های یک سیستم با کم‌ترین خطای ممکن است. این فیلتر با الگوریتمی بازگشتی و داشتن توزیع اختلال‌ها و به‌کارگیری عملگرهای ماتریسی حالت‌های جدید سیستم را در فضای گسسته پیش‌بینی می‌کند. در یکی از رویکردها [۲۳]، با استفاده از الگوریتم ژنتیک مبتنی بر فیلتر کالمن فرایند تولید داده آزمون را انجام می‌دهد.

الگوریتم تبرید شبیه‌سازی‌شده، ابتدا از یک جواب اولیه شروع می‌کند و سپس در یک حلقه تکرار به جواب‌های همسایه حرکت می‌کند. اگر جواب همسایه بهتر از جواب فعلی باشد، الگوریتم آن را به‌عنوان جواب فعلی قرار می‌دهد. در غیر این صورت، الگوریتم آن جواب را با احتمال $\exp^{-\Delta E/T}$ به‌عنوان جواب فعلی می‌پذیرد. در این رابطه ΔE تفاوت بین تابع هدف جواب فعلی و جواب همسایه است و T یک پارامتر به نام دما است. در هر دما، چندین تکرار اجرا می‌شود و سپس دما به آرامی کاهش داده می‌شود. در گام‌های اولیه، دما خیلی بالا قرار داده می‌شود تا احتمال بیشتری برای پذیرش جواب‌های بدتر وجود داشته باشد. با کاهش تدریجی دما، در گام‌های پایانی احتمال کمتری برای پذیرش جواب‌های بدتر وجود خواهد داشت و بنابراین، الگوریتم به سمت یک جواب خوب همگرا می‌شود. از ترکیب الگوریتم ژنتیک و الگوریتم تبرید شبیه‌سازی‌شده به‌منظور کارایی بهتر تولیدکننده داده آزمون استفاده شده است [۲۴]. طی روش دیگری [۲۵]، تنها با استفاده از الگوریتم تبرید شبیه‌سازی‌شده، فرایند تولید داده آزمون انجام شده است.

۴. روش تحقیق

روش تحقیق پیشنهادی، به‌صورت کلی از دو بخش تولید داده آزمون و تحلیل رفتاری برنامه تشکیل شده است. در بخش اول، داده آزمون تولید می‌شود و به‌منظور اجرا توسط برنامه و عملیات تحلیل رفتاری به بخش دوم ارسال می‌شود.

یکی از مشکلات در پوشش مسیرهای اجرایی، مسئله شناخته‌شده انفجار مسیر است که برای رفع آن پیشنهاد شده است که مسیرها با احتمال اجرای بیشتر یا به‌عبارت دیگر مسیرهای اصلی انتخاب شوند. یک برنامه بزرگ ممکن است دارای میلیون‌ها مسیر اجرایی در داخل خود باشد. اغلب، بسیاری از آن‌ها می‌توانند مسیرهای فرعی بوده و باید طی فرایندی از لیست مسیرهای اصلی برنامه تفکیک گردند. به این فرایند هرس کردن می‌گویند. بسیاری از روش‌های تولید داده آزمون از روش‌های مختلف به هرس کردن گراف اجرایی می‌پردازند. مشکل یافتن مسیرهای اصلی و تفکیک آن‌ها از سایر مسیرها است. یکی از روش‌های ابداعی این مقاله در مورد هرس کردن، هدایت کردن فرایند تولید داده آزمون به سمت مسیرهایی است که توابع آسیب‌پذیر بیشتری در آن‌ها مشاهده شده است تا بدین ترتیب احتمال بروز خطا در هنگام اجرای داده آزمون در آن مسیر بیشتر گردد و از سایر مسیرها صرف‌نظر شده است.

تولید داده آزمون به دو صورت انجام می‌شود: تولید داده آزمون با استفاده از الگوریتم شکست و تولید داده آزمون با استفاده از الگوریتم ژنتیک. متناسب با نتایج تحلیل رفتاری یکی از این دو الگوریتم مورد استفاده قرار می‌گیرند. درصورتی‌که در مسیر اجرایی برنامه، آسیب‌پذیری وجود داشته باشد، از الگوریتم شکست به‌منظور تولید داده آزمون بعدی استفاده می‌شود. داده آزمون تولیدشده توسط الگوریتم شکست باعث، رخ دادن آسیب‌پذیری موجود و درنتیجه شکست برنامه می‌شود. اگر در مسیر اجرایی برنامه، آسیب‌پذیری وجود نداشته باشد، از الگوریتم ژنتیک، به‌منظور تولید داده آزمون بعدی استفاده می‌شود.

در بخش دوم، ابتدا داده آزمون تولیدشده در بخش اول دریافت و به‌عنوان ورودی به برنامه داده می‌شود. با اجرای برنامه، عملیات تحلیل رفتاری روی آن آغاز می‌شود. به‌منظور تحلیل رفتاری، ابتدا کد ماشین برنامه، به کد اسمبلی تبدیل می‌شود. سپس برنامه در سطح دستورالعمل‌ها مستندگذاری می‌شود. آدرس‌های آلوده حافظه شناسایی و برچسب‌گذاری می‌شوند. از طرف دیگر، برنامه در سطح توابع API، مستندگذاری می‌شود. توابع API فراخوانی شده در هر مسیر اجرایی استخراج می‌شوند و با توابع ناامن مقایسه صورت می‌گیرد. در صورت وجود توابع ناامن، آدرس‌های پارامترهای مورد استفاده در این توابع استخراج می‌شود. درنهایت، آدرس‌های استخراج‌شده، با آدرس‌های آلوده حافظه مقایسه می‌شود. در صورت وجود تطابق، الگوریتم شکست فراخوانی می‌شود، در غیر این صورت الگوریتم ژنتیک اجرا خواهد

[1] Kalman Filter

شد. در صورت رخ دادن آسیب‌پذیری در کد، یک اشکال‌زدای تعبیه‌شده، آسیب‌پذیری و علت و مکان آن را نشان می‌دهد. شرط پایان الگوریتم، پوشش اکثریت مسیرهای برنامه است، که این شرط توسط الگوریتم ژنتیک مورد بررسی قرار خواهد گرفت. در شکل (۴) فلوچارت روش پیشنهادی نشان داده شده است.

شکل ۴. فلوچارت روش پیشنهادی

روش پیشنهادی را می‌توان روی انواع برنامه‌ها مانند برنامه‌های پردازشگر فایل (ویرایشگرهای متنی، پخش‌کننده فیلم و موسیقی، ابزارهای نمایش فایل‌های PDF و ...) برنامه‌های خط فرمان و غیره اعمال کرد. به‌منظور پیاده‌سازی روش، از برنامه‌های ویرایشگر متنی به‌عنوان مورد مطالعاتی استفاده شده است؛ بنابراین، داده آزمون‌های ورودی فایل‌های متنی در نظر گرفته شده است. در ادامه نحوه اجرای هر یک از مراحل، داده‌های خروجی تولید شده در آن مرحله و دلیل استفاده از روش‌های به کار گرفته شده توضیح داده می‌شود.

۴-۱. تولید داده آزمون با استفاده از الگوریتم ژنتیک

به‌منظور تولید داده آزمون، از دو الگوریتم ژنتیک و الگوریتم شکست استفاده شده است. متناسب با نتایج تحلیل رفتاری یکی از این دو الگوریتم داده آزمون بعدی را تولید می‌کنند. در ابتدا، عملیات تولید داده آزمون با الگوریتم ژنتیک آغاز می‌شود. پس از تحلیل رفتاری، در صورت وجود آسیب‌پذیری در مسیر اجرایی، داده آزمون بعدی توسط الگوریتم شکست تولید می‌شود. در غیر این صورت همچنان، الگوریتم ژنتیک عملیات تولید داده آزمون را بر عهده دارد. در ادامه به معرفی الگوریتم ژنتیک مورد استفاده، پرداخته شده است.

همان‌طور که اشاره شد یکی از ورودی‌های ابزار فازینگ ارائه شده، پوشه‌ای شامل فایل‌های متنی است. فایل‌های این پوشه به‌عنوان مجموعه داده آزمون‌های اولیه الگوریتم در نظر گرفته شده است. ابتدا یکی از اعضای مجموعه داده آزمون اولیه، به‌عنوان ورودی به برنامه مورد آزمون ارسال می‌شود. برنامه اجرا شده و مورد تحلیل رفتاری قرار می‌گیرد. اجرای برنامه منجر به پیمایش یک مسیر اجرایی می‌شود. در هر مسیر اجرایی مجموعه‌ای از توابع API فراخوانی می‌شوند. درنهایت لیست این توابع API از مسیر اجرایی، استخراج می‌شوند. این کار با مستندگذاری کد در سطح توابع API فراخوانی شده انجام می‌شود. این روند تا اتمام تمام اعضای مجموعه داده آزمون اولیه ادامه می‌یابد. به‌عنوان‌مثال اگر مجموعه داده آزمون اولیه شامل سه فایل با نام‌های 1.txt و 2.txt و 3.txt باشد، جدول (۱) اطلاعات مربوط به این مجموعه را نشان می‌دهد. پس از اجرای هر داده آزمون با برنامه مورد آزمون، در مسیر اجرایی پیمایش شده تعدادی توابع API فراخوانی می‌شوند. لیست این توابع API در ستون آخر جدول آمده است.

جدول ۱. مجموعه داده آزمون اولیه

شماره داده آزمون	نام فایل داده آزمون	توابع API فراخوانی شده
۱	1.txt	CreateFile, ReadFile
۲	2.txt	ReadFile
۳	3.txt	StrNCat

پس از اتمام مجموعه داده آزمون اولیه، ماتریسی مربعی، در ابعاد اندازه مجموعه اولیه (در این مثال ۳×۳) در نظر گرفته می‌شود. سطرها و ستون‌های این ماتریس، اعضای مجموعه هستند و درایه‌های این ماتریس از رابطه زیر به دست می‌آید:

$$D_{ij} = |(\text{توابع API فراخوانی شده در مسیر اجرایی متناظر با داده آزمون i}) \Delta (\text{توابع API فراخوانی شده در مسیر اجرایی متناظر با داده آزمون j})|$$

درواقع D_{ij}، اندازه مجموعه حاصل از تفاضل متقارن دو مجموعه توابع API فراخوانی شده در مسیر اجرایی متناظر با داده آزمون i و داده آزمون j است.

به‌عنوان‌مثال اندازه D_{12} مربوط به دو داده آزمون ۱ و ۲ به‌صورت زیر محاسبه می‌شود:

$$D_{۱۲} = D_{۲۱} = |\{\text{CreateFile, ReadFile}\} \Delta \{\text{ReadFile}\}| = |\{\text{CreateFile}\}| = ۱$$

اندازه D_{13} مربوط به دو داده آزمون ۱ و ۳ به‌صورت زیر محاسبه می‌شود:

$$D_{۱۳} = D_{۳۱} = |\{\text{CreateFile, ReadFile}\} \Delta \{\text{StrNCat}\}|$$
$$= |\{\text{CreateFile, ReadFile, StrNCat}\}| = ۳$$

اندازه D_{23} مربوط به دو داده آزمون ۲ و ۳ به‌صورت زیر محاسبه می‌شود:

$$D_{۲۳} = D_{۳۲} = |\{\text{ReadFile}\} \Delta \{\text{StrNCat}\}|$$
$$= |\{\text{ReadFile, StrNCat}\}| = ۲$$

درنتیجه ماتریس نهایی که ماتریس A نام دارد، به‌صورت زیر است:

	1.txt	2.txt	3.txt
1.txt	۰	۱	۳
2.txt	۱	۰	۲
3.txt	۳	۲	۰

به دلیل اینکه مسیر اجرایی دو داده آزمون مشابه، دنباله توابع فراخوانی API یکسانی را پیمایش می‌کنند، بنابراین، اندازه مجموعه حاصل از تفاضل متقارن آن‌ها صفر است و درنتیجه مقدار درایه‌های روی قطر اصلی ماتریس A صفر است.

$$D_{11} = D_{22} = D_{33} = 0$$

دو داده آزمون که مسیرهای اجرایی متناظر آن‌ها تفاوت بیشتری با هم داشته باشند، به این معنی است که توابع API مختلفی در مسیرهای اجرایی متناظر آن‌ها فراخوانی شده است. بنابراین، درصورتی‌که توسط الگوریتم ژنتیک به‌عنوان والد انتخاب شوند، احتمال اینکه داده آزمون تولید شده توسط آن‌ها منجر به پیمایش مسیر اجرایی متفاوتی نسبت به مسیرهای اجرایی قبلی شود، افزایش می‌یابد. از طرف دیگر داده آزمون که مسیرهای اجرایی متناظر آن‌ها مشابه با هم داشته باشند، به این معنی است که توابع API مشابهی در مسیرهای اجرایی متناظر آن‌ها فراخوانی شده است. بنابراین، درصورتی‌که توسط الگوریتم ژنتیک به‌عنوان والد انتخاب شوند، احتمال اینکه داده آزمون تولید شده توسط آن‌ها منجر به پیمایش مسیر اجرایی مشابه مسیرهای اجرایی قبلی شود، افزایش می‌یابد.

به‌منظور یافتن داده آزمون‌هایی که مسیرهای اجرایی متناظر آن‌ها تفاوت بیشتری با هم داشته باشند، یک سطر به ماتریس A اضافه می‌شود. درایه‌های این سطر شامل بیشینه مقدار درایه‌های هر ستون از ماتریس است:

	1.txt	2.txt	3.txt
1.txt	۰	۱	(۳)
2.txt	۱	۰	۲
3.txt	(۳)	۲	۰
Max	(۳)	۲	(۳)

درنهایت بیشینه سطر Max محاسبه می‌شود. دو داده آزمون مرتبط با مقدار بیشینه نهایی، دارای بیش‌ترین تفاوت در توابع API فراخوانی شده در مسیر اجرایی متناظرشان هستند. این دو داده آزمون توسط الگوریتم ژنتیک به‌عنوان والد داده آزمون بعدی انتخاب می‌شوند. در این مثال داده آزمون ۱ و داده آزمون ۳، بیش‌ترین اختلاف توابع فراخوانی API را دارا هستند. بنابراین، به‌عنوان والدین انتخاب می‌شوند. پس از اعمال مراحل الگوریتم ژنتیک روی این دو عضو، داده آزمون بعدی تولید خواهد شد. درنهایت به‌منظور جلوگیری از انتخاب دوباره این دو عضو، عدد اختلاف آن‌ها در ماتریس برابر صفر قرار داده می‌شود. ماتریس جدید به‌صورت زیر است:

	1.txt	2.txt	3.txt
1.txt	۰	۱	(۰)
2.txt	۱	۰	۲
3.txt	(۰)	۲	۰

داده آزمون جدید پس از مرحله تحلیل رفتاری باید به مجموعه داده آزمون‌ها اضافه شود. اگر فرض شود در مسیر اجرایی برنامه با داده آزمون جدید (نام آن 4.txt است)، توابع CreateFile و WriteFile فراخوانی شود، مجموعه داده آزمون به‌صورت جدول (۲) به‌روزرسانی می‌شود.

جدول ۲. مجموعه داده آزمون پس از افزودن داده آزمون 4.txt

شماره داده آزمون	نام فایل داده آزمون	توابع API فراخوانی شده
۱	1.txt	CreateFile, ReadFile
۲	2.txt	ReadFile
۳	3.txt	StrNCat
۴	4.txt	CreateFile, WriteFile

سپس به‌منظور اضافه کردن عضو جدید 4.txt به ماتریس، باید مقدار درایه متناظر آن در ماتریس محاسبه گردد.

اندازه D_{14} مربوط به دو داده آزمون ۱ و ۴ به‌صورت زیر محاسبه می‌شود:

$$D_{۱۴} = D_{۴۱} = |\{\text{CreateFile, ReadFile}\} \Delta \{\text{CreateFile, WriteFile}\}|$$
$$= |\{\text{ReadFile, WriteFile}\}| = ۲$$

اندازه D_{24} مربوط به دو داده آزمون ۲ و ۴ به‌صورت زیر محاسبه می‌شود:

$$D_{۲۴} = D_{۴۲} = |\{\text{ReadFile}\} \Delta \{\text{CreateFile, WriteFile}\}|$$
$$= |\{\text{ReadFile, CreateFile, WriteFile}\}| = ۳$$

اندازه D_{34} مربوط به دو داده آزمون ۳ و ۴ به‌صورت زیر محاسبه می‌شود:

$$D_{۳۴} = D_{۴۳} = |\{\text{StrNCat}\} \Delta \{\text{CreateFile, WriteFile}\}|$$
$$= |\{\text{StrNCat, CreateFile, WriteFile}\}| = ۳$$

پس از افزودن داده آزمون ۴، ماتریس اختلاف به‌صورت زیر نمایش داده می‌شود:

	1.txt	2.txt	3.txt	4.txt
1.txt	۰	۱	۰	۲
2.txt	۱	۰	۲	۳
3.txt	۰	۲	۰	۳
4.txt	۲	۳	۳	۰

حال، به‌منظور تولید داده آزمون بعدی، دو والد که بیش‌ترین اختلاف را دارند، انتخاب می‌شوند. دو انتخاب وجود دارد، داده آزمون ۲ و ۴ و داده آزمون ۳ و ۴، الگوریتم به تصادف یکی را انتخاب کرده و سایر مراحل روی آن اعمال می‌شود.

این روند همچنان ادامه می‌یابد. داده آزمون‌هایی که توابع API فراخوانی شده در مسیر اجرایی متناظرشان تفاوت بیشتری با هم دارند به‌عنوان والدین انتخاب و داده آزمون بعدی را تولید می‌کنند. زمانی که مسیرهای پیمایش شده توسط مجموعه داده آزمون‌ها مشابه هم باشند، به این معنی است که لیست توابع API فراخوانی شده در آن‌ها مشابه هم است. در این صورت اندازه مجموعه تفاضل متقارن لیست توابع API فراخوانی شده در هر دو مسیر اجرایی موجود به سمت صفر می‌رود. در این شرایط، در صورت ادامه فرایند تولید داده آزمون، داده آزمون‌هایی تولید می‌شوند که مسیرهای تکراری را پیمایش می‌کنند. به دلیل اینکه این مسیرها یک‌بار به‌طور کامل مورد بررسی و تحلیل رفتاری قرار گرفته‌اند، پیمایش و تحلیل دوباره آن‌ها علاوه بر اینکه اطلاعات جدید را ارائه نمی‌دهد، باعث افزایش سربار سیستم نیز می‌شود. به‌منظور جلوگیری از تولید مسیرهای تکراری، زمانی که دایره‌های موجود در ماتریس A از یک مقدار حد آستانه کمتر باشد، فرایند تولید داده آزمون متوقف می‌شود. در ابزار ارائه شده، آزمون‌کننده می‌تواند خود مقدار حد آستانه را تعیین کند. در این مثال حد آستانه برابر ۲ در نظر گرفته شده است. بدین معنی که زمانی که بیشینه کل برابر ۲ و یا کمتر از ۲ باشد فرایند تولید داده آزمون متوقف می‌شود. بنابراین، در زمان توقف فرایند تولید داده آزمون تمام درایه‌های موجود در ماتریس A مقدار ۲ و یا مقداری کمتر از ۲ دارند. به‌صورت خلاصه فرایند تولید داده آزمون با استفاده از الگوریتم ژنتیک شامل مراحل زیر است:

۱. دریافت لیست توابع API فراخوانی شده در مسیر اجرایی متناظر با مجموعه داده آزمون‌های اولیه.

۲. محاسبه اندازه مجموعه حاصل از تفاضل متقارن دو مجموعه توابع API فراخوانی شده در مسیرهای اجرایی متناظر با هر دو داده آزمون موجود در مجموعه داده آزمون‌های اولیه.

۳. ایجاد ماتریس A و قرار دادن مقادیر محاسبه‌شده در مرحله ۲ در درایه‌های آن. عناوین سطرها و ستون‌های ماتریس A داده آزمون‌ها هستند.

۴. محاسبه بیشینه مقدار درایه‌های ماتریس.

۵. درصورتی‌که مقدار بیشینه از حد آستانه تعیین‌شده برابر و یا کمتر باشد، مرحله ۱۳ اجرا شود. (بررسی شرط پایان فرایند تولید داده آزمون).

۶. انتخاب دو داده آزمون متناظر با مقدار بیشینه به‌عنوان والد.

۷. تولید داده آزمون جدید.

۸. تحلیل رفتاری برنامه در حین اجرای داده آزمون جدید.

۹. دریافت لیست توابع API فراخوانی شده در مسیر اجرایی متناظر با داده آزمون جدید.

۱۰. محاسبه اندازه مجموعه حاصل از تفاضل متقارن مجموعه توابع API فراخوانی شده در مسیر اجرایی متناظر با داده آزمون جدید با سایر داده آزمون‌های موجود در مجموعه داده آزمون‌ها.

۱۱. اضافه کردن مقادیر محاسبه‌شده در مرحله ۱۰ به ماتریس A.

۱۲. اجرای مرحله ۴.

۱۳. خروج.

۴-۲. اجرای برنامه با موارد آزمون تولیدشده

در این مرحله مورد آزمون تولیدشده، توسط برنامه اجرا می‌شود. در حین اجرا برنامه مورد تحلیل رفتاری قرار می‌گیرد و اطلاعات مورد نیاز از جمله لیست توابع API فراخوانی شده در مسیر اجرایی و آدرس‌های آلوده استخراج می‌شوند. اگر مورد آزمون باعث افشای آسیب‌پذیری موجود در برنامه شود، توسط ابزار اشکال‌زدای تعبیه‌شده، آسیب‌پذیری به همراه مکان و علت آن گزارش داده می‌شود.

۴-۳. تولید کد اسمبلی و مستند گذاری آن

همان‌طور که اشاره شد، ورودی فازر ارائه شده فایل اجرایی دودویی است. به‌منظور تحلیل رفتاری این برنامه، نیاز به تبدیل آن به زبان اسمبلی است. بدین منظور از کتابخانه PIN استفاده شده است. با استفاده از این کتابخانه با تبدیل کد ماشین برنامه کاربردی به زبان اسمبلی، می‌توان کد اسمبلی را در سطوح مختلفی مستندگذاری کرد. به‌منظور مستندگذاری برنامه کتابخانه PIN با اضافه کردن جملاتی به برنامه سعی دارد رفتار

برنامه را در زمان اجرای آن مورد بررسی و تحلیل قرار دهد. با توجه به اطلاعاتی مورد نیاز در زمان اجرا، PIN در سطوح مختلفی کد اسمبلی را مستندگذاری می‌کند.

در این مقاله، کد در سطح دستورالعمل‌ها و توابع API مستندگذاری می‌شود. مستندگذاری در سطح دستورالعمل‌ها باعث دسترسی به لیست دستورالعمل‌های اجرا شده در مسیر اجرایی برنامه می‌شود و مستندگذاری در سطح توابع API باعث دسترسی به لیست توابع API فراخوانی شده در مسیر اجرایی برنامه می‌شود. در مراحل بعدی از این نتایج استفاده می‌شود.

۴-۴. شناسایی آدرس‌هایی از حافظه شامل مورد آزمون خوانده‌شده توسط برنامه

هنگامی‌که برنامه اجرا می‌شود و مورد آزمون را به‌عنوان ورودی دریافت می‌کند، آن را در حافظه بارگذاری می‌کند. آدرس‌هایی از حافظه که شامل داده ورودی کاربر هستند، به‌عنوان نقاط آلوده حافظه در نظر گرفته می‌شوند. زیرا در صورت وجود آسیب‌پذیری در برنامه کاربردی، مهاجم از طریق این نقاط می‌تواند به برنامه نفوذ کند و به بهره‌برداری از آسیب‌پذیری بپردازد. به آدرس‌هایی از حافظه شامل مورد آزمون خوانده‌شده توسط برنامه هستند، آدرس‌های آلوده گفته می‌شود.

۴-۵. انتشار آدرس‌های آلوده در حافظه

پس از شناسایی آدرس‌های آلوده حافظه، مرحله بعدی مسیریابی و انتشار این آدرس‌ها در حافظه است. همه داده‌های متناظر با آدرس‌های آلوده، به‌عنوان داده آلوده در نظر گرفته می‌شوند. به‌عنوان‌مثال اگر x به‌عنوان داده آلوده در نظر گرفته شده است و داشته باشیم y=x+1 و z=y+1 در این مثال y و z هر دو به‌عنوان داده آلوده در نظر گرفته می‌شوند و آدرس‌هایی که y و z را ذخیره می‌کنند، به‌عنوان آدرس آلوده برچسب‌گذاری می‌شوند. درواقع، دستورالعمل‌های برنامه از طریق رونویسی کردن و تغییر دادن داده آلوده، آن را در حافظه انتشار می‌دهند.

در حین اجرای برنامه، هر دستورالعملی که اجرا می‌شود، مستند گذاری شده و توابع متناسب برای اعمال انتشار آلودگی در حافظه، فراخوانی می‌شوند. به‌منظور انجام عمل انتشار آلودگی در حافظه، دستورالعمل‌های اسمبلی در سه دسته زیر تقسیم‌بندی شده‌اند:

- دستورالعمل‌های خواندن از حافظه (بارگذاری): دستورالعمل‌هایی که محتوای حافظه را به ثبات‌های پردازنده منتقل می‌کنند. در زمان تحلیل برنامه، اگر دستورالعمل از نوع خواندن از حافظه باشد، باید بررسی شود، آیا آدرس حافظه‌ای که داده از آن خوانده می‌شود، جزء آدرس‌های آلوده است یا خیر. درصورتی‌که آدرس حافظه آلوده باشد، ثبات مقصد نیز به‌عنوان ثبات آلوده برچسب‌گذاری می‌شود. همچنین درصورتی‌که آدرس حافظه آلوده نباشد، ثبات مقصد اگر در لیست ثبات‌های آلوده است، باید از این لیست حذف شود.
- دستورالعمل‌های نوشتن در حافظه (ذخیره‌سازی): دستورالعمل‌هایی هستند که محتوای ثبات‌های پردازنده را به حافظه منتقل می‌کنند. در زمان تحلیل برنامه، اگر دستورالعمل از نوع نوشتن در حافظه باشد، باید بررسی شود، آیا ثباتی که داده از آن خوانده می‌شود، جزء ثبات‌های آلوده است یا خیر. درصورتی‌که ثبات آلوده باشد، آدرس حافظه مقصد نیز به‌عنوان آدرس آلوده برچسب‌گذاری می‌شود. همچنین درصورتی‌که ثبات مبدأ آلوده نباشد، آدرس حافظه اگر در لیست آدرس‌های آلوده است، باید از این لیست حذف شود.
- دستورالعمل‌های بین ثبات‌ها: دستورالعمل‌هایی هستند که بین ثبات‌های پردازنده اجرا می‌شوند. در زمان تحلیل برنامه، اگر دستورالعمل بین ثبات‌ها باشد، باید بررسی شود، آیا ثبات مبدأ، جزء ثبات‌های آلوده هست یا خیر. درصورتی‌که ثبات مبدأ آلوده باشد، ثبات مقصد نیز به‌عنوان ثبات آلوده برچسب‌گذاری می‌شود. همچنین درصورتی‌که ثبات مبدأ آلوده نباشد، ثبات مقصد اگر در لیست ثبات‌های آلوده است، باید از لیست حذف شود. در صورت وجود دستورالعمل‌های پاک‌کننده، مانند xor eax,eax و sub eax,eax که باعث پاک شدن محتوای ثبات خاصی می‌شوند، باید ثبات را از لیست ثبات‌های آلوده حذف کرد. پس از شناسایی و انتشار آدرس‌های آلوده، تعدادی از آدرس‌های حافظه، به‌عنوان آدرس آلوده برچسب‌گذاری می‌شوند، این آدرس‌ها، به‌منظور استفاده در مراحل بعد، در یک لیست نگه‌داری می‌شوند.

۴-۶. استخراج توابع API فراخوانی شده در مسیر اجرایی

همان‌طور که اشاره شد در حین تحلیل رفتاری برنامه، کد در سطح توابع API مستندگذاری می‌شود. از این طریق می‌توان توابع API در مسیری که توسط برنامه طی می‌شود را استخراج نمود. هر زمان که برنامه داده آزمون جدید را به‌عنوان ورودی دریافت کرده و اجرا می‌کند، مسیری از دستورالعمل‌ها را طی می‌کند. تفاوت این مسیرها بستگی به تفاوت موارد آزمون دارد. هر مقدار که موارد آزمون اختلاف بیشتری با هم داشته باشند، مسیرهای اجرایی طی شده توسط برنامه نیز تفاوت بیشتری با هم دارند. به‌منظور شناسایی آسیب‌پذیری در برنامه، باید سعی شود، همه مسیرهای اجرایی برنامه مورد تحلیل و بررسی قرار گیرد. برای نیل به این هدف باید موارد آزمون مختلفی تولید شود. علاوه بر این باید سعی شود که مسیرهای مشابه کمتری پیمایش

شود. زیرا هر مسیر یک‌بار به‌صورت دقیق مورد تحلیل قرار گرفته است و تحلیل دوباره آن اطلاعات جدیدی ارائه نمی‌دهد.

به‌منظور شناسایی مسیرهای طی شده، برای هر مسیر اجرایی توابع API فراخوانی شده در آن مسیر استخراج می‌شود. لیست توابع API استخراج‌شده علاوه بر استفاده در مرحله بعد، در اجرای الگوریتم ژنتیک نیز استفاده می‌شود.

۴-۷. بررسی وجود توابع API ناامن

توابع ناامن توابعی هستند که می‌توانند باعث ایجاد آسیب‌پذیری در کد شوند. در این مرحله از روش باید توابع ناامن، در لیست توابع API فراخوانی شده در هر مسیر اجرایی، جستجو شود. درصورتی‌که مسیر اجرایی شامل توابع ناامن باشد، آن مسیر به‌عنوان مسیر بحرانی برچسب‌گذاری می‌شود. به‌عنوان‌مثال اگر در مسیر اجرایی یک برنامه توابع API زیر فراخوانی شده باشند:

{SendMessageW, wsprintfW, Strcpy}

دو تابع wsprintfW و Strcpy جزء توابع ناامن هستند. یعنی در مسیر اجرایی این داده آزمون توابع ناامن فراخوانی شده‌اند.

۴-۸. استخراج آدرس‌های حافظه مورد استفاده در توابع ناامن موجود در مسیر اجرایی

هرکدام از توابع ناامن، دارای پارامترهایی هستند. این پارامترها اطلاعات خود را از حافظه دریافت و به آن منتقل می‌کنند. به‌عنوان‌مثال فرم کلی تابع Srcpy به‌صورت زیر است:

```
char *strcpy (char *strDestination, const char *strSource);
```

این تابع رشته strSource را در رشته strDestination رونویسی می‌کند. تابع Strcpy به دلیل اینکه قبل از انجام عمل رونوشت، اندازه بافر مقصد را اندازه‌گیری نمی‌کند، جزء توابع ناامن است. زیرا اگر اندازه میانگیر مقصد کوچک‌تر از رشته منبع باشد، فضای کافی برای رونوشت همه رشته در میانگیر مقصد وجود ندارد و درنتیجه میانگیر خروجی سرریز می‌شود.

۴-۹. جستجوی آدرس‌های حافظه مورد استفاده در توابع ناامن در آدرس‌های آلوده

همان‌طور که در مرحله قبل اشاره شد، هرکدام از توابع ناامن آدرس‌های از حافظه را به‌عنوان پارامتر دریافت می‌کنند. در این مرحله بررسی می‌شود که آیا این آدرس‌ها، جزء آدرس‌های آلوده (آدرس‌های شامل فایل ورودی) هستند یا خیر.

۴-۱۰. تولید داده آزمون با استفاده از الگوریتم شکست

به‌منظور ایجاد آسیب‌پذیری در برنامه باید شروط زیر برقرار باشد:

- توابع ناامن در مسیرهای اجرایی برنامه باشند.
- آدرس‌های مورد استفاده در پارامترهای توابع ناامن، جزء آدرس‌های آلوده باشند. یعنی بتوان از طریق ورودی آن‌ها را تغییر داد.

در صورت برقراری دو شرط بالا، الگوریتم شکست، داده آزمونی تولید می‌کند، که باعث افشای آسیب‌پذیری آشکارشده در برنامه می‌شود. درواقع پس از یافتن نقطه آسیب‌پذیری و استخراج شرایط رخ دادن آسیب‌پذیری، الگوریتم شکست با توجه به این اطلاعات، داده آزمونی تولید می‌کند که منجر به شکست برنامه می‌شود. وظیفه الگوریتم شکست، تولید داده آزمون مناسب به‌منظور افشای آسیب‌پذیری ناشی از توابع ناامن است. توابع ناامن در نظر گرفته شده در این پایان‌نامه، در صورت استفاده نادرست منجر به ایجاد آسیب‌پذیری سرریز میانگیر می‌شوند. درنتیجه الگوریتم شکست روی تولید داده آزمون‌هایی تمرکز دارد که باعث افشاء شدن آسیب‌پذیری سرریز میانگیر می‌شوند. داده آزمون‌های تولید شده توسط الگوریتم شکست، باعث تغییر اندازه پارامترهای توابع ناامن شده و از این طریق منجر به ایجاد آسیب‌پذیری و شکست برنامه می‌شود.

ورودی الگوریتم شکست اطلاعات مربوط به تابع ناامن است. این اطلاعات شامل نام تابع، آدرس(های) حافظه پارامتر مبدأ (میانگیر مبدأ)، اندازه میانگیر مبدأ و اندازه میانگیر مقصد است. این الگوریتم ابتدا، مکان میانگیر مبدأ را در فایل ورودی برنامه می‌یابد. سپس با در نظر گرفتن اندازه میانگیر مبدأ و اندازه میانگیر مقصد، اندازه داده‌ای که باید به مکان میانگیر مبدأ در فایل ورودی برنامه اضافه شود، محاسبه می‌شود. درنهایت الگوریتم با اضافه کردن داده محاسبه‌شده به برنامه، داده آزمونی جدیدی تولید می‌کند. درصورتی‌که مسیر اجرایی برنامه با ورودی داده آزمون جدید تغییر نکند، این داده آزمون منجر به شکست برنامه می‌شود. خروجی الگوریتم شکست، داده آزمونی است که مسبب افشای آسیب‌پذیری تابع ناامن موجود در مسیر اجرایی برنامه می‌شود. در ورودی الگوریتم شکست، از نتایج تحلیل پویای برنامه استفاده شده است. مراحل الگوریتم شکست در زیر آمده است:

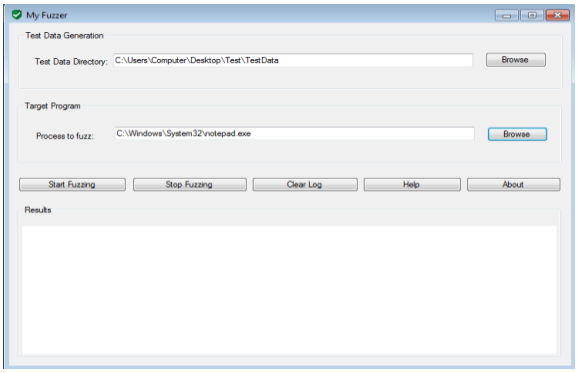
۱. دریافت ورودی‌های الگوریتم از بخش تحلیل پویای برنامه، ورودی‌ها شامل نام تابع، آدرس(های) حافظه پارامتر مبدأ (میانگیر مبدأ)، اندازه میانگیر مبدأ و اندازه میانگیر مقصد.
۲. به دست آوردن مکان میانگیر مبدأ در فایل ورودی برنامه.
۳. تفریق اندازه میانگیر مبدأ از اندازه میانگیر مقصد.
۴. اضافه کردن مقدار یک یا بیشتر از یک به مقدار به‌دست‌آمده در مرحله ۳.
۵. تولید داده به اندازه مقدار به‌دست‌آمده در مرحله ۴.

اضافه کردن داده تولیدشده در مرحله ۴ به مکان میانگیر

مبدأ را در فایل ورودی برنامه. (تولید داده آزمون مسبب آسیب‌پذیری سرریز میانگیر.)

۵. نتایج و بحث

با توجه به توضیحات بخش قبلی، ابزاری پیاده‌سازی شده است که در شکل (۵) صفحه اولیه آن ارائه شده است. به‌منظور ارزیابی روش پیشنهادی تعدادی برنامه نوشته شده است. نکته مهم در مورد برنامه‌های مورد آزمون، برخورداری از تعداد مسیرهای مناسب و فراخوانی توابع ناامن در این مسیرهاست. زیرا برای سنجش میزان کارایی روش، وجود مسیرهای متفاوت و وجود توابع ناامن در این مسیرها، ضروری است. مورد دوم در مورد این برنامه‌ها داشتن پارامترهای ورودی به‌صورت فایل متنی است. علی‌رغم این‌که روش پیشنهادی مستقل از نوع ورودی‌هاست، ولیکن تاکنون روش، برای پردازشگرهای فایل‌های متنی پیاده‌سازی شده است. بر این اساس، سه برنامه TextEditor_1، TextEditor_2، TextEditor_3 مورد آزمون قرار گرفته‌اند. این برنامه‌ها فایل متنی را از ورودی می‌خوانند و روی داده‌های خوانده‌شده عملیات انجام می‌دهند. تفاوت این سه برنامه در تعداد مسیرهای برنامه و تعداد توابع ناامن مورد فراخوانی است. این برنامه‌ها و جزئیات مربوط به آن‌ها در جدول (۳) آورده شده‌اند.



شکل ۵. ابزار پیاده‌سازی شده برای فازینگ

جدول ۳. برنامه‌های مورد آزمون و مشخصات آن‌ها

نام برنامه	تعداد مسیرهای ایستای برنامه	تعداد خطوط برنامه	تعداد توابع ناامن فراخوانی شده
TextEditor_1	۵۲	۴۱۲	۱۰
TextEditor_2	۱۰۸	۱۱۲۵	۲۰
TextEditor_3	۲۰۵	۲۳۱۰	۵۰

۵-۱. بررسی پوشش مسیرهای برنامه

هدف این مورد، ارزیابی روش پیشنهادی در پوشش مسیرهای برنامه است. در راهکار پیشنهادی به‌منظور پوشش حداکثری مسیرهای برنامه از الگوریتم ژنتیک استفاده شده است. روش کار این الگوریتم در فصل قبل توضیح داده شد. در اینجا روش ارائه شده با استفاده یک مثال واقعی تشریح می‌شود. بدین منظور برنامه TextEditor_1 در نظر گرفته شده است.

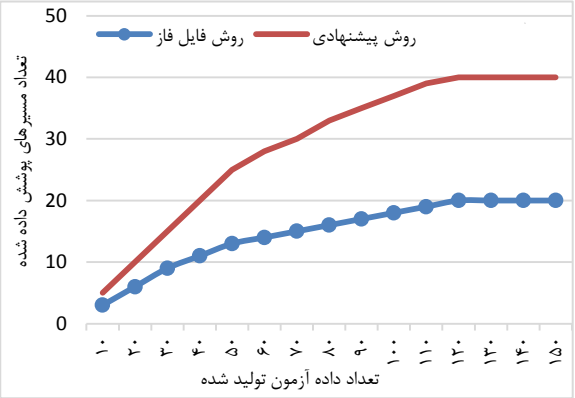
مقایسه روش پیشنهادی با ابزار فایل فاز انجام شده است [۱]. این فازر تحت سیستم‌عامل ویندوز بوده و دارای واسط کاربری است که به کاربر اجازه تولید فایل‌ها را به‌عنوان داده آزمون داده و امکان اجرای این فایل‌های آزمون را بر روی برنامه هدف و بررسی رفتار آن برنامه در قبال داده‌های آزمون را می‌دهد. در زمان انجام عملیات فازینگ، بخش‌هایی از فایل اصلی تغییر یافته و دوباره به برنامه هدف داده می‌شود. این فازر همچنین دارای امکان بررسی رفتار برنامه هدف در زمان وقوع اتفاقات خاص همانند کرش کردن آن است. دلیل انتخاب این فاز به خاطر شباهت زیاد روش پیشنهادی با روش موجود در این فازر از حیث تولید داده آزمون از جنس فایل (فایل فرمت) و همچنین مکانیسم فازینگ و روش بررسی رفتار برنامه هدف است. در ابزار فایل فاز، پس از شناسایی نقاط آسیب‌پذیر فایل هدف با استفاده از روش‌های جستجوی سطحی، از روش‌های سیل‌آسا همانند تولید داده‌های تغییر یافته به شکل تصادفی در نقاط آسیب‌پذیر به‌منظور تولید داده‌های بعدی استفاده می‌شود. اطلاعات مربوط به مسیرهای پوشش داده شده توسط روش ابزار فایل فاز و روش پیشنهادی مقایسه شده و در جدول (۴) نمایش داده شده است. در این شکل همچنین میزان معیار پوشش مسیرهای برنامه نیز آورده شده است.

جدول ۴. اطلاعات مربوط به برنامه‌های مورد آزمون

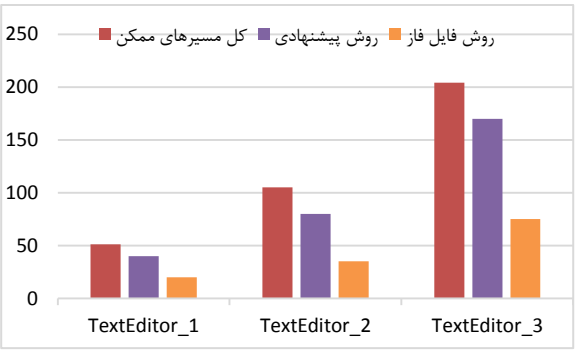
نام برنامه	تعداد مسیرهای ایستای برنامه	مسیرهای پوشش داده شده توسط روش فایل فاز	مسیرهای پوشش داده شده توسط روش پیشنهادی	درصد معیار پوشش روش فایل فاز	درصد معیار پوشش روش پیشنهادی
TextEditor_1	۵۲	۱۸	۴۱	۳۴ درصد	۷۸ درصد
TextEditor_2	۱۰۸	۳۶	۸۱	۳۳ درصد	۷۵ درصد
TextEditor_3	۲۰۵	۷۴	۱۶۷	۳۶ درصد	۸۱ درصد

هدف دیگر این مورد مطالعاتی، نمایش روند رشد پوشش مسیرهای اجرایی روش پیشنهادی در برنامه‌های مورد بررسی است. برای نمایش این روند از نمودار استفاده شده است. در شکل (۶) روند اجرا عملیات و افزایش پوشش برای برنامه

TextEditor_1 آورده شده است. در شکل (۷) هر برنامه ازنظر میزان تعداد مسیرهای تولید شده توسط دو روش فایل فاز و روش پیشنهادی مقایسه شده است.



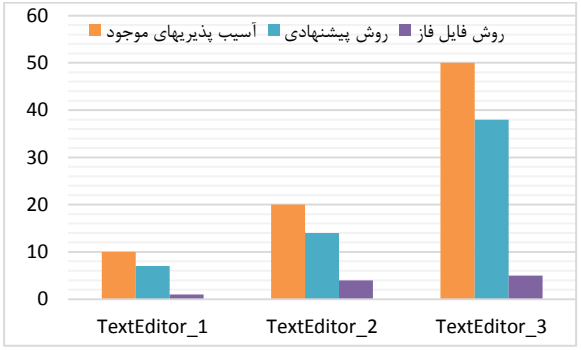
شکل ۶. روند پیشرفت پوشش مسیرهای برنامه TextEditor_1



شکل ۷. مقایسه هر سه برنامه در مورد پوشش مسیرها

۵-۲. شناسایی آسیب‌پذیری‌های موجود در مسیرهای اجرایی

در این مورد، هدف شناسایی آسیب‌پذیری‌های موجود در مسیرهای اجرایی است. همان‌طور که در مورد مطالعاتی قبل نشان داده شد، روش پیشنهادی معیار پوشش مسیر را نسبت به روش فایل فاز افزایش می‌دهد. درنتیجه با پیمایش مسیرهای بیشتر، آسیب‌پذیری‌های بیشتری را شناسایی می‌کند. مهم‌ترین مزیت روش پیشنهادی این است که با استفاده از اطلاعات به‌دست‌آمده از تحلیل رفتاری برنامه، بلافاصله بعد از شناسایی آسیب‌پذیری، داده آزمون متناسب به‌منظور افشای آسیب‌پذیری را تولید می‌کند. اما در روش فایل فاز به دلیل نادیده گرفتن رفتار برنامه، در صورت وجود آسیب‌پذیری در مسیر، افشای آن فقط توسط داده آزمون فعلی صورت می‌گیرد. در صورت پیمایش آن مسیر و افشا نشدن آسیب‌پذیری، ممکن است داده آزمون‌های تولید شده بعدی، این آسیب‌پذیری را شناسایی نکنند. درنتیجه، مسیری که شامل آسیب‌پذیری بوده است، پیمایش شده اما آسیب‌پذیری موجود شناسایی نشده است. به‌طورکلی روش فایل فاز علاوه بر پوشش مسیرهای کمتر، در همان مسیرها نیز در صورت وجود آسیب‌پذیری، ممکن است بدون شناسایی آسیب‌پذیری از آن‌ها عبور کنند. شکل (۸) تعداد آسیب‌پذیری‌های شناسایی شده در روش فایل فاز و روش پیشنهادی در برنامه‌های مورد آزمون را نشان می‌دهد.

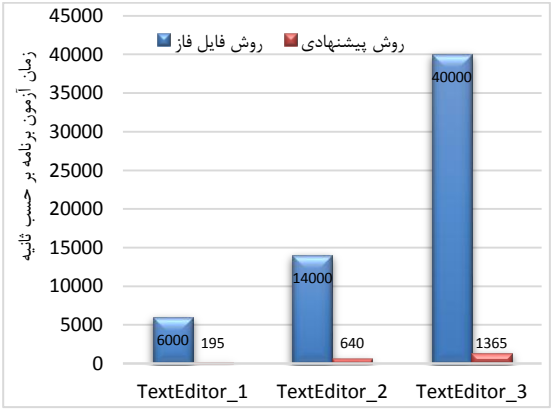


شکل ۸. مقایسه سه برنامه در مورد شناسایی آسیب‌پذیری‌های موجود

۵-۳. زمان مورد نیاز برای آزمون

در این مورد، زمان مورد نیاز برای آزمون نرم‌افزار در روش پیشنهادی و روش فایل فاز مورد بررسی قرار می‌گیرند. به‌منظور مقایسه دو روش، تا زمانی که تعداد آسیب‌پذیری‌های شناسایی شده در هر دو روش یکسان باشد، داده آزمون تولید می‌شود. همان‌طور که در شکل (۹) مشاهده می‌شود به‌منظور دست یافتن به یک نتیجه یکسان، روش پیشنهادی نسبت به روش فایل فاز، زمان بسیار کمتری را صرف می‌کند. این موضوع بر تولید هوشمندانه داده آزمون در روش پیشنهادی حکایت می‌کند.

شکل (۹) کارایی روش پیشنهادی را نسبت به روش فایل فاز نشان می‌دهد. در صورت استفاده از روش تولید داده آزمون پیشنهادی در فرایند آزمون نرم‌افزار، زمان مورد نیاز بسیار کمتر از استفاده از روش فایل فاز است. این کاهش زمان ناشی از تولید داده آزمون مؤثر است.



شکل ۹. مقایسه سه برنامه درباره زمان مورد نیاز آزمون نرم‌افزار

۶. نتیجه‌گیری

بررسی‌های انجام شده دلالت بر وجود آسیب‌پذیری در تعداد محدودی از توابع سیستمی رایج است. با توجه به چگونگی استفاده از این توابع در برنامه می‌توان مکان‌های مستعد نفوذ در کد برنامه‌ها در زمان اجرا را مشخص نمود. در فرایند تکاملی تابع هدف انتخاب مسیرهای جدید با تعداد بیشتر توابع آسیب‌پذیر در داخل آن مسیرها می‌تواند باشد. به‌این‌ترتیب، بعد از عمل تقاطع گیری و ایجاد کروموزوم‌ها در قالب مسیرهای جدید، کروموزوم‌های جدید در صورتی جایگزین پدر می‌شوند که تعداد توابع آسیب‌پذیر بیشتری در آن‌ها مشاهده گردد. به‌این‌ترتیب می‌توان مشکل شناخته‌شده انفجار مسیر را از میان برداشت. نقطه ضعف روش پیشنهادی، طولانی بودن فرایند تکاملی بخصوص در برنامه‌های بسیار بزرگ است. این مشکل را با روش‌های محاسبات با کارایی بالا مثل رایانش نرم توزیع شده از میان برداشت. این مسئله به‌عنوان کار آتی مطرح است. در بخش ارزیابی نشان داده شده که روش پیشنهادی به دلیل تولید داده آزمون کمتر، دارای سرعت و دقت بیشتری نسبت به روش‌های مشابه همانند روش ارائه شده در ابزار فایل فاز است. همچنین روش پیشنهادی به دلیل افزایش پوشش مسیرهای اجرایی برنامه، آسیب‌پذیری‌های پنهان بیشتری را در معرض افشاء شدن قرار می‌دهد. درنتیجه قابلیت اطمینان برنامه را افزایش می‌دهد. روش پیشنهادی، روی توابع ناامن موجود در زبان C/C++ به‌عنوان آسیب‌پذیری تمرکز دارد. در صورت به دست آوردن الگوهای آسیب‌پذیری سایر زبان‌ها می‌توان دقت روش را در یافتن همه انواع آسیب‌پذیری‌های موجود در برنامه‌های اجرایی افزایش داد.

۷. مرجع‌ها

[1] Takanen, A.; Demott, J. D.; Miller, C. “Fuzzing for Software Security Testing and Quality Assurance”; Artech. House, 2008.

[2] Godefroid, P.; Levin, M.; Molnar, Y. “Automated White Box Fuzz Testing”; Proceedings of Network and Distributed Systems Security 2008.

[3] Marcellino, B. A.; Hsiao, M. S. “Dynamic Partitioning Strategy to Enhance Symbolic Execution”; Design, Automation & Test in Europe Conference & Exhibition 2016, 774-779.

[4] Yang, S.; Man, T.; Xu, J.; Zeng, F.; Li, K. “RGA: A Lightweight and Effective Regeneration Genetic Algorithm for Coverage-oriented Software Test Data Generation”; Information and Software Technology 2016, 76, 19-30.

[5] Chen, T. Y.; Kuo, F. C.; Merkel, R. G.; Tse, T. H. “Adaptive Random Testing: The Art of Test Case Diversity”; J. Systems and Software 2010, 83, 60-66.

[6] Barus, A. C.; Chen, T. Y.; Kuo, F. C.; Liu, H.; Merkel, R.; Rothermel, G. “A Cost-Effective Random Testing Method for Programs with Non-Numeric Inputs”; IEEE Trans. Computers 2016, 99, 1-4.

[7] Liu, B.; Shi, L.; Cai, Z.; Li, M. “Software Vulnerability Discovery Techniques: A Survey”; Fourth International Conference on Multimedia Information Networking and Security 2012, 152-156.

[8] Nouman, M.; Pervez, U.; Hasan, O.; Saghar, K. “Software Testing: A Survey and Tutorial on White and Black-Box Testing of C/C++ Programs”; Region 10 Symposium IEEE 2016, 225-230.

[9] McNally, R.; Yiu, K.; Grove, D.; Gerhardy, D. “Fuzzing: The State of the Art”; Defense Science and Technology Organization Edinburgh (Australia), 2012.

[10] Mouzarani, M.; Sadeghiyan, B.; Zolfaghari, M. “A Smart Fuzzing Method for Detecting Stack-based Buffer Overflow in Binary Codes”; IET Software 2016, 10, 96-107.

[11] Chen, T.; Zhang, X. S.; Guo, S. Z.; Li, H. Y.; Wu, Y. “State of the Art: Dynamic Symbolic Execution for Automated Test Generation”; Future Generation Computer Systems 2013, 29, 1758-1773.

[12] Mouzarani, M.; Sadeghiyan, B.; Zolfaghari, M. “A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes”; IEEE 21st Pacific Rim Int. Symposium Dependable Computing 2015, 42-49.

[13] Fangquan, D.; Chaoqun, D.; Yao, Z.; Teng, L. “Binary-Oriented Hybrid Fuzz Testing”; I6th IEEE Int. Conf. Software Engineering and Service Science 2015, 345-348,.

[14] Pham, V. T.; Ng, W. B.; Rubinov, K.; Roychoudhury, A. “Hercules: Reproducing Crashes in Real-World Application Binaries”; 37th Int. Conf. Software Eng. 2015, 891-901.

[15] Khatun, S.; Rabbi, K. F.; Yaakub, C. Y.; Klaib, M. J. “A Random Search Based Effective Algorithm for Pairwise Test Data Generation”; Int. Conf. Electrical, Control and Computer Engineering 2011, 293-297.

[16] Huang, R.; Xie, X.; Chen, T. Y.; Lu, Y. “Adaptive Random Test Case Generation for Combinatorial Testing”; IEEE 36th Annual Computer Software and Applications Conference 2012, 52-61.

[17] Chen, T. Y.; Kuo, F. C.; Merkel, R. G.; Ng, S .P. “Mirror Adaptive Random Testing”; In Information and Software Technology 2004, 46, 1001-1010.

[18] Huang, R.; Liu, H.; Xie, X. “Enhancing Mirror Adaptive Random Testing Through Dynamic Partitioning”; Information and Software Technology 2015, 67, 13-29.

[19] Shuai, B.; Li, M.; Li, H.; Zhang, Q. “Test Case Generation for Vulnerability Detection Using Genetic Algorithm”; 4rd Int. Conf. Consumer Electronics, Communications and Networks 2015, 1198-1203.

[20] Shuai, B.; Li, M.; Li, H.; Zhang, Q. “Software Vulnerability Detection Using Genetic Algorithm and Dynamic Taint Analysis”; 3rd Int. Conf. Consumer Electronics, Communications and Networks 2013, 589-593.

[21] Shuai, B.; Li, H.; Zhang, L.; Zhang, Q. “Software Vulnerability Detection Based on Code Coverage and Test Cost”; 11th Int. Conf. Comput. Intelligence and Security 2015, 317-321.

[22] Yi, M. “The Research of Path-Oriented Test Data Generation Based on a Mixed Ant Colony System Algorithm and Genetic algorithm”; 8th Int. Conf. Wireless Communications, Networking and Mobile Computing 2012, 1-4.

[23] Aleti, A.; Grunske, L. “Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm”; J. Systems and Software 2015, 103, 343-352.

[24] Mann, M.; Sangwan, O. P.; Singh, S. “Automatic Goal-Oriented Test Data Generation Using a Genetic Algorithm and Simulated Annealing”; 6th Int. Conf. Cloud System and Big Data Engineering 2016, 83-87.

[25] Kun, W.; Yichen, W. “Software Test Case Generation Based on the Fault Propagation Path Coverage”; Annual Reliability and Maintainability Symposium 2016, 1-4.