

Improving Code Coverage Metrics for Discovering Vulnerabilities in Stateful Network Protocols using Hybrid Fuzzing

Hamid Rezaei Rahvard¹, Mehdi Salkhordeh Haghighi^{2*}

¹ Faculty of Computer Engineering and IT, Sadjad University, Mashhad, Iran

² Faculty of Computer Engineering and IT, Sadjad University, Mashhad, Iran

Received: 15 October 2023, Revised: 05 January 2024, Accepted: 02 February 2024

Paper type: Research

Abstract

Fuzzing software is a method for finding security vulnerabilities in applications. In this method, by sending random data to the program, attempts are made to find cases that lead to undesirable behaviors and errors such as memory corruption or unauthorized access. One of the proposed methods for improving and enhancing fuzzing is the use of symbolic analysis and dynamic-symbolic execution. In this method, in addition to generating random data, logical analysis of the program and its symbolic execution are used to generate data that can cover new paths in program execution. In this research, we have shown that the dynamic-symbolic execution method can be used for fuzzing network protocols and also improve this process. For this purpose, the first framework for hybrid fuzzing of network protocols has been designed and implemented. The results on two services dcmtk and dnsmasq show that hybrid fuzzing performs better in terms of code coverage compared to traditional fuzzing. Branch coverage in the dcmtk service improved by 2.71 percent compared to AFLNet, which was able to make the negative performance of NyxNet compared to AFLNet positive. Also, branch coverage in the dnsmasq service improved by 37.72 percent compared to AFLNet and by 11.82 percent compared to NyxNet.

Keywords: Fuzz Testing, Network Protocol Testing, Vulnerabilities, Symbolic Execution, Concolic Execution.

* Corresponding Author's email: haghighi@sadjad.ac.ir

بهبود معیار پوشش کد برای کشف آسیب‌پذیری در پروتکل‌های شبکه دارای حالت توسط فازینگ ترکیبی

حمید رضایی رهورد^۱، مهدی سالخورده حقیقی^۲*

^۱ دانشکده مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه سجاد، مشهد، ایران

^۲ دانشکده مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه سجاد، مشهد، ایران

تاریخ دریافت: ۱۴۰۲/۰۷/۲۳ تاریخ بازبینی: ۱۴۰۲/۱۰/۱۵ تاریخ پذیرش: ۱۴۰۲/۱۱/۱۳

نوع مقاله: پژوهشی

چکیده

فازینگ نرم‌افزار، روشی برای یافتن آسیب‌پذیری‌های امنیتی در برنامه‌های کاربردی است. در این روش با ارسال داده‌های تصادفی به برنامه، سعی می‌شود مواردی پیدا شود که منجر به رفتارهای نامطلوب و خطاهایی همچون خرابی حافظه یا دسترسی‌های غیرمجاز شود. یکی از روش‌های پیشنهادی برای بهبود و اثربخشی فازینگ، استفاده از تحلیل نمادین و اجرای پویا-نمادین است. در این روش علاوه بر تولید داده‌های تصادفی، از تحلیل منطقی برنامه و اجرای نمادین آن برای تولید داده‌هایی استفاده می‌شود که بتوانند مسیرهای جدیدی از اجرای برنامه را پوشش دهند. در این پژوهش نشان داده‌ایم که می‌توان از روش اجرای پویا-نمادین برای فازینگ پروتکل‌های شبکه استفاده نمود و همچنین این فرایند را بهبود بخشید. بدین منظور اولین چارچوب برای فازینگ ترکیبی پروتکل‌های شبکه طراحی و پیاده‌سازی شده است. نتایج بر روی دو سرویس dcmtk و dnsmasq نشان می‌دهد که فازینگ ترکیبی در معیار پوشش کد نسبت به فازینگ سنتی عملکرد بهتری دارد. پوشش شاخه در سرویس dcmtk مقدار ۲,۷۱ درصد نسبت به AFLNet بهبود داشته است که توانسته عملکرد منفی NyxNet نسبت به AFLNet را مثبت نماید. همچنین پوشش شاخه در سرویس dnsmasq نسبت به AFLNet مقدار ۳۷,۷۲ درصد و نسبت به NyxNet مقدار ۱۱,۸۲ درصد بهبود داشته است.

کلیدواژه‌گان: آزمون فازینگ، آزمون پروتکل‌های شبکه، آسیب‌پذیری، اجرای نمادین، اجرای پویا-نمادین.

* رایانامه نویسنده مسؤول: haghghi@sadjad.ac.ir

۱- مقدمه

موجود است.

۲- مفاهیم پایه

۲-۱- تست فایزینگ

در فایزینگ، داده‌های غیرمنتظره و تصادفی و گاه ساختارمند، به برنامه‌ها ارسال می‌شود. این داده‌ها می‌توانند شامل مقادیر نامعتبر، خارج از محدوده و یا الگوهای غیرمنتظره‌ای باشند که هدف از ارسال آن‌ها، بررسی واکنش برنامه در برابر ورودی‌های غیراستاندارد است. اگر برنامه به درستی کار نکند به عنوان مثال خراب شود یا به اشتباه عمل کند، نشان می‌دهد مشکلی^۳ وجود دارد. فایزینگ برای پیدا کردن مشکلات امنیتی مانند اجرای کد دلخواه، مشکلات مرتبط با حافظه و انکار سرویس مفید است.

۲-۲- اجرای نمادین^۴

اجرای نمادین تلاش می‌کند به طور سیستماتیک با بررسی یک برنامه به صورت نمادین، مسیرهای مختلف اجرای برنامه را شناسایی کند. این روش به طور کلی با تغییر مقادیر شرطی، حالت‌های مختلف اجرا را بررسی می‌کند و برای هر حالت اجرا، مسیر مجزایی ایجاد می‌شود. سپس حل‌کننده‌های SMT^۵ برای تعیین امکان‌پذیری مسیرها و انتخاب راه‌حل‌ها استفاده می‌شوند. در صورت امکان پذیر بودن یک مسیر، خروجی SMT منجر به تولید ورودی‌های متفاوت خواهد شد. یکی از مهم‌ترین مشکلات روش اجرای نمادین، انفجار مسیرها^۶ است که منجر به کاهش کارایی الگوریتم برای استفاده روی برنامه‌های بزرگ و واقعی می‌شود. افزایش تصاعدی تعداد مسیرهای اجرا با افزایش شاخه‌ها^۷ در کد، پیچیدگی فضای جستجو را به شدت افزایش داده و مانع از به کارگیری این روش بر روی برنامه‌های پیچیده واقعی می‌گردد.

۲-۳- اجرای پویا-نمادین^۸

در اجرای پویا-نمادین، برنامه تحت آزمون به طور همزمان هم با داده‌های واقعی و هم با داده‌های نمادین اجرا می‌شود و سعی بر مرتفع نمودن مشکلات اجرای نمادین دارد. داده‌های واقعی برای اجرای مسیر خاصی از برنامه و داده‌های نمادین برای ثبت رفتار برنامه استفاده می‌شوند. مراحل اجرای روش پیشنهادی پویا-نمادین به شرح زیر است که بخشی از چگونگی پیاده‌سازی این پژوهش را

فایزینگ به عنوان یکی از مهمترین تکنیک‌های آزمون امنیت نرم‌افزار، نقش بسزایی در شناسایی آسیب‌پذیری‌های امنیتی دارد. با توجه به بررسی‌های انجام شده طی ده سال گذشته [1]، روش فایزینگ به طور قابل توجهی نسبت به روش‌های دیگر همچون تحلیل ایستا و یا استفاده از روش تحلیل نمادین به تنهایی، مورد توجه قرار گرفته است. همچنین ترکیب روش فایزینگ با سایر روش‌ها، خروجی‌های بسیار بهبودیافته و قابل توجهی داشته است که می‌توان گفت استفاده از چند تکنیک، ضعف‌های سایر تکنیک‌ها را پوشش می‌دهد.

از آنجا که پروتکل‌های شبکه پایه و اساس ارتباطات شبکه‌ای هستند، فایزینگ آنها از اهمیت ویژه‌ای برخوردار است. با این حال، فایزینگ پروتکل‌های شبکه به دلیل ماهیت پیچیده آنها، همواره یک چالش بزرگ محسوب می‌شده است. تاکنون پژوهش‌های اندکی در زمینه فایزینگ پروتکل‌های شبکه صورت گرفته و عمدتاً محدود به بهبود حفظ وضعیت پروتکل و نیز نحوه ایجاد داده تست به صورت ساختارمند بوده‌اند. از طرفی، تحقیقات انجام شده در زمینه فایزینگ ترکیبی که یکی از روش‌های افزایش معیار پوشش کد می‌باشد، بیشتر معطوف به بهبود فایزینگ تجزیه‌گر فایل^۱ بوده و کمتر به پروتکل‌های شبکه پرداخته‌اند. فایزینگ تجزیه‌گر فایل‌ها یکی از انواع فایزینگ است که در آن فایل‌های ورودی برنامه‌های کاربردی مورد هدف تست قرار می‌گیرند. این فایل‌ها معمولاً شامل فایل‌های پیکربندی، فایل‌های ورودی XML، فایل‌های ورودی کاربر و فایل‌های داده‌ای هستند. با توجه به اهمیت فایزینگ پروتکل‌های شبکه و کمبود تحقیقات در این حوزه، در این مقاله سعی شده است با بررسی راهکارهای موجود و نقاط ضعف آنها، یک معماری فایزینگ ترکیبی متناسب با پروتکل‌های شبکه دارای حالت ارائه شود و گامی در جهت بهبود فرآیند فایزینگ برداشته شود. مهمترین چالش‌های بررسی شده در فرآیند فایزینگ این پروتکل‌ها شامل حفظ وضعیت برنامه‌های چندنخی^۲ در تحلیل پویا-نمادین و همچنین تحلیل نمادین از نقطه دلخواه در طی فرآیند اجرا و پردازش بسته‌ها می‌باشد. همچنین برخی از راهکارها در جهت افزایش سرعت اجرا و همچنین کاهش بار پردازشی در بخش‌های مختلف خواهد بود. پس از معرفی معماری پیشنهادی، نتایج حاصل از پیاده‌سازی و ارزیابی آن ارائه می‌شود که بیانگر برتری روش پیشنهادی نسبت به فازرهای

⁵ Satisfiability Modulo Theories

⁶ Path Explosion

⁷ Branch

⁸ Concolic Execution

¹ File Parser

² Multi-thread

³ Bug

⁴ Symbolic Execution

نیز تشریح می‌نماید: نمادین آماده می‌کند. سوم، داده‌های آزمونی را که بین فازر و اجرای پویا-نمادین اجرا می‌شوند، انتخاب و فیلتر می‌کند.

۲-۵- پروتکل‌های شبکه

پروتکل‌های شبکه به دو دسته دارای حالت^۳ و بدون حالت^۴ تقسیم می‌شوند. در پروتکل‌های بدون حالت مانند HTTP، اطلاعات مربوط به حالت ارتباط ذخیره نمی‌شود و هر بسته به طور مستقل ارسال و پردازش می‌گردد. اما در پروتکل‌های دارای حالت مانند TLS، اطلاعاتی در مورد حالت اتصال ذخیره می‌شود که هر بسته می‌تواند به بسته‌های قبلی وابستگی داشته باشد و نحوه پردازش هر بسته ارسالی متفاوت باشد.

۳- بررسی مطالعات پیشین

بزرگترین چالش در فازینگ پروتکل‌های شبکه دارای حالت، سرعت اجرا و حفظ حالت براساس ماشین حالت پروتکل است. اخیراً راه‌حل‌های متنوعی مطرح شده‌اند که تا حد زیادی چالش‌های اولیه در فازینگ این نوع محصولات نرم‌افزاری را برطرف کرده‌اند. حال گام بعدی در تکامل و پیشرفت این حوزه استفاده از تکنیک‌های بهبوددهنده و مکمل مانند اجرای پویا-نمادین در طول فرایند آزمون فازینگ برای تولید داده‌های آزمون با کیفیت‌تر است.

این پژوهش، اولین پیاده‌سازی اجرای پویا-نمادین بر روی اهداف سرویس‌های شبکه دارای حالت، بر پایه تحقیقات و کارهای انجام شده تاکنون خواهد بود که دسته‌بندی آن‌ها بر روی سه موضوع اصلی زیر انجام شده است:

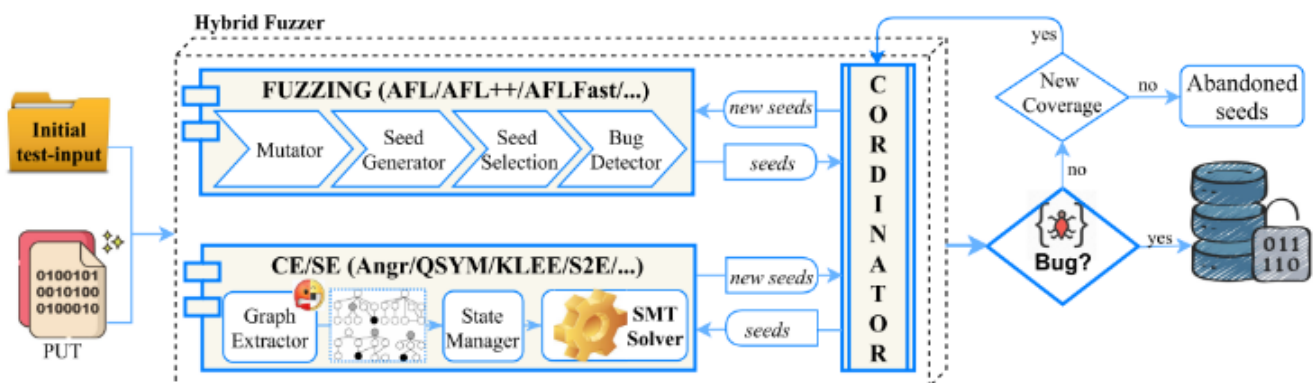
۱. یک داده ورودی واقعی برای بررسی مسیر اجرایی برنامه تحت آزمون انتخاب می‌شود.
۲. برنامه با این داده‌های ورودی واقعی اجرا می‌شود و همزمان داده‌ها و عملیات‌های مرتبط با داده ورودی به صورت نمادین ثبت می‌شوند.
۳. یک مدل نمادین از رفتار برنامه با استفاده از اطلاعات ثبت شده ساخته می‌شود.
۴. با استفاده از تحلیل نمادین، شرایطی که منجر به رسیدن به مسیرهای اجرایی جدید می‌شوند، استخراج می‌گردد.
۵. داده‌های ورودی واقعی جدیدی که این شرایط را برآورده می‌کنند، تولید می‌شوند.

۲-۴- فازینگ ترکیبی^۱

فازینگ ترکیبی با ترکیب دو تکنیک فازینگ و اجرای پویا-نمادین تلاش می‌کند تا پوشش کد^۲ و کارایی فرآیند فازینگ را افزایش دهد. شکل ۱ معماری یک سیستم آزمون فازینگ ترکیبی را نشان می‌دهد. این سیستم شامل سه بخش اصلی است:

۱. فازر
۲. موتور اجرای پویا-نمادین
۳. هماهنگ‌کننده

بخش هماهنگ‌کننده یک نرم‌افزار میانی است که تکنیک‌های فازینگ و اجرای پویا-نمادین را کنترل می‌کند و سه وظیفه اصلی دارد. ابتدا، فازر را نظارت می‌کند تا بتواند زمان اجرای پویا-نمادین را تشخیص دهد. دوم، محیط در حال اجرا را برای اجرای پویا-



شکل ۱. ساختار کلی یک فازر ترکیبی [3]

³ Statefull
⁴ Stateless

¹ Hybrid Fuzzing
² Code Coverage

۳-۱- فازینگ پروتکل‌های شبکه دارای حالت

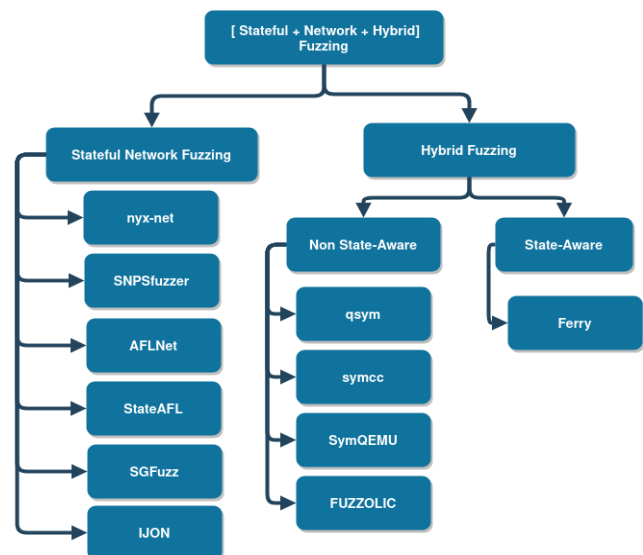
پس از موفقیت فازر AFL، برخی از محققین سعی کرده‌اند تا آن را برای سرویس‌های شبکه نیز به کار گیرند. بدین منظور سعی شده اهداف تحت آزمون را تغییر دهند تا به جای دریافت ورودی از سوکت شبکه، از فایل‌های ورودی دریافت کنند. اما این تغییرات ذات شبکه‌ای بودن اهداف را نقض کرده و برخی مشکلات را از دید فازر پنهان می‌کند. فازر AFLNet [5] سعی کرده است روش را معکوس کند؛ یعنی به جای تغییر اهداف تحت آزمون، خود فازر AFL را تغییر دهد تا متناسب با اهداف شبکه‌ای باشد. در بررسی کارایی طرح پیشنهادی این پژوهش AFLNet به عنوان State-of-the-art و پایه مقایسه‌ها در نظر گرفته می‌شود.

AFLNet از ترافیک ضبط‌شده بین سرویس‌دهنده و سرویس‌گیرنده به عنوان داده‌های اولیه استفاده می‌کند. به عنوان سرویس‌گیرنده عمل کرده و درخواست‌های جهش‌یافته را ارسال می‌کند. از کدهای پاسخ سرویس‌دهنده برای استنتاج ماشین حالت پروتکل و هدایت فازر استفاده می‌کند. در مقایسه با AFL برای پروتکل‌های پیچیده و دارای حالت موثرتر است. اما همچنان معایبی مانند نیاز به ترافیک اولیه، محدود بودن به پروتکل‌های خاص، عدم حفظ وضعیت و نیازمند اسکریپت cleanup و اجرای قبل از انجام هر تست، عملکرد ضعیف و همچنین نیاز به تجزیه‌گرهای سفارشی دارد.

فازر NYX [6] با استفاده از QEMU و KVM با ماشین مجازی یکپارچه شده و حالت آن را به یک پشتیبان‌گیری مقطعی^۱ تنظیم می‌کند. این فازر از یک عامل نماینده^۲ در ماشین مجازی برای کنترل چرخه فازینگ استفاده می‌کند. تکنیک پشتیبان‌گیری مقطعی در این فازر بر پایه Dirty-Page است که از امکانات سخت‌افزاری پردازنده‌های مدرن برای ردیابی صفحات تغییر یافته استفاده می‌کند. نسخه توسعه‌یافته این فازر به نام NyxNet [7] است که از قابلیت‌های آن برای فازینگ پروتکل‌های شبکه استفاده می‌کند. معماری این فازر در شکل ۳ قابل مشاهده است. پشتیبان‌گیری مقطعی افزایشی^۳ منجر می‌شود تا تنها صفحات تغییر یافته ذخیره و بازیابی شوند و سرعت را افزایش دهد. یکی دیگر از قابلیت‌های بسیار مفید فازر NyxNet هماهنگی کامل با زیرساخت و ساختار فازر AFL می‌باشد. همچنین اهدافی که برای این نوع فازر کامپایل شده‌اند، نیز به راحتی توسط NyxNet قابل استفاده خواهند بود و پیچیدگی و ابهاماتی در یادگیری و تسلط بر روی این فازر به

- فازینگ پروتکل‌های شبکه دارای حالت: فازرهای مرتبط لیست شده‌اند تا بهترین تحقیقات انجام شده به عنوان فازر پایه انتخاب شود.
- فازینگ ترکیبی بر روی اهداف دارای حالت: مقالاتی که فازینگ ترکیبی را بر روی تمام جریان پردازش و ارتباط بین حالت‌های تعریف شده درون برنامه انجام می‌دهند.
- فازینگ ترکیبی بر روی اهداف بدون حالت: مقالاتی که فازرهای ترکیبی از آن‌ها استفاده کرده‌اند و یا به عنوان بخشی از فازرهای ترکیبی می‌باشند که بر روی اهداف بدون حالت و یا با در نظر نگرفتن حالت‌های گذشته عمل می‌کنند.

با بهره‌گیری از دانش و تجربیات به دست آمده از تحقیقات پیشین، هدف انتخاب درست تکنیک‌ها و ابزارهای مورد استفاده در این پژوهش است. دسته‌بندی مقالات مطالعه شده در شکل ۲ قابل مشاهده است.



شکل ۲. دسته‌بندی کارهای پیشین

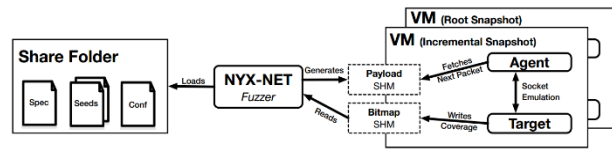
در بخش فازینگ ترکیبی آگاه به حالت بر روی اهداف دارای حالت تنها یک مقاله [4] وجود دارد که تحقیقات آن بر روی تجزیه‌گر فایل‌ها انجام شده است که نشان‌دهنده جدید بودن این موضوع و عدم تحقیقات کافی در این زمینه است. برای پیاده‌سازی فازینگ ترکیبی آگاه به حالت بر روی پروتکل‌های شبکه دارای حالت، ابتدا باید زیرساخت لازم برای فازینگ ترکیبی فراهم شود. بنابراین پیاده‌سازی فازینگ ترکیبی آگاه به حالت بر روی اهدافی که دارای حالت هستند، خارج از حوزه تحقیقاتی این پژوهش می‌باشد.

³ Incremental Snapshot

¹ Snapshot

² Agent

حداقل‌ترین حالت خود خواهد رسید.



شکل ۳. معماری NyxNet [7]

فازر SNPSFuzzer [8] راهکاری مبتنی بر پشتیبان‌گیری مقطعی برای تسریع فازینگ پروتکل‌های شبکه پیشنهاد می‌کند. این فازر با ذخیره و بازیابی اطلاعات مربوط به هر حالت، از تکرار ارسال پیشوندهای طولانی جلوگیری می‌کند. همچنین با استفاده از الگوریتم‌های تحلیل نقطه مقطع و زنجیره پیام، پوشش حالت‌ها را افزایش می‌دهد. اما پیچیدگی پیاده‌سازی، نیاز به تنظیم الگوریتم‌ها و کندی نسبی از محدودیت‌های این روش به شمار می‌روند. این فازر برای مدیریت پشتیبان‌گیری‌های مقطعی از CRIU استفاده می‌کند. کارایی CRIU را می‌توان با قابلیت مشابه در Qemu مقایسه کرد که جنبه‌های عملکردی مهم در فازینگ مانند کارایی و سرعت در آن مورد توجه نبوده است.

فازر StateAFL [9] یک فازر جعبه خاکستری برای سرورهای شبکه‌ای است که بدون نیاز به سفارشی‌سازی دستی، صرفاً با تجزیه و تحلیل ساده برنامه تحت آزمون کار می‌کند. این فازر کدهایی را درون برنامه هدف تزریق می‌کند تا اطلاعات حافظه و شبکه جمع‌آوری کند و پس از هر تعامل درخواست-پاسخ، یک پشتیبان‌گیری مقطعی از حافظه بلندمدت گرفته می‌شود. سپس هر پشتیبان‌گیری به یک شناسه حالت یکتا نگاشت می‌شود. این روش نیازی به تجزیه‌گرهای سفارشی پروتکل ندارد. نتایج نشان می‌دهد StateAFL بدون سفارشی‌سازی، می‌تواند پوشش و خرابی‌های قابل مقایسه و حتی بهتر از فازرهای سفارشی قبلی به دست آورد. اما محدودیت‌هایی مانند نیاز به دسترسی به کد منبع و افزایش زمان اجرا وجود دارد.

مقاله SGFUZZ [10]، روشی نوین بر پایه Libfuzzer برای فازینگ برنامه‌های دارای حالت همچون پروتکل‌های ارتباطی ارائه می‌دهد. راه حل پیشنهادی، شناسایی متغیرهای حالت برنامه با استفاده از الگویابی و پیگیری مقادیر آنها حین اجرای برنامه است.

الگوی استفاده شده بر این فرض استوار است که اغلب حالت‌های برنامه به صورت شمارشی ذخیره^۱ و در دستورات شرطی بررسی می‌شوند. از مزایای این روش می‌توان به کشف خودکار فضای حالتی

و عدم نیاز به دانش قبلی در مورد حالت‌های برنامه اشاره کرد. اما معایبی همچون کاهش سرعت اجرا، خطا در شناسایی متغیرها و عدم پوشش فضای حالت‌های ضمنی مانند تغییرات پایگاه داده‌ها، برای این روش وجود دارد. همچنین برای برنامه‌های با فضای حالتی بسیار بزرگ، کارایی کمتری خواهد داشت.

مقاله IJON [11] رویکرد پیشنهادی با الهام از مدل Human-in-the-loop ارائه می‌نماید که امکان دخالت کاربر در خروجی یک فرایند یا رویداد را فراهم می‌سازد. نتایج تحقیق نشان می‌دهد یک تحلیلگر انسانی می‌تواند فازر را در فضای حالتی برنامه هدایت کرده و بخش‌هایی از فضای حالتی را که نیاز به بررسی کامل دارند، مشخص نماید. با استفاده از قابلیت نشانه‌گذاری، می‌توان اطلاعات بیشتری در مورد وضعیت متغیرها و حالت‌های برنامه در اختیار فازر قرار داد که منجر به افزایش پوشش کد و تولید داده‌های آزمون باکیفیت‌تر می‌شود. اما مهم‌ترین محدودیت این روش، نیاز به نیروی انسانی متخصص و آشنا با کد منبع هدف است که نیازمند صرف زمان زیادی برای آشنایی با پیچیدگی‌های برنامه و معماری پیاده‌سازی آن است.

۳-۲- فازینگ ترکیبی

ابزار Angr [12] یک موتور اجرای نمادین کلاسیک است که از زبان میانی VEX استفاده می‌کند که باعث کاهش سرعت آن می‌شود، اما این کار باعث ساده شدن پیاده‌سازی آن شده است. معماری آن در شکل ۴ قابل مشاهده است. با توجه به اینکه Angr بر پایه VEX است، بنابراین می‌تواند تمام معماری‌هایی که VEX پشتیبانی می‌کند را به ارث برده و پشتیبانی نماید. از آنجایی که کد Angr در پایتون نوشته شده است، سرعت آن نسبتاً کند است.

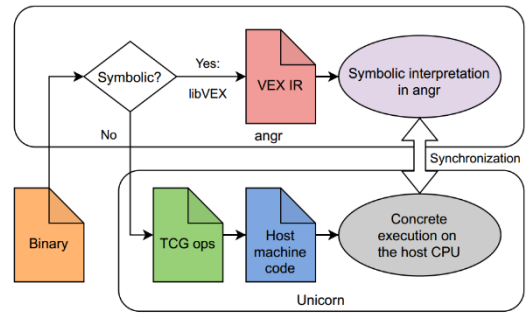
ابزار S2E [13] یک سیستم‌عامل کامل را درون QEMU شبیه‌سازی می‌کند و آن را به موتور اجرای نمادین KLEE متصل می‌کند. این کار باعث می‌شود S2E بتواند تجزیه و تحلیل جامعی از کد در تمامی لایه‌های سیستم‌عامل ارائه کند. اما پیچیدگی زیاد آن باعث شده استفاده از آن دشوار باشد.

در دو راهکار S2E و Angr به عنوان state-of-the-art، استفاده از یک زبان میانی در جهت کمتر شدن پیچیدگی پیاده‌سازی انتخاب شده است اما با کم شدن هزینه پیاده‌سازی، عملکرد و کارایی به شدت کاهش یافته است.

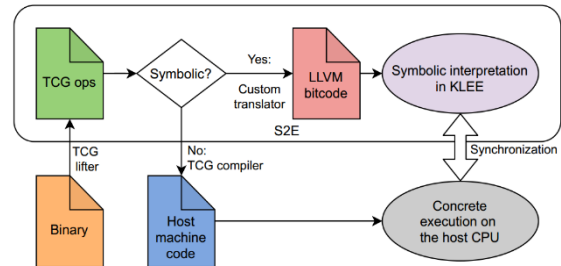
^۱ Enum

فرآیند کامپایل منجر به شکست گردد و همچنین دسترسی به کد منبع اهداف تحت آزمون به عنوان یک الزام مطرح می‌شود.

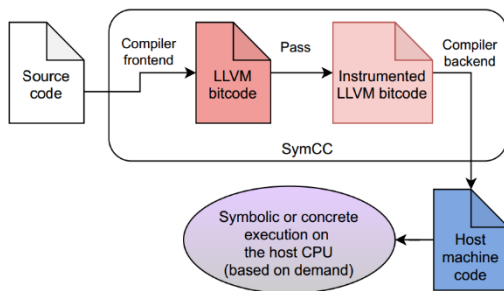
ابزار SymQEMU [16] تلاش نموده است تا تمامی مزایا موجود در تحقیقات انجام شده تاکنون را در کنار هم قرار دهد. از سرعت و کارایی symcc و همچنین از Qemu و TCG که زبان میانی آن است استفاده نموده تا قابلیت پشتیبانی معماری مختلف را بدست آورد. این ابزار به عنوان شبیه‌سازی سطح کاربر می‌باشد و عملکردی مشابه به Angr و SymCC/QSym خواهد داشت. همچنین راه حلی را پیشنهاد می‌دهد تا بتوانند ایده اصلی SymCC که برای مراحل و فرآیند کامپایل کد منبع بوده است را بر روی باینری اعمال کنند. ابزار SymQemu فقط بر روی معماری x86 پیاده‌سازی شده است. همچنین این پیاده‌سازی بر روی فایزینگ فایل تمرکز داشته است. نمی‌توان بصورت سفارشی از آن برای تجزیه و تحلیل در سطح توابع استفاده کرد و یا به صورت سفارشی شده بخش‌های تعیین شده‌ای در برنامه تحت آزمون مورد بررسی و اجرای نمادین قرار گیرد.



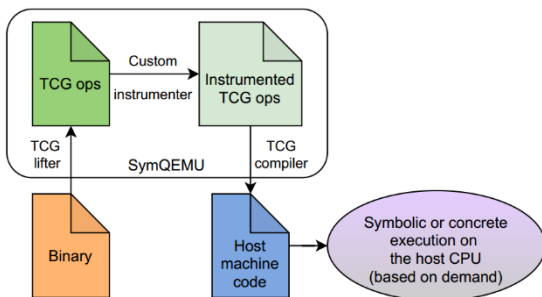
شکل ۴. معماری Angr [16]



شکل ۵. معماری S2E [16]

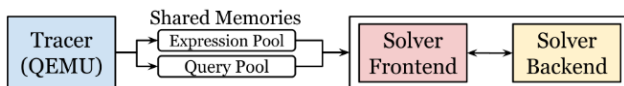


شکل ۷. معماری SymCC [16]



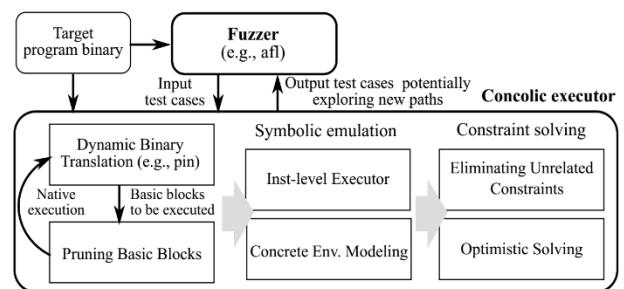
شکل ۸. معماری SymQemu [16]

ابزار Fuzzolic [17] همانند SymQEMU از زبان میانی TCG استفاده می‌کند، اما با جدا کردن بخش حل‌کننده، توانسته است عملکرد بهتری داشته باشد. Fuzzolic سه حالت مختلف تجزیه و تحلیل پویا دارد که باعث بهبود دقت آن شده است.



شکل ۹. معماری Fuzzolic [17]

ابزار QSym [14] بدون استفاده از زبان میانی و با Instrument کردن فایل باینری برنامه تحت آزمون در زمان اجرا، توانسته سرعت زیادی کسب کند. اما این روش محدود به معماری x86 است و پیاده‌سازی آن برای سایر معماری‌ها بسیار دشوار خواهد بود. ذات QSym برای فایزینگ فایل می‌باشد و هنگام تجزیه و تحلیل یک فایل ۲۱۸ بایتی عکس png در برنامه libpng برای اجرای تمام شاخه‌ها به بیش از دو ساعت زمان نیاز دارد. بزرگترین عیب این روش عدم سفارشی‌سازی و محدود نمودن شاخه‌های مورد بررسی می‌باشد.



شکل ۶. معماری QSym [14]

ابزار SymCC [15] به عنوان یک لفاف بر روی کامپایلر برپایه QSym ایجاد شده است که با افزودن کد مورد نیاز هنگام کامپایل، توانسته است سرعت اجرای خود را تا هزار برابر نسبت به سایر راهکارها افزایش دهد. اما نیازمند یک کامپایلر خاص است و قابلیت کار با باینری‌های موجود را ندارد. براساس مشاهدات تجربی ممکن است این کامپایلر خاص در برخی اهداف قابل استفاده نباشد و

چالش ۳، برخی ابزارهای بررسی شده بخش 3 در سطح باینری کار کرده و فرایند اجرا را در کتابخانه‌هایی مانند glibc نیز درگیر می‌کنند. این موضوع می‌تواند بر سرعت تجزیه و تحلیل و اجرای نمادین تأثیر منفی بگذارد. استفاده از توابع نمادین^۱ می‌تواند با کاهش حجم محاسبات، سرعت تحلیل و اجرای این ابزارها را افزایش دهد.

با توجه به معایب مطرح شده در بخش ۳ و همچنین سه چالش اصلی قبلاً ذکر شده در این بخش، در این پژوهش سعی بر آن است تا راهکاری جامع و منعطف برای اهداف سرویس‌های شبکه و پروتکل‌های دارای حالت ارائه شود.

به منظور مقابله با انفجار مسیر، بهینه‌سازی‌ها و روش‌هایی مانند محدود کردن عمق تجزیه و تحلیل، کاهش شرایط پیچیده و استفاده از ترکیب‌های تحلیل نمادین و آزمون‌های واقعی می‌توانند مورد استفاده قرار گیرند که به عبارت دیگر اجرای Concrete به نوعی مکمل تحلیل نمادین است به توان عیوب آن را مرتفع و از فواید آن استفاده نماییم. بدین منظور در این پژوهش با استفاده از آزمون‌های واقعی به همراه تحلیل نمادین که به آن تحلیل پویا-نمادین می‌گوییم، سعی شده است تا بتوان با استفاده از مزایای هر دو روش بهبود در عملکرد ایجاد داده ای آزمون جدید برای فازر NyxNet انجام شود.

بسیاری از پروتکل‌ها شامل مفاهیم سرویس دهنده-سرویس گیرنده هستند که در آن سرویس گیرنده اتصال را آغاز می‌کند و سرویس دهنده پس از درخواست سرویس گیرنده پاسخ می‌دهد. در مرحله آزمون سرویس دهنده، عملکرد به نسبت آسان است. نرم‌افزار هدف در انتظار دریافت ارتباطات ورودی قرار دارد و فازر تنها نیاز دارد که به سرویس دهنده هدف متصل شود و داده‌های آزمون را ارسال کند. اما آزمون سرویس گیرنده به طور معمول پیچیده‌تر است. در این حالت، فازر باید مانند سرویس دهنده عمل کند و منتظر درخواست‌های ورودی از سوی سرویس گیرنده باشد. هر زمان که سرویس گیرنده هدف به فازر متصل می‌شود، فازر با ارسال یک داده آزمون به درخواست سرویس گیرنده پاسخ می‌دهد. به طور معمول، نیاز است که در سرویس گیرنده تغییراتی ایجاد شود تا به صورت مکرر به فازر متصل شود و بتواند دریافت داده‌های آزمون را ادامه دهد. چنین مواردی با ایجاد تغییر در چند خط کد منبع سرویس گیرنده با استفاده از وصله‌ها^۲ مرتفع شده است.

با توجه به اهداف تحت آزمون، اعداد تصادفی برای مواردی مانند

با توجه به پیاده‌سازی مشابه Fuzzolic و SymQemu هر دو دارای معایب مشترکی می‌باشند.

اخیراً در کاربردهای مختلفی این نوع آزمون‌ها مورد استفاده بوده‌اند از جمله مرجع [۲۰]. همچنین ابزارها و روش‌های مختلفی بر مبنای آن توسعه یافته‌اند [۲۱] و [۲۲].

همچنین، روش‌هایی نیز برای توسعه نرم‌افزار مبتنی بر آزمون معرفی شده‌اند که روش پیشنهادی در این تحقیق نیز می‌تواند در آنها مفید باشد [۲۳].

۴- طرح پیشنهادی و پیاده‌سازی

با توجه به بررسی‌های انجام شده در بخش 3.1، ابزار NyxNet به عنوان بهترین گزینه برای استفاده به عنوان فازر پایه در جهت توسعه و بهبود ابزار پیشنهادی انتخاب می‌شود. این ابزار علاوه بر سرعت بالای اجرا و حفظ حالت در پروتکل‌های شبکه‌ای حالت‌دار، از برنامه‌های تحت تست باینری و نیز دارای کد منبع باز می‌باشد. علاوه بر معایب هر کدام از روش‌های مورد بررسی در بخش ۳، در ادامه ۳ چالش مطرح می‌شود.

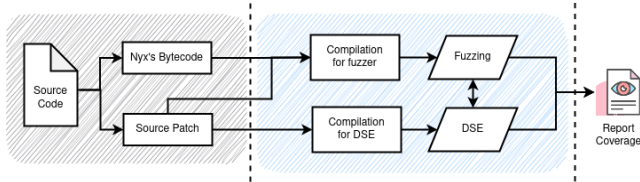
چالش ۱، همان‌طور که در بخش 3.2 بررسی شد، روش‌های موجود اجرای برنامه‌ها و توابع چندنخی را بصورت نمادین پشتیبانی نمی‌کنند. علت این محدودیت این است که اهداف این روش‌ها، تجزیه گره‌های فایل بوده و به طور ضمنی تنها برنامه‌های تک‌نخی را پشتیبانی می‌کنند. در حالی که اکثر برنامه‌ها و سرویس‌های شبکه‌ای به صورت چندنخی پیاده‌سازی می‌شوند تا بتوانند با سرعت بیشتری به درخواست‌ها پاسخ دهند. این محدودیت باعث می‌شود نتوان از نتایج کارهای گذشته در حوزه فازینگ ترکیبی برای اهداف شبکه‌ای چندنخی استفاده کرد. عدم پشتیبانی برنامه‌های چندنخی یکی از مهم‌ترین نقاط ضعف مشترک تمام این روش‌ها به‌شمار می‌رود.

چالش ۲، برنامه‌ها و سرویس‌های شبکه‌ای در طول اجرای خود و پردازش داده‌های یک ارتباط، چندین بسته دریافت کرده و هر بسته بر حالت برنامه تأثیر می‌گذارد. بنابراین در فازینگ ترکیبی نیاز است تا بتوان بخش‌های مختلف برنامه تحت تست را به صورت سفارشی و نمادین اجرا و بررسی کرد. به عنوان مثال، اگر داده تست شامل هفت بسته باشد، بخواهیم فقط روند اجرای بسته پنجم را به صورت نمادین بررسی کنیم. این امکان توسط روش‌های گذشته فراهم نبوده است.

² Patch

¹ Semantic Functions

سرویس^۲، اما بایت کدهای مختص به NYX و همچنین مفسرهای این بایت کدها باید توسط کاربر فراهم شده باشد تا بتوانند پکت‌های دریافت شده در طی فایزینگ را تفسیر و به داخل برنامه تزریق نمایند. برای فرآیند فایزینگ دو بخش بایت کدها و وصله‌ها نیاز است اما برای فرآیند اجرای پویا-نمادین تنها وصله‌ها لازم خواهد بود تا تغییرات لازمه را در فرآیند اجرای برنامه ایجاد نماید.



شکل ۱۰. طرح کلی پیاده‌سازی

اهداف تحت آزمون آماده شده برای اجرای فایزینگ در هنگام کامپایل با استفاده از کامپایلر پیش‌فرض afl-clang-fast توسط Sanitizerها تغییر خواهند کرد تا به عنوان Oracle بتوانند مشکلات و خطاهای حافظه را تشخیص دهند. در صورت نیاز می‌توان پارامترهای کامپایلر را تغییر داد و Sanitizerهای دیگری را به عنوان Oracle انتخاب نمود تا انواع دیگری از باگ‌ها و آسیب‌پذیری‌ها را با استفاده از فایزینگ کشف نماییم. اما با توجه به اینکه Sanitizerها کدهایی را به درون برنامه تزریق می‌نمایند، در جهت عدم پردازش این کدها در تحلیل نمادین نیاز خواهیم داشت تا اهداف تحت آزمون را بدون چنین ویژگی‌هایی کامپایل نماییم. بنابراین دو روش کامپایل خواهیم داشت و فایل باینری مورد استفاده توسط فازر و تحلیل نمادین متفاوت خواهند بود. در ادامه دو فرآیند فایزینگ و تحلیل پویا-نمادین همزمان اجرا شده و داده‌های خود را با یکدیگر به اشتراک خواهند گذاشت.

در ادامه شبه کد و منطق پیاده‌سازی انجام شده آورده شده است:

```

def coordinator():
    set_corpuses_notify()
    list_corpuses()
    while True:
        py_script = reproduce_corpus(
            get_one_corpus()
        )
        packets = load_packets(py_script)
        new_seeds = explore(packets)
        feed_to_nyxnet(new_seeds)

def explore():
    run_CE()
    handle_instructions()
    return solve_path_constraints()
    
```

رمزنگاری نقش بسیار مهمی را ایفا می‌کند. گاهی تولید اعداد تصادفی می‌تواند برنامه تحت آزمون را در وضعیتی نامعین قرار دهد و همچنین خطاهای برنامه با استفاده از داده‌های آزمون، می‌بایست قابل بازتولید باشند. بسیاری از محصولات نرم‌افزاری که بر روی بستر شبکه کار می‌کنند، با توجه به تأخیرات در انتقال داده‌ها از توابعی مانند sleep(), usleep() استفاده می‌کنند تا وقفه‌های مصنوعی ایجاد کنند. اما این وقفه‌ها می‌توانند عملکرد فازر را به شدت کاهش دهند. برای به حداقل رساندن تاثیر وقفه‌ها و تأخیرها، همچنین مواردی اعداد تصادفی که منجر به ایجاد مسیرهای متفاوت در هر بار اجرای برنامه تحت آزمون می‌شود، سعی شده است که از قابلیت‌های DBI^۱ به طور کامل بهره‌برداری شود تا تمامی فرآیندها و تغییرات مورد نیاز به صورت خودکار انجام شود.

ابزارهای DBI به وسیله ترجمه باینری در زمان اجرا، توانایی اعمال تغییرات در اجرای یک برنامه را داراست. به عنوان مثال، این قابلیت را دارد که بدون دسترسی به کد منبع، کدهای نظارتی را در نقاط مختلف درون باینری تزریق کند تا اطلاعاتی در مورد جریان اجرا جمع‌آوری کند. همچنین قادر است خطاها را شناسایی و اصلاح نماید یا فرآیند اجرا را با تغییرات مورد نیاز سازگار کند. با این حال، یکی از مشکلات اصلی این روش، سربار اجرایی بالای آن است. زیرا ترجمه باینری در زمان اجرا هزینه‌های زیادی دارد و باعث کندی در اجرای برنامه می‌شود. به همین دلیل، برای مواردی مانند آزمون فایزینگ که نیاز به اجرای سریع و مکرر برنامه دارند، این روش چندان مناسب نمی‌باشد. خوشبختانه، Dynamorio [18] با استفاده از مکانیزم‌های مختلفی که پیاده‌سازی نموده است، توانسته است سربار اجرایی را به حداقل ممکن، به حدی که تنها ۱/۱ برابر سرعت اجرای اصلی باشد، کاهش دهد.

طرح کلی و تغییراتی که این پژوهش در فرآیند فایزینگ NyxNet ایجاد خواهد کرد در شکل ۱۰ قابل مشاهده است. با توجه به شکل ۱۰ برای کامپایل و آماده‌سازی اهداف آزمون دو روش کامپایل نیاز است.

استفاده از Patch اجباری نمی‌باشد و می‌تواند هدف آزمون بدون تغییرات در کد منبع مناسب برای فایزینگ باشد. گاهی نیاز است که برخی تغییرات بسیار کمی در کد منبع ایجاد شود تا وضعیت پایداری در فرآیند اجرا داشته باشیم. برخی از این تغییرات گاهی توسط خود اهداف آزمون به صورت پارامترهای اجرایی در اختیار کاربر قرار می‌گیرند، به عنوان مثال اجرا نشدن برنامه تحت آزمون به صورت

² Daemon

¹ Dynamic Binary Instrumentation

عبارت دیگر، پکت‌ها توسط SE انجام خواهد شد و در صورت نیاز به دریافت پکت، CE می‌تواند آن را از طریق حافظه مشترک از SE درخواست و دریافت نماید.

در پیاده‌سازی فعلی تحلیل نمادین تنها بر روی آخرین پکت انجام می‌شود. در صورت دریافت آخرین بسته، حافظه مورد استفاده بصورت Taint شده خواهد بود. بدین ترتیب چالش ۲ مطرح شده در این بخش مرتفع خواهد شد و در صورت نیاز می‌توان برای هر بسته دریافتی شرایط و پردازش خاصی را مدنظر داشته باشیم که این عمل می‌تواند برای داشتن زیرساخت تحلیل نمادین آگاه به حالت^۳ بسیار کارآمد باشد.

در تابع `process_instruction` بررسی می‌شود اگر فراخوانی تابعی بر روی یکی از توابع نمادین شده به عنوان مثال `stcmp` باشد به جای پردازش دستورالعمل‌ها توابع نمادین اجرا خواهد شد و همچنین بخش CE ارسال دستورالعمل‌ها را تا خاتمه یافتن تابع نمادین شده برای SE متوقف خواهد کرد. بدین ترتیب چالش ۳ مطرح شده در این بخش مرتفع می‌گردد و بار پردازشی در تحلیل نمادین کاهش و سرعت اجرای CE افزایش خواهد یافت.

پس از پردازش کامل آخرین پکت توسط برنامه تحت آزمون، عملیات تحلیل نمادین و حل نمودن `Path Constraints` انجام می‌شود تا با حل نمودن محدودیت‌های موجود در طی مسیر پردازش آخرین پکت، داده‌های آزمون جدیدی ایجاد شود و به فرآیند فزینگ تزریق گردد تا بتوان پوشش کد را بهبود بخشید.

کدمنبع بخش `Coordinator` و همچنین SE به زبان پایتون نوشته شده است. اما بخش ارسال داده آزمون جدید به `NyxNet` به زبان `Rust` برنامه‌نویسی شده است تا بتوان براساس ساختارهای تعریف شده در این فازر داده‌های جدید را به فرآیند فزینگ تزریق نماییم. همچنین بخش CE به زبان `C/C++` نوشته شده است.

یکی از نکات بسیار مهم که در طول تحقیقات در نظر گرفته شده، این است که اهداف آزمون پروتکل‌های شبکه وضعیت دار هستند. بنابراین، نمی‌توان به همان روشی که در آزمون‌های رایج برای تجزیه گره‌های فایل‌ها استفاده می‌شود، اقدام به درون‌ریزی داده از ابتدا به داخل برنامه کرد و سپس تحلیل را انجام داد. بهمین دلیل با استفاده از قابلیت‌های DBI توابعی مهم که مرتبط با ارتباطات شبکه می‌باشند را `Hook` کرده و متناسب با نیاز تغییرات در فرآیند اجرا در آن ایجاد می‌شود. به عنوان مثال هنگام خواندن اطلاعات از

```
def handle_instructions():
    done = False
    while(not done):
        action = read_shm()
        if action == get_packet:
            packet = get_next_packet()
            if packet.is_last_one():
                taint memory(packet)
                send(packet)
            elif action == set_reg_mem:
                change_reg_mem_value()
            elif action == instruction:
                done = process_instruction()
```

بخش `Coordinator` وظیفه جمع‌آوری فایل‌های `Corpus` ایجاد شده توسط `NyxNet` را بر عهده دارد. سپس این فایل‌ها را بر اساس اندازه آنها مرتب می‌کند، به گونه‌ای که فایل‌های کوچکتر در اولویت بررسی قرار می‌گیرند. این اقدام به منظور بهینه‌سازی مصرف حافظه در فرآیند تحلیل نمادین انجام می‌شود. وظیفه دیگر این بخش دریافت اطلاعات پکت‌ها است تا بتواند آنها را به عنوان ورودی به پروسه اجرای پویا-نمادین ارسال کند.

در بخش SE^۱ همان تابع `explore`، ابتدا پروسه CE^۲ اجرا می‌شود و سپس شروع به ارسال/دریافت داده می‌کند. بخش CE با استفاده از ابزار `DynamoRio` که یک `DBI` می‌باشد، پیاده‌سازی شده است. CE وظیفه اجرا برنامه تحت آزمون و تغییرات لازم در طول فرآیند اجرا مانند کاهش تاثیر وقفه‌ها و اعداد تصادفی را اعمال می‌نماید تا اجرا آسان و سریع شود و به علاوه، همچنان بتواند همگرایی و یکپارچگی برنامه تحت آزمون که یک برنامه شبکه‌ای است، را حفظ کند.

ارتباط میان CE/SE با استفاده از حافظه مشترک `Shared Memory` برقرار خواهد بود و برای رفع مشکلات و تداخلات همزمانی در ارسال و دریافت داده‌ها و همچنین پشتیبانی از برنامه‌های تحت آزمون چندنخی، از `Semaphore` و `Mutex` در پیاده‌سازی CE/SE استفاده شده است. در طی پردازش دستورالعمل‌های دریافت شده توسط SE عمل `Context-Switching` برای برنامه‌های چندنخی با استفاده از اطلاعات دریافت شده از CE انجام خواهد شد، بنابراین چالش ۱ مطرح شده در این بخش بدین صورت مرتفع می‌گردد. اگر پردازش دستورالعمل‌های دریافت شده به داده خاصی از حافظه و یا برخی از رجیسترهای خاص مانند `XMM` و `YMM` نیاز داشته باشد، می‌تواند آن را از CE درخواست و دریافت نماید. همچنین مدیریت ورودی‌ها یا به

³ Satte-Aware

¹ Symbolic Executor

² Concrete Executor

همانطور که اعداد جدول ۱ برای هدف آزمون dcmk نشان می‌دهند، عملکرد فازر NyxNet در برابر AFLNet نسبتاً ضعیف بوده است. دلیل این امر می‌تواند پیچیدگی ساختار برنامه تحت آزمون و همچنین وجود شرایط خاص ورودی در dcmk باشد که منجر به کاهش کارایی فازر شده است و به دلیل تصادفی بودن ایجاد داده آزمون‌ها شرایط قابل پاس شدن نبوده‌اند. با این حال، فایزینگ ترکیبی توانسته است این عملکرد نسبتاً ضعیف را بهبود بخشد و درصد تفاوت در پوشش را مثبت کند. بنابراین می‌توان نتیجه گرفت که فایزینگ ترکیبی تأثیر مثبت خود را در طی فرایند ایفا نموده است.

اما برای هدف آزمون dnsmasq، عملکرد و بهبود هر دو فازر نسبت به AFLNet کاملاً مشهود است. فایزینگ ترکیبی همچنان موجب افزایش پوشش در این هدف آزمون شده است و این بهبود نزدیک به ۳۸ درصد می‌باشد.

شاخص بعدی ارزیابی پوشش کد می‌باشد که بر مبنای تعداد خط کد ارزیابی انجام می‌شود. این مقادیر با ابزارهایی مانند gcov قابل بدست آوردن است. جدول ۲ مقادیر عددی برای این شاخص ارزیابی را نشان می‌دهد.

جدول ۲. مقایسه عددی پوشش خط کد

NyxNet-Hybrid		NyxNet-Balanced		AFLNet		
4526	+1.95	4407	-0.7	4439		dcmk
1753	+28.89	1606	+18	1361		dnsmasq

با توجه به شاخص پوشش شاخه که یکی از معیارهای مهم در ارزیابی کیفیت فرایند آزمون نرم‌افزار می‌باشد، از نظر مثبت و منفی بودن خروجی، مفهوم کلی به دست آمده از جدول ۲ و جدول ۱ شبیه به یکدیگر است، اما از نظر عددی مقادیر دو جدول تفاوت‌هایی دارند. همچنان عملکرد نسبتاً ضعیف فازرها در هدف آزمون dcmk با وجود پیچیدگی‌های خاص این برنامه قابل مشاهده است. با این وجود، فایزینگ ترکیبی توانسته است تعداد ۸۷ خط کد بیشتری را در طی زمان ۳۰ دقیقه پوشش دهد که خود نشان‌دهنده بهبود نسبی در فرایند آزمون فایزینگ ترکیبی می‌باشد. اما این اعداد در هدف آزمون dnsmasq به مراتب بیشتر بوده و فایزینگ ترکیبی توانسته است تا نزدیک به ۴۰۰ خط کد بیشتری را نسبت به سایر فازرها پوشش دهد در شکل ۱۱ و ۱۲ نیز می‌توان نمودار پوشش خط کد هر دو سرویس تحت آزمون را مشاهده کرد.

Socket آن را به عنوان دریافت کننده ورودی تشخیص داده و درخواست دریافت پکت را به SE ارسال خواهد کرد بدین ترتیب اطلاعات با سرعت بیشتری نسبت به سوکت دریافت خواهد شد و بهبود در سرعت اجرا خواهیم داشت.

همچنین در جهت پیشگیری از احتمال انفجار مسیرها ورودی‌های با سایز کوچکتر ابتدا پردازش خواهند شد و همچنین بازه زمانی حداکثر دو دقیقه برای پردازش SE بر روی هر داده تست در نظر گرفته شده است تا در صورت پردازش ورودی‌هایی با سایز بزرگتر مشکلی در فرآیندها ایجاد نگردد.

۵- آزمایش‌ها و ارزیابی نتایج

شاخص ارزیابی و عملکرد میان ابزارها و تکنیک‌های فایزینگ میزان افزایش پوشش کد در اهداف تحت تست می‌باشد. در این پژوهش نیز به منظور حفظ اصول و همچنین ارزیابی منصفانه در برابر تحقیقات انجام شده، شاخص بهبود بر پایه مقدار پوشش کد در نظر گرفته شده است. تمامی بهبودها در فایزینگ سعی بر آن دارند تا به پوشش کد صد درصد دست یابند و هر بهبود کمک در جهت رسیدن به این هدف می‌باشد. بررسی و ارزیابی‌ها بر روی یک دستگاه با معماری ۱۲ هسته اینتل Intel Xeon(R) با فعال بودن hyper-threading (مجموعاً ۲۴ هسته منطقی) و ۳۲ گیگابایت حافظه RAM در حال اجرای بر روی سیستم عامل Ubuntu 22.04 LTS x64 انجام شده است. برای اطمینان و قابلیت استناد به خروجی‌های ارزیابی، اجراها به صورت ۲۰ مرتبه با مدت زمان ۳۰ دقیقه انجام شدند. این تعداد اجراها و زمان مورد نیاز برای هر بار اجرا، به ما امکان می‌دهد از داده‌های قابل اعتماد و قابل استناد برای تحلیل و ارزیابی استفاده کنیم.

داده‌های مرتبط با پیاده‌سازی این پژوهش با نام NyxNet-Hybrid در مقایسه‌ها و تحلیل‌ها قابل مشاهده است. شاخص بهبود عملکرد NyxNet-Hybrid در مقایسه با فازرهای AFLNet و NyxNet می‌باشد. فازر NyxNet سه روش در گرفتن پشتیبان‌گیری مقطعی را ارائه می‌دهد که با توجه به ارزیابی‌های درون مقاله بهترین عملکرد آن بصورت Balanced بوده است و از همین رو این مکانیزم را برای مقایسه انتخاب نموده ایم و همچنین پیاده‌سازی این تحقیق نیز با Nyx-Balanced اجرا شده است.

جدول ۱. مقایسه عددی پوشش شاخه

NyxNet-Hybrid		NyxNet-Balanced		AFLNet		
2948	+2.71	2869	-0.03	2870		dcmk
1139	+37.72	1042	+25.9	827		dnsmasq

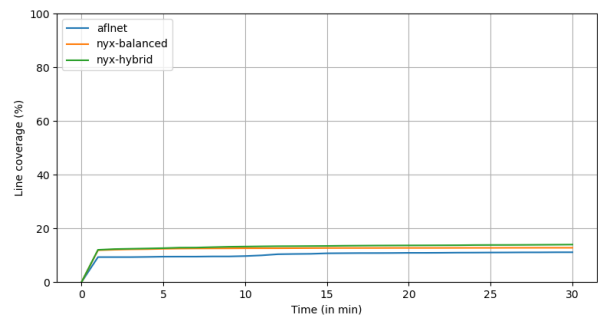
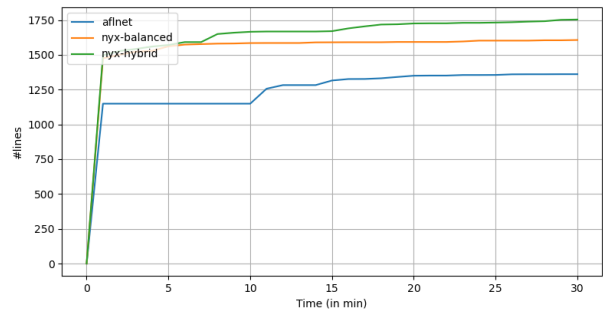
برنامه پیچیده با ویژگی‌های خاص خود می‌باشد، احتمالاً پیاده‌سازی فازینگ ترکیبی و اجرای نمادین برای آن دشوارتر بوده است. اما در آزمون‌های انجام شده روی برنامه dnsmasq شاهد بهبود قابل توجهی در معیارهای ارزیابی بودیم که نشان دهنده تاثیر مثبت روش پیشنهادی می‌باشد. بنابراین می‌توان نتیجه گرفت که میزان بهبود حاصل از فازینگ ترکیبی بستگی به ویژگی‌ها و پیچیدگی برنامه مورد نظر دارد.

علیرغم چالش‌های موجود در پیاده‌سازی این روش، با توجه به نتایج به دست آمده، به نظر می‌رسد از لحاظ صرفه جویی در زمان نتایج رضایت بخشی حاصل شده است. همچنین با بهینه‌سازی بیشتر در پیاده‌سازی، امکان ارتقای نتایج وجود دارد. در مجموع این پژوهش نشان داد که فازینگ ترکیبی می‌تواند گزینه مناسبی برای بهبود فرآیند فازینگ پروتکل‌های شبکه باشد و مسیر مناسبی را برای پژوهش‌های آتی در این حوزه هموار نماید.

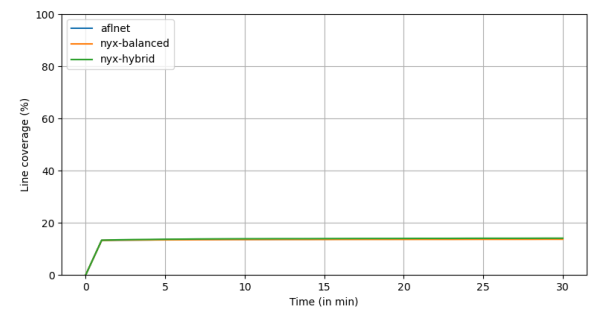
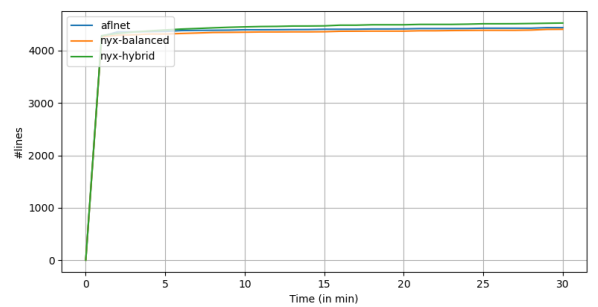
۶- چالش‌های فرارو و پیشنهادها

در این پژوهش سعی شده است تا آخرین و جدیدترین مقالات مرتبط با فازینگ ترکیبی پروتکل‌های شبکه مطالعه شود و بهترین راهکارهای موجود نیز مورد بررسی قرار گیرند. از آنجایی که تاکنون چنین پیاده‌سازی جامعی صورت نگرفته است، لذا باید بهترین روش از میان راهکارهای موجود انتخاب شود تا بتوان از آن برای رسیدن به اهداف مورد نظر بهره برد. البته پیاده‌سازی‌های موجود تمرکز بیشتری بر روی فرآیند فازینگ فایل‌ها داشته‌اند و کمتر با دید شبکه‌ای صورت گرفته‌اند. همان‌طور که در ارزیابی‌های انجام شده مشخص است، آنها در کنار فاکتورهای مربوط به فازینگ فایل، تنها بر اساس همان معیارها ارزیابی و پیش‌رفته‌اند. با این حال، کدمنع تمامی آنها بررسی و در برخی موارد اصلاح شد تا امکان‌سنجی استفاده از آنها در این پژوهش فراهم شود.

اما این پژوهش تلاش کرده است مسیر جدیدی را طی کند و با تمرکز بر پروتکل‌های شبکه، پیاده‌سازی اختصاصی متناسب با نیازهای فازینگ شبکه انجام دهد. دو ابزار DynamoRIO و Triton که بهترین در حوزه کاری خود هستند، مورد استفاده قرار گرفتند تا بهترین عملکرد و خروجی حاصل شود که خوشبختانه چنین هم شد. پس از پیاده‌سازی، ارزیابی‌ها نشان می‌دهند که انتخاب زیرساخت مناسب و همچنین تکنیک فازینگ ترکیبی، همچنان فرآیند فازینگ پروتکل‌های شبکه را بهبود می‌بخشد. بنابراین می‌توان از این تکنیک برای فازینگ پروتکل‌های شبکه بهره برد.



شکل ۱۱. پوشش خط کد سرویس dnsmasq



شکل ۱۲. پوشش خط کد سرویس dcmtk

با توجه به نتایج حاصل از ارزیابی‌های انجام شده، می‌توان نتیجه گرفت که روش پیشنهادی فازینگ ترکیبی با استفاده از ابزارهای Triton و DynamoRIO [19] توانسته است منجر به بهبود فرآیند فازینگ شود. البته این بهبود در تمامی موارد یکسان نبوده و بستگی به شرایط و ویژگی‌های برنامه مورد آزمایش داشته است. در آزمون‌های انجام شده روی برنامه dcmtk، اگرچه شاهد بهبود چشمگیری در معیارهای ارزیابی مانند پوشش کد و پوشش شاخه نبوده ایم، اما با این حال نتایج حاکی از آن است که فازینگ ترکیبی توانسته به نحوی شرایط را بهبود بخشد. از آنجایی که dcmtk یک

- [9] R. Natella, "Statefl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, 2022.
- [10] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in 31st USENIX Security Symposium (USENIX Security 22), pp. 3255–3272, 2022.
- [11] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in 2020 IEEE Symposium on Security and Privacy (SP), pp. 1597–1612, IEEE, 2020.
- [12] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in IEEE Symposium on Security and Privacy, 2016.
- [13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multipath analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
- [14] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in 27th USENIX Security Symposium (USENIX Security 18), pp. 745–761, 2018.
- [15] S. Poeplau and A. Francillon, "Symbolic execution with {SymCC}: Don't interpret, compile!" in 29th USENIX Security Symposium (USENIX Security 20), pp. 181–198, 2020.
- [16] S. Poeplau and A. Francillon, "Symqemu: Compilation-based symbolic execution for binaries," in NDSS 2021, Network and Distributed System Security Symposium, Internet Society, 2021.
- [17] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: Mixing fuzzing and concolic execution," *Computers & Security*, vol. 108, p. 102368, 2021.
- [18] D. Bruening and T. Garnett, "Building dynamic instrumentation tools with dynamorio," in Proc. Int. Conf. IEEE/ACM Code Generation and Optimization (CGO), Shen Zhen, China, 2013.
- [19] F. Soudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, pp. 31–54, 2015.
- [20] Leonardo Xompero, Tahir Mohammad, Jouni Isoaho, Jürgen Grossi, "Fuzz Testing Approach for Embedded Avionic Software", The 15th International Conference on Ambient Systems, Networks and Technologies (ANT), 2024.
- [21] Chi Zang et al. "TR-Fuzz: A syntax valid tool for fuzzing C compilers", *Science of Computer Programming*, vol. 238, 2024.
- [22] Haorn Xu et al. "Fuzzing JavaScript engines with a syntax-aware neural program model", *Computers & Security*, vol. 144, 2024.
- [۲۳] زهره مافی، سیدحسن میریان حسین‌آبادی. "انتخاب موارد آزمون مبتنی بر روش تولید آزمون رانه"، دو فصلنامه فناوری اطلاعات و ارتباطات ایران، دوره ۱۶ شماره ۵۹، ۱۴۰۳.
- از جمله چالش‌ها و محدودیت‌های پیش‌روی فازینگ ترکیبی پروتکل‌های شبکه می‌توان به موارد زیر اشاره کرد:
- پیاده‌سازی توابع بیشتر به صورت نمادین در SE که می‌تواند دقت بیشتر و همچنین سربار محاسباتی کمتری را به همراه داشته باشد.
 - استفاده از تکنیک‌های یادگیری ماشین برای اولویت‌بندی و انتخاب مناسب‌ترین داده‌های ورودی و پرهیز از انفجار مسیرها در بخش SE
 - به نظر می‌رسد با پیگیری این موارد در تحقیقات آتی، بتوان چالش‌های فازینگ ترکیبی پروتکل‌های شبکه را به طور مؤثری مدیریت کرد.

مراجع

- [1] Cui, Lei, Jiancong Cui, Zhiyu Hao, Lun Li, Zhenquan Ding, and Yongji Liu. "An empirical study of vulnerability discovery methods over the past ten years." *Computers & Security*, 2022.
- [2] M. Zalewski, "American fuzzy lop - a security-oriented fuzzer".
- [3] F. Rustamov, J. Kim, J. Yu, and J. Yun, "Exploratory review of hybrid fuzzing for automated vulnerability detection," *IEEE Access*, vol. 9, pp. 131166–131190, 2021.
- [4] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, "Ferry: {StateAware} symbolic execution for exploring {State-Dependent} program paths," in 31st USENIX Security Symposium (USENIX Security 22), pp. 4365–4382, 2022.
- [5] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 460–465, IEEE, 2020.
- [6] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in 30th USENIX Security Symposium (USENIX Security 21), pp. 2597–2614, 2021.
- [7] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in Proceedings of the Seventeenth European Conference on Computer Systems, pp. 166–180, 2022.
- [8] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2673–2687, 2022.