

# آزمایشگاه اینترنت اشیا

استاد: دکتر علیرضا اجللی

اعضا: محسن قاسمی، محمدرضا بدری، امیرحسین نقدعلی

پروژه دوم: عملیات fuzzing روی UEFI با استفاده از پلتفرم Simics و ابزار TSFSS

## گزارش نهایی

2	بخش تئوری
2	خلاصه‌ی مقاله‌ی UEFI Firmware Fuzzing with Simics Virtual Platform
4	خلاصه‌ی مقاله RSFuzzer
6	خلاصه‌ی مقاله‌ی «FUZZUER: Enabling Fuzzing of UEFI Interfaces on EDK-2»
13	معیار مقایسه‌ی ابزارهای fuzzing
15	تعریف Coverage
15	تحقیق درباره‌ی پیاده‌سازی گراف فراخوانی و CFG در FIRNESS
17	بخش عملی
17	نصب و راه‌اندازی ابزار شبیه‌سازی TSFSS
19	فاز کردن پروتکل‌های تکراری
19	پروتکل USB_IO
21	پروتکل SimpleNet
22	فاز کردن پروتکل‌های تازه
22	پروتکل DNS4
23	پروتکل BlockIO
24	پروتکل Credential (S3BootScriptLib)
24	پروتکل I2C_Master (HII Database)
25	جمع‌بندی

## بخش تئوری

در این بخش ابتدا خلاصه‌ی مقاله‌های بررسی‌شده ارائه می‌شود، سپس به بررسی برخی نکات تکمیلی پیرامون این مقالات و موضوع آن‌ها پرداخته می‌شود.

### خلاصه‌ی مقاله‌ی UEFI Firmware Fuzzing with Simics Virtual Platform

#### معرفی:

در ابتدای مقاله به اهمیت امنیت و تست برنامه‌های کامپیوتری، وظایفی که بر عهده‌ی firmwareهاست اشاره می‌شود. همچنین برای نشان دادن اهمیت UEFI بیان می‌شود که بر روی ۴ میلیارد دستگاه موجودند. یک اشتباهی که بعضاً در مورد این بخش دارند این است که با اجرای سیستم عامل کار آن به پایان می‌رسد ولی این مقاله به عنوان نمونه بیان می‌کند که در برخی سیستم‌های بر مبنای پردازنده‌های اینتل، سیستم عامل می‌تواند با ایجاد وقفه‌ی System Management Interrupt و System Management Mode (بالاترین سطح دسترسی ممکن که عملاً همه چیز بدون محدودیت در دسترس است) شود و مدیریت‌کننده‌های چنین وقفه‌هایی بر روی BIOS نصب هستند و برای اجرا باید به آن مراجعه کرد. سپس اشاره‌ای به روش‌های تست می‌کند و بیان می‌کند که چه مشکلاتی نسبت به تست سیستم عامل‌ها و نرم‌افزارهای کاربر در تست UEFI وجود دارد (برای مثال بحث حافظه) و در پاسخ به این مشکل Virtual Platform را معرفی می‌کند. سپس بیان می‌شود که چرا fuzzing از بین روش‌های تست روش مناسبی برای کاربرد مورد نظر ما است و دیگر روش‌ها به چه علت‌هایی ممکن است خوب کار نکنند (مانند وجود دستورات privileged و عملیات IO). در نهایت به روند کلی fuzzing و کارهایی که با استفاده از آن برای تست موارد مربوط به سیستم عامل‌ها استفاده شده است اشاره می‌کند.

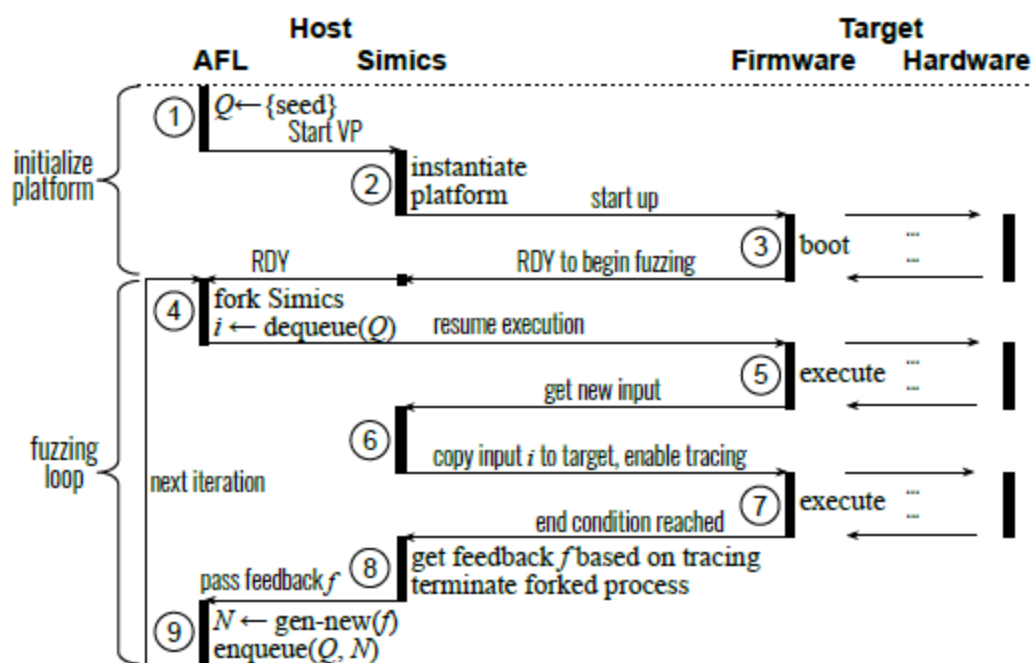
#### پیش‌زمینه:

ابتدا به توضیح امنیت UEFI و اینکه چرا مهم است و در معرض چه تهدیدهایی است پرداخته می‌شود. سپس دلایل اهمیت استفاده از یک محیط مجازی، تأثیری که در روند تست می‌توانند داشته باشند، و ابزارهایی که ممکن است در اختیار قرار دهند بیان می‌شود. در این مقاله از محیط مجازی Simics استفاده می‌شود. در نهایت به مفهوم fuzzing و انواع fuzzerها (generation-based و mutation-based) اشاره می‌شود. از دو نمونه fuzzer (AFL و LibFuzzer) نیز نام برده می‌شود ولی بیان می‌شود که برای نرم‌افزارهای سطح کاربر مناسب هستند و برای کار مورد نظر ما مناسب نیستند.

## فریمورک fuzzing:

در این بخش به ساختار این fuzzing پرداخته می‌شود و برخی پرسش‌ها مطرح می‌شود: چگونه استفاده از ورودی تولیدشده توسط fuzzer (توسط خود VP، در اینجا Simics، یا به درخواست خود firmware)، چگونه برگشت به حالت ابتدایی برنامه پس از انجام تست برای تست‌های بعدی (فورک کردن کل پردازشی Simics)، چگونه جمع‌آوری فیدبک (Simics را به این منظور پیکربندی می‌کنیم) و چگونه نظارت برای تشخیص رفتار غیرمنتظره از سیستم (Simics را به این منظور پیکربندی می‌کنیم). سپس بیان می‌شود که چرا در این کاربرد استفاده از fuzzer همانگونه که در دیگر کاربردها استفاده می‌شود ممکن نیست و به دو مورد محدودیت تغییر در در کد برای بررسی کد و محدودیت در انجام fork به دلیل استفاده از VP.

در نهایت جزئیات بیشتر پیاده‌سازی بیان می‌شود که تصویر آن در زیر آمده است. گام ۱، ۲ و ۳ به ترتیب گام آماده‌سازی برای شروع به کار fuzzer و VP و firmware است. در گام بعدی fuzzer پردازشی Simics فرزند خود را fork می‌کند تا بعداً بتوانیم به این حالت برگردیم و ورودی تست را آماده می‌کند. در دو گام بعدی firmware اجرای خود ادامه می‌دهد تا به ورودی نیاز پیدا کند و آن را از VP بگیرد. VP نیز روند اجرا را تحت نظر می‌گیرد. در گام ۷ firmware اجرای خود را ادامه می‌دهد تا به نقطه‌ای پایان از پیش مشخص شده برسیم. در گام بعدی VP اطلاعات جمع‌آوری‌شده را پردازش می‌کند و به عنوان فیدبک به fuzzer برمی‌گرداند و کار پردازش خاتمه می‌یابد. در گام آخر fuzzer (در اینجا AFL) بر حسب اطلاعات به دست‌آمده ورودی را تغییر می‌دهد و به گام ۴ برمی‌گردد. این فرایند تا زمان توقف آن توسط کاربر و یا تمام شدن محدودیت زمانی مشخص‌شده برای آن ادامه می‌یابد.



در ادامه نیز نکاتی درباره‌ی مصرف حافظه، کارایی و نحوه‌ی ارتباط اجزا با یکدیگر بیان می‌شود.

## نتایج:

در این بخش عملکرد این سیستم در سه سناریو ارزیابی می‌شود: تست مدیریت‌کننده‌های وقفه‌های System Management، تست ارتباط firmware با IO در حالتی که خود firmware درخواست ورودی کند و تست ارتباط firmware با IO در حالتی که VP با زیر نظر داشتن firmware ورودی را، پیش از رسیدن firmware به آن نقطه از کد، در محل مناسب قرار می‌دهد. در مورد نخست بیان می‌شود که ارتباط SMI با فراخواننده‌ی خود از طریق بافر است (آدرس و سائز آن را به عنوان ورودی دریافت می‌کند). در این تست با دادن ورودی‌های مختلف و بررسی دسترسی‌های به حافظه، دسترسی به خارج از فضای حافظه‌ی مورد نظر کشف شد. در مورد‌های بعدی گفته می‌شود که فرضی که در هنگام نوشتن firmware وجود دارد، کارکرد درست سخت‌افزار است و با بررسی موردی از USB IO حالتی کشف می‌شود که امکان کم بودن حافظه‌ی تخصیص داده شده می‌تواند باعث سرریز بافر، دسترسی غیرمجاز و... شود.

## جمع‌بندی:

در این مقاله ابتدا به مفاهیم ابتدایی و چالش‌های تست firmware پرداخته شد و fuzzing به عنوان یک گزینه‌ی مناسب برای این موضوع معرفی شد. سپس مدلی برای انجام fuzzing با استفاده از موتور AFL fuzzer و پلتفرم Simics ارائه شد. در نهایت نیز با بررسی چند سناریو کارایی این روش نشان داده شد.

## خلاصه‌ی مقاله RSFuzzer

در پردازنده‌های x86 قابلیت‌ی توسط UEFI به نام System Management Mode اضافه شده است که به کمک اینتراپت‌های SMI قابلیت دسترسی به داده‌های غیرقابل دسترس از جاهای دیگر یا دسترسی به قطعات سخت افزاری را فراهم می‌سازد، از اینرو وجود نقاط ضعف در SMI‌ها می‌تواند منجر به خرابی‌های بزرگی در دستگاه شود. برای برطرف کردن این مشکلات میتوان از روش‌های fuzzing استفاده کرد. در این راستا روش‌های مختلفی شکل گرفته مانند فازر Excite شرکت اینتل اما به دلایل زیر محدود هستند:

- تنها از طریق ورودی‌های معمول
- نبود اطلاع از متغیرهای داخلی هندلرها که باعث می‌شود نتوان به طور کامل نقاط ضعف را گشت و مشکلات را برطرف کرد.

این مقاله با ارائه یک فازر جدید با فرمت گریپاکس سعی میکند تا این مشکلات را برطرف کند.

## علت اهمیت:

در UEFI دو محیط برای اجرای پروسه ها شامل مود نرمال و مود SMM که مکانیزم SSM جدا سازی کامل این دو محیط است. برای اینکار قسمتی از رم به نام SMRAM جدا می شود که برای هر پدازه دیگری از دسترس خارج است، از این سو با امنیت این فضا حتی در صورت از دسترس خارج شدن محیط دیگر می توان امنیت را تضمین کرد اما با خدشه دار شدن امنیت این بخش سیستم دیگر قابل استفاده نیست، از این رو تضمین امنیت UEFI امری بسیار مهم است.

### پیاده سازی:

انجام عملیات fuzzing روی handlerها به علت دایورس بودن ورودی ها بهتر است، همچنین هندلرها تنها بخش های SMM هستند که از کرنل ورودی می گیرند. فازر اینتل به علت تنها بررسی ورودی های بین کرنل و SMM تک بعدی به شمار می رود و پیام های داخلی را بررسی نمی کند. فازر ارائه شده شامل دو بخش موتور فازر و موتور اجرایی می شود. هربار فازر یک سید جدید تولید می کند، این سید توسط موتور اجرایی روی یک هندلر انجام می شود و دو خروجی دریافت می شود: ۱. نوع دریافت ورودی و ۲. فرمت هر ورودی که با این اطلاعات فازر می تواند ورودی های معتبر برای هندلر ایجاد کند. با این حال معتبر بودن ورودی ها برای این کار کافی نیست و بعضی ورودی ها میان هندلرها جابه جا میشوند که می تواند منجر به ایجاد مشکل شود. در اینجا موتور فازر از دو روش single handler و cross handler استفاده می کند و با سوییچ کردن بین این دو مود برای هندلرهای مختلف می تواند اطلاعات مناسب و ورودی های مناسب را ایجاد کند.

### جزئیات:

یک SMM driver یک پدازه در محیط SMM هست که شامل سه بخش protocols برای برقراری ارتباط با دیگر درایورها و handler برای ورودی گرفتن از سمت کرنل و یوزر که به کمک حافظه CommonBuffer کار می کند و در نهایت initialization function برای شروع پدازه در هنگام بوت که بعد از بوت همه در SMRAM ذخیره می شوند.

دسترسی Ring-2 Privilege برای SMM Driver ها موجود است که اجازه دسترسی مستقیم با آدرس به مموری را فراهم می سازد. هنگام کال شدن یک پدازه SMM دیگر پدازه ها suspend می شوند و وضعیت پردازنده در SMRAM ذخیره می شود.

### فرض فرد متخاصم:

فرض می شود که فرد متخاصم دسترسی سطح 0 دارد، یعنی به طور مستقیم نمی تواند به SMRAM دسترسی داشته باشد و به کمک فازر مشکلات در این سطح مورد بررسی قرار می گیرد.

### چالش ها:

بعضی هندلرها ورودی های خود را از چند بخش مختلف دریافت می کنند، برای مثال برخی ممکن است از بخشی هاردکود شده ورودی را بخوانند و ... پیدا کردن فرمت مناسب برای ورودی که هندلر بتواند آنرا بخواند.

بعضی نقاط ضعف نیاز به اجرا شدن چندین هندلر دارند و فقط با اجرای یک پردازش انجام نمی‌شوند. بعضی نقاط ضعف با path constraints محافظت شده‌اند. بعضی نقاط ضعف منجر به کرش هندلر نمی‌شوند، در واقع silent corruption هستند. پس فازر باید شامل قابلیت‌های زیر باشد:

1. بتواند استراکچر و اینترفیس ورودی‌ها را شناسایی کند.
2. بتواند ارتباط میان هندلرهای مختلف را شناسایی کند.
3. بتواند path constraints را شناسایی کند.
4. قابلیت شناسایی silent corruptions.

## خلاصه‌ی مقاله‌ی «FUZZUER: Enabling Fuzzing of UEFI Interfaces on EDK-2»

### معرفی:

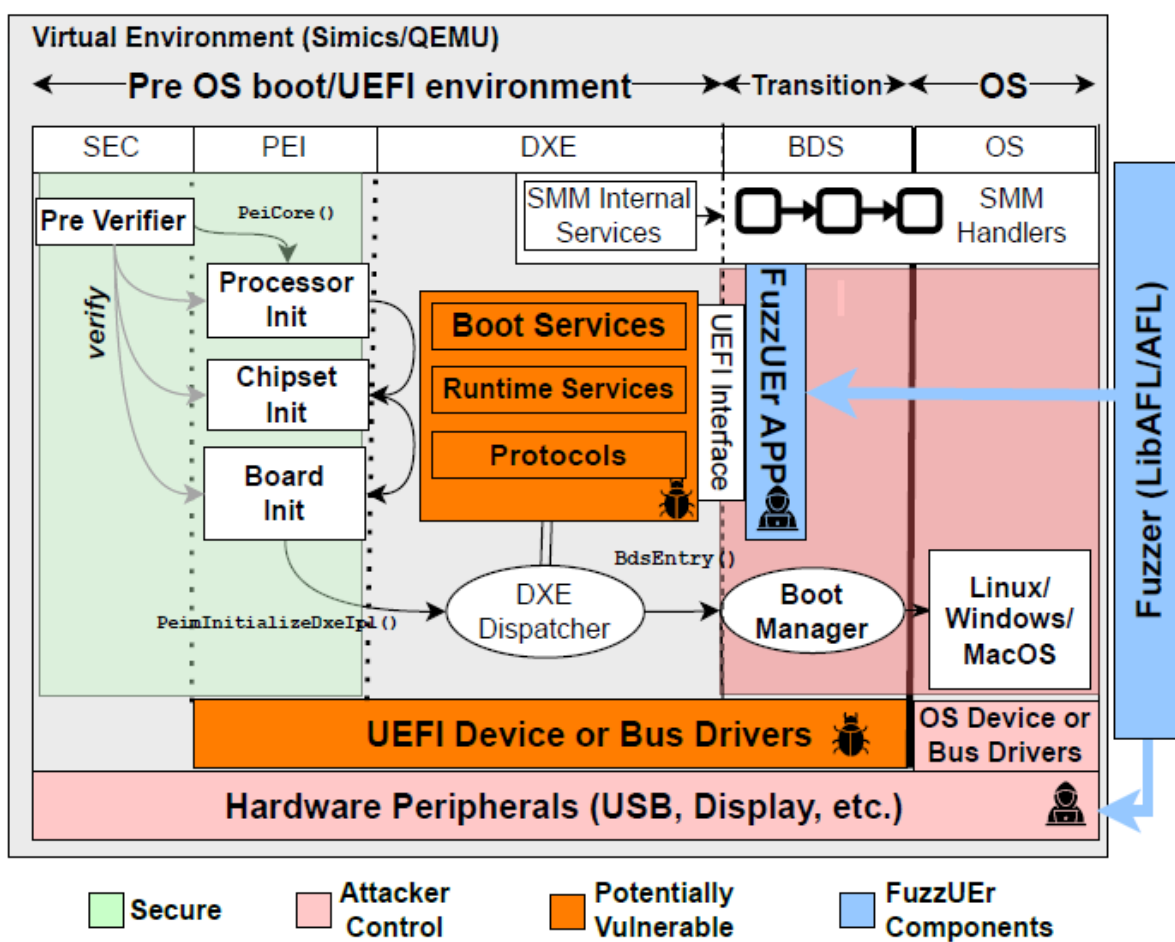
در این مقاله یک سیستم برای فاز کردن UEFI (به طور خاص EDK-2) ارائه شده است که از پلتفرم مجازی Simics و فازر LibAFL استفاده می‌کند. همچنین یک fuzzing harness، به نام FIRNESS، برای ساخت خودکار ورودی‌های مناسب (با بررسی کدهای موجود) برای تست تابع‌های مد نظر نوشته شده است. در نهایت کل این سیستم همراه با Address Sanitizer، پلتفرم FUZZUER را تشکیل می‌دهد. همچنین کد این سیستم به صورت متن‌باز در دسترس است و کارایی آن نیز با بررسی چندین تابع و مقایسه‌ی عملکرد آن با دیگر سیستم‌های موجود نشان داده شده است.

در ابتدا به دلایل اهمیت UEFI و EDK-2 و امنیت آن‌ها پرداخته می‌شود. همچنین مختصراً به کارهای مشابه دیگر، از جمله UEFI Firmware Fuzzing with Simics Virtual Platform که آن را پیش از این بررسی کردیم، اشاره می‌شود و تفاوت‌های آن‌ها با کار کنونی بیان می‌شود. برای مثال گفته می‌شود که تمرکز خیلی از کارهای انجام‌شده در این حوزه بر روی مدیریت‌کننده‌های SMI است و یا کدهای مقاله‌ی ذکرشده متن‌باز نیستند. گفته می‌شود که یکی از کارهای مهمی که سعی در انجام آن داشته‌اند خودکار کردن خیلی از این فرایندها در تست چنین سیستم‌هایی است.

### پیش‌زمینه:

در ابتدا به اهمیت و هدف UEFI و EDK-2 پرداخته می‌شود. سپس به فازهای مختلف UEFI پرداخته می‌شود: Security (SEC), Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), Boot Device Selection (BDS). فاز SEC مسئول بررسی امضای دیجیتال firmware و راه‌اندازی برخی ادوات ابتدایی سخت‌افزاری است. فاز PEI پس از SEC مسئول راه‌اندازی بخش‌های بیشتری از سخت‌افزار است. فاز DXE فاز اصلی راه‌اندازی سیستم است. در این فاز دیگر driverهایی که برای boot شدن یا کارکرد سیستم لازم است

راه‌اندازی می‌شوند. در نهایت فاز BDS سیستم عاملی برای راه‌اندازی را مشخص می‌کند. تمرکز اصلی در این کار بر روی فاز DXE است. رابط‌های این فاز به دو بخش سرویس‌ها و پروتکل‌ها تقسیم می‌شوند. سرویس‌ها واسطه‌هایی هستند که برای کارکرد ابتدایی و درست برای تمامی محیط‌های UEFI پیاده‌سازی شود (برای مثال AllocatePages). پروتکل‌ها به توسعه‌دهندگان اجازه‌ی تغییر یا ایجاد اجزای جدید را برای اضافه کردن کارکردی جدید به UEFI می‌دهند. دسترسی به پروتکل‌ها نیاز به GUID (Globally Unique Identifier) دارد. در مقاله بیان می‌شود که طبق بررسی انجام‌شده ۷۱٪ آسیب‌پذیری‌ها در پروتکل‌ها رخ می‌دهد که ۴۹٪ از آن‌ها مربوط به خرابی حافظه می‌شود. در این کار تمرکز اصلی بر فاز کردن واسطه‌های پروتکل‌ها است. در زیر تصویری کلی از فازهای مختلف بوت شدن سیستم مشاهده می‌شود.



مدل:

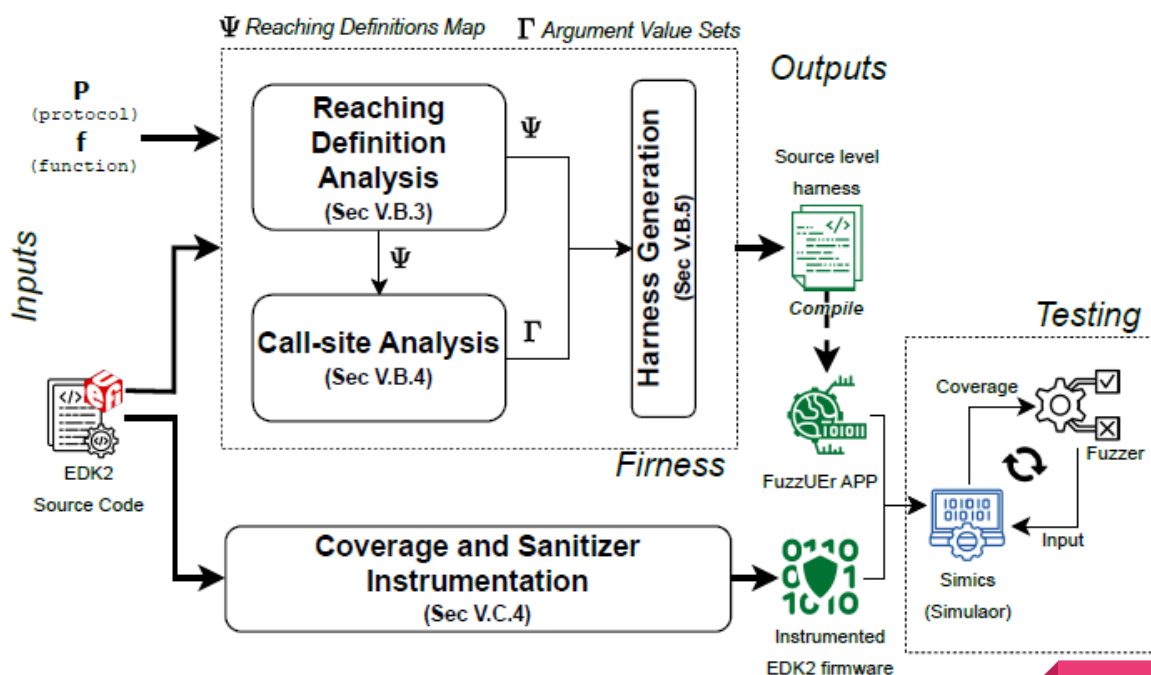
همان‌طور که در تصویر پیشین قابل مشاهده است، فرض شده است که فازهای SEC و PEI ایمن هستند و به تست فاز DXE می‌پردازیم. همچنین در این تصویر بخش‌هایی که قصد داریم آن‌ها را بررسی کنیم و جاهایی که ورودی به برنامه تزریق می‌کنیم، مشخص است.

### چالش‌ها:

برای پروتکلی خاص (GUID مشخص) و تابعی خاص از آن (یا توابعی از آن) قصد داریم عملیات فاز کردن را انجام دهیم که برخی چالش‌ها در این مسیر وجود دارد. ما نیاز داریم که برای این تابع مشخص ورودی‌های مناسب آن را تولید کنیم و (۱) نیاز است تا نوع داده‌ی ورودی هر پارامتر را تشخیص دهیم و (۲) تولید مقادیری از این نوع داده‌ها. این موارد بدیهی نیست زیرا ممکن است در تعریف تابع از نوع داده‌ی جنریک (\* void) استفاده شده باشد و یا برای تولید مقدار از نوعی خاص نیاز به فراخواندن تابعی باشد که خود ممکن است از پروتکلی دیگر باشد. دیگر چالش موجود وابستگی UEFI به سخت‌افزار که جمع‌آوری داده از آن برای ارائه‌ی فیدبک در حین فاز کردن را مشکل می‌کند.

### ساختار فریم‌ورک:

این سیستم سه بخش اصلی دارد: FIRNESS، تغییر (instrumentation) کد، و تست. این سه بخش در تصویر زیر مشخص است (به ترتیب بالاچپ، پایین چپ، و پایین راست). FIRNESS با انجام بررسی‌هایی بر روی کد پروتکلی خاص (تمامی توابع آن یا تابعی خاص از آن که می‌خواهیم تست کنیم)، harness تولید می‌کند که با تولید ورودی‌های مناسب تابع را فراخوانی می‌کند. در نهایت این کد C تولیدشده به UEFI اپلیکیشنی کامپایل می‌شود و در کنار FUZZUER قرار می‌گیرد. به این ترتیب ورودی‌های تصادفی تولیدشده توسط fuzzer با نوع مناسب به تابع مد نظر داده می‌شوند و تابع برای انجام تست فراخوانی می‌شود. بخش instrumentation نیز





تغییرات لازم برای بررسی پوشش تست (coverage) و خطاهای حافظه‌ای (sanitization) را در EDK-2 به وجود می‌آورد. در نهایت در بخش این برنامه‌ی FUZZUER بر روی این کد تغییر داده شده در محیط مجازی اجرا می‌شود.

حال به طور خاص به نحوه‌ی کارکرد FIRNESS می‌پردازیم. ابتدا تعدادی تعریف ارائه می‌کنیم. پارمترهای ورودی‌های یک تابع را پارمترهایی از آن در نظر می‌گیریم که تابع صرفاً از آن‌ها می‌خواند و پارمترهای خروجی تابع را آن‌هایی که در آن‌ها می‌نویسد (برای مثال پوینتری به تابع پاس داده می‌شود که بر روی آن نوشته شود). برای مثال در تابع strncpy با پارمترهای destination و source و num، پارمتر نخست از نوع خروجی و دوتای دیگر از نوع ورودی هستند. هم‌چنین توابع تولیدکننده‌ی یک نوع داده‌ی خاص را توابعی در نظر می‌گیریم که این نوع داده در خروجی یا پارمترهای خروجی این تابع باشد. valueها را نیز عبارت‌هایی تعریف می‌کنیم که می‌توانند در سمت چپ یک عبارت قرار گیرند.  $LRExp(f)$  را مجموعه‌ی تمامی valueها در تابع  $f$  و  $Lr(E)$  را بزرگترین زیرعبارت value عبارت  $E$  در نظر می‌گیریم.

به نحوه‌ی جمع‌آوری داده‌ی مورد نیاز خود می‌پردازیم. پارمترهای ورودی و خروجی تابع از روی annotationهای IN و OUT موجود در کد مشخص می‌شوند. برای تشخیص فراخوانی غیرمستقیم توابع، از طریق پوینتر به تابع، نیز تخصیص‌ها به پوینترهای مد نظر و مقداردهی‌های اولیه را در نظر می‌گیریم (در این روش تنها یک لایه غیرمستقیم را تشخیص می‌دهیم). می‌خواهیم در هر تابع برای هر value (نوع خاصی از) تعاریف را بیابیم تا از آن‌ها برای یافتن محل فراخوانی توابع استفاده کنیم. به این منظور برای متغیرهای از نوع پوینتر هم مقداردهی مستقیم ( $ptr=NULL$ ) و هم غیر مستقیم ( $ptr*=c$ ) و برای دیگر متغیرها تنها مقداردهی مستقیم را در نظر می‌گیریم. برای سادگی در طراحی و پیاده‌سازی نیز تنها تعاریفی که ثابت و یا تابع تولیدکننده هستند در نظر گرفته می‌شوند. در نهایت با استفاده از تکنیک reaching definition analysis بر روی گراف Control Flow هر تابع، reaching definition map آن را با  $C \cup GF \rightarrow (L \times LRExp(f)) : \psi$  به دست می‌آوریم که آن تعاریفی (نوع تعاریفی که مشخص کردیم) از valueای را که در خطی از تابع قابل دسترسی است مشخص می‌کند. حال برای هر تابع مورد نظر در هر مکان فراخوانی آن مقادیری که پارمترهای ورودی آن می‌توانند داشته باشند را نگه می‌داریم؛ به بیان دقیق‌تر  $C \cup GF \rightarrow (L \times (f_l \cup GF) \times N) : \Gamma$ . برای پیدا کردن این مقادیر نیز از  $\psi$  که به دست آوردیم استفاده می‌کنیم و مقادیری که پارمترها می‌توانند داشته باشند را می‌یابیم.

با استفاده از داده‌های جمع‌آوری شده‌ی خود می‌خواهیم harness را تولید کنیم که با دریافت ورودی‌های تصادفی مقدارهایی از نوع مناسبی را برای پاس دادن تابع تولید می‌کند. ابتدا می‌خواهیم نوع پارمترهای ورودی را تشخیص دهیم. به این منظور از  $\Gamma$  که به دست آوردیم استفاده می‌کنیم و برای یک تابع خاص و یک ورودی خاص از آن تمامی مقادیر  $\Gamma$  را یکی می‌کنیم و به این ترتیب تابع  $C \cup GF \rightarrow ((f_l \cup GF) \times N) : \Gamma_v$  را به دست می‌آوریم. برای تشخیص نوع داده‌ی ورودی به یک تابع نوع داده‌های آن در خروجی  $\Gamma_v$  را برای آن تابع و ورودی نگاه می‌کنیم. اگر تعداد نوع داده‌های متمایز بیشتر از مقداری بود (در این‌جا ۳ در نظر گرفته شده است) نوع آن

را جنریک (\* void) و در غیر این صورت نوع داده‌ای با بیشترین تکرار در نظر می‌گیریم. حال می‌خواهیم برای این نوع داده‌های تشخیص داده شده مقدار مناسب تولید کنیم تا به هنگام فراخوانی به تابع داده شوند. (۱) اگر داده از نوع ثابت بود خود آن را در نظر می‌گیریم، (۲) اگر از نوع primitive بود به سادگی با استفاده از ورودی تصادفی که در اختیار داریم برای آن مقدار تعیین می‌کنیم (مثلاً برای int32 صرفاً ۳۲ بیت از ورودی را در نظر می‌گیریم)، (۳) اگر از نوع داده‌های مرکب (مانند struct) بود، ابتدا بررسی می‌کنیم که آیا این نوع داده داخل خود به صورت بازگشتی استفاده شده است و یا بیش از دو لایه داده در آن تعریف شده است، اگر چنین نبود در صورت وجود توابع تولیدکننده برای آن از یکی از آن‌ها به صورت تصادفی برای تولید داده با این نوع استفاده می‌کنیم (ممکن است نیاز باشد برای خود تابع تولیدکننده نیز این فرایندها را تکرار کنیم تا بتوانیم از آن استفاده کنیم)، و (۴) برای نوع‌های پوینتر نیز ابتدا داده‌ای با این نوع مشخص با مقادیر مناسب تولید می‌کنیم و سپس از آدرس آن استفاده می‌کنیم. همچنین لازم به ذکر است برای انتخاب تابع تولیدکننده برای تولید مقدار مناسب توابعی را که با استفاده از آن‌ها ممکن است در حلقه بیفتیم را در نظر نمی‌گیریم. برای مثال برای تولید مقداری از نوع a از توابعی که ورودی‌ای از نوع a دارد استفاده نمی‌کنیم.

در پیاده‌سازی این سیستم برای بخش FIRNESS و تولید harness هزاران خط در زبان‌های مختلف (C, C++, Python and) کد نوشته شده است و برای پشتیبانی ASAN نیز چندین هزار خط از کد تغییر منبع تغییر داده شده است. ورودی سیستم محل کد EDK-2 و فایلی شامل واسطی از DXE است که می‌خواهیم آن را تست کنیم. این واسطها به دو دسته‌ی سرویس‌ها و پروتکل‌ها تقسیم می‌شوند که به ترتیب به وسیله‌ی نام و نوع، و GUID (و در صورت نیاز نام تابع) مشخص می‌شوند. هنگام پردازش کد علاوه بر مواردی که پیش از این گفته شد ماکروها، استراکت‌ها، فایل‌ها هدر و... نیز نگهداری می‌شوند که برای تولید کد برای تولید harness از آن‌ها استفاده می‌شود. آنالیزها همان‌طور که پیش از این توضیح داده شد مرحله به مرحله انجام می‌شوند و در نهایت داده‌های مورد نیاز در فایل fitness.json ذخیره می‌شوند که شامل نتایجی از هر نقطه‌ی فراخوانی از تابع مورد نظر می‌شود. از همین فایل در آینده برای تولید harness استفاده می‌شود. همچنین کدی شامل توابع مورد نیاز پیش از این به زبان C نوشته شده است. برای هر پارامتر ابتدا با استفاده از داده‌های به دست آمده به روش توضیح داده شده نوع آن را می‌یابیم و سپس با تولید کدی با استفاده از توابع تولیدکننده در صورت نیاز مقدار مناسبی برای آن تولید می‌کنیم. برای بهبود عملکرد سیستم نیز می‌خواهیم sanitizerهای مانند Address Sanitizer (ASAN) به آن بیفزاییم. از آنجایی که EDK-2 نحوه‌ی مدیریت حافظه‌ی خود را دارد و پیش از بوت شدن حافظه‌ی مجازی در دسترس نیست افزودن این ابزار بدون ایجاد تغییری در آن ممکن نیست. پیش از این کارهایی به این منظور انجام شده است ولی قدیمی شده‌اند. همان‌طور که پیش از این گفته شد این موضوع با تغییر ساختار حافظه‌ی EDK-2 و پیکربندی ASAN انجام شده است و همچنین با این تغییرات امکان استفاده از آن در محیط‌های مجازی مانند Simics نیز فراهم شده است. همچنین با توجه به تغییرات انجام شده باید توابع مدیریت حافظه در EDK-2 نیز بازنویسی شوند.

## ارزیابی:

در این بخش ۴ موضوع اصلی بررسی می‌شود: عملکرد FIRNESS، عملکرد FUZZUER، مقایسه با HBFA، و تأثیر هر کدام از تکنیک‌ها پیاده‌سازی شده بر عملکرد کلی سیستم fuzzing. ابتدا به setup سیستم می‌پردازیم. در این سیستم از TSFFS استفاده شده است که خود از فازر libAFL با اجرا در محیط مجازی از طریق Simics استفاده می‌کند. برای بررسی عملکرد سیستم ۳ آسیب‌پذیری شناخته شده از قبل که در سیستم وجود داشته است را به آن وارد می‌کنیم (تا ببینیم آیا این سیستم توانایی تشخیص آن‌ها را دارد یا خیر). این کار خود با توجه به تغییرات در نسخه‌های EDK-2 خود چالشی بوده است. برای بررسی عملکرد سیستم در حالت‌های مختلف و سهم هر کدام در عملکرد کلی چندین پیکربندی برای آن در نظر می‌گیریم: (۱) استفاده مستقیم از ورودی بدون توجه به نوع داده، (۲) استفاده نکردن از توابع تولیدکننده، (۳) استفاده از توابع تولیدکننده برای مقادیر با استفاده مستقیم از ورودی و بدون استفاده از نوع داده‌ها، (۴) بدون در نظر گرفتن پوینتر به تابع و فراخوانی غیر مستقیم آن، و (۵) سیستم در حالت کلی با تمام قابلیت‌های پیاده‌سازی شده. در این کار ۱۵۰ تابع از ۲۵ پروتکل (از ۶ دسته‌ی USB, text, controller, SMM, driver helper, network) بررسی شده‌اند. برای هر پروتکل harness تولید شده و برای ۲۴ ساعت برای هر کدام از پیکربندی‌ها تست شده است (انتخاب تست کردن تابع بر اساس بایت نخست خود ورودی تصادفی مشخص می‌شود).

برای بیان عملکرد در هر کدام از بخش‌ها میانگین عملکرد آن برای هر کدام از پارامترهای آن نوع (برای مثال مقدار ثابت) در نظر گرفته می‌شود (این بررسی به شکل دستی انجام می‌شود). تشخیص پارامترهای ثابت و توابع تولیدکننده در اکثر موارد با دقت بالایی انجام شده است ولی در برخی موارد به دلیل بررسی نکردن جریان اجرای کد متوجه خطا شده است (برای مثال  $f(a)$  if  $a=2$ ) تشخیص تابع فراخوانده شدن در پوینتر به تابع نیز با دقت خوبی به جز برای SMM موجه هستیم که علت آن وجود دو لایه فراخوانی غیرمستقیم است (\*\*fptr). عملکرد تشخیص نوع پارامترها برای نوع‌های scalar pointer, and struct pointer, and scalar, struct and struct pointer جداگانه بررسی شده است. برای scalar pointer و scalar دقت بسیار بالایی در تمامی پروتکل‌ها وجود دارد، ولی برای struct پروتکل‌های مربوط به SMM دقت پایین‌تری وجود دارد که آن هم به دلیل کمبود تعداد فراخوانده شدن برخی توابع‌ها برای تشخیص نوع پارامترهای آن‌هاست. در نهایت برای تولید harness حدوداً دقت ۸۰ درصدی برای تمامی پروتکل‌ها داریم. علت این موضوع وجود recursion و پارامترهای از نوع call-back است (که از آن‌ها پشتیبانی نمی‌شده است برای مثال به جای پارامتر call-back مقدار NULL قرار می‌گیرد).

برای بررسی عملکرد fuzzing، پوشش کد و توانایی یافتن باگ را در آن بررسی می‌کنیم. در کل FUZZUER پوشش حدود ۴۰ درصدی داشته است. این عدد کمتر از مقدار واقعی است زیرا استفاده از پوینتر به توابع در این پوشش حساب نمی‌شود که دلیل آن در پیوست مقاله توضیح داده شده است. هم‌چنین این سیستم به علت استفاده از مقدارهای مناسب برای پارامترها زود به این مقدار پوشش می‌رسد (در ۱۰ ساعت ابتدایی از ۲۴ ساعت). این سیستم از ۳ آسیب‌پذیری شناخته شده‌ی پیشین (که خود برای بررسی وارد EDK-2 کردیم) ۲ مورد از آن‌ها را

تشخیص می‌دهد. ۲۰ آسیب‌پذیری جدید از نوع‌های مختلف (null-pointer dereference, buffer overflow, use-after-free, ...) نیز با استفاده از این سیستم یافت شده است. برای نمونه نیز دو آسیب‌پذیری تشخیص داده‌شده بررسی شده‌اند.

برای استفاده از HBFA باید خود برای پروتکل مد نظر harness و برای بخش‌های وابسته به سخت‌افزار stub بنویسیم که کاری طاقت‌فرسا است. در حال حاضر برای ۳ پروتکل این شرایط از پیش برقرار شده است که در این شرایط آن را با FUZZUER مقایسه می‌کنیم. حجم کد harness در HBFA خیلی کمتر است آن هم به این دلیل که نمونه‌های ساده‌ای و اکثراً با داده‌های ثابت تولید می‌کند. حجم کد پوشش داده‌شده توسط FUZZUER چندین برابر HBFA است که علت آن استفاده از harness ساده و لزوم استفاده از stub در HBFA است. در این ۳ پروتکل مورد بررسی FUZZUER سه آسیب‌پذیری یافت ولی HBFA، به همان دلایلی پیش از این تشریح شد، موردی نیافت.

حال می‌خواهیم تأثیر هر کدام از بخش‌ها در عملکرد کلی سیستم را ببابیم. همان‌طور که انتظار داریم در حالت کلی به بیشترین پوشش در اکثر پروتکل‌ها و در حالت بدون harness به کمترین پوشش می‌رسیم. هم‌چنین در حالت‌های دیگر میزان پوشش به نوع پروتکل بستگی دارد. برای مثال در یک پروتکل بیشتر نوع داده‌ها تابع تولیدکننده ندارند و تأثیر استفاده نکردن از تابع تولیدکننده در آن‌ها کم است. در برخی حالت‌ها نیز پوشش سیستم در حالت کامل کمتر از دیگر حالت‌ها است که به علت وارد نشدن به بخش‌های مدیریت خطا به دلیل تولید مقادیر مناسب است. توانایی یافتن باگ نیز در حالت کامل از همه بیشتر و در حالت استفاده نکردن از harness از همه کمتر است. این مقدار برای حالت‌های دیگر بسته به پروتکل متفاوت است. در نهایت با بررسی این نتایج به این نتیجه می‌رسیم که تمامی بخش‌های FUZZUER در عملکرد آن تأثیر دارند.

### محدودیت‌ها:

در طراحی کنونی FIRNESS نوع داده‌ها و تابع‌های بازگشتی را پشتیبانی نمی‌کند. فاز کردن در این حالت coverage کمی دارد که بیان شده است این موضوع به ماهیت این سیستم وابسته نیست و با استفاده از فازرهای concolic بهبود می‌یابد.

### کارهای مرتبط:

دیگر کارهایی در زمینه‌ی تست و آسیب‌یابی UEFI انجام شده است که در اینجا به برخی از آن‌ها اشاره می‌شود (به دلیل تفاوت‌های ساختاری در بوت‌لودرهای دیگر مانند Smartphone به آن‌ها پرداخته نمی‌شود). مقاله کارهای انجام‌شده در این زمینه را به دو بخش بررسی ایستا و پویا تقسیم کرده است. کارهای ایستا بیشتر به تست System Management Interrupt ها می‌پردازند (از جمله یکی از مقاله‌هایی که پیش از این بررسی کردیم). این کارها از بررسی کد باینری موجود تا استفاده از الگوریتم‌های گراف و شبکه برای یافتن آسیب‌پذیری، مانند افزایش سطح دسترسی، را می‌تواند شامل شود. در کارهای پویا نیز از روش‌های مختلفی از جمله فاز کردن

استفاده می‌شود و ابزارهای مختلفی برای این موضوع توسعه داده شده‌اند که بین آن‌ها مقایسه‌ای انجام شده است. بیان می‌شود که بیشتر کارهای این بررسی پویا نیز به بررسی مدیریت‌کننده‌های SMI می‌پردازد که از آن‌جایی که به حالت سیستم وابسته نیست و با دریافت ورودی حالت آن را عوض می‌کند، بررسی آن‌ها از بررسی پروتکل‌ها، که در این مقاله انجام شده است، متفاوت است. پس از آن به چندین کار اشاره می‌شود که از روش‌های فاز کردن تا Symbolic Execution برای انجام این بررسی‌ها بهره برده‌اند. از جمله کارهایی که در این زمینه انجام شده است HBFA است. اما برای استفاده از این پلتفرم نیاز به تست پروتکل به عنوان یک برنامه‌ی جداگانه داریم و به این منظور باید ارتباط آن با دیگر بخش‌ها را خود دستی شبیه‌سازی کنیم که می‌تواند فرایندی سخت و زمان‌گیر باشد. از کارهای دیگر SIMFUZZER است که با دریافت فیدبک در هنگام فاز به صورت خودکار سیستم را بررسی می‌کند، ولی اهمیتی به نوع داده‌ها نمی‌دهد که باعث کاهش عملکرد آن می‌شود (در واقع مشابه همان FUZZUER بدون بررسی نوع داده‌ها و توابع تولیدکننده و... است). همچنین این سیستم، برخلاف FUZZUER، متن‌باز نیست که امکان استفاده از آن را محدود می‌کند.

در نوع دیگری از کارها نقاط مختلفی از برنامه را به حالت‌هایی مدل می‌کنند و سعی می‌کنند با استفاده از داده‌ای که آن‌ها در اختیار می‌گذارند، تست بخش بیشتری از کد را پوشش دهد. یکی از ابزارهایی که اینچنین کار می‌کند FUZZGEN است که گرافی از وابستگی‌ها تولید می‌کند و با کمک libFuzzer و حالت‌های نگه‌داری شده می‌توان با استفاده از آن سیستم‌ها پیچیده را فاز کرد. ولی این ابزار نیاز به دانستن نوع پارامترها و تعریف تابع تحت تست دارد که این موارد در EDK-2 که در آن از پوینتر به توابع زیاد استفاده می‌شود، برقرار نیست. از دیگر ابزارها INTELLIGEN و HOPPER است که مقایسه‌ی آن‌ها با FUZZUER در جدولی در مقاله انجام شده است و ما در این‌جا به آن‌ها نمی‌پردازیم. در نهایت به این اشاره می‌شود که چرا تکنیک‌ها و ابزارهایی که برای تست فراخوان‌های سیستمی در سیستم عامل‌ها انجام می‌شود، لزوماً برای تست EDK-2 جواب نمی‌دهند (برای مثال تشخیص void\* و به ابزارهای HFL و DIFUZE اشاره شده است).

### جمع‌بندی:

در این کار FUZZUER به عنوانی فریم‌ورکی برای فاز کردن، با استفاده از harness خودکار به نام FIRNESS معرفی شده است. در این فریم‌ورک از پلتفرم مجازی Simics و فازر LibAFL استفاده شده است. harness مد نظر با بررسی کد و تشخیص نوع و ایجاد مقادیر مناسب (همگی به صورت خودکار) عمل می‌کند. همچنین این فریم‌ورک متن‌باز است و عملکرد خود را با یافتن آسیب‌پذیری‌هایی در نسخه‌ی آخر EDK-2 نشان داده است.

## معیار مقایسه‌ی ابزارهای fuzzing

برای پاسخ به این پرسش، بخش ارزیابی مقاله‌ی «FUZZUER: Enabling Fuzzing of UEFI Interfaces on EDK-2» را بررسی می‌کنیم. در این مقاله برای بررسی عملکرد سیستم خود ۴ پرسش مطرح کرده است و یک‌به‌یک به آن‌ها پاسخ داده است:

- عملکرد FIRNESS: دقت این ابزار در تولید harness چه قدر است؟
  - عملکرد fuzzing: عملکرد FUZZUER در فاز کردن پروتکل‌های UEFI چگونه است (Code Coverage و توانایی باگیابی)؟ آیا این پلتفرم توانست آسیب‌پذیری جدیدی را کشف کند؟
  - عملکرد FUZZUER در مقایسه با HBFA چگونه است؟
  - تأثیر هر کدام از بخش‌های این پلتفرم در عملکرد کلی سیستم چگونه است؟
- برای بررسی عملکرد FIRNESS، در هر کدام از توابع میانگین عملکرد آن برای هر کدام از پارامترهای آن در نظر گرفته می‌شود. این اعداد نیز به شکل دستی و با مقایسه‌ی نوع مقدار به‌دست‌آمده با نوع مورد نظر به دست می‌آید.

همان‌طور که گفته شد برای بررسی عملکرد fuzzing، پوشش کد و توانایی یافتن باگ در آن بررسی می‌شود. پوشش به این شکل محاسبه می‌شود: به دست آوردن گراف فراخوانی (Call Graph) و توابع در دسترس با استفاده از آن؛ محاسبه‌ی درصد خط‌های کد پوشش‌داده‌شده به نسبت کل تعداد خطوط کد در دسترس (در تمامی توابع در دسترس). در کل FUZZUER پوشش حدود ۴۰ درصدی داشته است. این عدد کمتر از مقدار واقعی است زیرا استفاده از پوینتر به توابع در این پوشش حساب نمی‌شود و در نتیجه کدهای پوشش‌داده‌شده‌ی آن‌ها محاسبه نمی‌شود. هم‌چنین این سیستم به علت استفاده از مقدارهای مناسب برای پارامترها زود به این مقدار پوشش می‌رسد (در ۱۰ ساعت ابتدایی از ۲۴ ساعت). این سیستم از ۳ آسیب‌پذیری شناخته‌شده‌ی پیشین (که خود برای بررسی وارد EDK-2 کردیم) ۲ مورد از آن‌ها را تشخیص می‌دهد. ۲۰ آسیب‌پذیری جدید نیز با استفاده از این سیستم یافت شده است.

برای ۳ پروتکل شرایط برای استفاده از HBFA از پیش برقرار شده است که در این شرایط با FUZZUER مقایسه شده است. در این بررسی حجم کد harness، پوشش کد و آسیب‌پذیری‌های یافت‌شده بررسی شده است. حجم کد harness در HBFA خیلی کمتر است آن هم به این دلیل که نمونه‌های ساده‌ای و اکثراً با داده‌های ثابت تولید می‌کند. در این حالت برای محاسبه‌ی پوشش کد به این شکل عمل می‌کنیم: ابتدا Control Flow Graph کد را به دست می‌آوریم و سپس تعداد یال‌های یکتای دیده‌شده را به دست می‌آوریم. در این مقایسه، این عدد در کل کد و خود پروتکل به‌دست‌آورده و مقایسه می‌شود. تعداد حجم کد پوشش‌داده‌شده توسط FUZZUER چندین برابر HBFA است که علت آن استفاده از harness ساده و لزوم استفاده از stub در HBFA است. در این ۳ پروتکل مورد بررسی FUZZUER سه آسیب‌پذیری یافت ولی HBFA، به همان دلایلی پیش از این تشریح شد، موردی نیافت.

حال می‌خواهیم تأثیر هر کدام از بخش‌ها در عملکرد کلی سیستم را بباییم. به این منظور در حالت‌های مختلف پلتفرم عملکرد (در این‌جا پوشش) آن را بررسی می‌کنیم. در نهایت با بررسی این نتایج به این نتیجه می‌رسیم که تمامی بخش‌های FUZZUER در عملکرد آن تأثیر دارند.

برای جزئیات بیشتر در هر کدام از این بخش‌ها می‌توانید به بخش ارزیابی در خلاصه‌ی این مقاله (گزارش ۴) مراجعه کنید.

## تعریف Coverage

برای پاسخ به این پرسش نیز ، مقاله‌ی «FUZZUER: Enabling Fuzzing of UEFI Interfaces on EDK-2» را بررسی می‌کنیم. همان‌طور که در بخش قبل گفته شد پوشش به دو شکل در این مقاله بررسی شده است.

- **بر حسب Call Graph:** پوشش به این شکل محاسبه می‌شود: به دست آوردن گراف فراخوانی (Call Graph) و توابع در دسترس با استفاده از آن؛ محاسبه‌ی درصد خط‌های کد پوشش داده‌شده به نسبت کل تعداد خطوط کد در دسترس (در تمامی توابع در دسترس).
- **بر حسب Control Flow Graph:** در این حالت برای محاسبه‌ی پوشش کد به این شکل عمل می‌کنیم: ابتدا Control Flow Graph کد را به دست می‌آوریم و سپس تعداد یال‌های یکتای دیده‌شده را به دست می‌آوریم.

از تعریف نخست در بررسی پوشش کد این پلتفرم به خودی خود و از تعریف دوم در مقایسه‌ی آن با HBFA استفاده شد. همان‌طور که در گزارش قبلی دیدیم نیز خروجی تولیدشده‌ی این ابزار پوشش را با استفاده از تعریف دوم ارائه می‌کند.

## تحقیق درباره‌ی پیاده‌سازی گراف فراخوانی و CFG در FIRNESS

در این بخش به بررسی پیاده‌سازی Call Graph و Control Flow Graph می‌پردازیم. در گزارش قبلی به این دو معیار برای ارزیابی پوشش در عملکرد فازر اشاره شد.

- در Call Graph یا همان گراف فراخوانی پوشش ابتدا تمام توابع در دسترس محاسبه می‌شوند و درصد پوشش درصد خط‌های کد پوشش داده شده می‌شود.
- در Control Flow Graph ابتدا گراف کنترل جریان کد را بدست می‌آوریم و سپس تعداد یال‌های یکتای دیده‌شده نشانگر درصد پوشش خواهد بود.

در واقع به کمک Call Graph روابط بین توابع و بلوک‌های مختلف کد و نحوه‌ی تولید ورودی برای رسیدن به آن‌ها و در CFG رفتار داخلی توابع و بلوک‌ها برای بررسی مسیرها خطرناک و محتمل برای یافتن خطرات بررسی می‌شوند.



در این سیستم، کلاس call graph در [این آدرس](#) پیاده‌سازی شده است. این کلاس از [Recursive AST Visitor](#) برای گردش روی درخت سینتکس کد و خارج کردن اطلاعات فراخوانی‌های موجود در کد استفاده می‌کند. این بخش از کد توسط سازندگان سیستم طراحی شده است و صرفاً به کمک AST Visitor روی AST کد عملیات DFS را پیاده‌سازی می‌کنند.

همچنین در فایل [scripts/firness.py](#) به کمک دستور `opt -dot-cfg` خروجی CFG گراف گرفته می‌شود. در ادامه به کلیات نحوه‌ی استخراج CFG اشاره می‌کنیم و جزئیات بخش‌های مختلف را توضیح می‌دهیم.

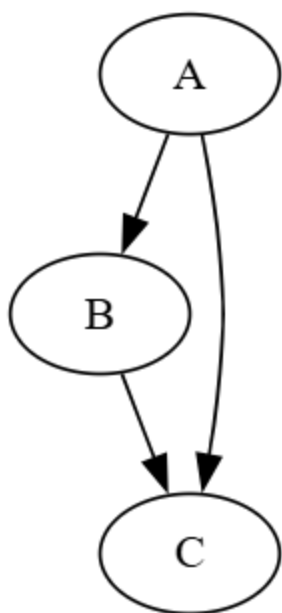
کتابخانه‌ی کامپایلر [LLVM](#):

در این سیستم به کمک `opt` گراف کنترل جریان ساخته می‌شود. ابزار `opt` در LLVM امکان تولید نمودار جریان کنترل (CFG) از کدهای LLVM IR را فراهم می‌کند. این نمودارها برای تحلیل و بهینه‌سازی کد بسیار مفید هستند. دستور پایه `input` `opt -dot-cfg` که در خط 295 فایل [firness.py](#) موجود است، دستور اولیه برای تولید فایل با پسوند `dot` برای CFG استفاده می‌شود.

پسوند DOT:

این فایل برای ذخیره‌سازی گراف‌ها استفاده می‌شود و قابلیت‌های متغیری برای نمایش اتصال بین رئوس، ایجاد زیرگراف‌های مختلف و یال‌های گوناگون دارد اما در این پروژه تنها نسخه‌ی ساده آن که شامل تعریف رئوس و سپس تعریف یال‌ها می‌شود انجام می‌شود. نمونه کد و گراف متناظر آن در شکل زیر آمده است.

گراف ارتباط بین توابع در پوشه خروجی‌ها قرار دارد و ارتباط بین این توابع را تصویر می‌کند. (به علت حجم زیاد قابل نمایش نمی‌باشد.) فایل‌های مربوط به این بخش در پوشه‌ی Call Graph در Miscellaneous قرار دارد.



```

1 digraph {
2     A;
3     B;
4     C;
5
6     A -> B;
7     B -> C;
8     A -> C;
9 }
  
```



## بخش عملی

در این بخش ابتدا به راه‌اندازی ابزارهای مورد نیاز پرداخته شده است و سپس نتایج و چالش‌های مربوط به فاز کردن پروتکل‌های مختلف آورده شده است. فایل‌های مربوط به فاز کردن‌ها در پوشه‌ی Fuzzing Results در Miscellaneous آمده است.

### نصب و راه‌اندازی ابزار شبیه‌سازی TSFFS

در قدم اول برای شروع کار عملی ابزار TSFFS را راه‌اندازی می‌کنیم. این ابزار با استفاده از شبیه‌ساز SIMICS و کتابخانه LibAFL فرآیند Fuzzing را بسیار ساده می‌کند. ما در این هفته این ابزار را راه‌اندازی می‌کنیم. به جهت سادگی و portable بودن، از نسخه داکری این ابزار استفاده می‌کنیم. در مرحله اول لازم است که سورس این ابزار را از گیت‌هاب دریافت کنیم. سپس با ایجاد تغییرات زیر ایمج داکری را برای build آماده می‌کنیم.

- استفاده از رجیستری آروان برای دانلود base image داکر.
  - استفاده از شکن برای دانلود مراحل میانی build شدن ایمج.
- سپس به صورت زیر محیط را build می‌کنیم.

```

[sdf/iot-lab/tsffs] main !1 docker build -t tsffs
[+] Building 248.4s (7/14)
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 4.79kB
=> [internal] load metadata for docker.arvancloud.ir/fedora:38@sha256:b9ff6f23cceb5bde20bb1f79b492b98d71ef
=> [internal] load .dockerignore
=> => transferring context: 266B
=> [ 1/10] FROM docker.arvancloud.ir/fedora:38@sha256:b9ff6f23cceb5bde20bb1f79b492b98d71ef
=> [internal] load build context
=> => transferring context: 34.29kB
=> CACHED [ 2/10] RUN dnf -y update && dnf -y install alsa-lib atk
=> CACHED [ 3/10] WORKDIR /workspace
=> [ 4/10] RUN mkdir -p /workspace/simics/ispn/ && curl --noproxy '*.intel.com' -L -o
=> => # % Total % Received % Xferd Average Speed Time Time Time Current
=> => # Dload Upload Total Spent Left Speed
=> => # 11 134M 11 14.8M 0 0 69787 0 0:33:35 0:03:43 0:29:52 83308

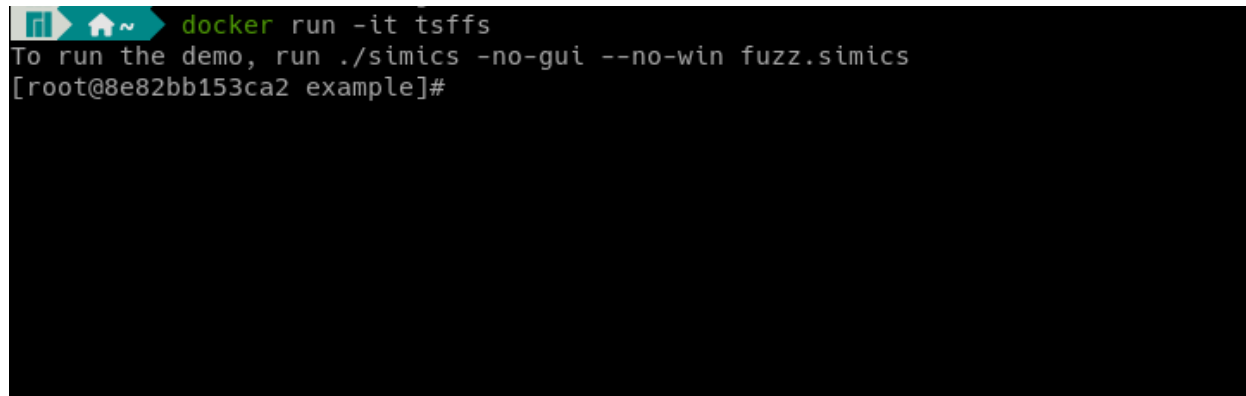
```

اقدامات و چالش‌هایی که هنگام بیلد شدن مواجه شدیم به صورت زیر است:

- ما برای اتصال به پروکسی جهت دانلود محتوای ابزار simics نیاز بود که شبکه داکر را به صورت host قرار دهیم.

- در هنگام بیلد شدن، منابع بسیار زیادی مصرف می‌شد و سیستم OOM می‌شد. برای رفع این مشکل ما حافظه Swap بیشتری به سیستم اختصاص دادیم.
- در نهایت پس از بیلد شدن می‌توانیم به صورت زیر شبیه‌ساز را استفاده کنیم:

```


A terminal window with a dark background. The prompt is [root@8e82bb153ca2 example]#. The command docker run -it tsffs is entered. Below it, the instruction "To run the demo, run ./simics -no-gui --no-win fuzz.simics" is shown. The prompt returns to [root@8e82bb153ca2 example]#.

```

حال عملیات فاز را روی نرم‌افزار مثال این ابزار انجام می‌دهیم:

```

<qsp.serconsole.con>000e7a7c3200001\r\n
[tsffs info] Simulation stopped with reason Magic { magic_number: StopNormal }
[tsffs info] Testcase: Testcase { testcase: "[51, 102, 102, 102, 102, 102, 241, 255, 64, 0, 0, 0, 48
}
[tsffs info] Resuming simulation
[tsffs info] Simulation stopped with reason Magic { magic_number: StartBufferPtrSizePtr }
[tsffs info] Resuming simulation
<qsp.serconsole.con>336666666666f1ff\r\n
[tsffs info] Simulation stopped with reason Magic { magic_number: StopNormal }
[tsffs info] Testcase: Testcase { testcase: "[102, 102, 102, 102, 65, 99, 102, 102, 102, 102, 65, 99
g: false }
[tsffs info] Resuming simulation
[tsffs info] Simulation stopped with reason Magic { magic_number: StartBufferPtrSizePtr }
[tsffs info] Resuming simulation
<qsp.serconsole.con>66666666641636666\r\n
[tsffs info] Simulation stopped with reason Magic { magic_number: StopNormal }
[tsffs info] Testcase: Testcase { testcase: "[255, 117, 122, 122, 105, 79, 103, 33, 52, 52, 52, 48,
}
[tsffs info] Resuming simulation
[tsffs info] Simulation stopped with reason Magic { magic_number: StartBufferPtrSizePtr }
[tsffs info] Resuming simulation
<qsp.serconsole.con>ff757a7a694f6721\r\n
[tsffs info] Simulation stopped with reason Magic { magic_number: StopNormal }
[tsffs info] Testcase: Testcase { testcase: "[37] (1 bytes)", cmplog: false }
[tsffs info] Resuming simulation
[tsffs info] Simulation stopped with reason Magic { magic_number: StartBufferPtrSizePtr }
[tsffs info] Resuming simulation
<qsp.serconsole.con>25\r\n
[tsffs info] Simulation stopped with reason Magic { magic_number: StopNormal }
[tsffs info] Testcase: Testcase { testcase: "[76, 76, 76, 76, 87, 255, 255, 255, 255, 255, 255,
}

```

همانطور که مشاهده می‌شود عملیات فاز روی UEFI بدون مشکل در حال انجام است و به واسطه تست کیس‌ها در حال انجام است.

## فاز کردن پروتکل‌های تکراری

در این بخش مراحل فاز کردن پروتکل‌های موجود در مقاله و replicate کردن خروجی آن‌ها آمده است.

### پروتکل USB\_IO

#### چالش‌ها

پس از بیلد کردن ایمپج داکری و اجرای firmness به مشکلی برای پیدا نکردن فایل‌های آنالیز خوردیم. پس از بررسی متوجه شدیم که در کد اصلی firmness بخش آنالیز به علت زمان‌بر بودن کامنت شده است. سپس با اجرای آنالیز این مرحله از اجرا جلو می‌رود.

پس از آن ساخت Harness برای ورودی‌های پیش‌فرض در input.txt به مشکل خورد. با بررسی بیشتر و حذف تعدادی از Function‌ها از input.txt متوجه شدیم که ساخت Harness برای یکی از Function فقط مشکل داشت. بنابراین هدفمان روی ساخت Harness برای پروتکل مقصد یعنی USB\_IO گذاشتیم. پس از آن تمام Harness‌ها بدون مشکل بیلد می‌شوند.

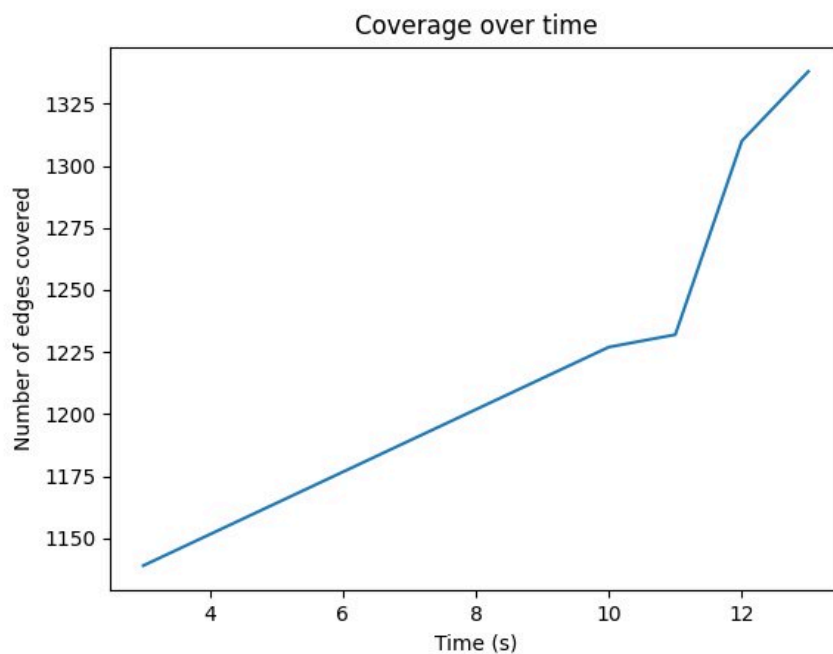
پس از آن برای اجرای fuzzing از فلگ -a استفاده می‌کنیم تا Fuzzer اجرا بشود.

#### نتایج

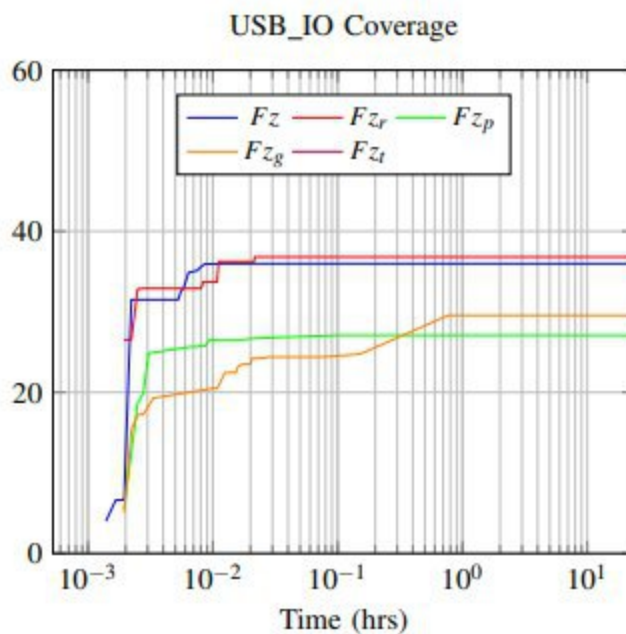
توابعی از پروتکل USB\_IO را فاز کرده‌ایم که لیست آن‌ها در فایل input.txt آورده شده است. همچنین ویژگی‌های بخش‌هایی که فاز می‌کنیم و FIRNESS آن‌ها را استخراج می‌کند، در فایل stats.csv آورده شده است. این فاز در حالتی که از تمامی ویژگی‌های harness استفاده شده است (fz)، و در مدت زمان ۴۳ دقیقه انجام شده است. لاگ‌های خروجی fuzzer نیز در زیر مشاهده می‌شود.

```
[tsffs info] Fuzzer message: [Testcase #0] run time: 0h-0m-12s, clients: 1, corpus: 4, objectives: 0, executions: 46, exec/sec: 5.090
[tsffs info] Interesting input for AFL indices [267, 314, 3726, 3790, 7787, 7864, 8116, 9656, 12915, 14674, 14756, 14809, 39697, 39904,
, 84145, 85002, 86156, 86171, 86183, 86195, 86204, 86247, 86422, 88490, 89366, 91150, 91167, 91196, 91221, 91244, 91255, 91268, 91329, 9
97388, 98857, 98884, 98927, 102775, 102792, 102901, 104906, 104932, 106031, 106036, 116111, 116140, 117940, 118106, 121494, 122532] wit
[tsffs info] 14 Interesting edges seen since last report (951 edges total)
[tsffs info] Fuzzer message: [Stats #0] run time: 0h-0m-13s, clients: 1, corpus: 4, objectives: 0, executions: 46, exec/sec: 4.766
[tsffs info] Fuzzer message: [Testcase #0] run time: 0h-0m-13s, clients: 1, corpus: 5, objectives: 0, executions: 48, exec/sec: 4.973
[tsffs info] Interesting input for AFL indices [7249, 92065, 92096, 102584] with input [190, 255, 178, 255, 236, 236, 236, 236, 255, 255
[tsffs info] 2 Interesting edges seen since last report (953 edges total)
[tsffs info] Fuzzer message: [Stats #0] run time: 0h-0m-13s, clients: 1, corpus: 5, objectives: 0, executions: 48, exec/sec: 4.671
[tsffs info] Fuzzer message: [Testcase #0] run time: 0h-0m-13s, clients: 1, corpus: 6, objectives: 0, executions: 50, exec/sec: 4.865
Execution time: 0hrs 43mins 5secsss 29secs
Execution time: 0hrs 43mins 6secs
```

در تصویر زیر coverage مشاهده می‌شود که همان‌طور که انتظار داریم به سرعت به نقطه‌ی اوج خود می‌رسد. منظور از «edge» یال‌های Control Flow Graph کد است.



در زیر نیز coverage فاز این پروتکل که در مقاله آمده است آورده شده است. همان‌طور که در این تصویر نیز مشاهده می‌شود، coverage در حالت fz در چند ثانیه تقریباً به نقطه‌ای اوج رسیده است و پس از آن ثابت است، که با داده‌های به دست آمده‌ی ما هم‌خوانی دارد. (محور عمودی درصد coverage را نشان می‌دهد که همان درصد کد پوشش داده‌ی تابع‌های قابل دسترس، که آن نیز از Call Graph کد به دست می‌آید، است).



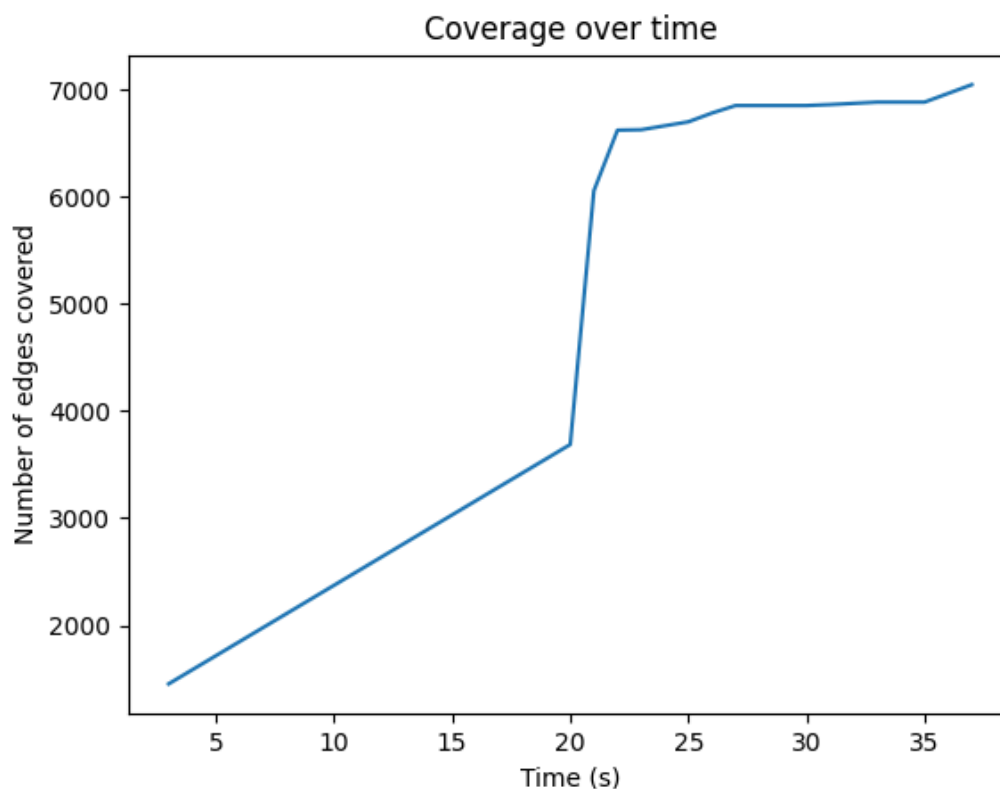
در فایل report.txt خطاهای کشف شده قرار دارد. دیده می شود که خطاها با پیام NullPointerException نمایش داده شده اند. این مورد با این که در مقاله نیز برای این پروتکل خطای Use After Free گزارش شده است هم خوانی دارد.

### پروتکل SimpleNet

توابعی از پروتکل SimpleNET را فاز کرده ایم که جزئیات آن در زیر می آید. فایل های مرتبط به آن نیز در پوشه ای با نام SimpleNET قرار دارد.

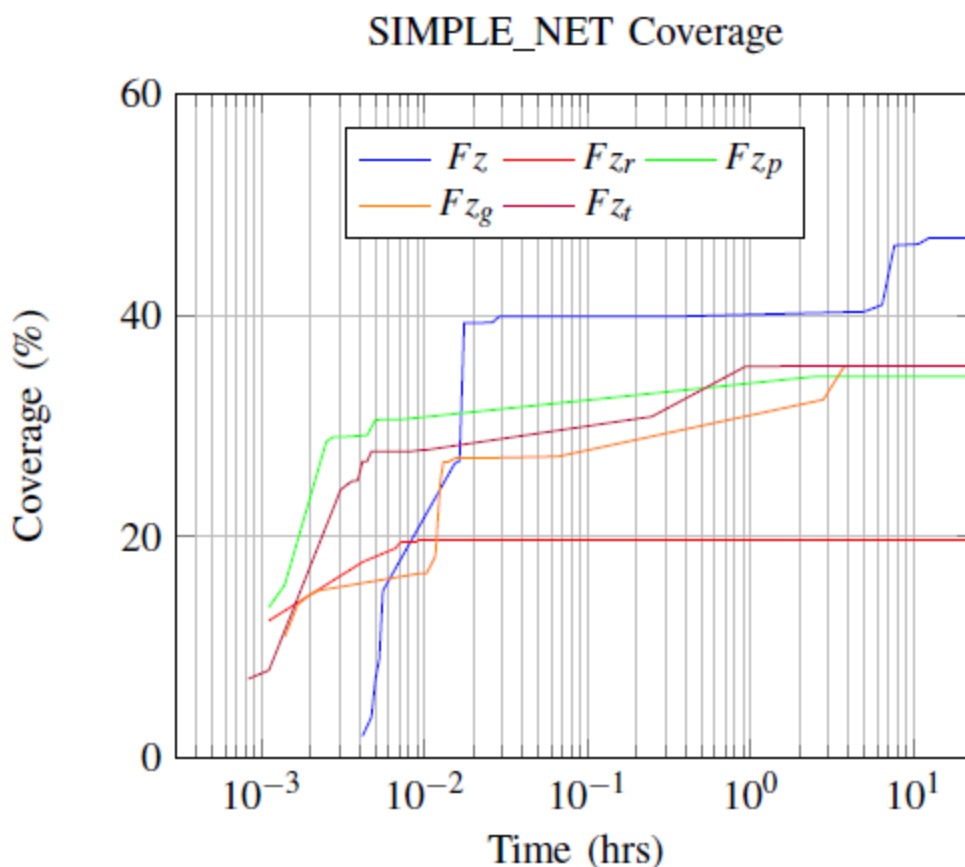
لیست توابع فاز شده از این پروتکل در فایل input.txt آورده شده است. این فاز در حالتی که از تمامی ویژگی های harness استفاده شده است (fz).

در تصویر زیر coverage مشاهده می شود که همان طور که انتظار داریم به سرعت به نقطه ای اوج خود می رسد.



منظور از «edge» یال های Control Flow Graph کد است.

در زیر نیز coverage فاز این پروتکل که در مقاله آمده است آورده شده است. همانطور که مشاهده می‌کنید مانند نمودار آزمایش ما، در ابتدا سرعت کم است اما در اواسط آزمایش سرعت کانورج رشد ناگهانی دارد و به اوج می‌رسد.



در فایل bugs.txt خطاهای کشف‌شده قرار دارد. دیده می‌شود که خطاها با پیام NullPointerUse و overflow در این فایل قرار دارند. همچنین توابع بررسی شده در فایل functions قرار دارند.

## فاز کردن پروتکل‌های تازه

در این بخش مراحل فاز کردن پروتکل‌هایی که در مقاله موجود نیست و خروجی آن‌ها آمده است.

### پروتکل DNS4

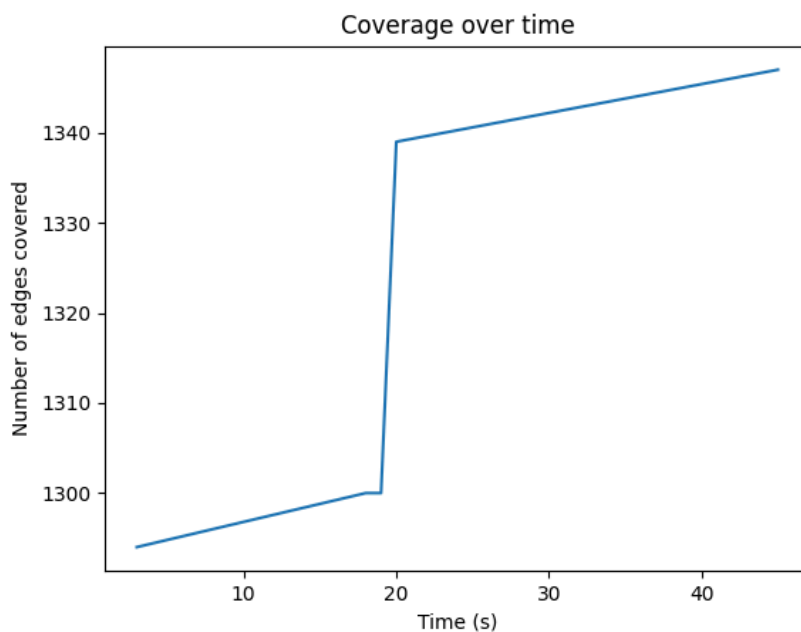
توابعی از پروتکل DNS4 را فاز کرده‌ایم که جزئیات آن در زیر می‌آید. فایل‌های مرتبط به آن نیز در پوشه‌ای با نام DNS4 قرار دارد.

لیست توابع فازشده از این پروتکل در فایل input.txt آورده شده است. این فاز در حالتی که از تمامی ویژگی‌های harness استفاده شده است (fz)، و در مدت زمان ۲۳ دقیقه انجام شده است. در لحظه شروع coverage مشاهده می‌شود که همان‌طور که انتظار داریم به سرعت به نقطه‌ی اوج خود می‌رسد. در ثانیه سوم به ۱۵۴۱ یال از یال‌های CFG می‌رسیم. (بنابر اینکه تنها یک نقطه داریم نمودار آورده نشده است و برای این بخش تنها شامل نقطه‌ی (۱۵۴۱، ۳) می‌شود). در فایل crashes.txt نیز خطاهای کشف‌شده قرار دارد که همان‌طور که مشاهده می‌شود خطای خاصی به دست نیامده است و تنها پیغام‌هایی که ASAN تولید کرده است در این فایل قرار دارد.

### پروتکل BlockIO

توابعی از پروتکل BlockIO را فاز کرده‌ایم که جزئیات آن در زیر می‌آید. فایل‌های مرتبط به آن نیز در پوشه‌ای با نام BlockIO قرار دارد.

لیست توابع فازشده از این پروتکل در فایل input.txt آورده شده است. هم‌چنین ویژگی بخش‌هایی که فاز می‌کنیم و آن‌ها را استخراج می‌کند، در فایل stats.csv آورده شده است. این فاز در حالتی که از تمامی ویژگی‌های harness استفاده شده است (fz) انجام شده است. در تصویر زیر coverage مشاهده می‌شود که همان‌طور که انتظار داریم به سرعت و پس از حدود ۴۵ ثانیه به نقطه‌ی اوج خود می‌رسد. اعداد دقیق تعداد edgeها برای لحظاتی از اجرا، در فایل coverage.csv آورده شده است.



در فایل crashes.txt نیز خطاهای کشف‌شده قرار دارد که همان‌طور که مشاهده می‌شود خطای خاصی به دست نیامده است و تنها پیغام‌هایی که ASAN تولید کرده است در این فایل قرار دارد.

### پروتکل (S3BootScriptLib Credential)

توابعی از پروتکل Credential را فاز کرده‌ایم که جزئیات آن در زیر می‌آید. فایل‌های مرتبط به آن نیز در پوشه‌ای با همین نام قرار دارد.

ویژگی بخش‌هایی که فاز می‌کنیم و FIRNESS آن‌ها را استخراج می‌کند، در فایل stats.csv آورده شده است. این فاز در حالتی که از تمامی ویژگی‌های harness استفاده شده است (fz) انجام شده است.

اعداد دقیق تعداد edgeها برای لحظاتی از اجرا، در فایل coverage.csv آورده شده است که مشاهده می‌شود در ثانیه‌ی سوم تقریباً به نقطه‌ی اوج خود با ۳۹۸ یال می‌رسد. (بنابر اینکه تنها یک نقطه داریم نمودار آورده نشده است و برای این بخش تنها شامل نقطه‌ی (۳، ۳۹۸) می‌شود).

در فایل crashes.txt نیز خطاهای کشف‌شده قرار دارد که همان‌طور که مشاهده می‌شود خطاهایی از نوع NullPointerException به دست آمده است که نشانگر دسترسی به متغیری خاص (از نوع SCRIPT\_TABLE\_PRIVATE\_DATA) با وجود Null بودن آن است.

### پروتکل (HII Database I2C\_Master)

توابعی از پروتکل I2C\_Master را فاز کرده‌ایم که جزئیات آن در زیر می‌آید. فایل‌های مرتبط به آن نیز در پوشه‌ای با همین نام قرار دارد. این فاز در حالتی که از تمامی ویژگی‌های harness استفاده شده است (fz) انجام شده است.

اعداد دقیق تعداد edgeها برای لحظاتی از اجرا، در فایل coverage.csv آورده شده است که مشاهده می‌شود در ثانیه‌ی سوم تقریباً به نقطه‌ی اوج خود با ۳۹۸ یال می‌رسد. (بنابر اینکه تنها یک نقطه داریم نمودار آورده نشده است و برای این بخش تنها شامل نقطه‌ی (۳، ۳۹۸) می‌شود).

در فایل crashes.txt نیز خطاهای کشف‌شده قرار دارد که همان‌طور که مشاهده می‌شود خطای خاصی به دست نیامده است و تنها پیغام‌هایی که ASAN تولید کرده است در این فایل قرار دارد.



## جمع‌بندی

در این گزارش به بخش‌های مختلف پروژه‌ی «عملیات fuzzing روی UEFI با استفاده از پلتفرم Simics و ابزار TSFFS» که در طی ترم انجام شد، پرداختیم. در بخش تئوری ۳ مقاله‌ی مرتبط بررسی شد و برخی بخش‌های آن‌ها که نیاز به توضیحات بیشتر داشت، توضیح داده شد. در بخش عملی نیز ابتدا به راه‌اندازی سیستم پرداختیم و سپس به فاز کردن پروتکل‌های تکراری و تازه پرداختیم که نتایج آن‌ها ارائه شده است. لازم به ذکر است مراحل مختلف این بخش‌ها به صورت توأمان انجام شده است و به سبب پیوستگی مطالب در بخش‌های جداگانه‌ای در این گزارش آورده شده است.