مجله علمی- پژوهشی "پدافند الکترونیکی و سایبری"
سال ششم، شماره ۱، بهار ۱۳۹۷، ص ۱۶-۱

مبهم‌سازی کد به منظور جلوگیری از اجرای نمادین

سعید پارسا[1*]، حمیدرضا صالحی[2]، محمد هادی علائیان[3]

۱- دانشیار، ۲- دانشجوی کارشناسی ارشد، ۳- دانشجوی دکتری، دانشگاه علم و صنعت ایران

(دریافت: ۹۴/۱۲/۲۲، پذیرش: ۹۵/۰۸/۱۰)

چکیده

امروزه حفاظت از نرم‌افزار در مقابل تحلیل‌گران به یکی از مسائل مهم عرصه رایانه تبدیل شده است. در این میان، روش اجرای نمادین به‌عنوان رویکردی برای کشف مسیرهای اجرایی و شروط وقوع برنامه، اخیراً مورد توجه قرار گرفته است. لذا، برنامه‌نویسان جهت محافظت از برنامه خود، مقابله با روش‌های تحلیل کد را نیز در نظر می‌گیرند. یک اجرای نمادین موفق، کلیه مسیرهای اجرایی برنامه به همراه شروط وقوع آن‌ها را در قالب یک درخت نمادین استخراج می‌کند. بنابراین، با جلوگیری از اجرای نمادین کد، می‌توان از یک برنامه حفاظت نمود و مسیرهای اجرایی آن را از دید تحلیل‌گران پنهان نگه داشت. در این میان، برخی روش‌ها با تکیه بر چالش‌های مختلف موجود در اجرای نمادین سعی بر پنهان‌سازی رفتار کد در مقابل اجرای نمادین دارند. در این مقاله، روشی جهت مبهم‌سازی شرط وقوع رفتارها در کد برنامه ارائه داده شده است تا درصورت تحلیل نمادین کد، کاربران از شروط واقعی یک رخداد مطلع نگردند. برای این منظور، یک راه‌کار نوین با اعمال معادلات خطی ارائه داده شده است. در این روش با جایگذاری برخی شروط غیرواقعی و جدید در مسیر برنامه و مرتبط‌نمودن متغیرهای آن با متغیرهای اصلی برنامه، حل‌کننده را دچار اشتباه می‌نماید. این امر موجب ایجاد شاخه‌های متعدد و غیرواقعی در درخت نمادین برنامه می‌گردد. لذا، تحلیل کد را دچار پیچیدگی می‌کند. ارزیابی‌ها نشان می‌دهد که این مبهم‌سازی، ابزار اجرای نمادین کد را در تولید همه مسیرهای اجرایی برنامه با شکست مواجه می‌کند.

واژه‌های کلیدی: اجرای نمادین، مبهم‌سازی کد، انفجار مسیر، درخت نمادین، تفسیر انتزاعی، تحلیل ایستای بازه

۱- مقدمه

اجرای نمادین به‌عنوان روشی برای تحلیل برنامه‌ها، نزدیک به سه دهه پیش معرفی شد [۱]. درسال‌های اخیر، به‌واسطه افزایش توان پردازشی سیستم‌ها، اجرای نمادین، پیشرفت زیادی نمود. این افزایش استفاده برای تحلیل برنامه‌ها باعث شد تا به‌عنوان روشی جهت تحلیل بدافزارها نیز مورد استفاده قرار گیرد. برای این منظور، عموماً اجرای نمادین در کنار روش‌های پویای تحلیل کد استفاده می‌گردد و همین مسئله باعث شده است تا روز به روز به روشی پخته‌تر و کامل‌تر بدل گردد. به طور خاص اجرای نمادین کارکرد خود را در تحلیل کدهای مبتنی بر ماشه[1] (که در اکثر موارد کدمخرب هستند) و یافتن شروط اجرای کد، نشان داد [۲].

به منظور مقابله با این نوع تحلیل، رویکردهای مختلفی مبتنی بر چالش‌های اجرای نمادین مطرح گردید. یکی از رویکردها با هدف مقابله با ابزارهای تحلیل بدافزار، ایجاد شروط تساوی در کد است. در این روش، ورودی‌های برنامه، با معرفی برخی توابع درهمساز[2] یک طرفه مبهم می‌شوند. این روش نشان می‌دهد که ابزارهای مبتنی بر تحلیل نمادین به سختی می‌توانند مقادیر ورودی‌ای که شروط تساوی دارند را شناسایی نمایند. البته این روش هم مشکلات زیادی دارد و در برخی حالات باعث ایجاد حساسیت در ابزارهای کشف کد بدخواه می‌گردد و به نوعی امضاء یا الگوی جدیدی برای کدهای بدخواه تولید می‌نماید [۳].

در این مقاله، با بررسی چالش‌های اجرای نمادین و روش‌های مبهم‌سازی، یک روش مبهم‌سازی کد با استفاده از ترکیب دو روش مبهم‌سازی ارائه می‌شود. اول روشی خطی با استفاده از ترکیب حل‌نشده ریاضی و دوم روشی بر مبنای تفسیر انتزاعی جهت مخفی‌سازی روش اول در مقابل ابزارهای تحلیل ایستای کد می‌باشد. استفاده از معادلات خطی حل‌نشدنی، به‌دلیل ماهیت خطی ریاضیاتی ابزارهای کشف بدافزار را حساس نمی‌نماید. بدین معنی که این ترکیبات یک سری معادله شفاف ریاضی بدون رفتار مبهم هستند و از این جهت ابزارهای کشف بدافزار به جریان رفتاری این معادلات حساس نخواهند شد.

* رایانامه نویسنده مسئول: parsa@iust.ac.ir

1- Trigger based

2- Hash functions

روش مبهم‌سازی ارائه‌شده در گام اول یک حلقه ساده محاسباتی به کد اضافه می‌نماید. اولاً، کد را به مقدار زیادی تغییر نمی‌دهد و دوماً، جریان اجرایی کد همانند همه کدهای معتبر و سالم دیگر باقی می‌ماند. سوماً، روش به واسطه ماهیت خطی خود جلوی اعمال روش‌های دیگر مبهم‌سازی را نمی‌گیرد. لذا می‌توان روش دوم مبهم‌سازی بر مبنای تفسیر انتزاعی را بر روی کد اعمال کرد. کد تولید شده نهایی از نظر حجم کمتر از پانصد بایت با کد اولیه تفاوت خواهد داشت. زیرا یک قطعه کد با مقدار حجمی ثابت بر روی تعدادی محدود از هر ساختار شرطی اعمال خواهد شد. البته روش تفسیر انتزاعی به عنوان گام دوم می‌تواند حجم کد را به مقدار خیلی زیاد یا کمی به صورت تصادفی تغییر دهد. این تغییر حجم در شرایط اعمال روش تفسیر انتزاعی مسئله منفی یا مثبتی نمی‌تواند تلقی شود چون میزان افزایش حجم کد در این مرحله معنای خاصی نمی‌دهد. این روش بر روی زبانی که ساختار شرطی دارد قابل اعمال است.

آن دسته از معادلاتی که راه‌حل نهایی و اثبات‌شده ریاضی ندارند را ترکیبات حل‌نشده ریاضی می‌گویند. این ت معادلات بر روی اعداد صحیح با اعمال مکرر برخی عملگرهای ساده خطی انجام می‌دهند [۴]. این معادله‌ها به صورت عام در الگوریتم‌های پایه علوم کامپیوتر نیز استفاده می‌شوند و اغلب به واسطه ساختار خطی از مرتبه O(n) هستند و سریع محسوب می‌شوند. از این‌رو، گزینه بسیار مناسبی برای مبهم‌سازی جهت مقابله با اجرای نمادین هستند. زیرا، عملیات اجرای نمادین معمولاً بر روی حلقه‌های بزرگ نامناسب و ضعیف عمل می‌نماید.

در ادامه روشی خودکار با استفاده از ترکیبات حل‌نشده و تزریق آن‌ها به روی شروط اجرایی برنامه، ارائه و پیاده‌سازی می‌شود. سپس به منظور جلوگیری از شناسایی کد تزریق‌شده و همچنین مقابله با تحلیل ایستای کد، روش مبهم‌سازی کد بر مبنای تفسیر انتزاعی بر روی آن اعمال می‌شود. محاسبات و ارزیابی‌ها نشان می‌دهد که ابزار تحلیل نمادین با صرف ساعت‌ها تحلیل نیز نمی‌تواند شرط اجرای رفتار مورد نظر را به درستی تشخیص دهد. همچنین، کد تزریق‌شده به واسطه تغییر محدوده شروط توسط تفسیر انتزاعی به صورت تصادفی کد را به‌هم می‌ریزد تا الگوی آن همواره یکسان نباشد. از طرفی، این مبهم‌سازی جدید تاثیر منفی بر روی مبهم‌سازی مرحله اول نخواهد داشت.

بخش‌های بعدی این مقاله به صورت زیر خواهند بود. در بخش دوم به پیش‌زمینه اجرای نمادین، معرفی برخی ترکیبات حل‌نشده، مفاهیم تفسیر انتزاعی و در نهایت کارهای مرتبط پرداخته می‌شود. در بخش سوم روش پیشنهادی و ترکیب دو روش مبهم‌سازی همراه با یک مثال ارائه می‌گردد. در بخش چهارم، الگوریتم پیاده‌سازی توضیح داده می‌شود و در نهایت نتایج به‌دست‌آمده از مبهم‌سازی در بخش پنجم ارائه خواهند شد. در بخش نهایی، نتایج حاصل‌شده مورد بررسی قرار خواهند گرفت.

۲- پیش‌زمینه و کارهای مرتبط

در این بخش، به‌طور خلاصه به معرفی اجرای نمادین و چالش‌های آن پرداخته خواهد شد و سپس روش‌های مبهم‌سازی کد که در مقابله با اجرای نمادین تمرکز کرده‌اند، مورد بررسی قرار خواهند گرفت. در نهایت ترکیبات حل‌نشده معرفی خواهند شد.

۲-۱- اجرای نمادین و کاربردهای آن

اجرای یک برنامه با استفاده از ورودی‌های نمادین و عملیات نمادین به‌جای ورودی‌های واقعی و اجرای واقعی برنامه را اجرای نمادین[1] می‌گویند. در این روش به ازاء هریک از ورودی‌های برنامه، یک ذخیره نمادین در نظر گرفته می‌شود. در اجرای نمادین، مقادیر متغیرهای برنامه در هر نقطه در یک مسیر از برنامه، با عبارات جبری نشان داده می‌شود. حالت نمادین[2] در یک نقطه خاص از یک برنامه، شامل مجموعه مقادیر نمادین برای همه متغیرهای نمادین در آن نقطه است. به مجموعه از محدودیت‌هایی[3] که متغیرها باید داشته باشند تا در یک مسیر از برنامه قرار بگیرند، شرایط مسیر[4] گفته می‌شود. با اجرای هر مسیر از برنامه، چندین شرط برنامه اجرا می‌شود. بسته به صحیح بودن و یا غلط بودن مقدار شرط، مسیر اجرای برنامه مشخص می‌گردد. اجرای نمادین به طور گسترده در بسیاری از روش‌های تحلیل امنیتی مورد استفاده قرار گرفته است [۵]. آزمون خودکار به منظور دست‌یافتن به پوشش همه مسیرهای ممکن و تولید داده خودکار از اجرای نمادین استفاده می‌کند. اکثر این موارد به منظور تولید ورودی برای تحریک و بررسی شروط وقوع خطا نظیر سرریز اعداد صحیح، خطای حافظه و اشاره‌گرهای سرگردان استفاده می‌شوند. در تحقیقات بعدی دیده شد که اجرای نمادین توانایی خوبی در تولید داده‌های دقیق و مناسب جهت بررسی بلاک‌های کد را داراست [۲، ۷-۶].

1- Symbolic Execution
2- Symbolic State
3- Constraint
4- Path Condition

۲-۲- چالش‌های اجرای نمادین

اجرای نمادین کد با چالش‌های متفاوتی روبرو است. البته چالش‌های موجود عموماً بر روی ابزارهای موجود اجرای نمادین بررسی شده‌اند که البته همگی زمینه تئوریک نیز دارند. از آن‌جا که روش اجرای نمادین پویا عمر چندانی ندارد، هنوز ابزارهای کامل بر اساس اجرای نمادین پویا جهت استفاده‌های تجاری وجود ندارند. عموم ابزارهای موجود نظیر Fuzzgrind [۸]، KLEE [۹]، EXE [۱۰]، DART [۱۱] و CUTE [۱۲] تنها برای اهداف تحقیقاتی تولید شده‌اند. ابزار SAGE یک ابزاری صنعتی است که توسط شرکت مایکروسافت تولید شده است، اما در خارج از آن شرکت در دسترس نیست. در ادامه به اختصار پنج چالش اجرای نمادین مطرح می‌شود و در نهایت بر روی چالش انفجار مسیر بررسی بیشتری خواهد شد.

توابع محاسباتی خارجی/پیچیده: در هنگام مقابله با توابع محاسباتی خارجی/پیچیده، اجرای نمادین پویا[1] بهتر از تولید نمونه ایستا عمل می‌نماید. اگرچه، توابع محاسباتی خارجی/پیچیده خارج از حیطه استدلال ثابت‌کننده تئوری[2] هستند، اجرای نمادین پویا از این مسئله در بسیاری حالات رنج می‌برد. برای نمونه، در قطعه کد زیر اگر خط سوم را با عبارت if (hash1(x) == hash2(y)) abort(); عوض کنیم، در این‌جا $hash1$ و $hash2$ دو تابعی هستند که نمی‌توان در مورد آن‌ها بحث و استدلال کرد. از همین رو، اجرای نمادین پویا قادر به تضمین کشف این خطا نیز نیست [۱۳].

```
1: void example2 (int x, int y)
2: {
3:         if (x == hash(y)) abort (); // error()!
4: }
```

شکل(۱): مثالی از چالش توابع محاسباتی پیچیده

محکم‌کاری[3] یک روش توافقی برای فرار از توابع محاسباتی خارجی/پیچیده است. محکم‌کاری شامل استفاده از مقادیر به‌هم‌پیوسته ورودی برای جایگزین‌نمودن مقادیر نمادین است که این توابع را به‌طور طبیعی بدون گردآوری محدودیت‌ها، اجرا می‌کنند. اگرچه محکم‌کاری در بعضی موارد خاص درست و قابل‌قبول است، درعین‌حال نتایج منفی کاذب[4] را به‌خوبی تخمین نمی‌زند.

محاسبات ممیز شناور: بسیاری برنامه‌ها بر روی رایانه‌های مدرن از قبیل رمزگشا/رمزگذارها[5]، نمایشگرهای تصویر[6] و پخش‌کننده‌های رسانه[7] تمایل به استفاده از دستورات ممیز شناور موجود دارند. ابزارهای امنیتی موجودی که بر اساس اجرای نمادین پویا هستند، هنوز محاسبات با ممیز شناور را پشتیبانی نمی‌کنند. دلیل عمده آن است که حل‌کننده‌های محدودیت، قادر به تحلیل محاسبات ممیز شناور نیستند. ابزارهای تحلیل‌گر در هنگام برخورد با دستورات ممیز شناور به‌منظور اجرای عادی برنامه نمونه، مقادیر نمادین را با مقادیر محکم (قطعی) جایگزین می‌نمایند [۱۴].

عملگر اشاره‌گر نمادین: یک عملگر اشاره‌گر نمادین بیانگر ارجاع یک اشاره‌گر به آدرسی است که مقدار آن به محاسبه یک عبارت نمادین (برخی ورودی‌های غیرقابل اطمینان) بستگی دارد. یک راه‌حل سرراست در مقابل این مسئله محکم‌کاری است. بدین معنی که هرگاه یک ابزار اجرای نمادین نمی‌داند که چگونه یک محدودیت نمادین برای یک عبارت برنامه که شامل عملیات اشاره‌گر نمادین است، تولید کند، متغیرهای نمادین با مقادیر قطعی جایگزین می‌شوند [۱۵].

تعاملات محیطی: از آن‌جایی‌که بسیاری از برنامه‌ها از نظر محیطی فشرده و متمرکز هستند، اجرای نمادین پویا از چالش‌های تحلیل کدی که با محیط پیرامون از قبیل سیستم عامل، شبکه و یا کاربر تعامل می‌کند، رنج می‌برد. در حقیقت نمونه آزمایشی اجرای برنامه نه تنها به ورودی کاربر، بلکه به محیط پیرامون آن نیز بستگی دارد. به همین سبب، اگر نتوان محیط اجرایی را به‌درستی شبیه‌سازی نمود، ممکن است به مسیرهای اجرایی که به محیط وابسته هستند، نرسید. درنتیجه، میزان پوشش کد برنامه‌های محیطی متمرکز که توسط اجرای نمادین پویا تحلیل می‌شوند بسیار پایین است.

انفجار مسیر: در نهایت، مهم‌ترین و فراگیرترین چالش یعنی انفجار مسیر است. مسیرهای اجرایی عملی برنامه با افزایش طول مسیر اجرا به‌طور نمایی افزایش می‌یابد. یکی از مکان‌هایی که انفجار مسیر در آن وجود دارد، پیمایش حلقه‌های تکرار طولانی است. اکثر روش‌های اجرای نمادین و طبع آن ابزارهای موجود هنگامی که حلقه تکرار با برخی متغیرهای نمادین همراه می‌شود در پیمایش و حل محدودیت‌های آن دچار مشکل می‌گردند. به

1- DSE – Dynamic Symbolic Execution
2- Theorem Prover
3- Concretization
4- False Negative
5- CODECs – COder/DECoder
6- Image Viewers
7- Media Player

طور معمول، این ابزارها به واسطه بروز مسئله کمبود حافظه و منابع تنها تعداد دفعات محدودی را صرف بررسی حلقه‌ها می‌نمایند، روش‌های زیادی جهت بهبود و اصلاح مسئله انفجار مسیر در اجرای نمادین مطرح شدند که هیچ‌کدام راه‌حل قطعی و جامعی نبودند[۱۷-۱۶]. این چالش باعث می‌شود که ابزارها در هنگام پیمایش مسیرهای اجرایی برنامه‌ها درخت‌هایی با عمق بسیار طولانی تولید نموده و عملاً دچار کمبود حافظه یا صرف زمان بالا در تحلیل گردند.

۲-۳- مبهم‌سازی کد

یکی از اولین هدف‌های مبهم‌سازی کد، مقابله با تحلیل ایستا است. تحلیل ایستا با بررسی و پیمایش خط به خط کد برنامه، سعی در استخراج رفتارهای احتمالی کد دارد. به این کد اجرایی یا باینری از آن‌جا که تعداد دستورات محدودی دارد و همچنین بر روی زبان ماشین مقصد ترجمه می‌شود برای مقاصد تحلیل مناسب و سریع هستند [۲۱-۱۸]. یکی از تلاش‌های معروف جهت مبهم‌سازی کد اجرایی تبدیل یا تغییر کد است [۲۲]. بعدها افراد دیگری با جابه‌جایی دستورالعمل‌های پرش با برخی کدهای خطادار سعی در انجام این کار داشتند [۱۹].

۲-۴- مبهم‌سازی کد جهت مقابله با اجرای نمادین

همه این روش‌های بالا روی یک مسئله تمرکز داشتند و آن تحلیل ایستای کد بود. اما مسئله این‌جاست که اجرای نمادین به کیفیت نوشتار یا میزان به‌هم‌ریختگی آن کاری ندارد. رویکرد اجرای نمادین حل محدودیت‌ها و استخراج مسیرهای اجرایی درست یا غلط است. هدف ما نیز مقابله با اجرای نمادین است که تفاوت اصلی این روش مبهم‌سازی با دیگر روش‌ها است. یکی از روش‌ها، مبهم‌سازی انشعاب نام داشت. دستورات پرش شرطی دودویی[1]، اطلاعات مسیر برنامه را در زمان اجرا فاش می‌سازند. به همین دلیل، به‌سادگی می‌توان توسط اجرای نمادین محدودیت‌های مسیر دنباله اجرای برنامه را استخراج نمود. روش مبهم‌سازی انشعاب با استفاده از آثار جانبی کد اجرایی یا دودویی، دستورات پرش شرطی را با دستوراتی که آثار جانبی دارند عوض می‌نماید. اگر در محیط اجرای فعلی، محدودیت‌های برنامه سیر اصلی را ارضاء نماید، تأثیرات جانبی دستورات منجر به بروز استثناء[2] خواهند شد. در غیر این صورت، برنامه به‌صورت ترتیبی و بدون بروز استثناء به اجرا ادامه می‌دهد. به‌منظور تحلیل، این استثناءها توسط سیستم‌عامل به عامل ثبت‌کننده ابزار اجرای نمادین فرستاده می‌شود [۲۳].

در ادامه، یک نوع روش‌پویا برمبنای درخواست اجرای نمادین معرفی شد [۲۴]. مزیت این روش افزایش سختی مهندسی معکوس، به‌وسیله گمراهی در زمان اجرا بود. در همین راستا، یک الگوریتم مبهم‌سازی پویا برمبنای درخواست[3] مطرح شد که بر تئوری تحلیل نمادین تکیه می‌کند. شیوه کار به این صورت است که در ابتدا تعدادی مسیر نامعتبر ایجاد می‌شود تا نتیجه اجرای نمادین را منحرف سازد. سپس برمبنای تئوری مبتنی بر درخواست یک مسیر خاص ایجاد می‌شود تا از امنیت نرم‌افزار محافظت نماید.

شریف[4] و همکاران [۲۵]، کدهای وابسته به ورودی، جهت حمله به اجرای نمادین را مبهم‌سازی کردند. ایده روشی که معرفی خواهیم نمود نیز از همین کار دریافت شده است. البته یک مسئله وجود دارد که مبهم‌سازی کد می‌تواند باعث بروز شک در تحلیل‌گران گردد. در ادامه نشان داده می‌شود که روش به واسطه فرایند خطی ساده، شک‌برانگیز نیست. البته جهت حفاظت هرچه بیشتر در مقابل کشف با یک روش تصادفی و فرمال برمبنای تفسیر انتزاعی، کد دوباره مبهم‌سازی می‌شود تا باعث ایجاد یک خانواده از کدهای یکسان نشود.

۲-۵- ترکیبات حل‌نشده

ترکیب کلاتز [۱۶] در شکل (۲) نشان داده شده است که با عنوان ترکیب 3x+1 نیز شناخته می‌شود. با شروع از هر عدد صحیح مثبت و تکرار حلقه، در نهایت، عدد ۱ را تولید می‌کند. این ترکیبات و انواع آن به‌آسانی بیان می‌گردند و به‌سختی اثبات می‌شوند. قبلاً اثبات شده است که ترکیبات نظیر 3x+1 تصمیم پذیرند. یعنی می‌توان الگوریتمی ارائه داد که به ازای ورودی خاص همیشه به یک جواب درست بله یا خیر برسد. در این ترکیب خاص دیده شده است که در اعداد کوچک‌تر از $20 * 2^{58}$ نتیجه همواره به ۱ ختم می‌شود [۶ و ۲۷].

(۱) ترکیب 7x+1 :

$$f(n)=\begin{cases} n/2 & \text{if } n\equiv 0 \pmod 2 \\ n/3 & \text{if } n\equiv 0 \pmod 3 \\ n/5 & \text{if } n\equiv 0 \pmod 5 \\ 7n+1 & \text{else} \end{cases}$$

(۲) ترکیب 5x+1:

$$f(n)=\begin{cases} n/2 & \text{if } n\equiv 0 \pmod 2 \\ n/3 & \text{if } n\equiv 0 \pmod 3 \\ 3n+1 & \text{else} \end{cases}$$

(۳) دنباله جاگلر:

$$a_i=\begin{cases} \lfloor a_{i-1}^{1/2} \rfloor & \text{if } a_{i-1}\equiv 0 \pmod 2 \\ \lfloor a_{i-1}^{3/2} \rfloor & \text{if } a_{i-1}\equiv 1 \pmod 2 \end{cases}$$

(۴) ترکیب متیوز:

$$f(n)=\begin{cases} 7n+3 & \text{if } n\equiv 0 \pmod 3 \\ (7n+2)/3 & \text{if } n\equiv 1 \pmod 3 \\ (n-2)/2 & \text{if } n\equiv 2 \pmod 3 \end{cases}$$

شکل (۲): مثال‌هایی از ترکیبات حل‌نشده

1- Binary Conditional Jump Instructions
2- Exception
3- Request-Driven Dynamic Obfuscation
4- Sharif

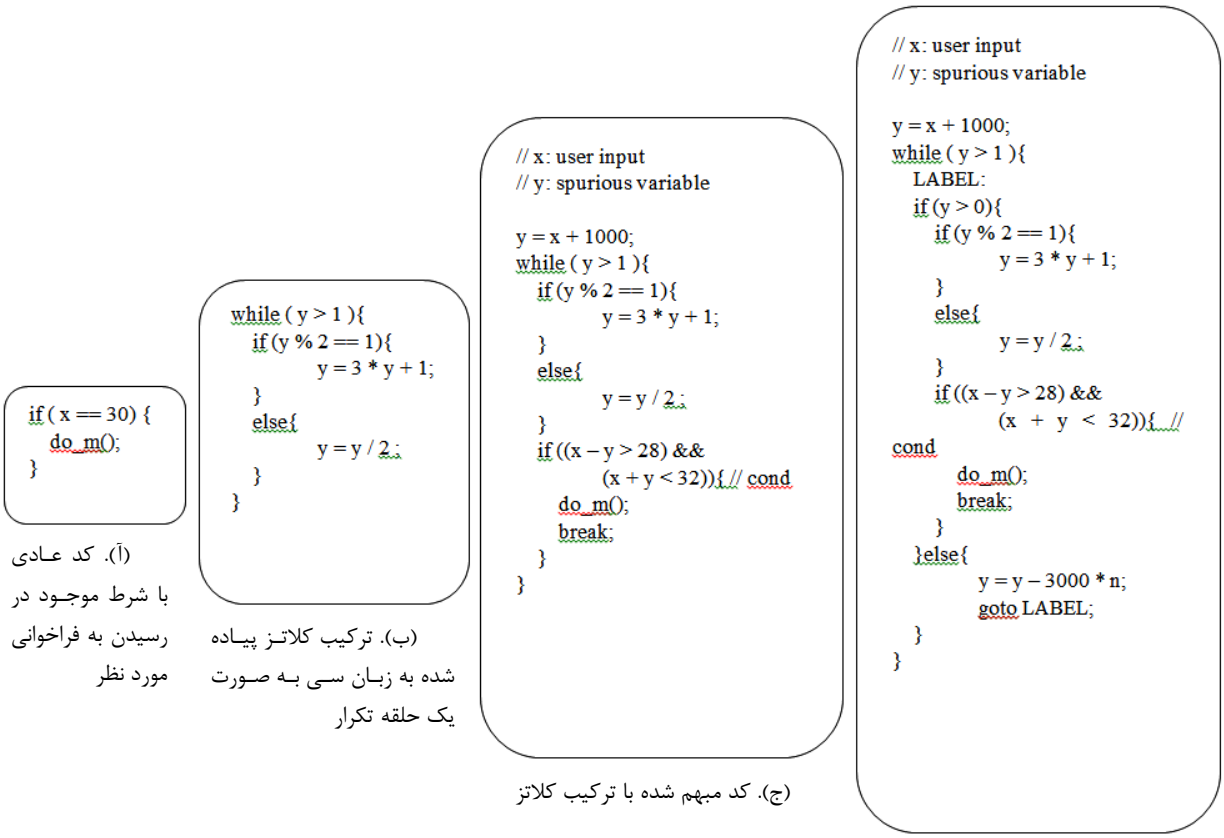
هم‌گرایی همه ترکیبات ارائه شده در شکل (۲) به یک ختم می شوند به جز ترکیب متیوز (معادله (۴)) که به صفر ختم می‌شود. البته دنباله جاگلر (معادله (۳)) نیز در تولید اعداد اعشاری با ممیز شناور نیز کاربرد دارد لذا گزینه خوبی برای استفاده در چالش محاسبات ممیز شنارو است.

۳-طرح پیشنهادی

این روش مبهم‌سازی، اجرای نمادین کد را به‌وسیله ایجاد یک متغیر اضافی وابسته به متغیر ورودی و حلقه‌ای از ترکیبات حل‌نشده، مشکل و پیچیده می‌نماید. متغیر اضافه‌شده بر روی گراف جریان کنترلی برنامه به شرط فعال‌کننده رفتار برنامه بستگی دارد. درنتیجه، متغیر ورودی باید به‌عنوان نماد در اجرای نمادین در نظر گرفته شود. حلقه‌ای که توسط ترکیبات حل‌نشده ایجاد شده است، به محل شرط فعال‌کننده به جریان کنترلی افزوده می‌شود. این حلقه تعداد بسیار زیادی مسیر ممکن اجرایی که به‌طور نمایی رشد می‌کند به اجرای برنامه اضافه می‌کند که موجب افزایش هزینه زمانی اجرای نمادین جهت فعال‌سازی شرط می‌شود. لازم به توجه است که افزودن متغیر ورودی جعلی و ترکیبات حل‌نشده، همانند بسیاری از روش‌های مبهم‌سازی موجود، رفتار برنامه را مخفی نمی‌نماید. لذا، تلاش بر این است کد برنامه را تحت شرط فعال‌کننده آن و نه رفتار آن مخفی شود. از همین رو، همچنان دو مسئله باقی مانده است. اول عیان‌ماندن رفتار که احتمالاً رفتار بدخواه است و دوم احتمال مشکوک‌شدن تحلیل‌گران به قطعه کد اضافه شده است. اما مزیت این روش تا این مرحله در این است که به واسطه فرایند خطی و همچنین عدم ترکیب‌شدن با رفتار اصلی، راه را برای اعمال دیگر روش‌های باز می‌گذارد. از همین رو، در تکمیل مرحله اول، مبهم‌سازی برمبنای تفسیر انتزاعی مطرح می‌شود که دو مزیت اصلی دارد. اول به‌واسطه تصادفی‌بودن الگوریتم، یک الگوی تکراری قابل شناسایی به هر نحوی به کد نمی‌افزاید. همچنین ساختار کلی کد را به نحوی مبهم می‌نماید که تحلیل و خوانایی آن به شکل قابل ملاحظه‌ای کاهش می‌یابد. از این‌رو، هم رفتار و هم مبهم‌سازی مرحله اول بر روی شرط اجرایی رفتار به طور تصادفی مبهم‌تر خواهند شد.

۳-۱- تشریح روش

در ادامه به کمک یک مثال عملی بر روی زبان C، روش تشریح خواهد شد. این نمایش صرفاً جنبه نظری داشته و پیاده‌سازی الگوریتم بر روی کد اسنبلی انجام شده است. شکل (۳) یک مثال عملی که از ترکیب خطی حل‌نشدنی کلاتز یا 3x+1 استفاده شده است را نشان می‌دهد. قسمت (الف) شکل، کدی ساده را نشان می‌دهد که تابع do_m() یک رفتار از برنامه را داراست و با شرط فعال‌سازی x==30 فراخوانی خواهد شد. در قسمت (ب) قطعه کد به همراه یک حلقه مشاهده می‌شود. در بدنه حلقه مقدار متغیر y براساس شرط موجود به‌روزرسانی می‌گردد. این قطعه کد، به‌واسطه تولید نمایی مسیر اجرایی جدید بسیار سخت توسط اجرای نمادین تحلیل می‌گردد. حال اگر بخواهیم کد قسمت (الف) را به‌وسیله یک متغیر ورودی جعلی نشان داده‌شده در قسمت (ب) مبهم‌سازی نماییم، مشاهده می‌شود که شرط فعال‌ساز کد برنامه دیگر ایستا نبوده و به یک متغیر جعلی بستگی دارد که مقدار آن متغیر نیز به تعداد دفعاتی که حلقه اجرا می‌گردد بستگی دارد. قسمت (ج). در مرحله بعدی با تفسیر انتزاعی کد، شروط و بازه‌های محتمل آن تحلیل‌شده و مبهم-سازی کد با استفاده از الگوریتم تفسیر انتزاعی، یک شرط جدید که همراه پرش به محل است به کد افزوده می‌شود. (د). البته مشکلات خاصی نظیر این‌که چگونه باید مطمئن شد که معنا و عملکرد برنامه بعد از مبهم‌سازی، تغییر نمی‌کند وجود دارد که در مورد آن، در ادامه بحث خواهد شد. به‌عبارت دیگر، اگرچه اجرای نمادین به‌سختی می‌تواند عملکرد برنامه و همه مسیرهای اجرایی آن را پیمایش کند، آیا برنامه‌سازان قادر به فهمیدن آن کد هستند؟ با توجه به ترکیبات حل‌نشده که در بالا توضیح داده شد، پاسخ مثبت است. یعنی ترکیب حل‌نشده تضمین می‌کند که بدون در نظر داشتن ورودی مقدار ترکیب، همواره به ۱ ختم می‌شود. در مرحله نهایی وظیفه مبهم‌سازی به عهده تفسیر انتزاعی است که در این مثال خاص، به‌واسطه کوچک‌بودن مسئله و کم‌بودن تعداد متغیرها عملاً دست‌کاری بیشتر در کد مقدور نیست. این مسئله در مثال‌های واقعی و بزرگ کاملاً متفاوت بوده و ساختار کد به میزان زیادی مبهم می‌شود. البته حتی در این شرایط نیز الگوی تولیدشده در مرحله سوم متفاوت خواهد بود.



(آ). کد عادی با شرط موجود در رسیدن به فراخوانی مورد نظر

(ب). ترکیب کلاتز پیاده شده به زبان سی به صورت یک حلقه تکرار

(ج). کد مبهم شده با ترکیب کلاتز

(د). مبهم‌سازی نهایی با اعمال روش مبهم سازی مبتنی بر تفسیر انتزاعی

شکل (۳): مثالی از مبهم‌سازی خطی

۳-۲- مبهم‌سازی بر مبنای تفسیر انتزاعی

تحلیل ایستای بازه، اطلاعات زیادی از سطح کد در اختیار روش مبهم‌سازی قرار می‌دهد. از این اطلاعات می‌توان در تولید شرط استفاده کرد [۲۸]. تحلیل ایستای بازه سعی دارد تا بازه متغیرهای برنامه را در هر خط از سطح کد به دست آورد. به مجموعه همه این بازه‌های متغیرها محیط گفته می‌شود و هر خط از برنامه یک حالت در نظر گفته می‌شود.

سعی بر آن است تا با استفاده از روش تفسیر انتزاعی، کد برنامه تحلیل شده و مقدار بازه‌ای متغیرهای برنامه (به همراه مقادیر ثابت‌ها) ایجاد گردد تا بتوان انتزاع کد را افزایش داد. انتزاع کد وقتی افزایش میباند که تحلیلگر انتزاعی نتواند بازه دقیق یک متغیر را محاسبه کند و یا به عنوان مثال بازه $[-\infty,\infty]$ حاصل گردد. چون مقدار بازه اصلی تغییر می‌کند. تمامیت یا قطعی بودن مقدار از بین می‌رود. همچنین، از صحت مقدار بازه حاصل شده کاسته می‌شود.

مبهم‌ساز انتزاعی در ابتدا گراف جریان کنترلی کد را استخراج می‌کند. پس در هر قسمت از برنامه مقدار ثبات eax یک مقدار بازه‌ای $[l, h]$ به خود اختصاص می‌دهد که در بعضی مواقع مقدار h با مقدار l می‌تواند یکسان باشد. در این صورت، با استفاده از این بازه می‌توان شرط‌های مختلفی از جمله " jge eax, l" را تولید کرد. برای گمراه‌کردن کاربران بدخواه و همچنین تحلیلگر ایستا، باید بلاک اولیه‌ای تولید شود که نقیض معنایی شرط را تولید کند. یعنی بلاک اولیه‌ای تولید شود که eax در آن مقداری کمتر از l را به خود بگیرد و آن را ورودی به بلاک شرط تولید قرار دهد. در این صورت، تحلیلگر ایستای بازه مقدار l' را به‌جای l در محیط مربوط به پرش تقریب خواهد زد که انتزاع بیشتر و صحت کمتری را خواهد داشت. در این صورت، تمامیت نیز از بین خواهد رفت. اگر مسیری که شرط، پرش را انجام می‌دهد را در نظر بگیرید، از آن‌جایی‌که همواره eax در بازه $[l, h]$ است، پس پرش همواره انجام خواهد شد. مسیر نادرست شرط برای گمراه‌کردن کاربران بدخواه و همچنین، تحلیلگر ایستای بازه است. با اتصال آن به بلاک اولیه نقیض معنایی شرطی دیگر که به

همین صورت ایجاد شده است، انجام خواهد شد. این عمل باعث می‌شود تا جای‌جای کد، به‌هم وابستگی پیدا کرده و پیچیدگی برنامه را بالا ببرد. این عمل چندین بار تکرار می‌شود تا شرط نهایی، خروجی از نقیض معنایی شرط دیگری شود. یعنی بخواهید شرطی را در بین یک شرط و نقیض معنایی‌اش قرار دهید.

۳-۳- تولید شرط از روی محیط

برای تولید شرط جهت استفاده در مبهم‌ساز انتزاعی، نیاز است تا محیط شرط با استفاده از تحلیلگر انتزاعی مورد تحلیل قرار بگیرد تا مقدار بازه‌ای متغیرهای برنامه، ثبات‌ها و فلگ‌ها حاصل گردد. اگر دستورات پرشی را در زبان ماشین مورد بررسی قرار دهید[۲۹]، هر پرش برای انجام، یک‌سری شرایط را باید داشته باشد. پس محیط‌های لازم برای اعمال پرش را باید شناسایی کرد. به‌عنوان نمونه، اگر یک حالت در محیط داشته باشید که متغیرهای $OF = 1$ و $SF = 1$ باشد، در این صورت، می‌توان برای اضافه‌کردن پرش در گراف جریان کنترلی از نوع‌های پرش موجود در JNL, JGE و یا JO استفاده کرد. با استفاده از این اطلاعات برروی هر محیطی می‌توان متناسب با مقدارهایی که دارد، پرش‌های شرطی تولید کرد.

جهت تولید شرط، ابتدا به‌صورت تصادفی از روی گراف جریان کنترلی یک محیط انتخاب می‌شود. در مرحله بعد متناسب با نوع محیط، هر پرشی که ممکن است در این محیط قرار بگیرد، انتخاب می‌شود. از روی این لیست پرش‌هایی که می‌توانند در این محیط قرار بگیرند، یکی به‌صورت تصادفی انتخاب شده و به‌عنوان شرط بین دو بلاک اولیه مرتبط با محیط قرار می‌گیرد.

۴-۳- تولید مسیرهای گمراه‌کننده

یک دنباله متناهی از دستورات رادارید که به‌صورت متوالی A1→A2→A3 در برنامه اصلی اجرا خواهند شد. فرض کنید با انجام تحلیل ایستای بازه مشاهده می‌شود که بعد از دستور A1 فلگ OF برابر بازه [۱،۱] و بعد از دستور A2 فلگ CF برابر بازه [۰،۰] است. در این صورت می‌توان نتیجه گرفت JO می‌تواند یکی از پرش‌های شرطی که همواره از مسیر درست بین A1→A2 عبور می‌کند، باشد. همچنین، در مسیر A2→A3 نیز می‌تواند JNC را قرار داد. ملاحظه می‌شود که مسیر اصلی برنامه تغییر نکرده است، چون A1 به JO تغییر مسیر داده است. مقدار AF همواره برابر ۱ است پس می‌توان نتیجه گرفت که JO همواره به A2 می‌رود. همین‌طور برای پرش JNC را نیز می‌توان اعمال کرد. برای این‌که تحلیلگر ایستای بازه، تقریب بالاتری داشته باشد، بلاک اولیه‌ای به نام C1 تولید می‌شود که در آن، نقیض معنایی $OF = 1$ یعنی $OF \neq 1$ را تولید می‌کند تا تحلیلگر، نتیجه تحلیل را برای ورودی به پرشJO، مقدار OF را [۰،۱] به دست آورد. پس در تحلیل با اعمال تغییرات در همه مسیرها، تحلیلگر ایستای بازه یک تقریب بالاتر و یا نادرستی را نسبت به نتیجه اصلی خواهد داشت و درنتیجه، مجبور به تحلیل در هر دو مسیر شرط خواهد شد و از آن‌جایی‌که همواره مجبور به تولید حلقه خواهد شد و از آن‌جایی‌که تحلیلگر ایستا خواستار تقریب حلقه است، پس از صحت اولیه حتماً کاسته خواهد شد.

تولید نقیض شرط حائز اهمیت است. از طرفی، نباید همواره یک دستور خاص باشد زیرا این عمل کمک به شناسایی زودتر روش می‌کند. بهترین عمل، استفاده از انتخاب‌های تصادفی است. زیرا از خود هیچ الگویی نشان نمی‌دهند. برای تولید نقیض معنایی به‌صورت تصادفی یک مجموعه دستورات به‌صورت متوالی تولید می‌گردد. این دستورات توسط تحلیلگر ایستا، تحلیل می‌شوند و هر کدام که نقیض معنایی را تولید کرد، می‌تواند مورد استفاده قرار گیرد.

۴- پیاده‌سازی

همان‌طور که در شکل (۳) مشاهده می‌شود، روش کار به‌صورت یک قطعه کد به زبان C توضیح داده شده است. حال به دنبال پیاده‌سازی این روش تحت زبان اسمبلی و روی کد ماشین هستیم. منظور از این کار، اولاً آزمایش بهتر کد خواهد بود زیرا ابزار ما که در ادامه معرفی خواهد شد بر روی کد اجرایی، تحلیل و اجرای خود را انجام می‌دهد. از طرفی، کد ماشین قابلیت بهتری در تحلیل در ابزارهای معروفی نظیر IDA PRO را دارا است. شیوه اجرا به این صورت خواهد بود که فایل متنی حاوی کد اسمبلی برنامه به‌عنوان ورودی به ابزار مبهم‌ساز داده می‌شود. سپس، ابزار مورد نظر با پیمایش کد اقدام به شناسایی محل‌های با پتانسیل بیشتر جهت مبهم‌سازی می‌کند. این محل‌ها به‌سادگی جاهایی هستند که برنامه با دو یا بیشتر شاخه مجزا تقسیم می‌شود. مثلاً محل‌های وجود شرط یا حلقه‌ها و همچنین محل‌هایی که توابع یا رفتارهای مورد نظر تحت شرطی خاص قرار خواهند گرفت. از طرفی، کلیه شروط که وابسته به بررسی صحت متغیرهای صحیح ورودی هستند در میان دیگران از بیشترین پتانسیل برخوردار خواهند بود. در این مرحله با انتخاب بین این محل‌ها کد مورد نظر درون کد اصلی تزریق می‌گردد. درنهایت، نوبت به مبهم‌سازی برمبنای تفسیر انتزاعی می‌رسد که کد از نو پیماش‌شده و گراف جریان کنترلی آن استخراج می‌گردد. بر روی گراف جریان کنترلی تحلیل ایستای بازه با همان تفسیر انتزاعی برای به‌دست آوردن اطلاعات، مقدار بازه متغیرهای برنامه در هر

خط از کد اعمال می‌شود تا محیط حالت‌های برنامه محاسبه شوند. درنهایت، برخی محل‌های مناسب به‌طور تصادفی انتخاب‌شده و الگوریتم تفسیر انتزاعی بر روی آن اعمال می‌گردد.

۴-۱- پیاده‌سازی روش روی یک برنامه

به‌منظور آزمایش روش پیشنهادی، ساده‌ترین برنامه ممکن برای اجرای نمادین بررسی خواهد شد تا میزان تأثیر اعمال روش مبهم‌سازی بر روی برنامه سنجیده شود. برنامه به‌صورت یک شرط ساده درست و غلط است که با دریافت یکی ورودی از اعداد صحیح تصمیم می‌گیرد که کدام یک از مسیرهای برنامه باید طی شود. به‌طور دقیق‌تر، این برنامه تنها دو مسیر اجرایی خواهد داشت که بنابر مقدار اولیه ورودی مشخص می‌شوند. ورودی که باعث وقوع شرط صحیح شود به‌طور ساده یک تابع را که نماد رفتار مورد نظر ماست فراخوانی می‌نماید و آن ورودی که باعث شرط غلط شود، برنامه را خاتمه خواهد داد. کد برنامه در شکل (۱۱) آمده است. مرحله اول مبهم‌سازی با اجرای الگوریتم ۱ انجام می‌گیرد. این الگوریتم بر روی قطعه کد شکل (۵) انجام می‌گیرد.

Algorithm 1. SymbolicObfuscation(S)

Input: assembly source code S
INS: list of instructions.
LABELS: list of labels.
INP_VARS: list of variables
OUT_INS: output instructions
Output: obfuscated code

```
1.  Input = splitSourceLineByLine()
2.  for each line in input do
3.  instruction = extractInstruction(line)
4.  push instruction into INS
5.      if (instruction is label)
6.          label =  extractLabel(line)
7.          push label into LABELS
8.  for each line in input do
9.      instruction = extractInstruction(line)
10.     if (instruction is label and operation is input call)
11.         var = extractVariable(line)
12.         push var into INP_VARS
13.     else
14.          extract functions
15.
16. // code analysis is ended now obfuscation starts
17. for each line in input do
18.     ins = extractInstruction(line)
19.     if (ins is compare stmnt and variable is input)
20.         injectCode()
21.     else
22.          push ins into OUT_INS
23.
24.  return  OUT_INS
```

شکل (۴): الگوریتم مبهم‌سازی جهت جلوگیری از اجرای نمادین

```
1       .intel_syntax noprefix
2       .section    .rodata
3   .LC0:
4       .string "YES CONDITION"
5       .text
6       .globl  do_job
7       .type   do_job, @function
8   do_job:
9       push    ebp
10      mov ebp, esp
11      sub esp, 24
12      mov DWORD PTR [esp], OFFSET FLAT:.LC0
13      call    puts
14      leave
15      ret
16      .size   do_job, .-do_job
17      .globl  main
18      .type   main, @function
19  main:
20      push    ebp
21      mov ebp, esp
22      and esp, -16
23      sub esp, 32
24      mov eax, DWORD PTR [ebp+12]
25      add eax, 4
26      mov eax, DWORD PTR [eax]
27      mov DWORD PTR [esp], eax
28      call    atoi
29      mov DWORD PTR [esp+28], eax
30      cmp DWORD PTR [esp+28], 30
31      jne .L3
32      call    do_job
33  .L3:
34      mov eax, 0
35      leave
36      ret
37      .size   main, .-main
```

شکل (۵): کد خام ورودی مبهم نشده

در این الگوریتم، بعد از دریافت فایل حاوی کد اسنبلی، دستورات به‌وسیله تشخیص علائم انتهایی خط و new line تشخیص داده می‌شوند. سپس، درخصوص خطوط ۲ الی ۷، کلیه برچسب‌های موجود در دستورات استخراج می‌گردد و در پشته LABELS اضافه می‌گردد. در یک حلقه تکرار، کد خام، خط به خط پیمایش شده و دستورالعمل‌های بامعنی از خطوط آن استخراج می‌شود. در ادامه، در خطوط ۸ الی ۱۴، دستورالعمل‌هایی که شامل مشخص‌نمودن عملگرها، کد اجرایی نوع دستور اعم از پرش، مقایسه یا دستورات دیگر خواهد بود. همچنین، در این مرحله لیستی از کلیه برچسب‌های استفاده‌شده درون کد استخراج می‌گردد. در حلقه تکرار دوم، کد به‌صورت خط به خط پیمایش شده و درصورتی‌که به یک دستور مقایسه cmp برسد، عملگرهای آن از متغیرهای ورودی باشند، پتانسیلی جهت مبهم‌سازی کشف شده است. بنابراین، متغیر مورد نظر به لیستی برای استفاده بعدی افزوده می‌شود. به عبارت دیگر، در صورتی‌که دستورالعمل دارای یک برچسب بوده و عملیات مرتبط با آن عملیات خواندن از ورودی باشد (فراخوانی input) متغیر موجود در آن دستورالعمل استخراج گردیده و در پشته INP_VARS اضافه می‌گردد. در غیر این صورت، کلیه خطوطی

که شامل دستور فراخوانی ورودی می‌باشند و نتیجه آن‌ها در یک متغیر نگهداری می‌شود، شناسایی می‌شوند. این متغیرها، همان متغیرهایی هستند که پتانسیل تحلیل نمادین را در صورت مشارکت در شروط برنامه دارا خواهند بود. در این مرحله، تحلیل کد به پایان رسیده است و در مرحله آخر، در خطوط ۱۷ تا ۲۲، نوبت به مبهم‌سازی کد است که دوباره با پیمایش خط به خط کد هر مرحله‌ای که به یکی از متغیرهای مورد نظر برسد، اقدام به تزریق کد خود می‌نماید وگرنه کد قبلی را به خروجی اضافه می‌کند. عملیات تزریق کد مبهم‌شده نیز به این صورت خواهد بود که کلیه عملیات‌های نتیجه عملیات شرطی تا برچسب بعدی را نگه داشته و متغیر اضافی به کد افزوده می‌شود. با استفاده از برچسب‌هایی که قبلاً استخراج شده است، برچسب‌های جدید تولید شده و عملیات باقی‌مانده در جای مناسب نوشته می‌شوند. با اعمال این الگوریتم کد ورودی به قطعه کد شکل (۷) و شکل (۸) تبدیل خواهد شد و سپس با اعمال الگوریتم ۲ موجود در شکل (۶) کد خروجی دوباره مبهم شده و قطعه کد شکل (۹) و شکل (۱۰) به‌دست خواهد آمد.

الگوریتم مبهم‌سازی با استفاده از تفسیر انتزاعی با دریافت لیست دستورالعمل‌ها و تبدیل آن لیست به گراف جریان کنترلی، یک لیست از بلاک‌های اولیه را ایجاد خواهد کرد که در خط یک نشان داده شده است. در خط دو، از بلاک‌های اولیه، حالت‌های برنامه استخراج می‌شود. حالت‌های برنامه بلاک‌های اولیه هستند که فقط یک دستورالعمل زبان ماشین و یک محیط که نشان‌دهنده بازه متغیرهای برنامه بعد از اعمال دستورالعمل می‌باشد را مشخص می‌کند. برای به‌دست‌آوردن مقادیر این محیط‌ها نیاز است تا با استفاده از روش تحلیل ایستای بازه، وضعیت‌ها مورد بررسی قرار گیرند. در خط سوم در تحلیل ایستای بازه، با استفاده از تقریبی که مد نظر است سعی می‌شود تا یک تقریب بازه‌ای برای هر خط از کد به‌دست آورد. در خط چهارم از الگوریتم، در هر بار تکرار برای تولید شرط به صورت تصادفی، یکی از محیط‌ها انتخاب می‌شود. متناسب با بخش تولید شرط از روی محیط یک حالت جدید برای برنامه ایجاد می‌شود که شرط مفروض در آن قرار می‌گیرد. این امر در خطوط پنجم و ششم قرار داده شده‌اند. در خط هشتم، این شرط بین دو حالتی که قبل و بعد از محیط انتخابی هستند، قرار می‌گیرد. حالت نقیض معنایی نیز در خط هفتم برای این محیط ایجاد می‌شود. این عمل موجب گمراه‌کردن و پایین‌آوردن دقت تحلیل تفسیر انتزاعی می‌گردد. این حالت به عنوان ورودی به حالت شرط داده می‌شود. تا حالت شرط از دو طریق، ورودی داشته باشد. از طرفی دیگر، شرط دو مسیر را برای برنامه اعمال می‌کند، مسیر درست همواره وارد برنامه شده تا روال اصلی برنامه بدون خدشه اجرا گردد و مسیر غلط وارد حالتی می‌شود که نقیض معنایی یک محیط دیگر است. در نهایت، کد اسمبلی برای تولید کد اجرایی بازگردانده می‌شود.

Algorithm 2. AbstractInterpretationObfuscation(S)

Input: list of instructions INS
Output: obfuscated code

```
1.  LBasicBlocks LB=
    geneateControlFlowGraphs(INS)
2.  Satates Pro_states= generateProgramStates(LB)
3.  Interval_static_analysis(Pro_Sates)
4.  While(maked_a_loop(Pro_Sates) do
5.      Environment e =
    select__random_env(Pro_Sates)
6.      state b = Make_jamp_block(e)
7.      state c = make_neg_env_block(e,b)
8.      Pro_Sates.add(b,c)
9.  End while
10. return Pro_Sates.ToAssembly()
```

شکل (۶): الگوریتم مبهم‌سازی با استفاده از تفسیر انتزاعی

```
1     .intel_syntax noprefix
2     .section    .rodata
3  .LC0:
4     .string "\n\nYES CONDITION\n"
5     .text
6     .globl  do_job
7     .type   do_job, @function
8  do_job:
9     push    ebp
10    mov ebp, esp
11    sub esp, 24
12    mov DWORD PTR [esp], OFFSET FLAT:.LC0
13    call    puts
14    leave
15    ret
16    .size   do_job, .-do_job
17    .globl  main
18    .type   main, @function
19 main:
20    push    ebp
21    mov ebp, esp
22    and esp, -16
23    sub esp, 32
24    mov eax, DWORD PTR [ebp+12]
25    add eax, 4
26    mov eax, DWORD PTR [eax]
27    mov DWORD PTR [esp], eax
28    call    atoi
29    mov DWORD PTR [esp+28], eax
30    mov eax, DWORD PTR [esp+28]
31    add eax, 1000
32    mov DWORD PTR [esp+24], eax
33    jmp .L3
34 .L7:
35    mov eax, DWORD PTR [esp+24]
36    cdq
37    shr edx, 31
38    add eax, edx
39    and eax, 1
40    sub eax, edx
41    cmp eax, 1
42    jne .L4
```

شکل (۷): کد مبهم شده در مرحله اول (ادامه در شکل (۸))

۴-۲- بررسی چالش‌های احتمالی

همان‌طورکه تا این‌جا بیان شد، این روش به دنبال مخفی‌نمودن شرط وقوع یک رفتار و نه خود رفتار است. با این‌که الگوریتم بر رفتار مورد نظر تاثیری مستقیم ندارد ولی حتی با مبهم‌نمودن شرط وقوع نیز ممکن است چالش‌هایی رخ دهد که در ادامه بیان خواهد شد.

```
1     .section    .rodata
2  .LC0:
3     .string "\n\nYES CONDITION\n"
4     .text
5     .globl  do_job
6     .type   do_job, @function
7  do_job:
8     push    ebp
9     mov ebp, esp
10    sub esp, 24
11    mov DWORD PTR [esp], OFFSET FLAT:.LC0
12    call    puts
13    leave
14    ret
15    .size   do_job, .-do_job
16    .globl  main
17    .type   main, @function
18 main:
19    push    ebp
20    mov ebp, esp
21    and esp, -16
22    sub esp, 32
23    mov eax, DWORD PTR [ebp+12]
24    add eax, 4
25    mov eax, DWORD PTR [eax]
26    mov DWORD PTR [esp], eax
27    call    atoi
28    mov DWORD PTR [esp+28], eax
29    mov eax, DWORD PTR [esp+28]
30    add eax, 1000
31    mov DWORD PTR [esp+24], eax
32    jmp .L3
33 .L4:
34    cmp DWORD PTR [esp+24], 0
35    jle .L5
36    mov eax, DWORD PTR [esp+24]
37    cdq
38    shr edx, 31
39    add eax, edx
40    and eax, 1
41    sub eax, edx
42    cmp eax, 1
```

شکل (۸): ادامه شکل (۷) - کد مبهم شده در مرحله اول

اعداد ثبات: این روش به‌منظور مخفی‌نمودن اعداد ثابت طراحی نشده و در واقع خود، ثابت‌های دیگری نیز علاوه‌بر ثابت‌های قبلی کد به آن می‌افزاید. این مسئله از دو جهت مورد توجه قرار می‌گیرد. اول این‌که شروع وقوع رفتار مورد نظر متکی به ورودی‌های برنامه بوده و به‌صورت پویا محاسبه شود. دوم این‌که خود، مقادیر ثابت هستند که توسط حل‌کننده‌های محدودیت به‌راحتی اداره می‌شوند. برای رفع این چالش پیشنهاد می‌شود تا از مقادیر ثابت مبهم به‌منظور مخفی‌سازی مؤلفه‌های مبهم‌سازی استفاده شود. مقادیر ثابت مبهم به‌عنوان نمونه می‌توانند اشاره‌گرهایی به محلی از حافظه با مقادیر ثابت باشند تا تحلیل‌گر نمادین در حین اجرای حل معادله با یکی از چالش‌های ذاتی خود یعنی ارجاع به اشاره‌گرها مواجه شود. البته در حالت معمولی نیز ابزار BitBlaze FuzzBall در تحلیل همه مسیرهای اجرایی ناموفق بود.

```
43    mov edx, DWORD PTR [esp+24]
44    mov eax, edx
45    add eax, eax
46    add eax, edx
47    add eax, 1
48    mov DWORD PTR [esp+24], eax
49    jmp .L5
50 .L4:
51    mov eax, DWORD PTR [esp+24]
52    mov edx, eax
53    shr edx, 31
54    add eax, edx
55    sar eax
56    mov DWORD PTR [esp+24], eax
57 .L5:
58    mov eax, DWORD PTR [esp+24]
59    mov edx, DWORD PTR [esp+28]
60    sub edx, eax
61    mov eax, edx
62    cmp eax, 28
63    jle .L3
64    mov eax, DWORD PTR [esp+24]
65    mov edx, DWORD PTR [esp+28]
66    add eax, edx
67    cmp eax, 31
68    jg  .L3
69    call    do_job
70    jmp .L6
71 .L3:
72    cmp DWORD PTR [esp+24], 1
73    jg  .L7
74 .L6:
75    mov eax, 0
76    leave
77    ret
78    .file   "my-program2-obf-abs.c"
79    .intel_syntax noprefix
```

شکل (۹): کد مبهم شده نهایی (ادامه در شکل (۱۰))

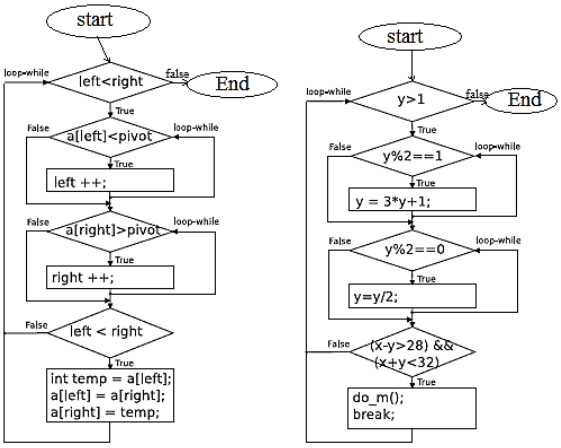
رفتار مخرب: گفته می‌شود که هدف این مبهم‌ساز مخفی‌سازی رفتار مخرب یا هر رفتار خاصی نیست بلکه هدف آن است تا شرط وقوع و اجرای این رفتار در هنگام تحلیل هرچه کمتر نمایان شود یا به عبارتی مخفی گردد. در صورتی‌که قرار باشد این روش اعمال شود، نویسنده برنامه می‌تواند از روش‌های مبهم‌سازی دیگر برای مخفی‌سازی رفتار مورد نظر خود بهره ببرد. این مطلب البته تا مرحله دوم مبهم‌سازی نیز قابل بحث بود اما با اعمال روش مبهم‌سازی برمبنای تفسیر انتزاعی تا حد زیادی

این مسئله را برطرف نمودیم. دلیل این مسئله این است که تفسیر انتزاعی و الگوریتم آن روش تصادفی بوده که بر روی کد اعمال می‌شود و برای اطمینان از مخفی‌سازی صددرصد رفتار مخرب می‌توان از روش‌های مستقیم دستی استفاده نمود. در هر حال، مبهم نمودن رفتار مخرب جزء مسائلی نبوده که این روش به دنبال حل آن باشد.

```
43      jne .L6
44      mov edx, DWORD PTR [esp+24]
45      mov eax, edx
46      add eax, eax
47      add eax, edx
48      add eax, 1
49      mov DWORD PTR [esp+24], eax
50      jmp .L7
51  .L6:
52      mov eax, DWORD PTR [esp+24]
53      mov edx, eax
54      shr edx, 31
55      add eax, edx
56      sar eax
57      mov DWORD PTR [esp+24], eax
58  .L7:
59      mov eax, DWORD PTR [esp+24]
60      mov edx, DWORD PTR [esp+28]
61      sub edx, eax
62      mov eax, edx
63      cmp eax, 28
64      jle .L3
65      mov eax, DWORD PTR [esp+24]
66      mov edx, DWORD PTR [esp+28]
67      add eax, edx
68      cmp eax, 31
69      jg  .L3
70      call    do_job
71      jmp .L8
72  .L5:
73      mov eax, DWORD PTR [esp+28]
74      imul    eax, eax, -3000
75      add DWORD PTR [esp+24], eax
76      jmp .L4
77  .L3:
78      cmp DWORD PTR [esp+24], 1
79      jg  .L4
80  .L8:
81      mov eax, 0
82      leave
83      ret
```

شکل (۱۰): ادامه شکل (۹) - کد مبهم شده نهایی

کشف الگو: استفاده از ترکیبات حل‌نشدنی در الگوی مورد نظر ارائه‌شده در این روش مبهم‌سازی ممکن است منجر به تولید یک الگوی جدید شود که قابل شناسایی توسط ابزارهای تحلیل بدافزار و تشخیص الگو باشند. اما این مسئله با دو استدلال قابل رفع است. اول این‌که همان‌طور که قبلاً گفته شد، شرط وقوع و اجرای رفتار مبهم می‌گردد نه خود رفتار. از این جهت، اگر فرض کنیم رفتار مخربی به الگوی a در کد وجود دارد و ترکیب حل‌نشده این روش مبهم‌سازی دارای الگوی b باشد اعمال آن بر روی کد منجر به تولید الگوی a+b نمی‌شود. به این دلیل که الگوی کد مبهم‌سازی بر روی رفتار و جریان رفتاری الگوی کد مخرب تأثیری ندارد. اما آیا خود الگوی منحصربه‌فرد b نیز می‌تواند قابل تطبیق بر الگوهای خطرناک و قابل شناسایی باشد؟ در ادامه بیان خواهد شد که قابل تطبیق نیست. با توجه به شکل (۱۱)، می‌توان گراف جریان برنامه مورد نظر را دید و با کمی دقت و تطابق با الگوهای دیگر، می‌توان فهمید که الگوی این قطعه کد مطابق با الگوی عادی و معمولی روند برنامه‌های دارای حلقه خصوصاً الگوریتم‌های مرتب‌سازی است. از این‌رو، نمونه کد مبهم‌شده در زبان C موجود در شکل (۳) و همچنین گراف جریان کنترلی quick sort را در کنار هم قرار داده‌ایم. در واقع، منظور این است که الگوی این قطعه کد آن‌قدر عادی و مرسوم است که ممکن است در هر برنامه‌ای چندین بار رخ دهد و وقوع چنین الگوی رفتاری دال بر وجود رفتاری خاص یا مخرب در برنامه نخواهد بود.



شکل (۱۱): آ) گراف جریان کد مبهم شده مرحله دو، ب) گراف جریان quick sort

علاوه‌بر مطالب فوق، هنوز مسئله تکرار الگوی مورد نظر در کد باقی مانده است. به عبارت دیگر، این الگو با این‌که بر رفتار کد تاثیری ندارد و به‌طور ذاتی نیز رفتار مخربی نیست، همانند هر الگوی عادی برنامه دیگری می‌تواند باشد اما یک مسئله این است که چنین الگوی به صورت تکراری در یک کد یا خانواده‌ای از کدها رخ دهد. برای حل این مسئله نیز با اعمال مبهم‌سازی مبتنی بر تفسیر انتزاعی به‌واسطه تصادفی‌بودن روش، عملاً الگوهای ثبات کد به‌هم ریخته و الگوهای متعدد و معتبری

به‌وجود خواهد آمد.

مجموعه‌های بزرگ‌تر از ورودی فعال‌ساز: در این روش، فرض بر این است که شرط فعال‌ساز رفتار مورد نظر یک عدد صحیح است و نشان داده شد که ابزار اجرای نمادین نمی‌تواند همه مسیرهای اجرایی را با موفقیت پیمایش نماید. با این‌حال، مجموعه و بازه وقوع شرط اجرای رفتار ممکن است با افزایش میزان متغیرهای تأثیرگذار بر شرط بیشتر شوند. البته این مطلب به‌طور نسبی بر روی روش تأثیرگذار است اما فقط شانس وقوع شرط صحیح را در هنگام تحلیل نمادین افزایش می‌دهد و کماکان ابزارهای نمادین در پیمایش همه مسیرهای ممکن دچار مشکل خواهند بود. به این صورت که فرض نمایید به‌جای بررسی برابری یا نابرابری شرط اجرا در حالت جدید با یک بازه از اعداد مواجه هستیم که باعث اجرای رفتار مورد نظر می‌گردند. حال اگر ابزار تحلیل‌گر بتواند این بازه را کشف نماید عملاً به‌شرط وقوع رفتار نیز پی برده است اما استفاده از ترکیبات حل‌نشدنی باعث می‌شود که حل‌کننده محدودیت نتواند با استفاده از حل معادله بازه را بیابد بلکه مجبور است در مسیر بازگشتی کد بازه را تولید نماید که با هر ورودی نمادین بازه جدید و شاید بزرگ‌تری تولید شود.

سربار اجرایی: این روش به‌واسطه حلقه افزوده‌شده به کد، یک سربار اجرایی بر روی کد اصلی تحمیل می‌نماید. البته این مطلب مخصوصاً وقتی بر روی یک رفتار مخرب اضافه می‌شود مطلب مهمی تصور نمی‌گردد اما در هرحال بروز سربار محاسباتی مطلبی است که واقع می‌شود. برای درک تصوری از میزان سربار می‌توان فرض کرد یک معادله خطی به میزان ۱۰۰۰ بار اجرا شود. میزان این سربار به قدرت پردازشی سیستم مقصد مرتبط است که حتی در ماشین‌های ضعیف نیز احتمالاً کمتر از ۱۰۰ میلی‌ثانیه خواهد بود.

۵- ارزیابی

برای ارزیابی روش مبهم‌سازی ارائه‌شده در این مقاله، میزان مقاومت، اجرای نمادین مثال ارائه شده در بخش قبل مورد بررسی قرار می‌گردد. شاید در نگاه اول این مثال بسیار ساده به نظر برسد اما در مقابل به‌خوبی میزان تأثیر روش مبهم‌سازی را بر روی کد نمایش می‌دهد. در بخش قبل، کد مبهم‌نشده در مقابل کد مبهم‌شده نمایش داده شده است. حال با نمایش نتایج اجرای هر دو کد بر روی ابزار FuzzBall، نشان می‌دهد که ابزار قادر نیست کلیه مسیرهای اجرایی تولیدشده را بپیماید و عملیات جستجو و پیماش خود را به نتیجه برساند. در مرحله بعد به این موضوع پرداخته خواهد شد که آیا چنین نتیجه‌ای بر روی برنامه‌های پیچیده‌تر نیز عملی خواهد بود یا خیر؟ برای آزمایش این روند از یک Ubuntu 14.0.4 x86 بر روی ماشین مجازی VMWare Workstation 11.1.0 استفاده شد که سیستم‌عامل میزبان آن نیز Window 10 x64 بود. منابع تخصیص داده‌شده به ماشین مجازی شامل 1.5 GB حافظه RAM، پردازنده دو هسته‌ای Intel Core i5-3337U 1.80GHz و 20 GB HDD است.

۵-۱- استراتژی مورد استفاده توسط ابزارهای تحلیل‌گر

به‌منظور ارزیابی میزان تأثیر روش مبهم‌سازی مورد نظر در فریب و شکست ابزارهای خودکار تحلیل و اجرای نمادین، در ابتدا باید شیوه استفاده‌شده توسط این ابزارها را دانست. همچنین، به یاد داشته باشید که هدف ابزار، مخفی‌سازی شرط اجرا می‌باشد نه رفتار مخرب آن. به‌طور ساده، اگر به‌واسطه شرط x رفتار مخرب t اتفاق بیافتد، در این‌جا، هدف، مخفی‌سازی حالات اتفاق و صحت x است نه اجرای t که خواه‌ناخواه در یکی از مسیرهای اجرای نمادین ممکن است رخ بدهد. همچنین، همان‌طورکه قبلاً بیان شد، ابزار تحلیل‌گر، قادر به کشف الگوی رفتاری کد نیست. زیرا اولاً کد مبهم‌شده جدای از رفتار مخربی است که ممکن است الگوی خاصی داشته باشد، عمل می‌نماید و از طرفی، الگوی آن، به‌صورت منحصربه‌فرد با الگوی اجرایی یک حلقه ساده جستجو تفاوتی نخواهد داشت. بنابراین، از نظر ابزارهای خودکار، یک الگوی مخرب نمی‌تواند شناسایی شود. در هر حال، شیوه تحلیل و اجرای نمادین ابزارها به‌صورت زیر است:

۱. انتخاب مقدار اولیه y_0 برای ورودی.
۲. رصد اجرای برنامه تحت متغیر نمادین y_i به‌صورت پویا. در صورتی‌که رفتار مخرب دیده شود، پس شرط اجرا معادل y_i خواهد بود.
۳. جمع‌آوری شروط انشعاب[1] به همراه دنباله اجرایی[2] و برعکس‌نمودن آخرین شرط دنباله.
۴. استفاده از حل‌کننده محدودیت به‌منظور حل ورودی جدید برنامه که دنباله جدید شروط را ارضاء می‌نماید (همان شرط آخر که معکوس‌شده یک دنباله جدید می‌سازد) به همراه شرط منجر به رفتار مخرب.

فرض کنید y_{i+1} معادل ورودی جدید خواهد بود درصورتی‌که شروط ارضاء شوند و سپس به مرحله ۲ باز می‌گردیم. اگر مقدار جدید نتواند پیدا شود شرط بعدی (به سمت نقطه آغازین برنامه)

1- Branch Conditions
2- Execution Trace

برعکس (منفی) شده و به حل‌کننده محدودیت فرستاده می‌شود تا ورودی جدید یافت شود.

در مثال جدول (۱) شرط اجرا، مقدار y = 1030 (همان x == 30) است. فرض کنیم که ابزار تحلیل‌گر مقدار y = 1158 را به‌عنوان ورودی اولیه انتخاب نماید که منجر به دنباله‌ای با نتایج true/false در محاسبه شرط y % 2 == 1 در هر بار اجرای حلقه خواهد شد.

جدول (۲)، مقدار y و نتایج ارزیابی حلقه برای شرط اجرای y == 1030 و ورودی y == 1158 را نمایش می‌دهد. این جدول نتایج حل‌کننده محدودیت را هنگامی‌که تحلیل‌گر قصد یافتن ورودی بعدی را دارد، نمایش می‌دهد. حل‌کننده محدودیت STP به‌طور مدام تا قبل از اجرای یازدهم حلقه، مقدار $false$ برمی‌گرداند. به این معنا که قادر نیست ورودی معتبری جهت ارضاء دنبال شرط داده‌شده پیدا نماید. به‌محض این‌که STP ورودی بعدی برنامه را برگرداند ابزار تحلیل‌گر به گام دوم بازگشته و دوباره تلاش می‌کند.

جدول (۱): دنباله پویا با ورودی اولیه ۱۱۳۰ و ۱۱۵۸

y = 1158			y = 1030		
STP result	y % 2 == 1	y	iteration	y % 2 == 1	y	iteration
	false	۱۱۵۸	۱	false	۱۰۳۰	۱
	true	۵۷۹	۲	false	۵۱۵	۲
	false	۱۷۳۸	۳	false	۱۵۴۶	۳
	…	…	…	…	…	…
	true	۱۶۳	۹	false	۱۴۵	۹
	false	۴۹۰	۱۰	false	۴۳۶	۱۰
true	true	۲۴۵	۱۱	false	۲۱۸	۱۱
…	…	…	…	…	…	…
false	false	۴	۱۲۳	false	۴	۱۲۳
false	false	۲	۱۲۴	false	۲	۱۲۴
false	true	۱	۱۲۵	true	۱	۱۲۵

جدول (۲): ارزیابی مثال‌های مختلف تحت آزمایش ابزار اجرای نمادین

نام برنامه	دفعات تلاش	زمان اجرا	حجم فایل (بایت)	نرخ موفقیت	نرخ اتمام	اتمام موفق
Simple if	۲	2.13 s	۷۴۲	۵۰%	۱۰۰%	بلی
Simple if (obfsct)	**۵۴۵**	**~ 1hr**	**۱۳۰۰**	**۱/۴۸۱%**	**۴/۹۷%**	**خیر**
Simple if (branch)	۱۵۴۶	~1hr	-	0%	۱۰۰%	خیر
fibonatchi	۱	2.34 s	۱۰۴۸	۱۰۰%	۱۰۰%	بلی
Fibonatchi (obfsct)	**۳۶۹**	**~ 1hr**	**۱۶۸۴**	**۱/۷۴%**	**۴/۴۴%**	**خیر**
Fibonatchi (branch)	۱۴۳۸	~ 1hr	-	0%	۱۰۰%	خیر
Pi number	۲	2.788 s	۲۹۰۹	۵۰%	۱۰۰%	بلی
Pi number (obfsct)	**۳۴۶**	**~ 1hr**	**۳۵۵۳**	**۱/۷۱%**	**۵/۱۸%**	**خیر**
Pi number (branch)	۱۱۶۲	~ 1hr	-	0%	۱۰۰%	خیر
Maze	۲	4.38 s	۵۸۳۶	۵۰%	۱۰۰%	بلی
Maze (obfsct)	**۲۸۴**	**~ 1hr**	**۶۴۵۸**	**۱/۶۴%**	**۴/۱۲%**	**خیر**
Maze (branch)	۸۱۴	~ 1hr	-	0%	۱۰۰%	خیر

این فرایند تا زمانی‌که رفتار مخرب مشاهده شده و شرط اجرا پیدا شود، ادامه پیدا می‌کند. دلیل این‌که آخرین شرط دنباله در ابتدا معکوس(منفی) می‌گردد این است که فرض می‌کنیم ابزار تحلیل‌گر قادر است ورودی اولیه نزدیک به‌شرط اجرا و درستی را حدس بزند. این فرض از این جهت منطقی به نظر می‌رسد که شرط درستی وابسته به زمینه[1] است. تحت چنین شرایطی، تحلیل‌گر ترجیح می‌دهد y بعدی از بین یکی از نتایج بسیار نزدیک به آخرین اجرا انتخاب نماید، زیرا این انتخاب به‌احتمال قوی‌تری به‌شرط درستی نزدیک و نزدیک‌تر خواهد شد

۵-۲- اجرا روی مثال واقعی

در این قسمت با توجه به توضیح مراحل بالا، اطلاعاتی از اجرای الگوریتم بر روی سه مثال مختلف را ارائه می‌شود. دقت شود مثال‌های زیر به‌دلیل تفاوت الگوریتم‌ها انتخاب شده‌اند و هدف افزایش حجم فایل نبوده است. به‌علاوه،در این قسمت، برای مثال ساده تشریح‌شده دو الگوریتم معروف تولید اعداد Fibonacci و همچنین محاسبه عدد pi را برای نمایش شیوه عملکرد واقعی الگوریتم نشان داده می‌شود. در این بخش، شش فاکتور در مورد مثال‌ها ارزیابی شده‌اند.

۱. تعداد دفعات اجرای STP: این مقدار صرفاً به تعداد مسیرهای یافت‌شده نیست بلکه می‌تواند ترکیبی از مسیرهای یافت‌شده و نیز تلاش‌های ناموفق STP باشد. البته در حالت کد مبهم‌نشده چون همه تلاش‌ها موفق هستند تعداد مسیرها نیز برابر با همان مقدار است.
۲. زمان اجرا: زمانی که طول می‌کشد تا ابزار BitBlaze کلیه مسیرهای اجرایی کد مورد نظر را کشف کرده و اجرای برنامه را به اتمام برساند. برای حالتی‌که کد مبهم‌شده از آن‌جا که یافتن همه مسیرهای اجرایی به اتمام نمی‌رسد زمان اجرای ۱ ساعت برای همه در نظر گرفته شده تا نتایج قابل مقایسه باشند.
۳. حجم فایل: حجم کد منبع برنامه قبل و بعد از مبهم‌شدن آمده است تا نشان داده شود این روش همواره حجم تقریباً ثابتی را به کد می‌افزاید (بین ۵۵۰–۶۵۰ بایت) و با افزایش حجم کد اصلی تغییر چشم‌گیری در حجم افزوده‌شده پس از مبهم‌سازی به‌وجود نمی‌آید.
۴. نرخ موفقیت: این نرخ نشان می‌دهد که حل‌کننده محدودیت توانسته است شرط درستی به رفتار مخفی‌شده را در یکی از مسیرهای اجرایی پیدا نماید.
۵. نرخ اتمام: این عدد نمایان‌گر این است که ابزار BitBlaze تا چه اندازه توانسته است مسیرهای اجرایی یافت‌شده را با موفقیت تا پایان برنامه اجرا کند و نتیجه را چاپ نماید.
۶. اتمام موفق: این مورد نیز صرفاً نمایش می‌دهد که آیا ابزار اجرای نمادین توانسته کلیه مسیرها را یافته و کار را به اتمام برساند یا خیر؟

در بررسی‌های انجام‌شده، روش‌های مختلفی از مبهم‌سازی کد برای مقابله با اجرای نمادین وجود داشت که روش مبهم‌سازی انشعاب نیز دارای پارامترهای قابل مقایسه با این روش بود. از طرفی، روش مبهم‌سازی پویا برمبنای درخواست نمادین با وجود نتایج بسیار خوب اعلام‌شده برای مقایسه، در دسترس قرار نداشت. برای این منظور، الگوریتم معرفی‌شده در روش مبهم‌سازی انشعاب به‌صورت محدود روی سه مثال معرفی‌شده پیاده شد. در جدول (۲)، نتایج حاصل از اجرای مثال‌ها تحت ابزار BitBlaze به نمایش درمی‌آید. در این جدول، نتایج را همان‌گونه که انتظار آن را داشتیم می‌بینیم. سطر اول مخصوص اجرای برنامه بر روی کد عادی و مبهم نشده است که به‌طور واضح ابزار اجرای نمادین به‌راحتی کار را به پایان رسانده و در یک‌زمان منطقی، کلیه مسیرهای اجرایی برنامه را استخراج می‌نماید.

در سطر سوم هر مثال، اجرای برنامه‌ها با اعمال روش مبهم‌سازی انشعاب که در کارهای مرتبط به آن اشاره شد، انجام گردید. از آن‌جاکه با اعمال این روش کد اجرایی ما به دو فایل مجزا تبدیل می‌شد، عملاً اندازه‌گیری حجم فایل بی‌معنی تلقی می‌شد. به همین جهت، بر روی پارامترهای درصد موفقیت و تعداد مسیرهای تولیدشده، تمرکز گردید. همان‌طورکه انتظار می‌رفت در کلیه مثال‌ها، تعداد مسیرهای اجرایی بیشتری تولید شد اما از آن‌جاکه تابع مبهم‌شده بر روی فایل اصلی قرار نداشت (شبیه‌سازی سیستم خارجی) ابزار اجرای نمادین نمی‌توانست کار را به‌طور نمادین ادامه دهد و به همین سبب، هر بار اجرای حل‌کننده محدودیت معادل یک‌بار اجرای برنامه با مقادیر واقعی بود. بنابراین، کلیه تلاش‌های حل‌کننده محدودیت، به جواب منتهی گردید اما به‌واسطه نبود کد مبهم‌شده برای اجرا، در عمل هیچ‌گاه رفتار مبهم شده رخ نداد.

سطر دوم کلیه مثال‌ها مربوط به روش معرفی‌شده است. همان‌طورکه در کلیه سطور دیده می‌شود، حجم کد به اندازه یک مقدار ثابت حدوداً ۵۵۰ بایتی افزایش داشت. همچنین، نرخ موفقیت نشان‌گر موفقیت حل‌کننده محدودیت دریافتن یکی از شروط اجرا بود که حدود مقدار ۱/۵ ٪ نوسان داشت. نرخ اتمام یعنی تعداد دفعاتی که ابزار اجرای نمادین توانست یک مرحله از اجرا را به ازای ورودی t به پایان برساند. تقریباً در ۹۵ ٪ موارد ابزار نمی‌توانست درخت نمادین ایجادشده را تا پایان آن ادامه دهد.

1- Context

۶- نتیجه‌گیری

در این مقاله، یک روش مبهم‌سازی کد، جهت جلوگیری از تحلیل خودکار نمادین با استفاده از متغیرهای ورودی به برنامه ارائه شده است. استفاده از متغیرهای ورودی به برنامه، موجب شد تا روش مبهم‌سازی بتواند شرط اجرای کد مورد نظر خود را از دید ابزار تحلیل‌گر نمادین مخفی نماید. همچنین، نشان داده شده است که انتخاب مقادیر مختلف بر روی زمان و نتیجه اجرایی تفاوت چندانی ندارد. این در حالی بود که ابزار اجزای نمادین BitBlaze FuzzBall در حالت غیرمبهم در حدود چند ثانیه توانست ساده‌ترین برنامه مورد نظر را حل کند ولی بعد از مبهم‌سازی تحلیل کد مورد نظر تا مدت نامعلومی طول کشید که عملاً مقایسه زمان اجرایی آن دو حالت بی‌معنی گردید. از طرفی، حجم کد نیز تغییر محسوسی نکرد و حدود ۵۵۰ الی ۶۵۰ بایت تغییر داشت که به‌دلیل قراردادن حجم ثابتی از کدی، ثابت است. روش معرفی‌شده، شرط اجرای رفتار مورد نظر را تا حد قابل قبولی مخفی نمود و عملاً باعث شد حل‌کننده محدودیت نتواند کلیه مسیرهای اجرایی برنامه را استخراج نماید. البته به‌دلیل استفاده از روش تفسیر انتزاعی جهت مبهم‌سازی درمقابل استخراج الگو از نمونه‌های مبهم‌سازی‌شده در جهت کشف روش مبهم‌سازی نیز جلوگیری شده است و موجب به‌هم‌زدن چرخه عادی و جریان عادی برنامه و عدم اضافه‌نمودن الگوی جدید به کد که باعث حساسیت تحلیلگرهای کشف الگو می‌شود که یک مزیت ویژه به حساب می‌آید. البته از دیگر مزایای روش نیز می‌توان عدم وابستگی به دستگاه و یا سروری جهت ارتباط و استفاده از روش رمزنگاری، توابع درهم‌سازی است که برای ابزارهای تحلیلگر حساسیت‌برانگیز است. همچنین، از معایب روش نیز پویانبودن قطعه کد اضافه شده است. یا این‌که امضای جدید ایجاد نمی‌شود اما به دفعات زیاد در کل، قابل استفاده نیست زیرا با تکرار رفتار تحلیلگر می‌تواند به دست‌کاری‌شدن کد شک نماید. برای مواردی مناسب است که از چند شرط اجرای کد در قسمت‌های مختلف کد استفاده گردد و به طور مستقل قرار می‌گیرد همچون، قطعه کدهای مرتبط با شماره سریال.

۷- کارهای آتی

حوزه مورد بحث در این پروژه به سبب وسعت، همچنان با چالش‌هایی متعدد و مختلفی روبه‌رو است. سعی بر آن شد تا به تعدادی از این چالش‌ها در این پژوهش پاسخ داده شود اما هنوز چالش‌ها و مسائل متعددی وجود دارد که می‌بایست در تحقیقات آتی مورد بحث قرار بگیرد. تعدادی از این چالش‌ها تحت عنوان پیشنهاد برای تحقیقات آتی در ادامه معرفی شده‌اند.

مقاوم‌سازی کد در مقابل اجرای کانکالیک یا ترکیبی: روشی که در این پروژه بر روی آن بحث شد حول اجرای نمادین کد بود که ابزار از ابتدا تا انتهای کد را به‌صورت نمادین اجرا می‌نمود. برای گام بعدی می‌توان روش را جهت فریب ابزارهای ترکیبی تکمیل نمود. به‌عنوان مثال، ابزار S2E [۳۰] ادعا می‌کند هر زمان که متوجه شود حل‌کننده محدودیت زمان زیادی را در محاسبه مسیر خود تلف می‌کند یا مسیرهای تکراری زیادی می‌رود ادامه آن مسیر را به‌صورت محکم و با مقادیر واقعی اجرا خواهد نمود.

استفاده از ترکیبات حل‌نشدنی با مقادیر اعشاری: همان‌طورکه می‌دانیم اعداد اعشاری یکی از چالش‌های اجرا نمادین کد هستند. بنابراین، می‌توان با ترکیب روش پیش‌رو با معادلاتی با متغیرهای اعشاری چالش دیگری برای ابزار اجرای نمادین ایجاد نمود.

۸- مراجع

[1] J. C. King and T. J. N. Yorktown Heights, "Symbolic execution and program testing," Communications of the ACM, vol. 19, pp. 385-394, 1976.

[2] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, J. Poosankam, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in Botnet Analysis in Defense, p. 36, 2007.

[3] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding malware analysis using conditional code onfuscation," in Annual Network and Distributed System Security Symposium, 2008.

[4] J. Lagarias, "The 3x+1 problem and its generations," Amer. Math. Monthly, vol. 92, pp. 3–23, 1985.

[5] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution," in IEEE Symposium on Security and Privacy, 2010.

[6] R. Crandall, "On the 3x + 1 problem," Mathematics of Computation, vol. 32, pp. 1281–1292, 1978.

[7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in IEEE Symposimu on Security and Privacy, 2006.

[8] The R&D lab team, "Fuzzgrind an automatic fuzzing tool Fuzzgrind," [Online]. Available: http://esec-lab.sogeti.com/pages/fuzzgrind.html. Last seen 3/1/2018.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in Proceedings of the USENIX Symposium on Operating System Design and Implementation, 2008.

[10] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: a system for automatically generating inputs of death

[11] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005.

[12] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C," in ESEC, and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE-13, pp. 263–272, 2005.

[13] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," Future Generation Computer Systems, vol. 29, no. 7, pp. 1758-1773, 2013.

[14] A. G. M. Bernard Botella, "Symbolic execution of floating-point computations," Software Testing, Verification & Reliability, vol. 16, no. 2, pp. 97 - 121, June 2006.

[15] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.

[16] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '98), 1998.

[17] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: obstructing static analysis of programs," Technical report CS-2000-12, Department of Computer Science, University of Virginia, 2000.

[18] J. Marciniak, "Encyclopedia of software engineering," J. Wiley & Sons, 1994.

[19] M. H. Halstead, "Elements of software science," Elsevier North-Holland, 1977.

[20] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," SIGPLAN Notices, vol. 16, pp. 63–74, 1981.

[21] E. I. Oviedo, "Control flow, data flow and programmers complexity," in Proc. of COMPSAC 80, 1980.

[22] H. Lai, "A comparative survey of Java obfuscators," in Computer Science Department, The University of Auckland, 2001.

[23] H. Lin, X. Zhang, M. Yong, and B. Wang, "Branch Obfuscation Using Binary Code Side Effects," Advances in Intelligent Systems Research, 2013.

[24] Y. Yubo, Y. Yixian, F. Wenqing, H. Wei, and L. Zhongxian, "Dynamic Obfuscation Algorithm based on Demand-Driven Symbolic Execution," Journal of Multimedia, vol. 9, no. 6, 2014.

[25] M. D. Preda, "Code Obfuscation and Malware Detection by Abstract Interpretation," Verona: Universit`a di Verona, 2005.

[26] J. M. C. J. D. G. Zhi Wang, "Linear Obfuscation to Combat Symbolic Execution," Computer Security – ESORICS, vol. 6879, pp. 210-226, 2011.

[27] R. Guy, "Unsolved problems in number theory," in Problem Books in Mathematics, 2004.

[28] M. Alaeiyan, "Code obfuscation by abstract interpretation," Iran university of sience and technology, tehran, 2015. (In persion)

[29] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in Proceedings of the 4th ACM Symp. on Principles of Programming Languages (POPL '77), New York, 1977.

[30] V. Chipounov, V. Kuznetsov, and G. Candea, "Systems, S2E: A Platform for In-Vivo Multi-Path Analysis of Software," in ASPLOS XVI, New York, 2011.

using symbolic execution," in Proceedings of the ACM Conference on Computer and Communications Security, 2006.

Journal of Electronical & Cyber Defence
Vol. 6, No. 1, 2017, Serial No. 21

Code Obfuscation to Prevent Symbolic Execution

S. Parsa*, H. Salehi, M. H. Alaeiyan

*Iran University of Science and Technology

(Received: 13/03/2016, Accepted: 31/10/2016)

ABSTRACT

The software protection against analysis has become an important issue in the field of computer engineering. The symbolic execution method, as an approach to explore execution paths and conditions of the program, is recently considered. Therefore, developers try to protect their code to prevent against symbolic execution. A successful symbolic execution has extracted the provisions of all paths in the form of a symbolic tree. Therefore, we can prevent the symbolic execution of a program to protect the code in several different ways and hide paths from the view of analysts. This paper focused on obfuscating the condition for behavior so that in the case of symbolic execution analyst can not find the right conditions of a behavior. For this purpose, a new agenda is presented to add some new variables to the execution path that they are related to the program variables to confuse constraint solvers and build many new fake paths in the form of the symbolic tree. Results showed that symbolic analysis tools are unable to obtain all paths after obfuscation.

Keywords: Symbolic Execution, Path Explosion, Code Obfuscation, Symbolic Tree, Abstract Interpretation, Interval Static Analysis

*Corresponding Author Email: parsa@iust.ac.ir