مروری بر روش‌های محافظت از نرم‌افزار در برابر حملات ایستا و پویا

افشین رشیدی[1]، رضا ابراهیمی آتانی[2] و حمید نصیری[3]

[1]دانش‌آموخته کارشناسی ارشد مهندسی کامپیوتر،نرم‌افزار دانشگاه گیلان، گیلان، رشت، ایران
afshinashidi1368@gmail.com
[2]استادیار گروه مهندسی کامپیوتر دانشگاه گیلان، گیلان، رشت، ایران
rebrahimi@guilan.ac.ir
[3]دانش آموخته کارشناسی ارشد مهندسی کامپیوتر، نرم‌افزار دانشگاه گیلان، گیلان، رشت، ایران
hamidnasiri2010@gmail.com

چکیده

در دهۀ گذشته با توزیع نرم‌افزارهایی مانند مرورگرها، فروشگاه‌های برخط، بانک‌داری اینترنتی، سامانه‌های رایانامه روی اینترنت، حملات گسترده‌ای برای انجام مهندسی معکوس، استفاده غیر قانونی از نرم‌افزار و یا تکثیر غیر قانونی آن انجام شده است. به‌دلیل ماهیت غیر قابل اعتماد محیط ماشین‌های میزبان، بحث محافظت از نرم‌افزار در برابر حملات تحلیل، دست‌کاری و دزدی نرم‌افزار افزایش پیدا کرده است. روش‌های محافظتی متعددی که تاکنون ارائه شده‌اند با آن که به‌طور مقطعی مقاومت نسبی ایجاد نموده است، امّا پس از گذشت مدتی، حملات جدید، آن روش‌ها را با شکست مواجه کرده و این باعث ایجاد رقابت میان مهاجمان و توسعه دهندگان نرم‌افزار شده است. تاکنون راه‌حل‌های زیادی برای محافظت مبتنی‌بر معماری سخت‌افزار و نرم‌افزاری معرفی شده است که هر کدام، از جنبه‌ای به محافظت از نرم‌افزار می‌پردازند. در این مقاله ابتدا انواع تهدیدهای موجود در برابر امنیت کدهای نرم‌افزاری را معرفی می‌کنیم و سپس سعی در دسته‌بندی و مرور تکنیک‌های معرفی‌شده برای محافظت از نرم‌افزار داریم.

واژگان کلیدی : دست‌کاری، تحلیل نرم‌افزار، دزدی نرم‌افزار، حملات ایستا و پویا، محافظت از نرم‌افزار، مبهم‌سازی.

۱- مقدمه

با توسعه سرعت اینترنت و همچنین خدمات ابری، امکان دسترسی به اطلاعات از طریق نرم‌افزار به‌عنوان خدمت، توسعۀ چشم‌گیری پیدا کرده است. به‌طوری که درعمل در دهۀ گذشته، بیشتر نرم‌افزارها روی اینترنت توزیع شده‌اند. به‌عنوان مثال نرم‌افزارهایی مانند مرورگرها، سامانه‌های رایانامه، بازی‌های برخط، بانکداری الکترونیکی و فروشگاه‌های برخط از جملۀ این نرم‌افزارها هستند. با توزیع‌شدن این نرم‌افزارها روی ماشین مشتری درواقع صاحب نرم‌افزار کنترل برنامه (روی ماشین مشتری) را از دست می‌دهد. به‌طورطبیعی در چنین محیط‌های ناهمگنی برای افزایش پایداری و سودآوری شرکت‌ها، امنیت اهمیت بسیار ویژه‌ای خواهد داشت. بحث امنیت برنامه‌های کاربردی به‌طورکلی در سه حوزه باید مورد توجه قرار گیرد:

۱. اطلاعات حساس و محرمانۀ برنامه‌ها
۲. کد برنامه
۳. داده‌های زمان اجرای برنامه

قبل از هر چیز چنین نرم‌افزارهایی حاوی اطلاعات بسیار حساس و محرمانه‌ای از افراد یا شرکت‌ها مانند پرونده، شمارۀ کارت اعتباری و رمز عبور و غیره هستند که حفظ محرمانگی و حریم خصوص یکی از اولویت‌های مهم این نرم‌افزارهاست. برای محافظت از چنین داده‌هایی الگوریتم‌های متعددی برای رمزنگاری[1] و احراز هویت[2] تاکنون طراحی و استفاده شده است که متاسفانۀ این الگوریتم‌ها براساس کلیدهای مخفی عمل می‌کنند که خود این کلیدها نیاز به محافظت دارند. بحث بعدی که باید برای امنیت آن چاره‌ای اندیشیده شود، کد برنامه‌های کاربردی است. به‌عنوان مثال مهندسی معکوس یا دست‌کاری کد تهدیدهایی در برابر کد برنامه‌های کاربردی هستند که باید

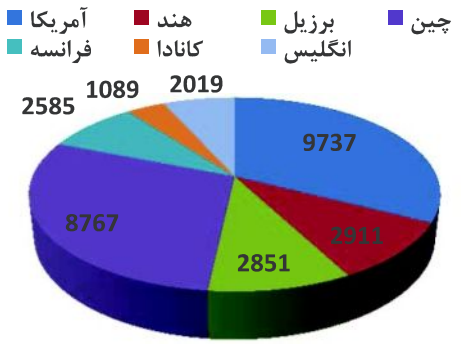
[1] Encryption
[2] Authentication

برای مقابله با این تهدیدها از نرم‌افزار محافظت انجام شود. درنهایت مسئله سوم اجرای برنامه‌هاست. در طول اجرای برنامه‌های کاربردی داده‌های محرمانه‌ای مورد دستیابی قرار می‌گیرند. در طول اجرای برنامه باید از کد و داده‌ها در مقابل مفاهیم مخربی مانند تحلیل پویا و دست‌کاری، محافظت صورت پذیرد. تمامی این موارد باید برای تضمین اجرای درست و ایمن برنامه‌ها در سمت کاربر تقویت شوند [۱].

امنیت محصولات نرم‌افزاری در برابر حملات مختلفی مورد تهدید قرار می‌گیرد. به‌طوراصولی نباید مفهوم امنیت نرم‌افزار را با محافظت از نرم‌افزار مشابه دانست. رویکرد امنیتی در طراحی نرم‌افزار، بخشی از فرآیند محافظت از آن است و هدف اصلی آن جلوگیری از آسیب‌پذیری‌های امنیتی و حملات روز، صفر است؛ درحالی‌که محافظت از نرم‌افزار یک مفهوم عام‌تر بوده که علاوه بر موارد امنیتی، چالش‌های سرقت کد و حملات استفادهٔ مجدد کد را برای کاربری جدید شامل می‌شود. دزدی نرم‌افزار[1] (تهیهٔ رونوشت‌های غیر قانونی از نرم‌افزار)، مهندسی معکوس[2] (یا تحلیل نرم‌افزار، که فهم چگونگی عمل‌کرد نرم‌افزار و استفاده از تمام یا بخشی از آن در جایی دیگر) و دست‌کاری[3] (تهدید مالکیت معنوی نرم‌افزار با تغییر بخشی از آن) از مهم‌ترین تهدیدهای محافظت از نرم‌افزار است.

دزدی نرم‌افزار معضلی است که صنعت نرم‌افزار با آن روبه‌رو است. این تهدید، هر ساله خسارت‌های جبران ناپذیری را بر صنعت نرم‌افزار وارد می‌کند. شکل (۱)، خسارت واردشده به صنعت نرم‌افزار در کشورهای مختلف را نشان داده است. روند خسارت واردشده به صنعت نرم‌افزار در حال رشد است. میزان خسارت کلی واردشده به صنعت نرم‌افزار در جهان، از ۵۳۰۰۰ میلیون دلار در سال ۲۰۰۸ میلادی به ۶۲۷۰۹ میلیون دلار در سال ۲۰۱۴ میلادی رسیده است[۳۵].

تاکنون روش‌های مختلفی برای محافظت از نرم‌افزار در برابر این حملات ارائه شده‌اند که با عناوین مختلفی مانند تکنیک‌های مقابله با دست‌کاری[4]، حفاظت از سرمایهٔ دیجیتال[5] و اغلب، به‌صورت مشترک، حفاظت از نرم‌افزار شناخته می‌شوند. برای محافظت در برابر حملات دست‌کاری، تکنیک‌های مقابله با دست‌کاری نرم‌افزار را برای مشاهده تغییرات نظارت می‌کنند و درصورتی‌که تغییراتی را در آن مشاهده کنند، اقدامات تهاجمی یا دفاعی را در برابر آن انجام می‌دهند. برای حفاظت در برابر حملات مهندسی معکوس، تکنیک‌های مبهم‌سازی سعی در ویرایش یک برنامه (گاهی در زمان کامپایل یا به‌طور مداوم در زمان اجرا) به‌منظور سخت‌کردن فهم آن دارند تا مهاجم نتواند یا به‌سختی بتواند آن را تجزیه و تحلیل کند. برای محافظت در برابر حملات شبیه‌سازی، تکنیک‌های نهان‌نگاری نرم‌افزار[6]، تغییراتی در نرم‌افزار ایجاد می‌کنند که آن نرم‌افزار تنها برای کاربران مشروع قابل استفاده باشد. برخی تکنیک‌های محافظت از نرم‌افزار فقط نرم‌افزاری هستند؛ درحالی‌که برخی دیگر از تکنیک‌ها از سخت‌افزارهای مقابله با دست‌کاری برای حفاظت از نرم‌افزار بهره می‌گیرند که نمونه‌هایی از آن‌ها کارت‌های هوشمند[7]، سکوهای با المان‌های قابل اعتماد[8] یا پردازنده رمزگذار[9] هستند.



(شکل ۱): خسارت واردشده بر صنعت نرم‌افزار در کشورهای مختلف در سال ۲۰۱۳ میلادی برحسب میلیون دلار[۳۵]

امروزه محافظت از نرم‌افزار به‌طور فزاینده‌ای تبدیل به یک نیاز مبرم در تولید نرم‌افزارهای صنعتی شده است؛ به‌خصوص هنگامی که سامانه‌های مورد نظر برای دفاع نظامی، زیرساخت‌های ملّی بانکی یا سلامت الکترونیکی باشد. هر فروشندهٔ نرم‌افزار باید از چنین حملاتی آگاهی داشته باشد و تکنیک‌هایی برای مقابله با آن در نظر گرفته باشد. به‌کارگیری تکنیک‌های محافظت از نرم‌افزار می‌تواند به معنای تفاوت بین بقای کسب و کار و یا ورشکستگی مالی باشد[۱].

[1] Software Piracy
[2] Reverse engineering
[3] Tampering
[4] Anti-tamper techniques
[5] Digital asset protection
[6] Software watermarking techniques
[7] Smart cards
[8] Trusted platform modules
[9] Crypto-processors

الگوریتم‌های حفاظت از نرم‌افزار را می‌توان به اشکال مختلف دسته‌بندی کرد. از لحاظ نوع تکنیک می‌توان آنها را به سه دسته تکنیک‌های محافظت از معماری، تکنیک‌های مبتنی بر نرم‌افزار و تکنیک‌های مبتنی بر سخت‌افزار تقسیم کرد. از لحاظ کارکرد هم این تکنیک‌ها در چهار دستهٔ اصلی قرار می‌گیرند. مبهم‌سازی کد[1] برای سخت‌کردن عمل مهندسی معکوس روی برنامه، تکنیک‌های مقاومت در برابر دست‌کاری برای سخت‌کردن ویرایش برنامه، نهان‌نگاری برای اجازهٔ ردیابی‌دادن به برنامه‌ها و علامت اصلیت[2] هم برای تشخیص این‌که کدی از یک برنامه در برنامه‌ای دیگر استفاده شده است. از لحاظ زمانی هم این تکنیک‌ها به دو دسته تکنیک‌های ایستا و پویا تقسیم می‌شوند که تکنیک‌های ایستا در زمان کامپایل و تکنیک‌های پویا در زمان اجرای برنامه روی آن اعمال می‌شوند.

در ادامهٔ این مقاله ابتدا در بخش دوم، انواع تهدیدها در برابر نرم‌افزار را بررسی می‌کنیم. بخش سوم به تکنیک‌های محافظت از نرم‌افزار در برابر حملات تحلیلی اختصاص یافته است. در بخش چهارم به معرفی تکنیک‌های مقابله با دست‌کاری می‌پردازیم. در بخش پنجم تکنیک‌های مقابله با دزدی نرم‌افزار را ارائه می‌دهیم. بخش ششم به دسته‌بندی و مقایسهٔ تمامی تکنیک‌های معرفی‌شده می‌پردازد و در بخش هفتم نتیجه‌گیری را ارائه خواهیم کرد.

۲- انواع تهدیدها در مقابل نرم‌افزار

امنیت سامانه‌های رایانه‌ای می‌تواند از جهات و روش‌های مختلفی تهدید شود. حمله منع سرویس[3] می‌تواند سروری را از کار بیندازد، یک کرم می‌تواند باعث از بین رفتن داده‌های شخصی کاربر شود یا استراق سمع[4] می‌تواند در ارتباط بین مشتری و بانک توسط حملهٔ مرد در میانه[5] اطلاعات کارت اعتباری کاربر را فاش کند. مفهومی که تمام این سناریوها به‌صورت مشترک وجود دارد، وجود دشمن یا موجودیتی غیرقابل اعتمادی است که به سامانه از خارج آن حمله می‌کند. در اینجا فرض می‌کنیم رایانه‌ای که مورد حمله قرار گرفته است، توسط کاربری که خوش‌خیم[6] و قابل اعتماد[7] است کار می‌کند؛ اما اگر این فرض را نادیده بگیریم، اگر به هرکسی اجازهٔ کار با سامانه رایانه‌ای را بدهیم (از مدیران سامانه[8] تا یک کاربر عادی)، برای تأمین امنیت چنین سامانه‌ای، در سناریوی جدیدی قرار می‌گیریم که متأسفانه توجه کمتری به آن شده است. حملاتی که توسط یک کاربر مورد اعتماد انجام می‌شود، حملات مرد در انتها[9] نامیده می‌شود، که می‌تواند اشکال مختلفی داشته باشد [۱].

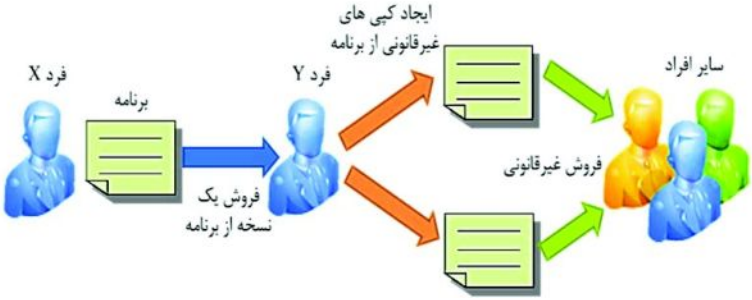
به‌طوراصولی دزدی نرم‌افزار[10] به معنای تهیه رونوشت‌های غیر قانونی از نرم‌افزار و استفادهٔ مجدد و فروش آنها توصیف شده است. به‌طور متوسط سالیانه دوازده میلیارد دلار به صنعت نرم‌افزار خسارت وارد می‌کند. بنابراین دزدی نرم‌افزار یکی از مهم‌ترین نگرانی‌های توسعه‌دهندگان نرم‌افزار است. متأسفانه تا زمانی که روش‌هایی آسان برای سرقت نرم‌افزار وجود داشته باشد و انجام آن برای سوء استفاده‌کنندگان سود فراوانی داشته باشد، ادامه خواهد یافت. پژوهش‌گران تکنیک‌های مختلفی برای جلوگیری یا حداقل سخت‌کردن دزدی نرم‌افزار ارائه داده‌اند.

همچنین بسیاری از توسعه‌دهندگان نرم‌افزار در مورد مهندسی معکوس[11] نرم‌افزارهایی که توسعه داده‌اند، نگرانند. موارد زیادی مشاهده شده که یک قطعهٔ خاص، ماژول یا الگوریتمی از یک نرم‌افزار توسط رقبا با مهندسی معکوس استخراج شده و در نرم‌افزارهای دیگر مورد استفادهٔ مجدد قرار گرفته است. چنین تهدیدهایی به‌تازگی بیشتر به نگرانی توسعه‌دهندگان تبدیل شده است؛ زیرا برنامه‌ها به جای فرمت کد دودویی بومی به فرمت‌هایی که به‌سادگی قابل مهندسی معکوس‌شدن هستند، توزیع شده‌اند. به‌عنوان مثال می‌توان به فرمت فایل طبقهٔ جاوا و ANDF در این خصوص اشاره کرد.

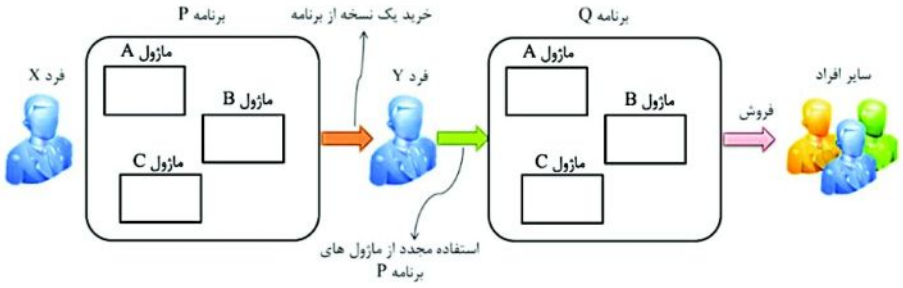
تهدید مرتبط دیگر، دست‌کاری[12] در نرم‌افزار است. بسیاری از نرم‌افزارهای تلفن همراه و برنامه‌های کاربردی تجارت الکترونیکی به‌صورت طبیعی دارای کلیدهای رمزنگاری و اطلاعات مخفی هستند. دزدان نرم‌افزار می‌توانند با استخراج، ویرایش و دست‌کاری آن‌ها، زیان‌های قابل توجهی به مالکیت معنوی صاحبان و تولیدکنندگان چنین نرم‌افزارهایی وارد کنند.

[1] Code obfuscation
[2] Birthmarking
[3] Denial-of-service attack
[4] Eavesdropper
[5] Man-in-the-middle
[6] Benign
[7] Trusted
[8] System administrators
[9] Man-at-the-end attacks
[10] Software Piracy
[11] Reverse engineering
[12] Tampering

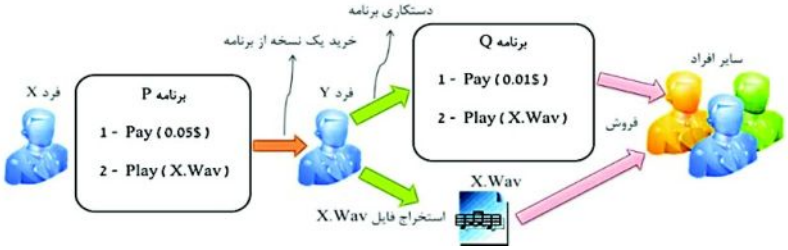
حال هرکدام از تهدیدهای معرفی‌شدهٔ بالا را با ذکر مثال و رسم تصویر بیشتر توضیح می‌دهیم. همان‌طور که اشاره شد، سه نوع تهدید در برابر نرم‌افزارها وجود دارد: دزدی نرم‌افزار، مهندسی معکوس (که گاهی تجزیه و تحلیل نرم‌افزار هم گفته می‌شود) و دستکاری نرم‌افزار.



(شکل۲): دزدی نرم‌افزار با تکثیر غیر قانونی محصول



(شکل۳): تحلیل نرم‌افزار و استفاده از بخشهایی از آن در محصولات دیگر



(شکل۴) دستکاری نرم‌افزار و استفاده از محتویات نرم‌افزار

در حالت نخست، فرد Y یک رونوشت از نرم‌افزار را به‌صورت قانونی از فرد X خریداری می‌کند و سپس چندین رونوشت غیر قانونی از نرم‌افزار را به افراد دیگر می‌فروشد. نمونه‌های زیادی از چنین رونوشت‌های غیر قانونی نرم‌افزار مانند سیستم‌های عامل و سایر برنامه‌های کاربردی موجود در بازار را می‌توان یافت که با قیمتی بسیار پایین‌تر از قیمت واقعی نرم‌افزار قابل خریداری هستند.

در حالت دوم ، فرد Y یک رونوشت قانونی از نرم‌افزار فرد X را خریداری می‌کند و آن را ترجمه معکوس کرده و پس از مهندسی معکوس، آن را در برنامه‌های کاربردی نوشته‌شدهٔ خودش می‌تواند مورد استفاده مجدد قرار دهد؛ سپس این بخش از نرم‌افزار مهندسی معکوس‌شده را که می‌تواند یک ماژول، الگوریتم یا بخشی از کد باشد، به همراه سایر ماژول‌های خود در قالب یک نرم‌افزار ارائه کرده و به فروش برساند. به‌عنوان مثال یک نرم‌افزار OCR را از توسعه‌دهندهٔ آن خریداری کرده، بخش تشخیص متن‌های دست‌نوشته را با مهندسی معکوس استخراج کرده و در نرم‌افزاری که تشخیص متن، تنها بخشی از آن است، مورد استفادهٔ مجدد قرار داده و به همراه امکانات اضافی دیگر دوباره به فروش می‌رساند.

در حالت سوم، فرد Y نرم‌افزاری را از فرد X تهیه می‌کند که به‌عنوان مثال دارای محتوای خاص و روشی برای

پرداخت مبلغی جهت دریافت آن محتوا است. اینجا فرد Y می‌تواند محتوا را از نرم‌افزار فرد X استخراج کند و جداگانه به فروش برساند یا با دسترسی به اطلاعات محرمانه موجود در نرم‌افزار، آنها را دست‌کاری کند. به‌عنوان مثال یک سامانهٔ فروش برخط محتوای چندرسانه‌ای که فرد Y می‌تواند با استخراج محتوا آنها را به‌صورت رایگان در اختیار کاربران قرار دهد یا آنها را به فروش برساند. همچنین می‌تواند با دست‌کاری نرم‌افزار اقدام به اخذ مبالغ بیشتر از شخص سوم کند[۲].

۳- تکنیک‌های مقابله با حملات تحلیل یا مهندسی معکوس

در این بخش به معرفی تکنیک‌های مقابله با حملات نرم‌افزاری می‌پردازیم.

۳-۱- مبهم‌سازی کد[1]

کولبرگ و همکاران در [۳] تکنیک مبهم‌سازی کد را معرفی کردند. براساس پیشنهاد آنها، مبهم‌سازی به عمل نامفهوم‌کردن یا حداقل، سخت‌کردن فهم کد گفته می‌شود. فرآیند مبهم‌سازی کد، انجام عملیات تبدیل‌ها روی کد است؛ به‌نحوی که ظاهر محتوایی کد را تغییر دهد؛ درحالی‌که ویژگی‌های جعبهٔ سیاه‌بودن برنامه را به همان شکل اصلی حفظ کند. تکنیک‌های مختلفی برای مبهم‌سازی کد تاکنون ارائه شده است. تکنیک‌های عمومی مبهم‌سازی کد، سعی در آشفته و گیج‌کردن تحلیل‌گر از فهم نحوه عملکرد برنامه دارند. این کار می‌تواند از تبدیلات ساده تا تغییرات پیچیده در جریان کنترل و دادهٔ برنامه باشد. کاربردهای تکنیک مبهم‌سازی کد، به حفاظت از مالکیت معنوی نرم‌افزار محدود نمی‌شود، بلکه این روش، همچنین به‌طور گسترده توسط نویسندگان کدهای مخرب و بدافزارها برای جلوگیری از تشخیص توسط آنتی‌ویروس‌ها هم مورد استفاده قرار می‌گیرد. بسیاری از ویروس‌ها از مبهم‌سازی کد و تبدیل‌های مبهم‌سازی با تغییر مداوم کدشان به‌منظور فریب ضدویروس‌ها استفاده می‌کنند [۳]. بر طبق [۴] کیفیت مبهم‌سازی براساس معیارهای قدرت[2](چقدر ابهام به برنامه اضافه می‌کند؟)، بازگردانی[3](تبدیل‌ها برای بازگردانی به حالت اوّلیه، توسط ابزارهای از ابهام خارج‌ساز[4] خودکار چقدر مشکل است؟)، نهان‌بودن[5](کد مبهم شده چگونه با بقیه برنامه مخلوط شده است؟) و هزینه[6] (چقدر سربار محاسباتی به برنامه مبهم‌شده افزوده شده است؟) قابل سنجش است. تاکنون تبدیل‌های گسترده‌ای برای اعمال مبهم‌سازی روی کدهای موجود معرفی شده است که مهم‌ترین آن‌ها تبدیل‌های عمومی برای مبهم‌سازی است که توسط کولبرگ و همکاران [۴] معرفی شده است. الگوهای پیشنهادی شامل تبدیل‌های مبهم‌سازی کنترلی، تبدیل‌های محاسباتی، مبهم‌سازی تجرید داده، مبهم‌سازی تجریدهای رویه‌ای[7](که خود شامل تفسیر جدول[8]، روش‌ها inline و outline و Clone Methods است) و مبهم‌سازی انواع دادهٔ داخلی[9] (شامل شکستن متغیرها، تبدیل داده‌های ایستا به داده رویه‌ای و ادغام متغیرهای اسکالر) است. مبهم‌سازی کد با امکان جلوگیری از تحلیل ایستا برنامه که توسط وانگ و همکارانش [۵] معرفی شده است شامل تبدیل‌های جریان کنترل و تبدیل‌های جریان داده است. در مبهم‌سازی کد در مرحلهٔ هم‌گذاری معکوس[10] که توسط وانگ و همکاران [۵] معرفی شده است درج بایت‌های آشغال[11]، جلوگیری از روش هم‌گذاری معکوس خطی و جلوگیری از هم‌گذاری معکوس بازگشتی (شامل توابع پرشی، تبدیل فراخوانی[12]، گزاره‌های مبهم و دست‌انداختن جدول پرش[13]) مورد استفاده قرار می‌گیرد.

۳-۲- رمزنگاری جعبهٔ سفید[14]

این روش به‌طور فزآینده‌ای در برنامه‌های کاربردی که در محیط‌ها و دستگاه‌های باز اجرا می‌شوند (مانند رایانه‌های شخصی، تبلت‌ها و گوشی‌های تلفن همراه هوشمند) مورد استفاده قرار می‌گیرد. از آنجایی که در چنین محیط‌هایی مهاجم کنترل کاملی بر روی پلتفرم و پیاده‌سازی نرم‌افزار دارد، نرم‌افزار در برابر حملات بسیار آسیب‌پذیرتر است. به عبارت دقیق‌تر مهاجم به‌راحتی می تواند کد دودویی برنامه و صفحات متناظر آن را در حافظه، در زمان اجرای برنامه تحلیل، فراخوانی‌های سیستمی برنامه را پیگیری، کد دودویی و اجرای برنامه را دستکاری و انواع حملات مورد

[1]Code Obfuscation
[2]Potency
[3]Resilience
[4]automatic deobfuscator

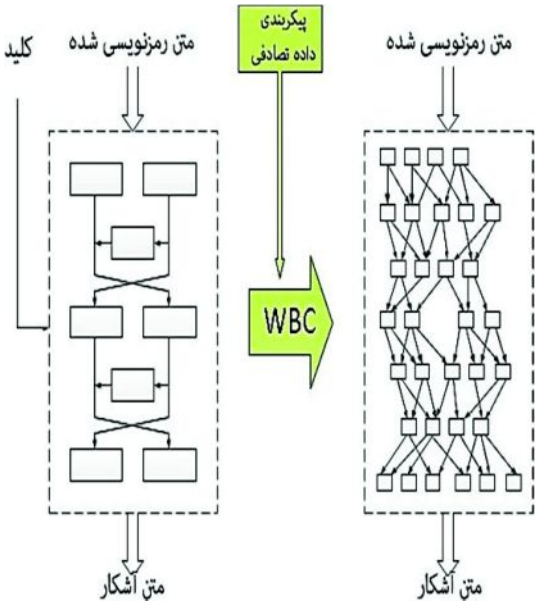
[5]Stealth
[6]Cost
[7]Procedural Abstractions
[8]Table Interpretation
[9]built-in data types
[10] Disassembly
[11]Junk Insertion
[12]Call conversion
[13]Jump Table Spoofing
[14]White-box cryptography

نظر خود را روی برنامه اعمال کند. چنین زمینه‌های حمله همان‌طور که در قبل اشاره شد، به‌عنوان حملات جعبهٔ سفید شناخته می‌شوند. چالش اصلی رمزنگاری جعبهٔ سفید در فرآیند اجرایی یک الگوریتم رمزنگاری، توانایی ایمن‌نگاه‌داشتن اطلاعات کلیدی مربوط به رمزنگاری با وجود حملات جعبهٔ سفید است. بنابراین رمزنگاری جعبهٔ سفید، یک راه حل بسیار مهم در رویکردهای حفاظت از نرم‌افزار است. این تکنیک اجازه می‌دهد تا عملیات رمزنگاری بدون افشای اطلاعات محرمانه، مانند کلید رمزنگاری انجام شود. بدون استفاده از این روش، مهاجم می‌تواند به‌راحتی از روی پیاده‌سازی دودویی، از روی حافظه یا پی‌گیری اطلاعات در زمان اجرا، کلیدهای مخفی مورد استفاده را به‌دست آورده و به اطلاعات حساس دسترسی داشته باشد.

(**شکل** (۵) مفهوم رمزنگاری جعبهٔ سفید را به‌صورت سطح بالا در ارتباط با پیاده‌سازی کلید ثابت نشان می‌دهد. در چنین پیاده‌سازی، کلید در کد به‌صورت پیاده‌سازی جاسازی‌شده قرار دارد. تبدیل‌های رمزنگاری جعبهٔ سفید، کد را برای برنامه کاربردی تولید[1] می‌کند، اما استخراج کلید که در کد تعبیه شده است، مشکل است.

۳-۳- رمزنگاری کد[2]

ابزارهایی مثل بستهٔ رمزنگاری[3] کد یک برنامهٔ نرم‌افزاری را جهت جلوگیری از دسترسی مهاجمان به کدها رمزگذاری می‌کنند. تکنیک رمزنگاری کد از نرم‌افزار در مقابل مهندسی معکوس ایستا و حملات دست‌کاری محافظت می‌کند و مهاجم زمانی که کد در دیسک ذخیره شده یا از طریق شبکه منتقل می‌شود کد را نمی‌تواند بازرسی کند یا هرگونه تغییر ساختاری در آن به‌وجود آورد. در طول اجرای برنامه بخش‌هایی از کد رمزگشایی می‌شوند که این کار با یک کلید مخفی انجام می‌شود. متأسفانه، در این لحظه کد به‌طور واضح در حافظه خواهد بود که می‌توان آن را مشاهده کرد. کد مشاهده‌شده را می‌توان بعدها اشکال زدایی[4] یا ترجمهٔ معکوس[5] کرد و این چالش مهم‌ترین آسیب‌پذیری این تکنیک است. در کار ساندر و همکاران [۶] روشی برای رمزگذاری برنامه جهت حفاظت از این الگوریتم‌ها در برابر آشکارسازی ارائه شده است. علاوه‌براین با استفاده از این روش، پروتکلی تعریف شده است که تحت شرایط مناسب اطمینان حاصل می‌کند که تنها کاربران مجاز توانایی مشاهدهٔ خروجی متن واقعی[6] برنامه را داشته باشند. نقطهٔ کلیدی در این تکنیک این است که توابع به‌شکلی رمزگذاری می‌شوند که قابل اجرا باقی می‌مانند. همچنین در این روش نشان داده شده است که چگونه از برنامهٔ به‌دست‌آمده به‌عنوان خروجی، اثر انگشت[7] گرفته می‌شود. این تکنیک به‌طورکامل مبتنی بر نرم‌افزار و تنها برای جلوگیری از تحلیل نرم‌افزار است و برای مقابله با دستکاری آن کاربردی ندارد.



(شکل۵): تشریح نحوه اعمال رمز نگاری جعبه سفید[۳۹]

۴-۳- روش‌های ویرایش سرخود[8]

یکی از انواع تبدیلات مهم مبهم‌سازی، تولید کد به‌صورت پویا[9] است. این کدها به‌عنوان کدهای ویرایش سرخود شناخته می‌شوند که باعث بالارفتن پیچیدگی اضافی برنامه می‌شوند. این کار همچنین به‌عنوان مانعی برای تطبیق الگوی برنامه با یک برنامهٔ عادی است. کدهای ویرایش سرخود دارای سابقه طولانی در مبهم‌سازی کد هستند. برنامه‌های DOS اولیه از این تکنیک برای مخفی‌کردن عمل‌کردهای محافظتی خود استفاده می‌کردند و برنامه‌های نرم‌افزاری مشروط[10] با استفاده از این روش از تغییر ساختار

[1] Generate
[2] Code Encryption
[3] cryptographic wrappers
[4] Debug
[5] Decompile

[6] Cleartext
[7] Fingerprint
[8] Self-Modifying Techniques
[9] Dynamic code creation
[10] Shareware

داخلی خود جلوگیری می‌کنند [۷]. تاکنون چندین کار مختلف در رابطه با تکنیک‌های ویرایش سرخود انجام شده است که از مهم‌ترین آن‌ها می‌توان به تکنیک‌های زیر اشاره کرد:

- روش‌های Kanzaki [۸]: در این روش دستورهای برنامه با رونویسی دستورهای ساختگی[1] پیچیده شده و فهم آن مشکل می‌شود. این ایده را می‌توان به این شکل خلاصه کرد که نرم‌افزار دارای بخشی است که در زمان اجرا، کد ساختگی بقیهٔ بخش‌ها را با کدهای اولیه بازیابی می‌کند و پس از اجرای آنها دستورهای ساختگی را با دستورهای اصلی جایگزین می‌کند.

- روش Madou [۹]: این روش یک موتور بازنویسی[2] را پیشنهاد داده است که توابع برنامه را براساس یک قالب، قبل از اجرای آن بازنویسی می‌کند. از یک مولد عدد شبه‌تصادفی[3] برای کدگذاری قالب بازنویسی مورد استفاده قرار می‌گیرد، و تکنیک‌هایی مانند مبهم‌کردن متغییرها براساس عدد شبه‌تصادفی عمل می‌کنند. موتور بازنویسی در داخل برنامه تعبیه شده است. عدد شبه‌تصادفی به‌عنوان رمز جریان و عدد تولیدی به‌عنوان کلید مورد استفاده قرار می‌گیرد.

- روش Cappaert و همکارانش [۱۰]: آن‌ها راه‌هایی برای جلوگیری از تحلیل ایستا کد اجرایی پیشنهاد داده‌اند که بر رمزگذاری کد، تأکید دارد. آن‌ها بین روش‌های رمزگذاری حجیم[4] که کل برنامه را در زمان اجرا باهم رمزگذاری می‌کند و رمزگشایی بر حسب تقاضا[5] که فقط قطعه‌ای از کد را که برای اجرا لازم است، رمزگذاری می‌کند، تمایز قائل شده‌اند. تمرکز اصلی آنها بر روی رمزگذاری بر حسب تقاضا است؛ به‌طوری که توابع را قبل از این‌که اجرا شوند، رمزگشایی کرده و پس از استفاده دوباره رمزگذاری می‌کنند. در طراح پیشنهادی حفاظ‌های رمزی[6] معرفی شده است. این حفاظ‌ها روتین‌هایی هستند که یک تابع را براساس یک‌پارچگی با کد توابع دیگر رمزگشایی می‌کنند. توابع چکیده‌ساز رمزگذاری[7] برای به‌دست‌آوردن کلید از کد توابع دیگر مورد استفاده قرار می‌گیرد. هر تابع با استفاده از کلیدی محافظت شده است که تابع فراخواننده آن را فراهم می‌کند و زمانی که تابع به دستور بازگشت[8] می‌رسد دوباره رمزگذاری می‌شود.

[1] Dummy instructions
[2] Rewriting engine
[3] Pseudo-random number generator
[4] Bulk encryption
[5] On-demand encryption
[6] Crypto guards
[7] Cryptographic hash

۳-۵- حفاظت نرم‌افزار مبتنی بر شبیه‌سازی[9]

کیمبال در [۱۱] روش حفاظت نرم‌افزار مبتنی بر شبیه‌سازی را پیشنهاد داد. این تکنیک نرم‌افزار را در برابر مهندسی معکوس کد محافظت می‌کند. برای این منظور از امضای کد و کد رمزنگاری‌شده استفاده می‌شود. این سازوکارهای حفاظتی بر روی شبیه‌سازهای مطمئن اجرا می‌شوند. یک‌پارچگی و قابلیت اطمینان سازوکارهای حفاظتی مشروط و وابسته به دسترسی مهاجمان به محیط شبیه‌سازی است. در ادامه دو نمونه از سازوکار حفاظتی را بررسی می‌کنیم:

- **مدل امولاتوری کیسهٔ شن**[10]: در این سازوکار، مسئله اصلی این است که بدافزار هسته قادر است تا سازوکارهای حفاظتی هسته را تغییر دهد (منظور حمله به این سازوکارها می‌باشد). روال زیر مدل امولاتوری کیسه شن را توصیف می‌کند:

۱) سیستم عامل میزبان، دستورالعمل‌ها را از حافظهٔ مربوط به سیستم عامل مهمان به حافظهٔ خود رونوشت می‌کند (این مرحله در سازوکارهای حفاظتی برای سایر مراحل مورد نیاز است).

۲) دستورالعمل‌های سیستم عامل مهمان به مجموعه دستورالعمل‌های سیستم عامل میزبان ترجمه و یا تفسیر می‌شود. وقتی‌که این مجموعه دستورالعمل‌های ترجمه‌شده اجرا می‌شوند، وضعیت حافظه و ثبات‌های سیستم عامل مهمان تغییر پیدا می‌کند، به‌گونه‌ای که گویا دستورالعمل‌ها بر روی ماشین مهمان اجرا می‌شوند.

[8] Return
[9] Emulation-based software protection
[10] Sandbox Emulation

۳) فرایند ترجمه تضمین می‌کند که دستورالعمل‌های سیستم عامل مهمان، حافظهٔ سیستم عامل مهمان را خوانده و نوشته‌اند. حافظهٔ سیستم عامل میزبان برای دستورالعمل‌های سیستم عامل مهمان غیرقابل دسترسی است.

(**شکل** ۶ مدل امولاتوری کیسه شن را برحسب روال توضیح داده شده، نشان می‌دهد.

(شکل۶:) مدل امولاتوری کیسه شن [۱۱]

- **اجرای کد رمزنگاری‌شده براساس شبیه‌سازی:** این تکنیک‌ها، فرایند مهندسی معکوس کد را کند می‌کنند. بنابراین کدها رمزنگاری می‌شوند. رمزنگاری به‌صورت ایستا کد را محافظت می‌کند؛ درنتیجه بایستی کدها را قبل از اجرا رمزگشایی کرد. این سازوکار حفاظتی کدهای رمزنگاری‌شده را در طول اجرا در داخل یک محیط شبیه‌سازی‌شده نگهداری می‌کند. روال‌های رمزگشایی و کلیدهای سری خارج از این محیط شبیه‌سازی هستند. روال کار این سازوکار به‌صورت زیر است:

۱) سیستم عامل میزبان، دستورالعمل‌های رمزنگاری‌شدهٔ سیستم عامل مهمان را به حافظهٔ خود رونوشت می‌کند. دستورالعمل‌های رمزنگاری‌شده سیستم عامل مهمان در حافظهٔ سیستم عامل میزبان رمزگشایی می‌شوند. دستورالعمل‌های رمزگشایی‌شده برای همیشه خارج از سیستم عامل مهمان می‌باشند و برای سیستم عامل مهمان غیرقابل دسترس است.

۲) دستورالعمل‌های رمزگشایی‌شده سیستم عامل مهمان به مجموعه دستورالعمل‌های سیستم عامل میزبان ترجمه یا تفسیر می‌شوند. وقتی‌که این دستورالعمل‌های ترجمه‌شده اجرا شدند، حافظهٔ مربوط به سیستم عامل مهمان تغییر خواهد کرد.

۳) فرآیند ترجمه اطمینان حاصل می‌کند که سیستم عامل مهمان هرگز دستورالعمل‌های رمزگشایی‌شده را نمی‌خواند. دستورالعمل‌های رمزنگاری‌شده در داخل سیستم عامل میزبان به‌وسیلهٔ روال‌های مربوط به سیستم عامل میزبان رمزگشایی می‌شوند. به‌دلیل این‌که حافظهٔ سیستم عامل میزبان برای سیستم عامل مهمان غیرقابل دسترسی است، بنابراین می‌توان نتیجه گرفت که دستورالعمل‌های رمزگشایی‌شده و روال‌های رمزگشایی‌شده خارج از محیط حمله می‌باشند، بنابراین حملات کم‌تری صورت می‌گیرد.

۴)

(شکل۷): اجرای کد رمزنگاری شده [۱۱]

(***شکل*** (۷) نحوهٔ اجرای کد رمزنگاری‌شده را نشان می‌دهد. روال‌های توصیف‌شده در شکل (۷) برای حفاظت از نرم‌افزار در مقابل مهندسی معکوس کد است. بدین صورت که کد رمزنگاری‌شده در یک ماشین مطمئن رمزگشایی و اجرا می‌شود و درنهایت نتایج حاصل از اجرا به ماشین غیرمطمئن بازگردانده می‌شود.

۴- تکنیک‌های مقابله با حملات دست‌کاری

۴-۱- امضای کد[1]

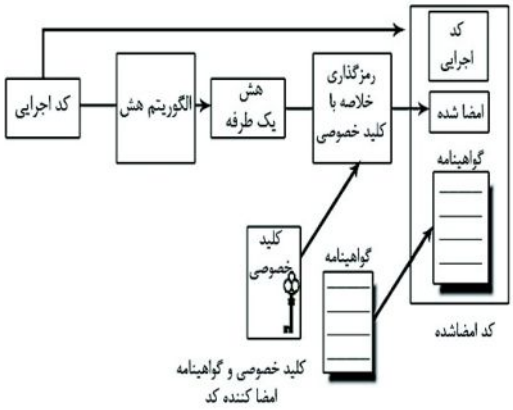
امضای کد، سازوکار مشترکی است که نویسندگان کدهای اجرایی از آن برای دفاع از حق نویسندگی خود و اطمینان از جامعیت کد که توسط اشخاص ثالث به‌صورت غیر مجاز مورد ویرایش قرار نگرفته باشد، استفاده می‌کنند. این تکنیک به‌صورت گسترده برای محافظت از نرم‌افزارهایی که روی اینترنت توزیع شده‌اند، استفاده می‌شود [۱۲]. تکنیک امضای کد که گاهی امضایشی[2] هم نامیده می‌شود، زیرمجموعه‌ای از امضای مدارک الکترونیکی[3] است. امضا اصالت کد را یعنی جایی که به‌درستی کد از آن آمده است، مشخص می‌کند. جامعیت هم با یک تابع درهم‌سازی امضا شده فراهم می‌شود که ثابت می‌کند کد، مورد دست‌کاری قرار نگرفته است.

امضای کد براساس زیرساخت کلید عمومی[4](PKI) کار می‌کند. PKI یک زیرساخت توزیع شده است که از توزیع و مدیریت کلیدهای عمومی و گواهینامه‌های دیجیتال[5] استفاده می‌کند. گواهینامه دیجیتال یک اظهارنامه امضا شده (با امضای دیجیتال) توسط شخص ثالث مورد اعتماد است که گواهینامهٔ تصدی[6] (CA) نامیده می‌شود و بین کلید عمومی و برخی از اطلاعات دیگر مانند نام دارنده مشروع کلید خصوصی که مرتبط با آن کلید عمومی است، ارتباط برقرار می‌کند. از اجتماع این اطلاعات درنهایت برای تشخیص هویت آن فرد استفاده می‌شود. همه شرکت‌کنندگان در سامانه می‌توانند از زوج نام-کلید هر گواهینامه موجودی تنها با اعمال کلید عمومی CA برای تأیید امضای دیجیتال CA استفاده کنند. این فرایند تأیید بدون نیاز به دخالت CA انجام می‌شود [۱۲].

در سامانه‌های رمزگذاری نامتقارن از دو کلید مرتبط، اما مختلف استفاده می‌شود که عبارتند از کلید عمومی[7] که آن را همه می‌دانند و کلید خصوصی[8] که تنها دارنده مشروع کلید عمومی از آن اطلاع دارد.

گواهینامهٔ مورد نیاز برای کد امضاشده را می‌توان به دو روش به دست آورد. یا توسط خود امضاکنندهٔ کد که با استفاده از ابزارهای امضای کد[9] ایجاد شده است و یا از یک CA به دست آمده است.

امضای کد به‌طور خلاصه به این شکل کار می‌کند: توسعه‌دهنده از یک تابع درهم‌سازی در کد استفاده می‌کند تا یک خلاصه[10] از کدش به دست آورد که به‌عنوان درهم‌سازی یک طرفه[11] شناخته می‌شود. این تابع درهم‌سازی، کدی با طول دلخواه را به‌صورت کدی با طول ثابت در می‌آورد. اغلب از "الگوریتم درهم‌سازی امن"[12] (SHA)، "الگوریتم خلاصه پیام۴"[13] (MD4) و MD5 برای این منظور استفاده می‌شود. طول نتیجه یا همان خلاصه بستگی به الگوریتم مورد استفاده دارد، اما اغلب به‌صورت ۱۲۸ بیتی است. پس از محاسبهٔ خلاصه، این خلاصه توسط کلید خصوصی توسعه‌دهنده رمزگذاری می‌شود که بخشی از گواهینامه توسعه‌دهنده است. بسته حاوی خلاصه رمزگذاری‌شده و گواهینامه دیجیتال توسعه‌دهنده در یک ساختار خاص کپسوله[14] می‌شود که بلاک امضا[15] نامیده می‌شود؛ سپس این بلاک امضا به‌عنوان امضای کد به انتهای کد اجرایی افزوده می‌شود. شکل ۸ مراحل کار را نشان می‌دهد.



شکل (۸): فرآیند امضای کد [۱۲]

مراحل بررسی به‌صورت زیر انجام می‌شود:

[1]Code Signing
[2]Object signing
[3]Electronic document signing
[4]Public key infrastructure
[5]Digital certificates
[6]Certificate Authority
[7]Public key
[8]Private key

[9]Code-signing toolkits
[10]Digest
[11]One-way hash
[12]Secure Hash Algorithm
[13]Message Digest Algorithm4
[14]Encapsulate
[15]Signature block

۱. گواهی‌نامه از روی بلاک امضا، بررسی می‌شود تا تأیید شود که توسط سامانه بررسی امضای کد به‌عنوان یک گواهی‌نامه با فرمت درست قابل تشخیص است.

۲. اگر فرمت صحیح باشد، گواهی‌نامه الگوریتم تابع درهم‌سازی‌ای که برای ساخت خلاصهٔ امضاشده داخل بلاک امضای دریافت‌شده مورد استفاده قرار گرفته، شناسایی می‌کند. با این اطلاعات، همان الگوریتم که خلاصهٔ اصلی ساخته شده است، روی کد اجرایی دریافتی اعمال می‌شود، مقدار خلاصه ایجاد و به‌طورموقتی در جایی ذخیره می‌شود. اگر فرمت صحیح هم نباشد، فرایند بررسی با شکست مواجه می‌شود.

۳. مقدار خلاصهٔ امضاشده از بلاک امضا گرفته می‌شود و با کلید خصوصی امضاکننده، رمزگشایی می‌شود، مقدار خلاصه‌ای که ابتدا توسط امضاکننده محاسبه شده بود، آشکار می‌شود. شکست در رمزگشایی، نشان‌دهندهٔ این است که کلید عمومی مورد استفاده صحت نداشته است. در این حالت مشخص می‌شود که امضا تقلبی است و فرایند بررسی با شکست مواجه می‌شود.

۴. مرحلهٔ ۳ بر روی خلاصهٔ محاسبه‌شده در مرحلهٔ دو هم انجام می‌شود. اگر دو مقدار با هم یکسان نباشند، مشخص می‌شود تغییراتی در کد به وجود آمده است و فرایند بررسی با شکست مواجه می‌شود. در غیر این صورت هویت امضاکنندهٔ کند شناسایی شده است.

۵. اگر عملیات موفق باشد، گواهی امضاکنندهٔ کد از بلاک امضا رونوشت و به گیرنده نشان داده می‌شود؛ سپس گیرنده می‌تواند مشخص کند امضاکننده مورد اعتماد است یا خیر. اگر باشد کد اجرا می‌شود و در غیر این صورت اجرا نمی‌شود.

۴-۲- نرم‌افزار مقابله با دست‌کاری Aucsmith

آقای Aucsmith در [۱۳] مدل حمله و اصول طراحی برای مقابله با حملات مطرح شده است. درنتیجه، معماری و پیاده‌سازی نرم‌افزار مقابله با دست‌کاری براساس اصول مطرح‌شده، توصیف شده است. معماری مطرح‌شده شامل سگمنت کد[1] است. این سگمنت، به‌صورت خوداصلاح، خودرمزگشا است و به‌صورت واحدی مستقر می‌شود. این سگمنت کد، هستهٔ بررسی یک‌پارچگی[2] نامیده می‌شود. درنهایت این سگمنت با سایر سگمنت‌ها ارتباط برقرار می‌کند تا یک مدل مطمئن متحد ایجاد شود. اصول طراحی برای نرم‌افزارهای مقابله با دست‌کاری، مبتنی بر نیاز به مخفی‌کردن اطلاعات سری نرم‌افزار و سختی تغییر این اطلاعات سری، است. چهار اصول مطرح‌شده عبارتند از:

- **پراکنده‌کردن اطلاعات سرّی از دریچهٔ زمان و فضا** : اطلاعات سرّی نباید در یک ساختار واحد حافظه قرار گرفته باشند؛ درغیراین‌صورت می‌توان با پویش حافظهٔ فعال به آن دسترسی پیدا کرد. علاوه‌براین، اطلاعات سرّی نبایستی توسط یک عمل واحد پردازش شود در غیر این‌صورت هر ناظر اجرای کدی می‌تواند به‌آسانی به آن دسترسی پیدا کند.
- **مبهم‌سازی عملیات درهم تنیده**[3] : یک وظیفهٔ کامل که با نرم‌افزار اجرا می‌شود، بایستی درهم تنیده باشد تا کوچک‌ترین بیت هر بخش به‌طور کامل اجرا شود. هدف از این اصل دست‌یابی به ویژگی تجزیه‌ناپذیری نرم‌افزار[4] است. بدین صورت که همهٔ بخش‌های اجرایی اجرا شوند یا هیچ یک از بخش‌ها. برای این منظور می‌توان از محیط‌های چندپردازنده‌ای استفاده کرد.
- **ایجاد کد منحصربه‌فرد** : برای جلوگیری از حملات سراسری بایستی هر نمونه از نرم‌افزار شامل عناصر منحصربه‌فرد باشند. این یکتایی یا منحصربه‌فردبودن می‌تواند به‌صورت ایجاد برنامه با توالی کدهای متفاوت یا کلیدهای رمزنگاری، به برنامه اضافه شود.
- **اتحاد مطمئن** : عمل‌کرد صحیح یک توالی از کد، بایستی به‌طور متقابل وابسته به عملکرد صحیح سایر توالی‌های کد باشد.

ذکر این نکته ضروری است که این اصول هر کدام به‌تنهایی، استقامت دست‌کاری نرم‌افزار را تضمین نخواهد کرد؛ بنابراین تمامی این اصول باید رعایت شوند. تمامی این اصول برای ایجاد هسته‌های بررسی یک‌پارچگی(همان IVK) اعمال می‌شوند.

معماری نرم‌افزار استقامت دست‌کاری شامل دو قسمت است:

۱) **هسته‌های بررسی یک‌پارچگی** : این هسته‌ها، تکه کدهای کوچکی هستند که با اصول طراحی اشاره‌شده

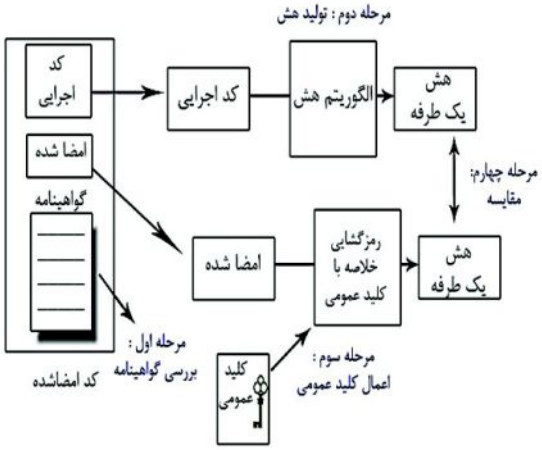
[1] Code segment

[2] Integrity Verification Kernel
[3] Interleaved operations
[4] Software atomicity

درهم آمیخته است (به عبارت دیگر مجهز به اصول بالاست) بنابراین به این راحتی‌ها قابل دست‌کاری نیست.

۲) **سازوکار اتحاد مطمئن** : این سازوکارها از ویژگی‌های IVK ها در یک پروتکل قوی استفاده می‌کنند، بنابراین IVK ها قادرند تا سایر IVK ها را بررسی کنند. این بررسی‌های متقابل استقامت دست‌کاری نرم‌افزار را به‌طور کلی افزایش می‌دهند.

در (شکل فرآیند تأیید اعتبار مشاهده می‌شود.



(شکل ۹): فرآیند تأیید اعتبار کد [۱۲]

۴-۳- حفاظ‌های نرم‌افزاری[1]

چانگ و همکاران در [۱۴] ایده سنتی داشتن یک واحد امنیتی که کد را بررسی کرده و خود را ویرایش می‌کند به این حالت گسترش داده‌اند که برنامه توسط تعدادی از این واحدهای امنیتی که حفاظ نامیده می‌شوند و همراه برنامه اجتماع یافته‌اند محافظت می‌شود که تکنیک آنها تحت عنوان حفاظ‌های نرم‌افزاری معرفی شده است. حفاظ‌های موجود در یک شبکه برای دفاع از خود در برابر حملات، از یکدیگر محافظت می‌کنند؛ که این کار به روشی به‌هم‌پیوسته انجام می‌شود. شکستن شبکه‌ای از موجودیت‌های امنیتی مشکل‌تر از یک ماژول امنیتی است؛ زیرا امنیت در میان تمام آنها به اشتراک گذاشته شده است؛ و به‌علاوه آنها از یکدیگر محافظت می‌کنند؛ چون راه‌های زیادی برای ساختاردهی به شبکه‌ای از حفاظ‌ها وجود دارد؛ برای مهاجمان پیش‌بینی‌کردن فرم قرارگیری و نحوهٔ عملکرد آنها سخت است؛ به‌علاوه درصورتی‌که نیاز به بالابردن سطح امنیت برنامه باشد، می‌توان حفاظ‌های جدیدی را به برنامه اضافه کرد تا بهتر از آن محافظت به عمل آید.

۴-۴- درهم‌سازی فراموش‌کار[2]

چنگ و همکاران [۱۵] روش درهم‌سازی فراموش‌کار را پیشنهاد داده‌اند. در یک مدل محاسباتی ساده، یک تابع یا برنامه به‌صورت دنباله‌ای از دستورالعمل‌های ماشین $I = \{i_1, i_2, \ldots, i_n\}$ که خانه‌های حافظه $M = \{m_1, m_2, \ldots, m_k\}$ را خوانده و یا می‌نویسند، مطرح می‌شود. در این مدل پیکربندی اولیه حافظه با M^0 و شمارنده دستورالعمل با C و مقدار اولیه‌اش با C^0 نشان داده می‌شود. ایدهٔ اصلی در این روش تسخیر ردیابی اجرای توابع (T) برای محاسبه مقدار درهم‌سازی H است که در شکل (۱۰) نشان داده شده است. به‌دلیل این‌که ردیابی برنامه مسیر اجرای واقعی برنامه را منعکس می‌کند، مقدار درهم‌سازی محاسبه‌شده به‌عنوان یک امضای قوی برای رفتار توابع است. این مقدار تابعی از کد، داده، پیکربندی اولیهٔ ماشین و پارامتر ورودی P است.

$$H \leftarrow H(T) \leftarrow H(I, M, C0, M0, P)$$

علامت ← نشان‌گر وابستگی است. در این مدل، محیط خارجی تابع در پیکربندی اولیهٔ حافظه، رمزنگاری شده است. تغییر مستقیم کد یا داده، منجر به تغییر مقدار درهم‌سازی فراهم‌شده با T است که شامل اطلاعات کافی در مورد اجرای واقعی برنامه است.

۴-۵- مجازی‌سازی[3]

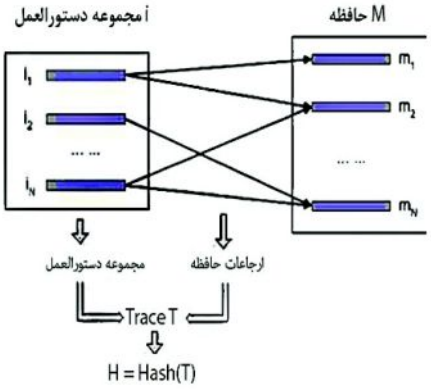
قوش و همکارانش [۱۶] روش مجازی‌سازی که روشی قوی و امن برای مقاومت در برابر دست‌کاری نرم‌افزار با استفاده از مجازی‌سازی سطح فرآیند است، پیشنهاد دادند. ماشین مجازی[4] (VM) به‌عنوان یک روتین رمزگشایی در زمان[5] عمل می‌کند. علاوه‌بر این، VM به‌صورت دوره‌ای کلیه کدهایی را که درقبل رمزگشایی شده است، رها می‌کند که این کار با تخلیه‌کردن کشی که حاوی کدهای رمزگشایی شده است، انجام می‌شود. تکنیک آنها همچنین برای جلوگیری از حملات پویا به‌صورت مداوم کد برنامه را تغییر می‌دهد و همچنین سازوکارهای پنهانی دارد که باعث

[1] Software Guards

[2] Oblivious Hashing
[3] Virtualization
[4] virtual machine
[5] just-in-time

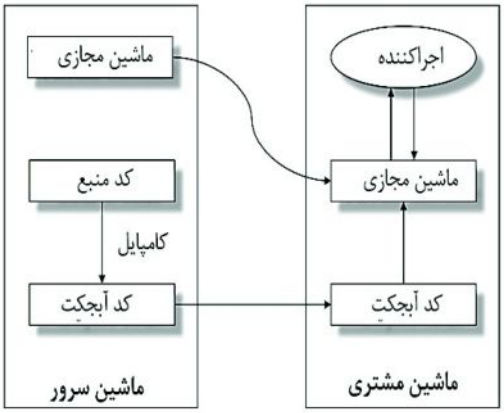
به‌راه‌انداختن عملیات پاک‌سازی می‌شود. این تکنیک شامل موارد زیر است:

- مجازی‌سازی : یک VM کدها را رمزگشایی کرده و کش خود را به‌صورت دوره‌ای پاکسازی می‌کند.
- رمزگذاری کد : بلاک‌های کد رمزگذاری می‌شوند.
- تغییر مداوم کد



(شکل ۱۰): مدل انتزاعی برای درهم‌سازی فراموش‌کار [۱۵]

مجازی‌سازی مشکل را تا حدی تغییر می‌دهد، طوری‌که به‌عنوان مثال VM یک زمینهٔ دیگر برای حمله ایجاد می‌کند، که یک سازوکار خودبررسی اضافی، جامعیت VM را بررسی می‌کند. (شکل ۱) نشان می‌دهد که مالک نرم‌افزار نیاز دارد دانشی نسبت به VM داشته باشد تا کد میانی را تولید کند. کاربر نهایی نیاز دارد تا هر دوی کد میانی و VM را درک تا برنامه واقعی را اجرا کند.



(شکل ۱۱) در مورد مجازی‌سازی، یک برنامه به یک VM و کد آبجکت شکسته می‌شود. هر دو بخش نرم‌افزاری باید برای اجرای برنامه به ماشین مشتری ارسال شوند [۱۶]

۴-۶- نظارت پویا[1]

Blietz [۱۷] تکنیک نظارت پویا را پیشنهاد داده که در آن، یک مدل با دو جریان دستورالعمل[2] (دو فرایند) برای مقابله با دست‌کاری مورد استفاده قرار می‌گیرد. فرایند ناظر[3] (M-process) به‌صراحت برای نظارت بر جریان کنترل فرایند برنامه اصلی[4] (P-process) طراحی شده است. در مرحلهٔ کامپایل، برنامه به دو فرایند کامپایل می‌شود: P-process و M-process. فرایند ناظر شامل شرایط سازگاری جریان کنترل برای P-Process است. P-process اطلاعات جریان کنترل را در بازه‌های زمانی ثابت به M-process ارسال می‌کند که سربار قابل قبولی دارد. درصورتی‌که تخلفی نسبت به شرایط جریان کنترل قرار گرفته در M-process شناسایی شود، M-process یک عکس‌العمل ضد دست‌کاری مانند پایان‌دادن به فرایند P-process انجام می‌دهد. با این طراحی، انتظار می‌رود فرایند ناظر فشرده باشد. ازاین‌رو می‌توان برای محافظت از M-process از یک تکنیک خیلی گران‌تر استفاده کرد که در پیاده‌سازی Blietz از نسخه‌ای از طرح Aucsmith استفاده شده است و با مترجم gcc زبان C پیاده‌سازی شده است. همچنین چندین تکنیک مبهم‌سازی، نظارتی و رمزگشایی پویا دیگر در این پیاده‌سازی تعبیه شده است.

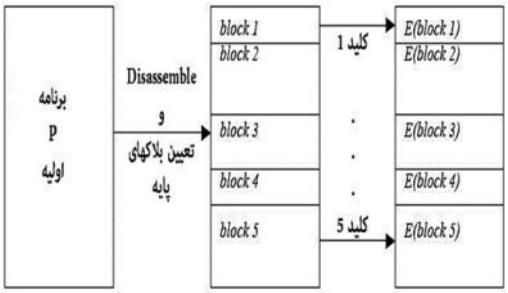
۴-۷- درهم‌سازی چندبلوکی[5]

Wang در [۱۸] روش درهم‌سازی چندبلوکی را معرفی کرد. در این راه‌کار، برنامهٔ دودویی به چندین بلاک مستقل با اندازه‌های متفاوت تقسیم می‌شود که هیچ یک از این بلاک‌ها حاوی دستور پرش به بلاک بعدی است. بلاک‌ها با عدد درهم‌سازی بلاک قبلی کدگذاری می‌شوند. به‌عنوان مثال آخرین بلاک با عدد درهم‌سازی بلاک ماقبل آخر رمزنگاری می‌شود. بلاک نخست رمزنگاری نمی‌شود و به‌عنوان بلاک پایه در نظر گرفته می‌شود. یک کنترل‌گر برنامه شامل روال رمزگشایی ساخته شده و در انتهای برنامه قرار داده می‌شود. هر بلاک شامل اشاره‌گری به برنامهٔ کنترل‌گر است تا بلاک بعدی با عدد درهم‌سازی بلاک فعلی رمزگشایی شود. وقتی‌که برنامه اجرا شد، بلاک پایه عدد درهم‌سازی خود را محاسبه کرده و به برنامهٔ کنترل‌گر

[1] Dynamic Monitoring
[2] Instruction-stream
[3] Monitor process
[4] Main Program process
[5] Multi-block Hashing Scheme

می‌دهد تا بلاک بعدی رمزگشایی شود. این روال آن‌قدر ادامه می‌یابد تا به انتهای برنامه برسیم. مقدار درهم‌سازی در طول اجرای برنامه به‌صورت پویا محاسبه می‌شود؛ بنابراین مهاجم نمی‌تواند کدهای دودویی را بعد از حفاظت آنها تغییر دهد [۱۸و۱۹].

تعداد بلاک‌ها می‌تواند از یک بلاک برای کل برنامه تا یک بلاک به‌ازای هر دستورالعمل متفاوت باشد. این موضوع واضح است که در محیط‌های دوردست تخصیص یک بلاک به‌ازای هر دستورالعمل منجر به دسترسی به حداکثر امنیت می‌شود. دستورالعمل می‌تواند در داخل پردازندهٔ خاص رمزگشایی شود. به هر حال گذاشتن چنین بار کاری عظیمی بر دوش پردازنده غیرمعقول است. بنابراین بایستی تعداد بلاک‌ها به‌طور قابل توجهی توازنی بین سطح امنیت و سرعت برنامه باشد. نکتهٔ قابل طرح این است که تعداد بلاک‌ها به روند کنترلی واقعی برنامه، وابسته است. (شکل ، الگوی درهم‌سازی بلاک‌های چندگانه را نشان می‌دهد[۱۸و۱۹].



(شکل ۱۲): الگوی درهم سازی بلاک‌های چندگانه [۱۸]

۵- تکنیک‌های مقابله با دزدی نرم‌افزار

۵-۱- تصدیق از راه دور[1]

از آنجایی که صاحبان محصولات نرم‌افزاری، علاقه‌مند به بررسی صحت و درستی ماشین‌های میزبانی که نرم‌افزار روی آنها اجرا می‌شود هستند همانطور که در روش مشتری-سرویس دهنده[2] نرم‌افزار روی سرور مورد اعتماد اجرا می‌شود، تصدیق از راه دور تکنیکی است که اجازه می‌دهد اشخاص ثالث[3] مورد اعتماد از راه دور حالت یک سامانهٔ رایانه‌ای را بررسی کنند، به‌عنوان مثال، نرم‌افزار نصب و اجراشده روی آن رایانه را بررسی کنند. اگر تغییرات غیر قانونی و غیر مجاز در نرم‌افزار اعمال شده باشد، تصدیق از راه دور باید این تغییرات را شناسایی و اطلاع‌رسانی کند تا اقدامات متقابل انجام شود. درحالی‌که این سناریو به‌نحوی با بستر حملات جعبهٔ سفید متفاوت است؛ اما بخش راه دور می‌تواند در برابر حملات جعبهٔ سفید آسیب‌پذیر باشد. ازاین‌رو اغلب تکنیک‌های حفاظتی با سایر تکنیک‌ها که مورد استفاده قرار می‌گیرند، مرتبط هستند و ترکیب می‌شوند [۲۰].

به‌دلیل این‌که اغلب اعتماد به صحت نرم‌افزار روی میزبان‌های غیر قابل اعتماد سخت است، تصدیق از راه دور گاهی اوقات بر یک ماژول پلتفرم قابل اعتماد متکی است (به‌عنوان مثال ماژول سخت‌افزاری). با این حال، Seshadri و همکارانش[۲۱] یک روش تصدیق از راه دور برای وسایل تعبیه شده معرفی کردند که به یک ماژول پلتفرم قابل اعتماد نیازی ندارد. اخیراً آنها یک روش سازگار با سیستم‌های Legacy PC پیشنهاد دادند که پیشگام[4] نامیده می‌شود[۲۲،۲۳]. این روش متشکل از یک پروتکل چالش-و-پاسخ[5] دو مرحله‌ای است. ابتدا، تصدیق‌کننده توسط یک عامل تصدیق که در میزبان غیر قابل اعتماد قرار دارد، به بررسی صحت نرم‌افزار می‌پردازد؛ سپس، این عامل تصدیق جامعیت برنامه اجرایی را گزارش می‌دهد.

بحث دیگر این است که ممکن است، دشمن سعی در نشان‌دادن برنامهٔ دست‌کاری‌شده به‌عنوان برنامهٔ اولیه و مجاز داشته باشد. به‌عنوان مثال وقتی دشمن سعی در تولید یک جمع‌آزمای[6] صحیح در زمان اجرای کد دست‌کاری‌شده دارد، این مسئله به دلیل رکود اجرا[7] قابل کشف است. وقتی یک حمله رونوشت حافظه انجام می‌شود، یک رکود اجرا به‌دلیل ترکیب مقدار شمارندهٔ برنامه و یا مقدار اشاره‌گر داده به محاسبه جمع‌آزما رخ می‌دهد. چون دشمن به جعل این مقادیر نیازمند است، این کار باعث افزایش زمان اجرا می‌شود.

با این حال، طراحی تابع جمع‌آزما به‌صورت پیش‌گام چندین محدودیت دارد:

- تابع جمع‌آزما باید خیلی کارا باشد. اگر دشمن قادر به بهینه‌سازی تابع جمع‌آزما باشد، او زمان خوبی برای انجام اعمال خراب‌کارانه به دست خواهد آورد.

[1] Remote Attestation
[2] Client/Server
[3] Third party
[4] Pioneer
[5] Challenge-Response
[6] Checksum
[7] Execution slowdown

- برای به حداکثررساندن سربار دشمن، تابع جمع‌آزما حافظه را به‌صورت پیمایش شبه‌تصادفی می‌خواند. این از پیش‌بینی خواندن حافظه از قبل توسط دشمن جلوگیری می‌کند.
- زمان اجرای تابع جمع‌آزما باید تجدیدپذیر باشد. ازاین‌رو، پیش‌گام باید در مد ناظر و در حالتی اجرا شود که وقفه‌ها غیر فعال شده باشند.

امنیت تکنیک پیش‌گام به سه فرض مهم وابسته است: نخست‌این‌که، تصدیق‌کننده نیاز به فهم کامل پیکربندی سخت‌افزار پلتفرم غیر قابل اعتماد دارد که شامل مدل CPU، سرعت کلاک و تأخیر حافظه می‌باشد تا زمان اجرایی را که انتظار می‌رود در حالت بدون دستکاری پیش آید محاسبه کند. اگر دشمن بتواند CPU را جایگزین یا کلاک را تغییر دهد، می‌تواند زمان اجرا را مورد نفوذ قرار دهد؛ ازاین‌رو، سامانهٔ پیش‌گام فرض می‌کند که پیکربندی سخت‌افزار توسط موجودیت تصدیق مشخص شده است و نمی‌تواند تغییر کند.

دوم‌این‌که، دشمن می‌تواند به‌عنوان یک شخص در وسط[1] عمل و کد را به یک واحد محاسباتی سریع‌تر محول کند تا جمع‌آزما را از طرف آن محاسبه کند. این حالت را حملات پراکسی[2] می‌نامند. برای اجتناب از این، پروتکل پیش‌گام فرض می‌کند کانال ارتباطی تصدیق‌شده بین موجودیت تصدیق و پلتفرم اجرایی غیر قابل اعتماد وجود دارد.

سوم این‌که، یک مشکل عمومی‌که باقی می‌ماند، تأخیر است. بنابراین، پیشگام فرض می‌کند موجودیت تصدیق درنزدیکی پلتفرم اجرای غیر قابل اعتماد قرار دارد.

در مطالعه‌ای دیگر Galay و همکارانش[۲۴] بر اساس زمان اجرای یک عامل تصدیق در "سیستم‌های عامل اجرایی زمانی"[3] سعی در کسب قابلیت اعتماد دارند. در تکنیک آنها برخلاف سامانه پیش‌گام، که از یک تابع تصدیق استفاده می‌کند، عامل تصدیق آنها حالت متحرک و قابل جابه‌جایی دارد.

۵-۲- تنوع نرم‌افزار[4]

این تکنیک ایده‌ای است که درواقع مبتنی بر بخشی از فرهنگ نرم‌افزار است، اما به‌تازگی جایگاه با ارزشی در جامعهٔ امنیت پیدا کرده است. ایدهٔ مربوط به تنوع ساده می‌باشد. در طبیعت تنوع ژنتیکی باعث می‌شود که تمامی گونه‌ها توسط یک بیماری یا ویروس به خطر نیفتند. همین ایده دربارهٔ نرم‌افزار صادق است؛ بدین صورت که یک ویروس یا حمله قادر نباشد تمامی نسخه‌های برنامه را تحت تأثیر قرار دهد. تنوع نرم‌افزار یک سازوکار حفاظتی در برابر ویروس‌های رایانه‌ای و حملات نرم‌افزاری است. معماری ابزارهای تبدیل کد خودکار را که درواقع منجر به مبهم‌سازی نرم‌افزار می‌شود، کمی می‌توان تغییر داد تا تنوع نرم‌افزار حاصل شود. در اینجا لازم نیست یک نمونهٔ جدید از نرم‌افزار ایجاد شود که از لحاظ عمل‌کرد معادل با برنامه اصلی باشد (و مهندسی معکوس آن سخت باشد). در اینجا سختی مهندسی معکوس یا دست‌کاری یک نمونه، برنامه یک فاکتور امنیتی است [۱۸].

از مهم‌ترین تکنیک‌هایی که تاکنون از تنوع نرم‌افزار استفاده کرده‌اند، به موارد زیر می‌توان اشاره کرد: Cohen [۲۵] یکی از نخستین مطرح‌کنندگان تنوع کد بود. او این روش را "تکامل نرم‌افزار"[5] نامید و در زمینه محافظت سیستم عامل در برابر حملات خودکارشده مطرح کرد. کولبرگ و همکاران[۲۶] از مدل‌هایی تاریخی و مبتنی‌بر زیست الهام گرفته و آنها را در اصولی دسته‌بندی کرده و نشان دادند چگونه این روش‌ها به یک جهان دیجیتال نگاشت پیدا می‌کنند. در سال ۱۹۹۷ Forrest و همکارانش [۲۷] با انگیزه اینکه چرا تنوع در سامانه‌های رایانه‌ای برای امنیت ارزشمند است، تحقیق کردند. نویسندگان پیشنهادشان را با تصادفی‌کردن مقدار حافظه اختصاص‌داده‌شده به یک قاب استک جهت مختل‌کردن حملات سرریز بافر توجیه کردند. Cox و همکارانش [۲۸] یک سامانه N نسخه‌ای پیشنهاد دادند که از سرور در برابر اختلاف میان چندین نمونه از یک فرایند که روی آن اجرا می‌شد، محافظت می‌کرد. یک سال بعد، Anckaert و همکارانش [۲۹] یک ماشین مجازی خودبررسی پیشنهاد دادند که در مقابل دست‌کاری، محافظت انجام می‌داد. در مدل آنها یک ماشین مجازی سفارشی در نظر گرفته شده بود که بایت کد سفارشی را می‌خواند.

۵-۳- نهان‌نگاری نرم‌افزار و اثر انگشت[6]

این تکنیک روشی دیگر برای مقابله با دزدی نرم‌افزار است. در طول سال‌ها، مشخص شده که افزونگی در برنامه‌های

[1] Man in the middle
[2] Proxy attacks
[3] Timed executable agent systems
[4] Software Diversity
[5] Program evolution
[6] Software Watermarking and fingerprinting

اجرایی می‌تواند برای جاسازی پیام‌های پنهان مورد استفاده قرار گیرد. علاوه‌براین، تفاوت میان برنامه‌ها با کارکردهای مشابه می‌تواند به‌عنوان یک آب‌نشان[1] عمل کند. این تفاوت شامل داده ایستا، داده پویا (مقادیر)، الگوریتم‌ها، ساختارهای داده زمان اجرا است. یک آب‌نشان نرم‌افزاری یک مقدار یا ویژگی است که در داخل یک نرم‌افزار تعبیه شده است که مالکیت نرم‌افزار را اثبات می‌کند. هرجایی که یک نمونه نرم‌افزار یک مقدار تعبیه‌شده داشته باشد، که کاربر مشروع نرم‌افزار را شناسایی می‌کند، این مقدار را به‌عنوان یک اثر انگشت معنا می‌کنیم.

کولبرگ و همکاران [۲] اهمیت آب‌نشان نرم‌افزاری را به‌عنوان مکملی برای اثبات دست‌کاری و مبهم‌سازی مطرح کردند. در دهۀ اخیر، تولیدات زیادی برای آب‌نشان‌های سخت افزاری انجام شده است: آب‌نشان مبتنی بر گراف[2]، آب‌نشان مبتنی بر مسیر[3] و آب‌نشان براساس کدگذاری ثابت[4]. با این حال، تعبیه یک آب‌نشان قوی برای نرم‌افزار ساده نیست. نوعاً، همان تبدیل‌هایی که آب‌نشان را تعبیه می‌کند، می‌تواند برای تحریف یا حذف آن مورد استفاده قرار گیرد.

۵-۴- سرریز بافر و استخراج[5]

ویروس‌ها و کرم‌های اینترنتی اغلب از طریق سیستم عامل و برنامه‌های کاربردی به سامانه‌های نرم‌افزاری رخنه می‌کنند. با توجه به یکسان‌بودن سیستم عامل اکثر سامانه‌های رایانه‌ای آلوده‌ساختن سامانه‌ها با استفاده از ویروس، به‌راحتی امکان‌پذیر خواهد بود. به‌عبارت دقیق‌تر جامعۀ نرم‌افزاری در حال تکامل است و اغلب افراد از سیستم‌های عامل مشابه و بسته‌های نرم‌افزاری مشابه استفاده می‌کنند که دارای خطاهای مشابهی هستند. این یکی از دلایلی است که ویروس‌ها که از یک یا چند حفره امنیتی بهره‌برداری می‌کنند و موفق هستند. سرریز بافرها [۳۰] یکی از شایع‌ترین حفره‌هاست. درحالی‌که راه‌هایی برای حل این نوع خطا مانند وارسی‌کردن محدوده[6]، زبان‌های نوع امن[7] وجود دارد، هنوز هم سرریز بافر وجود دارد و مهاجمان راه‌های زیرکانۀ زیادی برای بهره برداری از آنها پیدا می‌کنند [۳۱].

[1] Watermark
[2] Graph-based watermark
[3] Path-based watermark
[4] watermarks based on constant encoding
[5] Buffer Overflows and Exploits
[6] bound checking
[7] type safe languages

با این حال تکنیک‌های مقابله با دست‌کاری و تنوع نرم‌افزار می‌توانند از سرریز بافر جلوگیری و آنها را کشف یا حملات خودکارسازی آنها را کند کنند. Forrest و همکارانش [۲۷] مقایسه‌ای بین تنوع در سامانه‌های رایانه‌ای و تنوع در سامانه‌های زیست محیطی را انجام داده‌اند. مقالۀ آنها برخی از نتایج مقدماتی در تصادفی‌سازی طرح‌بندی پشته با افزایش شکاف‌های خاص در زمان‌های تصادفی هشت بایتی را نشان داده است. چنین تغییرات ساده‌ای می‌تواند یک نمونه برنامه را در برابر حملات سرریز بافر مقاوم کنند. تکنیک دیگر، برای نبرد با حملات سرریز بافر مبهم‌سازی نشانی[8] نام دارد که براساس ایده تنوع کد [۳۲] استوار است. این تکنیک کد و بخشهای داده را در پشته با تصادفی‌سازی تمام نشانی‌های مبنا و شروع، مکان‌های روتین‌ها و داده‌های ایستا و معرفی فاصله بین اشیا به‌صورت تصادفی درمی‌آورد؛ درحالی‌که تصادفی‌سازی طرح‌بندی فضای نشانی (ASLR) امروزه بیش‌تر توسط سیستم‌های عامل اتخاذ می‌شود، برخی مهاجمان پیشنهادهایی [۳۳] داده‌اند. با این حال، برخی کارها نشان داده است که ASLR مقاومت خوبی در برابر انجام حملاتی که سرریز بافر عادی انجام می‌دهند، ایجاد کرده است.

۵-۵- پیری نرم‌افزار[9]

تکنیک پیری نرم‌افزار [۳۴] تکنیکی است که استفاده از نسخۀ قدیمی را که ممکن است به خطر افتاده باشد، توصیه نمی‌کند. در این مورد، مقدار یک نسخۀ قدیمی در مقایسه با نسخۀ جدید کاهش می‌یابد. به هر حال نسخه‌های جدیدتر با نسخه‌های قدیم‌تر سازگار هستند. به‌عنوان مثال نسخه‌های جدیدتر می‌توانند ورودی نسخه‌های قدیمی را بخوانند. در نگاه نخست ممکن است به نظر برسد که این تکنیک ربطی به امنیت نرم‌افزار ندارد؛ ولی این روش بر روی فرضیات مشخصی متکی است. به‌عبارت دقیق‌تر، در این روش مهاجم نمی‌تواند حمله به نسخه جدید نرم‌افزار را مشابه نسخه قدیم اعمال کند و لذا با هر به‌روزآوری نسخه برنامه مهاجم باید تدبیر جدیدی برای حمله به نرم‌افزار بیندیشد [۲۰].

در این تکنیک، به‌روزآوری‌های دوره‌ای نرم‌افزار که سازگار با نسخه‌های قبلی نرم‌افزار هستند به مشتری فرستاده می‌شوند. به‌روزآوری‌ها اشکالات را برطرف کرده

[8] address obfuscation
[9] Software Aging

و ویژگی‌های جدیدی را عرضه می‌کنند، همچنین برنامه را با سایر برنامه‌های وابسته به نرم‌افزار همگام می‌کند. به‌هرحال، این راه‌کار برای برنامه‌های اسنادی مفید است (مانند نرم‌افزار واژه‌پرداز شرکت مایکروسافت که متکی بر داده‌های با فرمت مشخص است). پیری نرم‌افزار، قالب یا فرمت نسخهٔ بعدی را تغییر می‌دهد. تکنیک‌های رمزنگاری به این تکنیک کمک می‌کند تا به اطمینان برسیم که نسخه‌های بعدی نرم‌افزار می‌توانند اسناد مربوط به نسخه‌های قدیمی را بخوانند و یا استفاده کنند (ولی معکوس این رابطه صادق نیست) [۳۴].

۶- مقایسه و دسته‌بندی روش‌های مقابله

در (جدول ۱) دسته‌بندی کاملی از تمام تکنیک‌های نرم‌افزاری برای محافظت از نرم‌افزار در برابر حملات تحلیل، دست‌کاری و دزدی نرم‌افزار نمایش داده شده است. لازم به ذکر است که در این جدول تکنیک‌هایی که علاوه‌بر مقابله با تحلیل نرم‌افزار در دسته‌ای دیگری هم کاربرد دارند، در ستون "ویژگی امنیتی ثانویه" مشخص شده‌اند. به‌عنوان مثال تکنیک رمزنگاری جعبهٔ سفید هم می‌تواند از نرم‌افزار در برابر حملات تحلیل محافظت و هم از دست‌کاری نرم‌افزار جلوگیری کند. ارائه‌کنندگان هر کدام از تکنیک‌ها در ستون مربوطه مشخص شده است که البته برای اختصار تمامی کارهای مرتبط با هر تکنیک ذکر نشده است. در برخی تکنیک‌ها نفراتی که بعدها آن تکنیک را گسترش داده‌اند، معرفی شده است. به‌عنوان مثال در تکنیک مبهم‌سازی کدوانگ و همکاران تبدیلات مبهم‌سازی جدیدی را معرفی کرده‌اند. در ستون "محافظت ایستا" مشخص شده است آیا آن تکنیک در برابر حملات ایستا (زمان کامپایل) مقاوم است یا خیر و ستون "محافظت پویا" توانایی محافظت از نرم‌افزار در برابر حملات پویا (زمان اجرا) را مشخص کرده است.

۷- نتیجه‌گیری

در این مقاله ابتدا مخاطرات و تهدیدهای نرم‌افزاری مورد بررسی و طبقه‌بندی قرار گرفت. در ادامه نشان داده شد که یکی از مهم‌ترین دلایل نگرانی در مورد ایمنی نرم‌افزارها، توزیع‌شدن آن در ماشین‌های میزبان است. به‌طوراصولی تحلیل نرم‌افزار عمل مهندسی معکوس و فهم نحوه عملکرد برنامه است که در چنین شرایطی برای مهاجمان فرصت بیشتری برای حملات فراهم می‌آورد. در ادامه تکنیک‌های نرم‌افزاری برای مقابله با این تهدیدها معرفی شد و مزایا و معایب هر کدام مورد بررسی قرار گرفت؛ سپس نشان داده شد که هر کدام از تکنیک‌ها توانسته‌اند تا حدودی با حملات مقابله کنند؛ اما با پیشرفت دانش مهاجمان و ابداع روش‌های حملهٔ جدید، تکنیک‌های قدیمی‌تر با شکست مواجه شده و تکنیک‌های جدید معرفی شده‌اند. این امر باعث به‌وجودآمدن رقابت بین مهاجمان و توسعه‌دهندگان نرم‌افزار شده است. هدف این تکنیک‌ها درواقع شناسایی، جلوگیری و یا به تأخیرانداختن انجام حمله بر روی نرم‌افزارها است. اگرچه هیچ تضمینی وجود ندارد که نرم‌افزار مورد نظر به‌طورکامل در مقابل حملات تحلیل (مهندسی معکوس)، دست‌کاری و دزدی ایمن باشد؛ اما هدف این تکنیک‌ها سخت‌ترکردن فرآیند حمله به نرم‌افزار و صرف زمان و منابع بیشتر برای فرد مهاجم است. بدیهی است پژوهش‌های زیادی درخصوص ایمن‌کردن نرم‌افزارها در حال انجام بوده و این موضوع پژوهش در دانشگاه‌ها و در قالب واحدهای پژوهش و توسعه شرکت‌های نرم‌افزاری مورد توجه بیشتری قرار گیرد.

(جدول ۱) دسته بندی و مقایسه تکنیک‌های محافظت از نرم‌افزار

دسته	تکنیک	ارائه دهنده(گان)	کارایی همچنین در	محافظت ایستا	محافظت پویا
مقابله با تحلیل	مبهم سازی کد	Collberg و همکاران Wang و همکاران Davidson و همکاران	مقابله با دستکاری	بلی	بلی
	رمزنگاری جعبه سفید	Bringer و همکاران Joye Tehan Wyseur	مقابله با دستکاری	بلی	بلی
	رمزگذاری کد	Sander و همکاران	مقابله با دستکاری	بلی	بلی
	تکنیک‌های خود ویرایش	Kanzaki Madou Cappaert و همکاران	-	بلی	بلی
	محافظت مبتنی بر شبیه سازی	Madou و همکاران	-	بلی	بلی
مقابله با دستکاری	امضای کد	Fleischman	-	بلی	خیر
	نرم‌افزار Aucsmith	Aucsmith	مقابله با تحلیل	بلی	بلی
	حفاظ‌های نرم‌افزاری	Chang و همکاران	-	بلی	بلی

	درهم سازی فراموشکار	Chen و همکاران	-	بلی	بلی
	مجازی سازی	Ghosh و همکاران	مقابله با تحلیل	بلی	بلی
	نظارت پویا	Blietz	-	بلی	بلی
	درهم سازی چند بلاکی	Wang و همکاران	-	بلی	بلی
مقابله با دزدی نرم‌افزار	تصدیق از راه دور	Seshadri و همکاران Galay و همکاران	-	بلی	خیر
	تنوع نرم‌افزار	Cohen Collberg و همکاران Forrest و همکاران Cox و همکاران Anckaert و همکاران	-	بلی	خیر
	نهان نگاری و اثر انگشت	Collberg و همکاران	-	بلی	خیر
	سر ریز بافر و استخراج	Forrest و همکاران	-	بلی	خیر
	پیری نرم‌افزار		-	بلی	خیر

۸- مراجع

[1] P. Falcarin, C. Collberg, M. Atallah and M. Jakubowski, "Software Protection," IEEE COMPUTER SOCIETY, 2011.

[2] J. Cappaert, "Code Obfuscation Techniques for Software Protection," Katholieke Universiteit Leuven, 2012.

[3] C. S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation Tools for Software Protection," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 28, no. 8, 2002.

[4] A. Balakrishnan, C. Schulze, "Code Obfuscation Literature Survey," *CS701 Construction of Compilers,* December 2005.

[5] C. Collberg, C. Chomborson and D. Low, "A taxonomy of obfuscation transformation," Department of Computer Science, The University of Auckland, 1997.

[6] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," 2000.

[7] T. Sander and C. F. Tschudin, "On Software Protection via Function Hiding," in *the Second Workshop on Information Hiding*, 1998.

[8] N. Mavrogiannopoulos, N. Kisserli, B. Preneel, "A taxonomy of self-modifying code for obfuscation," *computers & security,* 2011.

[9] Y. Kanzaki, A. Monden, M. Nakamura, K. Matsumoto, "Exploiting self-modification mechanism for program protection," in *IEEE computer software and applications conference*, 2003.

[10] M. Madou, B. Anckaert, P. Moseley, S. Debray, BD. Sutter, "Software protection through dynamic code mutation," in *6th International workshop on information security applications*, 2005.

[11] J. Cappaert, N. Kisserli, D. Schellekens, B. Preneel, "Self-encrypting code to protect against analysis and tampering," in *1st Benelux workshop on information and system security*, 2006.

[12] E. Fleischman, "Code Signing," *Internet Protocol Journal,* vol. 5, no. 1, March 2002.

[13] H. Chang and M. J. Atallah, "Protecting Software Code by Guards," in *Digital Rights Management Workshop*, 2001.

[14] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha and Mariusz H. Jakubowski, "Oblivious Hashing : A Stealthy Software Integrity Verification Primitive," *Springer,* pp. 400-414, 2003.

[15] S. Ghosh, J. D. Hiser, and J. W. Davidson, "A secure and robust approach to software tamper resistance," in *12th international conference on Information hiding*, 2010.

[16] B. Blietz, "Software Tamper Resistance Through Dynamic Monitoring," Iowa State University, 2004.

[17] P. WANG, "Tamper resistance for software protection," School of Engineering Information and Communications University, Daejeon, 2005.

[18] J. M. Memon, A. Khan, A. Baig, A. Shah, "A Study of Software Protection Techniques," *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering,* p. 249–253, 2007.

[19] F. B. a. Cohen, "Operating system protection through program evolution," *Computers & Security,* p. 565–584, 1993.

[20] C. Collberg, J. Nagra, and F.-Y. Wang, "Surreptitious software: Modelsfrom biology and history," *Computer Network Security,* pp. 1-21, 2007.

[21] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *the Sixth Workshop on Hot Topics in Operating Systems*, 1997.

[22] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006.

[23] B. Anckaert, M. H. Jakubowski, and R. Venkatesan, "Proteus: virtualization for diversified tamper-resistance," in *Digital Rights Management Workshop*, 2006.

[24] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade,"

2000.

[25] S. Checkoway, L. Davi, A. Dmitrienko, A. R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security*, 2010.

[26] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *12th USENIX Security Symposium*, August 2003.

[27] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *ACM Conference on Computer and Communications Security*, 2004.

[28] M. Jakobsson and M. K. Reiter, "Discouraging software piracy using software aging," in *Digital Rights Management Workshop*, 2001.

[29] D. Aucsmith, "Tamper resistant software: An implementation," *Information Hiding,* vol. 1174, p. 317–333, 1996.

[30] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla, "Swatt: Software-based attestation for embedded devices," *IEEE Symposium on Security and Privacy,* p. 272, 2004.

[31] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. K. Khosla, "Externally verifiable code execution," in *ACM*, 2006.

[32] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *SOSP*, 2005.

[33] J. A. Garay and L. Huelsbergen, "Software integrity protection using timed executable agents," in *ACM Symposium on Information, Computer and Communications Security*, Taipei, Taiwan, 2006.

[34] W. Kimball, "Emulation-based Software Protection," in *Black Hat DC*, 2009.

افشین رشیدی متولد ۱۳۶۸، مدرک کارشناسی خود را در شهریور ماه ۱۳۹۰ در رشته مهندسی تکنولوژی نرم‌افزار در دانشگاه بناب و درجه کارشناسی ارشد را در شهریور ماه ۱۳۹۴ در دانشگاه گیلان و در رشته مهندسی کامپیوتر گرایش نرم‌افزار دریافت کرده است. زمینهٔ پژوهشی مورد علاقه ایشان امنیت نرم‌افزار، مبهم‌سازی کد و معماری و طراحی نرم‌افزار است. در زمینه حفاظت از کد نرم افزار تاکنون سه مقاله از ایشان در کنفرانس‌های ملی و بین المللی ارائه شده است.

رضا ابراهیمی آتانی استادیار گروه مهندسی رایانهٔ دانشکده فنی و مهندسی دانشگاه گیلان است. نامبرده دکترای خود را در سال ۱۳۸۹ در رشتهٔ مهندسی الکترونیک از دانشگاه علم و صنعت ایران دریافت کرد. ایشان عضو پیوستهٔ انجمن رمز ایران و انجمن‌های بین المللی IACR و IEEE هستند. از ایشان تاکنون دو عنوان کتاب و بیش از یکصد مقاله در مجلات و کنفرانس‌های ملی و بین‌المللی به چاپ رسیده است. زمینهٔ پژوهشی مورد علاقهٔ وی طراحی و پیاده‌سازی الگوریتم‌های رمزنگاری، امنیت شبکه و امنیت نرم‌افزار است.

حمید نصیری متولد ۱۳۷۰ هجری شمسی است. در مرداد ماه ۱۳۹۱ موفق به اخذ مدرک کارشناسی‌در رشتهٔ مهندسی تکنولوژی نرم‌افزار از دانشگاه اردبیل شد. ایشان دانشگاه گیلان را برای ادامه تحصیل در مقطع کارشناسی ارشد و در رشته مهندسی نرم افزار- کامپیوتر برگزید و در شهریورماه سال ۱۳۹۴ از این دانشگاه فارغ التحصیل شد. ایشان هم اکنون در مؤسسات غیرانتفاعی و دانشگاه سمای اردبیل مشغول تدریس هستند. از زمینه‌های پژوهشی مورد علاقهٔ ایشان می‌توان به محافظت از نرم‌افزار، بهینه‌سازی کد در کامپایلر و الگوهای برنامه‌نویسی اشاره کرد. در زمینه محافظت از کد نرم افزار سه مقاله از ایشان در کنفرانس ملی کامپیوتر و کنفرانس بین‌المللی رمز ایران ارائه شده است.