فصلنامه علمی-ترویجی پدافند غیرعامل
سال هشتم، شماره ۳، پاییز ۱۳۹۶، (پیاپی ۳۱): صص ۳۴-۲۳

استفاده از روش برنامه‌نویسی پدافندی جهت افزایش امنیت نرم‌افزار در زبان C#

سامان کشوری[۱]، محمدرضا حسنی آهنگر[۲*]، ساناز کشوری[۳]

تاریخ دریافت: ۱۳۹۵/۰۴/۲۸
تاریخ پذیرش: ۱۳۹۵/۱۱/۲۴

چکیده

با توجه به اهمیت امنیت نرم‌افزارهای رایانه‌ای در سازمان‌های نظامی، برنامه‌نویسان و طراحان این سیستم‌ها ملزم به رعایت مسائل امنیتی در برنامه‌نویسی هستند. این مقاله بر روی مرحله کدنویسی در چرخه تولید نرم‌افزار تمرکز دارد. هدف کدنویسی پدافندی، جلوگیری از بروز حوادث ناشی از کدنویسی نامناسب است. این حوادث شامل ازکارافتادن برنامه، دسترسی به کد منبع، استفاده بیش‌ازحد از منابع سخت‌افزاری و افزایش زمان نیاز جهت به‌روزرسانی برنامه هستند. در ادامه استفاده از آرگومان‌های نامی در فراخوانی تابع معرفی گردید که این عوامل منجر به افزایش خوانایی و درنتیجه کاهش زمان به‌روزرسانی برنامه می‌شود. جهت مدیریت خطاهای حین اجرا، روش شکست سریع استثناءها و نحوه مدیریت مناسب آن‌ها ارائه شده است. این اقدامات موجب افزایش امنیت برنامه و عدم ازکارافتادن برنامه به‌دلیل کدنویسی نامناسب گردیده و از طرفی موجب افزایش تعداد توابع، کلاس‌ها و درنتیجه افزایش تعداد خط کد می‌شود. این مسئله ممکن است موجب نگرانی درباره افزایش زمان اجرای برنامه و هزینه پردازشی شود. این مقاله به پیاده‌سازی یک برنامه در دو حالت کدنویسی پدافندی و غیرپدافندی پرداخته است. نتایج نشان می‌دهد که میانگین عمر نخ‌ها در حالت پدافندی ۸/۲۳ ثانیه است، درحالی‌که این مقدار برای حالت غیرپدافندی، ۸/۸۰ ثانیه بود. از طرفی، میزان درصد استفاده از پردازنده در حالت‌های پدافندی و غیرپدافندی به ترتیب ۵۴/۳۲٪ و ۵۴/۷۰٪، از طرف دیگر برای هزینه پردازشی کاربر نیز به‌ترتیب ۲۴/۲۳٪ و ۲۴/۴۱٪ بود. این اعداد حاکی از هزینه پردازشی و زمان اجرای یکسان برنامه در دو حالت پدافندی و غیرپدافندی است.

کلیدواژه‌ها: امنیت نرم‌افزار، برنامه‌نویسی پدافندی، کدنویسی امن، برنامه‌نویسی #C.

۱- دانشجوی کارشناسی ارشد نرم‌افزار، دانشگاه جامع امام حسین(ع)
۲- دانشیار دانشگاه جامع امام حسین(ع)، (mrhasani@ihu.ac.ir)- نویسنده مسئول
۳- دانشجوی کارشناسی ارشد نرم‌افزار، دانشگاه رازی کرمانشاه

۱- مقدمه

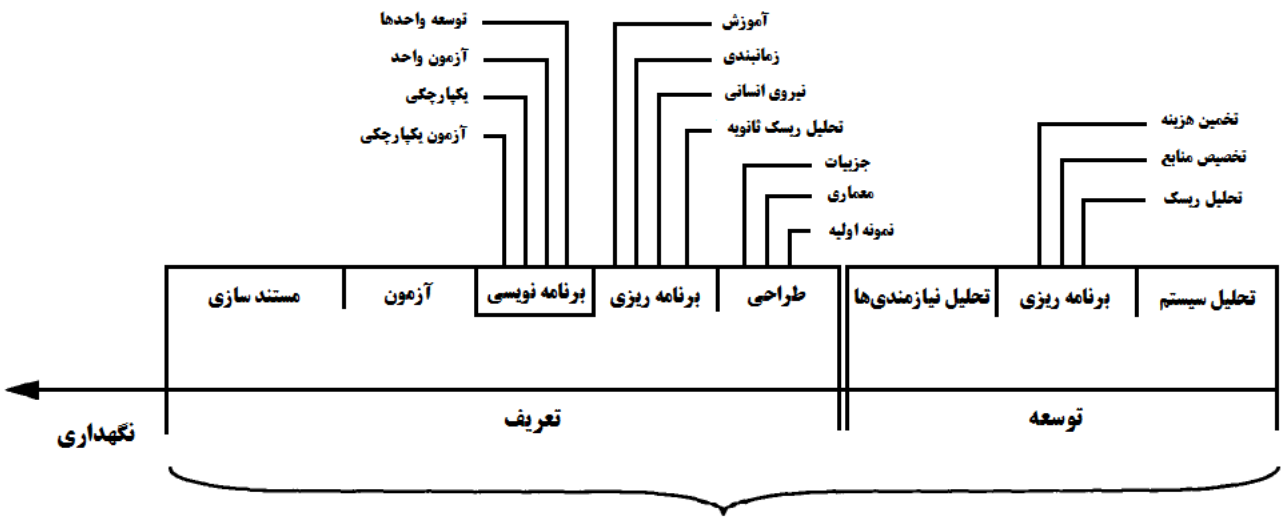
روش برنامه‌نویسی پدافندی یک قالب دفاعی جهت اطمینان از عملکرد مقاوم نرم‌افزار در مقابل اتفاقات غیرقابل پیش‌بینی است[۱]. استفاده از روش برنامه‌نویسی پدافندی این تضمین را می‌دهد که برنامه در برابر سوءاستفاده دیگران مقاوم است. برنامه‌نویسی پدافندی یک رویکرد جهت بهبود نرم‌افزار و کد منبع، ازنظر موارد زیر است [۲]:

- کیفیت عمومی[1]، شامل کاهش تعداد اشکالات نرم‌افزار شده که گاهی اوقات توسط ابزارهای بررسی کد به‌صورت خودکار بررسی می‌شود.
- کد منبع قابل‌فهم، کد مناسب به‌گونه‌ای نوشته می‌شود که با مراجعه توسط توسعه‌دهنده دیگر قابل فهم است. این کار موجب افزایش سرعت به‌روزرسانی کدها می‌شود. این مبحث تحت عنوان کدنویسی شفاف بیان شده که در این مقاله به آن خواهیم پرداخت.
- کدهای برنامه باید با پیش‌بینی ورودی‌های غیرمنتظره و درنظرگرفتن رفتار کاربر، اقدامات لازم در برابر حملات احتمالی را انجام دهد. در این مقاله راه‌حل‌های آن را مورد بررسی قرار می‌دهیم.

گاهی از برنامه‌نویسی پدافندی، با عنوان برنامه‌نویسی امن[2] یاد می‌شود. در این نوع توسعه نرم‌افزار، برنامه به‌صورت بالقوه در برابر حملاتی همچون تزریق کد، انکار سرویس یا دیگر حملات است، دچار اشکال نشود[۳]. چرخه تولید نرم‌افزار[3] شامل مراحل مختلفی است که در شکل(۱) مشاهده می‌شود[۴]. جهت تولید نرم‌افزار امن در این مقاله، بر روی فاز برنامه‌نویسی و دوشاخه توسعه واحدها و آزمون واحد تمرکز شده است. نتایج این مقاله فاز نگهداری نرم‌افزار را نیز تحت تأثیر قرار می‌دهد. کارهای مرتبط با این مقاله، شامل تحقیقاتی است که در آن‌ها راه‌کارهایی برای افزایش امنیت در چرخه تولید نرم‌افزار ارائه کرده‌اند، پژوهش‌هایی که تأثیرات برنامه‌نویسی پدافندی را شامل شده‌اند نیز به‌عنوان کارهای مرتبط با این مقاله محسوب می‌شوند.

جهت افزایش امنیت درطول چرخه نرم‌افزار در یکی از تحقیقات انجام‌شده به چالش‌های موجود در سیستم‌های جاسازی‌شده[4] پرداخته که برای تأمین امنیت در چرخه تولید نرم‌افزار و مقابله با خطر سرریز بافر در فاز تجزیه و تحلیل راه‌کاری ارائه شده است[۵]. در گزارشی که در آغاز همایش ملی امنیت سایبری منتشرشده، مشکلات کلیدی و توصیه‌های مربوط به آن‌ها برای مراحل چرخه توسعه نرم‌افزار ارائه شده است[۶]. در مقاله دیگری مدل بهبود یافته‌ای از چرخه تولید نرم‌افزار ارائه شده است. در این مقاله، مدل M-SDLC[5] ارائه شده که منجر به بهبود نگهداری نرم‌افزار می‌شود [۷]. در کار پاریس و همکارانش [۸]، عوامل مهم در کیفیت نرم‌افزار بیان شده است. در این مقاله عواملی که در آینده در مهندسی نرم‌افزار تأثیرگذار خواهند بود نیز آمده است. در مقاله دیگری ده روش جهت کاهش عیب‌های کدنویسی مطرح شده است [۹]. طبق بررسی دیگری که به تأمین امنیت در چرخه تولید برنامه‌های تحت وب کمک می‌کند؛ رویکرد تحلیل‌های ایستا جهت کشف تزریق کدهای SQL در این برنامه‌ها بررسی شده است [۱۰]. جهت مدیریت خطا و عدم تأثیر بروز آن‌ها در سایر اجزای سیستم در مقاله چن و همکاران [۱۱] مدلی احتمالی ارائه شده است. این مقاله حالات اجرایی مختلف یک برنامه را در یک محیط رقابتی[6] قرار داده و آرگومان‌های مختلف را محاسبه می‌کند. اگر درباره صحت آن‌ها دلایل قوی وجود نداشت، حالت‌های خطا را کشف می‌کند. در این مقاله با معیار زیمنس[7] نیز آزمایش‌های فشرده‌ای انجام شده و نتیجه آن‌ها اثبات شده است؛ که نه تنها رویکرد این مقاله قالب روشنی برای کشف علت بروز شکاف[8] در حالات مختلف است، بلکه به‌صورت آماری روش آن‌ها به‌لحاظ محلی‌سازی خطاهای رخداده بهتر از روش‌های Tarantula [۱۲]، SOBER [۱۳]،CT [۱۴] و PPDG [۱۵]، بهتر عمل می‌کند. در مقاله جرجم و دهقانی [۱۶]، ضمن مرور انواع آسیب‌پذیری‌های شناخته‌شده ناشی از کدنویسی، بر اهمیت تولید نرم‌افزارهای امن تأکید شده و رهنمون‌هایی برای کدنویسی امن پیشنهاد شده است. با ایجاد آسیب‌پذیری‌های ساختگی، شاخص‌های امنیتی SDLC در فاز کدنویسی محاسبه شده است. تحقیقی دیگر اهمیت کدنویسی استاندارد، بررسی کدها، آزمون واحد و مدیریت خطاها را در فاز کدنویسی SDLC جهت ایجاد برنامه امن برشمرده است [۱۷]. در مقاله آلکس و همکارانش بر اهمیت روش برنامه‌نویسی پدافندی در ظرفیت بالای برنامه در تحمل خطا تأکید شده است [۳].

1- General quality
2- Secure Programming
3- SDLC (Software Development Life Cycle)
4- Embedded Systems
5- Maintainable-Software Development Life Cycle
6- Dispatch
7- Simense benchmark
8- Bug



شکل (۱): چرخه تولید نرم‌افزار [۴]

در مقاله خلیلی و همکارانش [۲۱]، اولین تلاش برای استقرار اصول برنامه‌نویسی امن براساس اولویت‌های سیستم کنترل صنعتی است. نتایج تجربی ارائه‌شده در این مقاله، نشان‌دهنده آسیب‌پذیری مدیریت حافظه در سیستم‌های کنترل صنعتی است لذا این مسئله اهمیت مدیریت حافظه را در پی داشته و این مورد در اولویت قرار گیرد. نتایج استفاده از برنامه‌نویسی امن در این مقاله شامل حل آسیب‌پذیری‌های مدیریت حافظه و سرریزها در کنار کاهش استفاده از منابع برمبنای تفاوت زمان بارگذاری و استفاده از حافظه اصلی و پردازنده با آزمون‌های آماری اثبات شده است. علاوه‌بر این، نتایج حاصل از آزمون نفوذ در این تحقیق حاکی از بهبود هم‌زمان تمامیت و دسترسی منابع و هم‌چنین محرمانگی برنامه ایجادشده با کدنویسی امن در کاربردهای صنعتی دارد. در مقاله دیگری روش استفاده از برنامه‌نویسی پدافندی در زبان جاوا آمده است [۱۹]. در تحقیق دیگری تأثیرات انجام آزمون‌های مختلف در بهبود امنیت و کیفیت نرم‌افزار آمده است [۲۰]. پژوهش دیگری بر اهمیت یکپارچه‌سازی آموزش برنامه‌نویسی امن در دروس مقدماتی رایانه تأکید می‌کند. نتایج این تحقیق که شامل آموزش ۴۰ ماژول تزریق امنیتی در قالب کارهای آزمایشگاهی بر روی ۱۱۳۵ دانشجو که در مؤسسات مختلف مشغول به آموزش درس برنامه‌نویسی بودند انجام شده، نشان می‌دهد که افراد به درک، اهمیت و کاربرد برنامه‌نویسی امن در دوره‌های مقدماتی پی برده‌اند [۲۲].

گاهی در طول حیات چرخه نرم‌افزار در برنامه اشکالاتی رخ می‌دهد که نیاز به تغییرات در آن وجود دارد. در این صورت برنامه‌نویس باید در اسرع وقت اشکالات یا تغییرات را در برنامه اعمال نماید. در زمان مراجعه به کدهای برنامه گاهی به‌دلیل نداشتن حضور ذهن درباره نحوه نوشتن کدها و عملکرد هر بخش از کد دچار سردرگمی شده یا شخصی که کدها را نوشته اکنون در دسترس نیست. گاهی برنامه‌نویس دیگری موظف به اعمال تغییرات است. درصورت وقوع یکی از این حالات، زمان اعمال تغییرات بیش از زمان مورد نیاز برای انجام آن‌ها شده و حتی گاهی اعمال تغییرات با شکست مواجه می‌شود. جهت کاهش این زمان در این مقاله با معرفی امکانات زبان برنامه‌نویسی #C، محیط برنامه‌نویسی ویژوال استودیو و نحوه نوشتن کدها توسط برنامه‌نویس زمان اعمال تغییرات و رفع اشکالات برنامه در راستای افزایش امنیت، کاهش می‌دهیم. در این مقاله ابتدا نحوه نوشتن کد و تابع شفاف بیان‌شده است و در ادامه، روش استفاده از آرگومان‌های نامی در فراخوانی توابع که در #C تعبیه شده و کمک شایانی در خوانایی کد کرده، بیان می‌شود. نحوه مدیریت‌کردن استثناءها در کنترل ورودی توابع که روش شکست سریع در آن نقش به‌سزایی دارد شرح داده شده است. در پایان با پیاده‌سازی یک برنامه در دو حالت پدافندی و غیرپدافندی اثبات می‌کنیم برنامه پدافندی با این‌که تعداد تابع و کلاس و درنتیجه تعداد خط کد بیشتری نسبت به حالت عادی دارد، زمان و میزان هزینه مشابهی با آن دارد. این درحالی است که برنامه‌نویسی پدافندی دارای مزایای متعددی نسبت به حالت عادی کدنویسی (که در آن مسائل پدافندی رعایت نمی‌شود) است.

```
1 reference
133     private void linkLabel2_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
134     {
135          OpenFileDialog openFileDialog1 = new OpenFileDialog();
136         if (openFileDialog1.ShowDialog() == DialogResult.OK)
137         {
138             attachfilepatch = openFileDialog1.FileName;
139             FileInfo file = new FileInfo(attachfilepatch);
140             linkLabel2.Text = file.Name + " ,    " + (file.Length / 1024).ToString("N0") + " KB";
141             btndelete.Show();
142         }
143     }
144
```

شکل (۲): شبه کد نوشته‌شده با زبان C# در ویرایش‌گر ویژوال استودیو مایکروسافت

۲- کدنویسی شفاف[1]

برای کدنویسی شفاف باید ویژگی‌های زیر را مدنظر داشت[۲۲]:

- به‌دست‌آوردن منطق مناسب قبل از کدنویسی، با این کار قبل از این‌که به آزمون و خطاکردن انتخاب‌های مختلف پرداخته شود، چند نمودار جریانی[2] یا شبه‌کد[3] که از دانش و تجربه‌های قبلی شخص به‌دست‌آمده، تهیه می‌شود. نوشتن آن‌ها، موجب کاهش شک‌ها یا ناامنی‌های ناشی از پیچیدگی عملیاتی کار شده که صرفه‌جویی در زمان را در پی خواهد داشت. این اقدامات موجب درک سریع‌تر کد و جلوگیری از آشفتگی کدها می‌شود. نمایش واضحی از ساختار صفحه، استفاده از برنامه‌هایی برای کدنویسی که به‌ظاهر کدنویسی کمک می‌کنند، مفید است. به‌عنوان‌مثال، در شکل (۲) تکه کدی نشان داده شده که به کمک ویرایشگر کد ویژوال استودیوی شرکت مایکروسافت نوشته شده است. در این تکه کد، هر خط با یک شماره نشان داده شده که برای بررسی کدها در کار گروهی مؤثر است. علاوه بر آن، محتوای تعیین‌شده به‌طور مشخص نام‌گذاری شده که به‌روزرسانی کد را ساده‌تر می‌کند.
- فاصله‌گذاری مناسب جهت رعایت ساختار مناسب با مجزاکردن شروع و پایان هر قسمت دلیلی بر افزایش خوانایی برنامه است که در شکل (۲) نمونه آن مشاهده می‌شود.
- نوشتن توضیح‌ها[4]، با توجه به این‌که توضیحات در روبروی کد قرار دارند می‌توانند با اشاره مستقیم به اصل مطلب و دلیل نوشتن آن قطعه کد بپردازند. این اقدام منجر به رفع ابهامات شده و قابلیت استفاده مجدد آن کد را در پی دارد. باید توجه داشت که هدف از نوشتن توضیحات شیوه عمل‌کردن کد، توضیح علت قراردادن متغیرها و نتایج آن کد است. نوشتن مواردی از قبیل تاریخ نوشتن کد، نویسنده کد، نوشتن عبارات نامفهوم و ارجاع‌دادن به نویسنده آن قطعه کد در توضیحات ضرورتی ندارد.
- پرهیز از ترکیب دو زبان برنامه‌سازی به‌صورت پی‌درپی، به‌عنوان‌مثال برای طراحی صفحات وب قراردادن CSS[5]های درون‌خطی و یا جاوا اسکریپت‌های[6] پراکنده با رویه‌های کوچک در کد HTML[7]، نمونه‌ای از ترکیب نابجای زبان‌های برنامه‌سازی است. عدم توجه به این مسئله، باعث ایجاد حجم عظیمی از علامت‌های[8] المان‌ها به همراه CSSهای درون آن‌ها شده و درنتیجه منجر به کاهش خوانایی برنامه می‌شوند. جهت جلوگیری از این مشکل معماری سه‌لایه[9] برای برنامه‌های تحت وب پیشنهاد می‌شود. در این معماری لایه طراحی، کدنویسی و ارتباط با بانک اطلاعاتی از هم جدا شده‌اند و به‌طور مجزا نوشته می‌شوند [۲۳].

۲-۱- تابع شفاف[10]

تابعی که عملکرد قابل پیش‌بینی و انجام آزمون به‌صورت خودکار بر روی آن میسر است، تابع شفاف نامیده می‌شود [۲۴]. جهت رسیدن به این اهداف، معیارهایی برای نوشتن تابع وجود دارد که بیان می‌شود:

- هدف شفاف و روشن[11]: هر تابع باید برای تنها یک هدف مشخص ایجاد شده تا به‌راحتی قابل فهم[12] بوده و در زمان کمتری اشکال‌زدایی[13] شود. این مسئله به نوشتن تابعی که هیچ‌گونه اشکالی نداشته و تغییر آن بدون ایجاد اشکال جدید همراه است

1- Clean Coding
2- Flow diagram
3- Pseudo-Code
4- Comments

5- Cascading Style Sheets
6- Java Script
7- Hyper Text Markup Language
8- Tag
9- Model-View-Controller
10- Clean Method
11- Clean Purpose
12- Predictable
13- Debug

کمک خواهد کرد. اگر برنامه‌نویس در نوشتن تابع بر روی یک هدف خاص تمرکز کند، آن‌گاه به‌راحتی کدی با حداقل اشکال را ایجاد خواهد کرد.

- استفاده از نام‌گذاری استاندارد برای توابع و متغیرها: در نام‌گذاری استاندارد توابع و متغیرها، نام آن‌ها توضیح‌دهنده نیاز و عملکرد آن‌ها است. در هر زبان، استانداردهایی برای نوشتن کد وجود دارد که زبان C# نیز دارای استانداردهای خاص خود است [۲۵].

- پرهیز از نوشتن توابع بزرگ: در حین کدنویسی ممکن است اندازه یک تابع بیش‌تر از حد استاندارد شود که باید از وقوع چنین اتفاقی جلوگیری نمود. یک راه‌حل مناسب، شکستن این توابع بزرگ به توابع کوچک‌تر است. معمولاً بعضی رویه‌ها به‌صورت تکرارپذیر در میان بخش‌های مختلف برنامه دیده می‌شوند که گروه برنامه‌نویسی می‌تواند برای استفاده بهینه، آن‌ها را به‌صورت توابعی جداگانه تعریف کند.

- قابلیت آزمون کد خودکار[1]: در این حالت تابع به‌گونه‌ای نوشته می‌شود که قابل فراخوانی توسط ابزارهای آزمون واحد[2] خودکار را دارد. این امر منجر به افزایش کیفیت کدنویسی و به‌حداقل‌رساندن اشکالات خواهد شد. به‌عنوان مثال، در کنار زبان برنامه‌نویسی C#، مایکروسافت یک نوع پروژه با عنوان آزمون واحد در ابزار ویژوال استودیو قرار داده است که توانایی انجام این کار را به برنامه‌نویسان این زبان می‌دهد.

- استفاده از توابع هم‌نام[3]، مقادیر ورودی که در کدنویسی تابع استفاده می‌شوند پارامتر و مقادیری که در هنگام فراخوانی به تابع ارسال می‌شوند، آرگومان[4] نام دارند. توابعی که با آرگومان‌های مختلف یک عمل مشخص را انجام می‌دهند، توابع هم‌نام هستند. مثلاً برای جمع دو یا چندین عدد به ازای ورودی‌های مختلف چندین تابع با ورودی‌های مختلف نوشته می‌شود. با این کار هنگام صدازدن تابع جمع، کامپایلر[5] یا مفسر[6] با توجه به پارامترهای ارسال‌شده در فراخوانی، تابع مورد نظر را اجرا می‌کند.

- روش استفاده از آرگومان‌های نامی[7]، با تبدیل کلاس نوشته‌شده به فایل DLL[8] کد منبع کلاس در دسترس نخواهد بود. با فرض وجود کلاسی با نام شخص[9] که شامل کاربران است و قرار داشتن تابعی با عنوان ارسال[10] که از آن برای ارسال اعلانات از طریق ایمیل و پیامک به کاربران استفاده می‌شود. نمونه فراخوانی ساده این تابع در شکل (۳) آمده است.

```
Person Costumer = new Person();
Costumer.Send("Ali", true, false);
```

شکل (۳): نمونه فراخوانی ساده تابع Send

زمانی که قطعه کد شکل (۳) مشاهده می‌شود، برنامه‌نویسی که در آینده به این قطعه کد مراجعه کند، متوجه دلیل قرارگیری عبارات True و False در آرگومان دوم و سوم نمی‌شود. با توجه به این‌که کلاس Person به فایل DLL تبدیل شده است، کد منبع نیز در دسترس نیست. برای فهم کارکرد این تابع، یا باید سعی و خطا کرد یا به مستندات مراجعه نمود که منجر به افزایش زمان به‌روزرسانی کد می‌شود. راهی که برنامه‌نویس برای افزایش خوانایی برنامه می‌تواند انجام دهد نمونه کد منبع شکل (۴) است.

```
string Name = "Ali";
bool SendSMS = true;
bool SendEmail = false;
Person Costumer = new Person();
Costumer.Send(Name, SendSMS, SendEmail);
```

شکل (۴): نمونه فراخوانی تابع Send با خوانایی بالا

با مشاهده شکل (۴)، مشخص می‌شود پارامتر دوم و سوم مشخص‌کننده ارسال و یا عدم ارسال پیامک و ایمیل برای نام کاربری که در پارامتر اول ذکر شده است. در این شکل عملیات فراخوانی، سه عدد متغیر استفاده شده و سه خط کد نیز به برنامه اضافه شده است. این کار موجب افزایش فضای حافظه اشغال‌شده جهت خوانایی برنامه می‌شود، هم‌چنین کد منبع طولانی‌تری نیز ایجاد شده است؛ به همین دلیل، بهتر است از آرگومان‌های نامی استفاده شود. در شکل (۵) نمونه فراخوانی تابع با استفاده از این ترفند که در زبان C# وجود دارد، نشان داده شده است.

```
Person Costumer = new Person();
Costumer.Send(Name:"Ali", SMS:true, Email:true);
```

شکل (۵): فراخوانی تابع Send با آرگومان نامی

- توضیح‌گذاری توابع، در نوشتن توابع در زبان C# نوعی فرمت برای نوشتن توضیحات تابع وجود دارد که به برنامه‌نویسان کمک می‌کند خوانایی کد خود را افزایش دهند. همان‌طورکه در شکل (۶) نشان داده شده در این نوع توضیح گذاری، علاوه بر خلاصه‌ای که از عملکرد تابع نوشته می‌شود، برنامه‌نویس می‌تواند توضیحاتی

1- Automatic Code Testing
2- Unit Test
3- Method Overloading
4- Argument
5- Compiler
6- Interpreter
7- Named Arguments
8- Dynamic Link Library
9 -Person
10- Send

درباره پارامترها و مقداری که تابع برمی‌گرداند قرار دهد. با مراجعه به کد این تابع، عملکرد ورودی و خروجی تابع مشخص می‌شود. با این نوع توضیح‌گذاری در کد منبع هنگام فراخوانی آن تابع به‌صورت پنجره مانند شکل (۷) به برنامه‌نویس توضیحات مورد نظر نشان داده می‌شود.

```
/// <summary>
/// محاسبه تقسیم
/// </summary>
/// <param name="numerator">صورت</param>
/// <param name="divisor">مخرج</param>
/// <returns>پاسخ تقسیم</returns>
1 reference
private Decimal Division_Function(string numerator,
                                  string divisor)
```

شکل (۶): نمونه توضیح‌گذاری تابع در C#

```
Division_Function()
decimal Form1.Division_Function(string numerator, string divisor)
محاسبه تقسیم
numerator: صورت
```

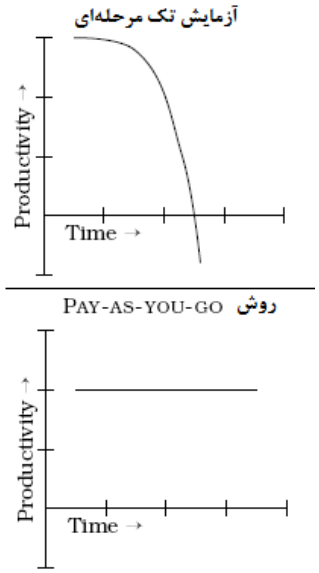
شکل (۷): نمونه فراخوانی تابع توضیح‌گذاری‌شده استاندارد C#

۲-۲- نوشتن کد شفاف

در برنامه‌نویسی پدافندی، علاوه‌بر داشتن کدی که در برابر حملات و اتفاقات غیرمنتظره مقاوم است، داشتن کدی که بتوان سریع آن را به‌روزرسانی کرد نیز اهمیت دارد زیرا از دست‌دادن زمان در به‌روزرسانی برنامه، گاهی خطرناک و هزینه‌بر است [۲۶]. ازاین‌رو، در کدنویسی باید نکاتی را رعایت نمود که خوانایی آن کاهش نیابد. به‌عنوان مثال، در این حالت کدنویسی دستورات شرطی و حلقه‌های تودرتو به‌صورت پیچیده نوشته نمی‌شوند [۲۷].

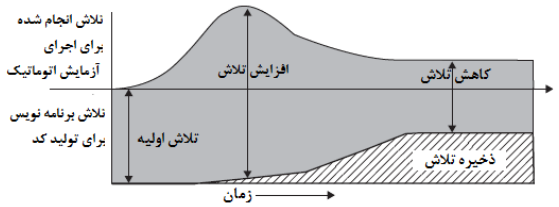
۲-۳- آزمون واحد

آزمون واحد، بررسی صحت عملکرد قطعه‌ای از کد به‌وسیله کدهای دیگر است. کدهای این آزمون توسط برنامه‌نویس نوشته می‌شوند. هدف آزمون واحد، بررسی کیفیت نرم‌افزار نیست، بلکه هدف بررسی صحت عملکرد قطعه کدی است که اخیراً به برنامه اضافه شده است. در آزمون کد خودکار نیازی به اجرای چندین مرتبه برنامه نبوده و کافی است کدها به‌گونه‌ای نوشته شود که بتوان آزمون خودکار بر روی آن‌ها اعمال کرد. با معرفی برنامه‌ای که قصد انجام آزمون واحد بر روی آن وجود دارد به پروژه آزمون واحد خودکار، این آزمون به‌صورت خودکار بر روی آن انجام می‌شود [۲۰]. مهم‌ترین دلیلی که برنامه‌نویس‌ها به‌موجب آن از نوشتن آزمون‌های واحد امتناع می‌کنند، زمان‌گیربودن آن است. مشکل از آن‌جاست که اکثر افراد به آزمون به‌عنوان مرحله آخر توسعه فکر می‌کنند. در این حالت، نوشتن آزمون‌های واحد سخت و زمان‌گیر خواهند بود. به‌همین جهت، برای جلوگیری از این مسئله روش pay-as-you-go مطرح شده است. در این روش با اضافه‌شدن هر واحد کوچکی به سیستم، آزمون واحد آن نیز تهیه می‌شود. به این صورت در طول توسعه سیستم اشکال کم‌تری رخ خواهد داد چون اجزای آن درحین توسعه به تفصیل مورد بررسی قرار گرفته است. اثر این روش در شکل (۸) ملاحظه می‌شود [۲۸]. نوشتن آزمون‌های واحد زمان‌بر هستند اما توسعه پیوسته آن‌ها با به‌تأخیرانداختن آزمون‌ها به انتهای پروژه، همانند شکل (۸) تأثیر بسیار قابل توجهی در بهره‌وری خواهند داشت. بالابودن زمان اجرا و اشکال‌زدایی برنامه، زمان‌بربودن اصلاح کدهایی که عملکرد مناسبی از آن‌ها دیده نمی‌شود و وقت‌گیربودن یافتن منشأ اشکال در کد باعث می‌شود که اگر آزمون واحد در حین انجام پروژه صورت نگیرد، بهره‌وری کاهش می‌یابد [۲۹].



شکل (۸): مقایسه زمان مصرف‌شده برای آزمون تک‌مرحله‌ای و استفاده از روش Pay-AS-You-Go [۲۰]

شکل (۹) بیانگر کاهش زمان و هزینه کدنویسی درطول زمان با رعایت اصول آزمون‌های واحد است. در آزمونی‌که در شرکت مایکروسافت انجام‌شده، در مدت یک سال بر روی نسخه دوم از یک برنامه که در دو حالت ایجادشده، صورت پذیرفته است. در یک حالت آزمون‌های واحد انجام‌نشده، درحالی‌که در دیگری آزمون‌های واحد به‌صورت پیوسته انجام شده است. طبق نتایج کدهای نسخه دوم ۲۰/۹٪ عیب کم‌تری نسبت به نسخه اول داشته و زمان توسعه آن ۳۰٪ افزایش یافته است [۳۰].



شکل (۹): روند هزینه کدنویسی در زمان با رعایت آزمون واحد [۲۰]

۳- مدیریت استثنا

برنامه‌نویس نهایت تلاش خود را در جهت ایجاد برنامه‌ای که با خطا یا اشکال مواجه نشود، انجام می‌دهد ولی گاهی در هنگام اجرا، برنامه با اشکالاتی مواجه می‌شود. برنامه‌نویس باید پدافندهای لازم برای مقابله با خطا و یا اشکالات رخداده پیش‌بینی کند تا برنامه از کار نیفتاده و استثناء مدیریت شود. استثناءها به سه دسته تقسیم می‌شوند:

استثناءهای ساختار برنامه: این استثناءها در زمان ترجمه برنامه رخ می‌دهند. زمانی که یک برنامه استثنای ساختار دارد، کامپایلر آن را اجرا نخواهد کرد و درهمان لحظه کامپایلر استثناءها را نمایش می‌دهد.

استثناءهای منطق برنامه: استثناءهای منطقی به عهده برنامه نویس است. اگر برنامه‌نویس الگوریتم را درست طراحی نکند این استثناء رخ می‌دهد.

استثناءهای زمان اجرا: این استثناءها در زمان اجرای برنامه رخ می‌دهند که در این صورت برنامه متوقف می‌شود و نوع استثناء را به‌صورت یک عدد صحیح بلند گزارش می‌کند. ساختار مدیریت استثناءها در زبان #C به‌صورت شکل (۱۰) است.

بلوک try بلوکی از کدها را شناسایی می‌کند که برای استثناءهای خاص فعال خواهند شد. Try با یک یا بیش‌تر از یک بلوک catch دنبال می‌شود

```
private void Test_Exception()
        {
            try
            {
                //اجرای کد برنامه
            }
            catch (Exception)
            {
                //مدیریت استثنا در صورت وقوع
            }

            finally
            {
                //  اجرا در هر صورت
            }
        }
```

شکل (۱۰): ساختار مدیریت استثناءها در زبان #C

برنامه یک استثناء را با کنترل‌کننده استثناء[1]، در مکانی‌که به مشکل رسیدگی‌شده شناسایی می‌کند. کلمه کلیدی catch نشان‌دهنده فریبنده[2] یک استثناء هست. بلوک finally برای اجرای مجموعه ارائه‌شده از بیانیه‌ها استفاده می‌شود. این قسمت از کد درصورت بروز یا عدم بروز استثناء همواره اجرا خواهد شد.

۳-۱- کنترل ورودی‌های تابع

ورودی‌های تابع گاهی مطابق انتظار برنامه‌نویس نیست، به‌عنوان مثال، با اجرای تکه کد شکل (۱۱)، جواب عملیات تقسیم دو عدد ورودی با استفاده از این تابع برگردانده می‌شود.

```
private Decimal Division_Function(string numerator,
                                  string divisor)
{
    return (Convert.ToDecimal(numerator) /
            Convert.ToDecimal(divisor));
}
```

شکل (۱۱): کد منبع تقسیم اول

در کد شکل (۱۱)، با وجود این‌که ورودی‌های متنی به عددی تبدیل شده‌اند ولی ممکن است که کاربر نهایی عدد دوم را صفر وارد کند. در این حالت، خطای تقسیم بر صفر رخ خواهد داد و باید صفرنبودن عدد دوم بررسی شود. برای رفع این مشکل ممکن است برنامه‌نویس کد را به شکل (۱۲) تغییر دهد. در این حالت نیز ممکن است کاربر هیچ عددی را به‌عنوان مخرج یا صورت کسر وارد نکرده لذا در صورت یا مخرج مقدار Null قرار خواهد گرفت؛ در این صورت برنامه با خطا مواجه خواهد شد. در زبان #C امکان استفاده از تابع TryParse علاوه‌بر تبدیل ورودی‌های تابع به متغیر از نوع دلخواه، Null بودن یا حتی غیرعددی بودن کاراکترهای واردشده توسط کاربر نیز بررسی می‌شود. نمونه اجرایی برنامه با استفاده از تابع TryParse در شکل (۱۳) مشاهده می‌شود.

```
private Decimal Division_Function(string numerator,
                                  string divisor)
{
    Decimal Result = 0;
    Decimal numerator_Value = Convert.ToDecimal(numerator);
    Decimal divisor_Value = Convert.ToDecimal(divisor);
    if (divisor_Value > 0)
    {
        Result = (numerator_Value / divisor_Value);
    }
    return Result;
}
```

شکل (۱۲): کد منبع تقسیم دوم

1- Exception handler
2- Catching

```
1 reference
private Decimal Division_Function(string numerator,
                                  string divisor)
{
    Decimal Result = 0;
    decimal divisor_Value = 0;
    decimal.TryParse(divisor, out divisor_Value);

    decimal numerator_Value = 0;
    decimal.TryParse(numerator, out numerator_Value);

    if (divisor_Value > 0)
    {
        Result = (numerator_Value / divisor_Value);
    }
    return Result;
}
```

شکل (۱۳): کد منبع تقسیم سوم

۳-۲- روش شکست سریع

هنگام بروز خطا در برنامه، بهتر است کاربر برنامه‌نویس و یا حتی کامپایلر سریعاً متوجه بروز آن شده تا از ادامه اجرای برنامه که گاهی خطرناک یا هزینه‌بر است، جلوگیری شود؛ به این روش، شکست سریع گفته می‌شود. در این روش، برنامه‌نویس کد منبع را به‌گونه‌ای می‌نویسد که در جاهایی‌که ممکن است خطا رخ دهد، علاوه‌بر بررسی خطا و جلوگیری از بروز آن، علت وقوع آن‌را نیز گزارش دهد [۳۱]. در زبان C# این امکان قرار داده شده که در زمان اجرا اگر برنامه با خطایی روبرو شد با کلمه کلیدی Throw این خطا اعلان می‌شود؛ در این کلاس می‌توان متنی را جهت اعلام خطای مورد نظر نشان داد. نمونه کد منبع تابع تقسیم ساده در شکل (۱۴) نشان داده شده است.

```
decimal divisor_Value = 0;

if (string.IsNullOrWhiteSpace(divisor)){
    throw new ArgumentException(
        "مخرج کسر باید وارد شود!","divisor");
}
if (!decimal.TryParse(divisor,out divisor_Value)){
    throw new ArgumentException(
        "کاراکتر غیرعدد وارد شده است!","divisor");
}
if (divisor_Value == 0){
    throw new ArgumentException(
        "مخرج کسر صفر وارد شده است!", "divisor");
}
```

شکل (۱۴): نمونه کد با استفاده از روش شکست سریع

در این کد، منبع خطاهای مربوط به مخرج کسر بررسی شده‌اند. این نوع بررسی‌ها به آرگومان‌هایی که برای تابع فرستاده‌شده پرداخته است و با عنوان استثنای آرگومان نام‌گذاری می‌شوند. همان‌طورکه مشاهده می‌شود تمام خطاها به‌طور دقیق بررسی شده و درصورت وقوع به کاربر اطلاع داده می‌شوند. در روش شکست سریع به‌محض دیدن خطا در برنامه یک استثناء با کلمه کلیدی Throw توسط برنامه‌نویس تولید می‌شود، آن استثناء به قسمت catch آمده و در آن‌جا مدیریت می‌شود. گروه فایل سرآیند ApplicationException استثناءهای تولیدشده توسط برنامه‌های کاربردی را پشتیبانی می‌کنند، بنابراین استثناءهای تعریف‌شده توسط برنامه‌نویس باید از این گروه استخراج شوند. علاوه‌بر این‌که این امکان به‌صورت دستی به برنامه‌نویس داده شده، به‌صورت خودکار زبان C# کلاس‌ها و کتابخانه‌هایی جهت مدیریت خطا دارد که در جدول (۱) مشاهده می‌شوند. این دسته به فایل سرآیند System Exception مربوط می‌شوند.

جدول (۱): انواع کلاس مدیریت خطاهای سیستمی

رسیدگی به خطاهای	کلاس استثنا
ورودی/خروجی	IO.IOException
خروج شاخص از محدوده آرایه	IndexOutOfRangeException
عدم سازگاری با نوع آرایه	ArrayTypeMismatchException
تولید شی پوچ	NullReferenceException
تقسیم بر صفر	DivideByZeroException
تبدیل نوع	InvalidCastException
کمبود فضا	OutOfMemoryException
سرریز	StackOverflowException

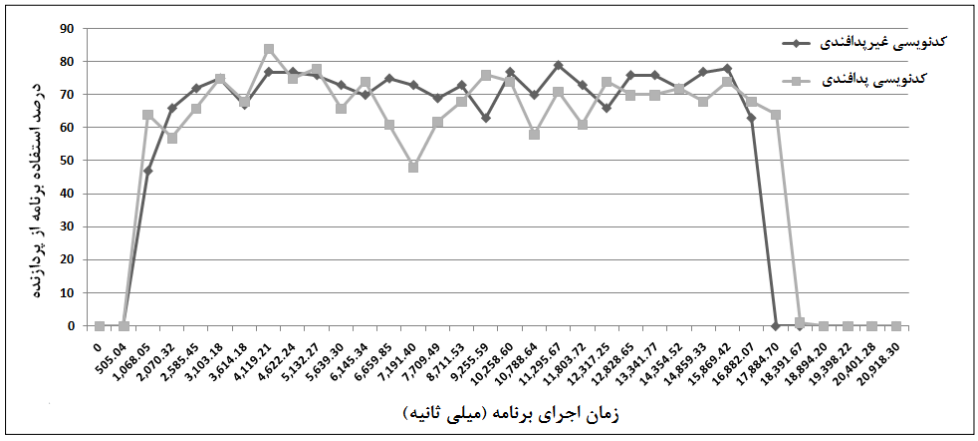
۴- پیاده‌سازی و ارزیابی

جهت ارزیابی کارایی کدنویسی پدافندی و کدنویسی عادی -که در این مقاله غیرپدافندی نامیده می‌شود- یک برنامه را با دو حالت پدافندی و غیرپدافندی در سیستمی با مشخصات پردازنده CORE^{TM}i7 شرکت اینتل و حافظه اصلی ۸ گیگابایت اجرا شده است. در حالت غیرپدافندی هیچ یک از مواردی که در مقاله گفته شد در کدنویسی رعایت نشده و کل عملیات درنظر گرفته شده در یک تابع نوشته شده و هنگام فراخوانی آن تابع صدا زده می‌شود. برنامه دارای یک بانک اطلاعاتی، شامل یک جدول با شش فیلد است. برای مقایسه بهتر، برنامه در هر دو حالت پدافندی و غیرپدافندی به مدت ۲۰ ثانیه اجرا می‌شود. در برنامه مورد نظر، ابتدا عملیات درج ۳۵ سطر در بانک اطلاعاتی انجام می‌شود، پس از آن، ۳۵ سطر واردشده در برنامه با اعمال تغییراتی به‌روزرسانی می‌شوند، سپس مقادیر از بانک اطلاعاتی حذف شده و شناسه جدول از سر گرفته می‌شود. در واقع طی اجرای ۲۰ ثانیه‌ای برنامه عملیات درج، به‌روزرسانی و حذف ۳۵ سطر در بانک اطلاعاتی انجام می‌شود.

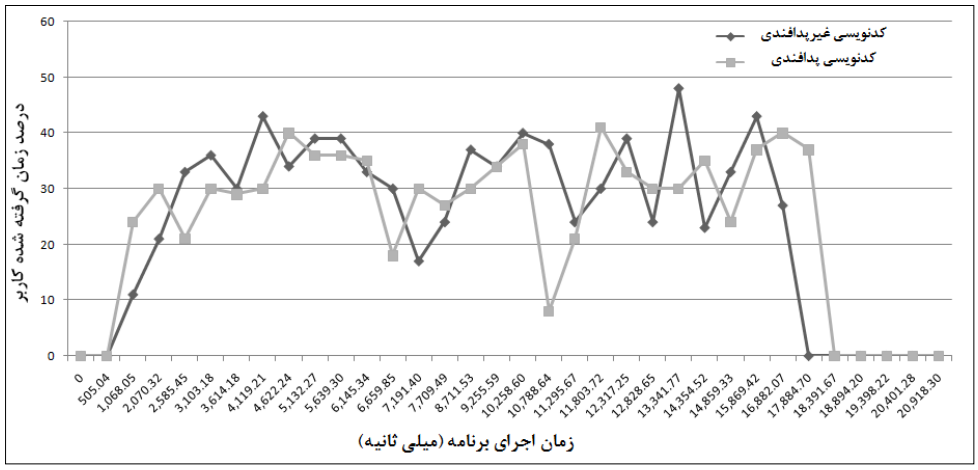
در حالت پدافندی یک کلاس با عنوان Person ایجاد شده که برای هر سطری که قرار است درج یا به‌روزرسانی شود، متغیرها و عملیات لازم را در خود دارد. در حالت پدافندی تمام مواردی که در مقاله گفته شد در کدنویسی رعایت شده است. درنتیجه، تعداد خط و فراخوانی‌های استفاده در حالت پدافندی بیشتر از حالت غیرپدافندی است. لذا این تصور پیش می‌آید که برنامه‌نویسی پدافندی منجر به افزایش زمان و هزینه اجرا شود.

جهت بررسی این موضوع ما از ابزار تجزیه تحلیلی که در ویژوال استودیو ۲۰۱۳ تعبیه شده است استفاده کردیم. این ابزار براساس معیارهای مختلف، از زمان شروع اجرای برنامه تا زمانی که برنامه بسته می‌شود برنامه را مورد ارزیابی قرار می‌دهد. شکل (۱۵) ارزیابی، درصد استفاده برنامه از پردازنده در دو حالت پدافندی و غیرپدافندی است. منظور از این درصد، کل درصد استفاده از پردازنده مرکزی نبوده و شامل درصد استفاده برنامه از پردازنده است، زیرا برنامه‌ها در دو زمان متفاوت اجرا شده‌اند و در هر حالت میزان استفاده سایر برنامه‌ها از پردازنده متفاوت است. با توجه به این‌که در این شکل نمی‌توان تعیین کرد که میزان استفاده از پردازنده مرکزی در کدام حالت بیش‌تر از دیگری است -تنها تفاوت حدود ۲۵ میلی‌ثانیه است که در حالت پدافندی پردازنده بیشتر مشغول بوده است (حدود ثانیه ۱۸/۱۳۹)- ما از میانگین زمان استفاده از پردازنده، جهت مقایسه استفاده کردیم. میانگین درصد استفاده برنامه از پردازنده در حالت غیرپدافندی ۵۴/۷۰ بود، درحالی‌که این مقدار برای حالت پدافندی ۵۴/۳۲ درصد است که این نشان از ۰/۰۴٪ بار پردازشی کم‌تر حالت پدافندی است. با توجه به کم‌بودن این اختلاف در جدول (۲) این معیار برابر در نظر گرفته می‌شود.

جهت بررسی بهتر موضوع، میزان درصد هزینه پردازشی کاربر در هنگام پردازش محاسبه شده که در شکل (۱۶) نشان داده شده است. در این حالت، میانگین درصد هزینه برای حالت غیرپدافندی و پدافندی، به‌ترتیب ۲۴/۴۱٪ و ۲۴/۲۳٪ است. این مقایسه نیز کاهش ۰/۰۲٪ هزینه حالت کدنویسی پدافندی را نشان می‌دهد، اما جهت مقایسه، می‌توان این از اختلاف جزئی صرف‌نظر نمود.



شکل (۱۵): درصد استفاده برنامه از پردازنده در دو حالت پدافندی و غیرپدافندی



شکل (۱۶): میزان درصد هزینه پردازشی کاربر در هنگام پردازش در دو حالت کدنویسی پدافندی و غیرپدافندی

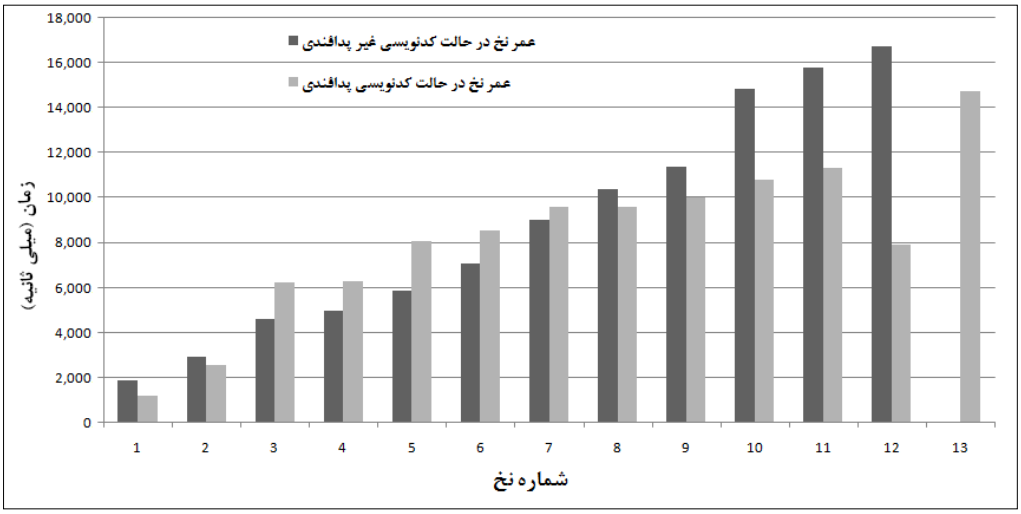
به‌دلیل پیمانه‌ای‌بودن کدنویسی پدافندی، جهت پاسخگویی به نیازهای برنامه، نخ‌های بیشتری در سطح پردازنده ایجاد می‌شود. لذا این تصور ایجاد می‌شود که این میزان نخ‌ها در روند اجرای برنامه تأثیرگذار است. معیار بعدی جهت ارزیابی، میزان عمر نخ‌های تشکیل‌شده در هسته پردازنده است. این مقایسه در شکل (۱۷) آمده است. در این حالت ما تنها نخ‌هایی که توسط کدنویسی ایجاد شده‌اند را درنظر گرفته و از سایر نخ‌هایی که کامپایلر تولید می‌کند، صرف‌نظر کرده‌ایم. در کدنویسی پدافندی به‌دلیل پیمانه‌ای‌بودن، در برنامه یک

نخ بیش‌تر از حالت غیرپدافندی ایجاد شده است، ولی طول عمر نخ‌ها در حالت پدافندی کم‌تر از حالت غیرپدافندی است. در حالت کدنویسی پدافندی ۱۳ نخ با میانگین عمر ۸/۲۳ ثانیه و غیرپدافندی ۱۲ نخ با میانگین عمر ۸/۸۰ ثانیه ایجاد شد.

از ارزیابی‌ها و مقایسه‌ها می‌توان نتیجه گرفت که کدنویسی پدافندی باعث افزایش زمان اجرای برنامه نمی‌شود. در جدول (۲) مقایسه بین دو حالت کدنویسی پدافندی و غیرپدافندی آمده است. معیارهای اول تا چهارم تا این‌جا بحث شد. با توجه به کنترل خطا و ورودی‌های کاربر در حالت پدافندی، احتمال ازکارافتادن برنامه به‌دلیل اشتباهات در کدنویسی بسیار کاهش می‌یابد. این ویژگی موجب افزایش امنیت در کدنویسی پدافندی نسبت به غیرپدافندی شده است.

معیار دیگر ارزیابی، تعداد توابع و کلاس است که در برنامه استفاده می‌شوند. در حالت پدافندی تعداد توابع و کلاس به نسبت غیرپدافندی بالاست. این مسئله باعث افزایش تعداد خط کدی که برای ایجاد برنامه به‌کاررفته، می‌شود.

با توجه به قراردادن توضیحات در کدنویسی پدافندی، این حالت از کدنویسی دارای خوانایی بیش‌تر و درنتیجه پیچیدگی کم‌تر از حالت غیرپدافندی است. پیچیدگی کم‌تر و خوانایی بیش‌تر منجر به کاهش زمان به‌روزرسانی و رفع اشکالات برنامه خواهد شد. در حالت پدافندی، برنامه به‌صورتی نوشته می‌شود که قابلیت انجام آزمون به‌صورت خودکار را دارد. در نتیجه این رویکرد، کاهش چشمگیر زمان انجام آزمون نرم‌افزار نسبت به حالت غیرپدافندی را در پی دارد.



شکل (۱۷): تعداد و عمر هر نخ ایجادشده در برنامه در دو حالت کدنویسی پدافندی و غیرپدافندی

جدول (۲): مقایسه کدنویسی پدافندی و غیرپدافندی

معیار کدنویسی	پدافندی	غیرپدافندی
زمان پردازش	برابر	
هزینه پردازشی	برابر	
تعداد نخ تشکیل شده	↑	↓
عمر نخ‌ها	↓	↑
کنترل خطا	✓	×
کنترل ورودی	✓	×
احتمال ازکارافتادن برنامه	↓	↑
امنیت	✓	×
تعداد توابع و کلاس	↑	↓
تعداد خط	↑	↓
پیچیدگی	↓	↑
خوانایی برنامه	↑	↓
زمان به‌روزرسانی	↓	↑
زمان آزمون کد	↓	↑

۵- نتیجه‌گیری

امنیت در کدنویسی نرم‌افزارها، مخصوصاً در سازمان‌های نظامی از اهمیت بالایی برخوردار است. کدنویسی پدافندی سعی بر جلوگیری از بروز حوادثی دارد که از کدنویسی نامناسب نشات می‌گیرد. این حوادث شامل از کار افتادن برنامه، دسترسی به کد منبع، اجرای سنگین همراه با استفاده بیش از حد از منابع سخت‌افزاری و حتی افزایش زمان نیاز برای به‌روزرسانی و رفع ایرادهای برنامه هستند. این مقاله با معرفی امکانات زبان برنامه‌نویسی C# و برنامه ویژوال استودیو درصدد رفع این ایرادها است. یکی از راه‌کارهای معرفی‌شده در این مقاله، معرفی نحوه کنترل ورودی‌های کاربر و خطاهای زمان اجرا در زبان C# بود. این راه‌کارها موجب کاهش احتمال از کار افتادن برنامه می‌شود. استفاده از روش شکست سریع در حین اجرای برنامه نیز معرفی شد که موجب مدیریت مناسب استثناءهای رخ داده در برنامه می‌شود.

هنگام به‌روزرسانی و رفع ایرادات برنامه، یکی از دغدغه‌های برنامه‌نویسان، پیچیده‌بودن درک و فهم کدهای برنامه است. در تابع شفاف، توضیحات به‌طور مؤثر نوشته شده و ورودی‌های تابع کنترل می‌شوند، هم‌چنین فراخوانی آن‌ها با آرگومان‌های نامی انجام می‌شود که این عوامل موجب افزایش خوانایی کد، کاهش پیچیدگی و درنتیجه کاهش زمان به‌روزرسانی برنامه می‌شوند. با توجه به درنظر گرفتن امکان اجرای آزمون خودکار در کدنویسی پدافندی، زمان اجرای آزمون در برنامه در این حالت کدنویسی کاهش چشمگیری نسبت به حالت عادی کدنویسی در پی دارد.

استفاده از برنامه‌نویسی پدافندی افزایش تعداد خط کد را در پی دارد. این امر ممکن است باعث نگرانی درباره زمان اجرای آن شود؛ در این مقاله با پیاده‌سازی یک برنامه در دو حالت پدافندی نشان دادیم با این‌که در حالت پدافندی ۱۳ نخ و در غیرپدافندی ۱۲ نخ ایجاد شد، ولی میانگین عمر نخ‌ها در حالت پدافندی ۰/۵۷ ثانیه کم‌تر از حالت دیگر است. تعداد بیش‌تر نخ حاکی از پیمانه‌ای شدن و تقسیم بار پردازشی بین نخ‌های متعدد در هسته پردازنده است. میانگین درصد استفاده برنامه از پردازنده در حالت غیرپدافندی ۵۴/۷۰ بود، درحالی‌که این مقدار برای حالت پدافندی ۵۴/۳۲ درصد بود. میزان درصد هزینه پردازشی کاربر در هنگام پردازش در دو حالت کدنویسی غیرپدافندی و پدافندی به‌ترتیب ۲۴/۴۱٪ و ۲۴/۲۳٪ است. این اختلاف‌های ناچیز در درصد استفاده از پردازنده و هزینه پردازشی کاربر را می‌توان درنظر نگرفته و درنتیجه این معیارها برای دو حالت کدنویسی پدافندی و غیرپدافندی برابر بوده و افزایش تعداد کلاس، توابع و تعداد خط کد در حالت کدنویسی پدافندی تأثیری در افزایش هزینه پردازشی و زمان اجرای دستورات برنامه ندارد.

۶- مراجع

1. P. S. Gilmour, “Defensive programming,” Embedded Syst. Program. vol. 3, pp. 60-68, 1989.
2. D. Y. Cheng, J. T. Deutsch, and R. W. Dutton, “Defensive programming in the rapid development of a parallel scientific program,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 9, pp. 665-669, 1990.
3. A. A. J. Zumalde, J. M. Secall, and J. B. C. Junior, “Comparative Analysis on the Impact of Defensive Programming Techniques for Safety-Critical Systems,” LADC '09. Fourth Latin-American Symposium, pp. 95-102, 2009.
4. K. W. Collier and J. S. Collofello, “Issues in software cycle time reduction,” IEEE Fourteenth Annual International Phoenix Conference, pp. 302-309, 1995.
5. P. Chul Su, L. Jae Hee, S. C. Seo, and B. K. Kim, “Assuring software security against buffer overflow attacks in embedded software development life cycle,” Advanced Communication Technology (ICACT), The 12th International Conference on, pp. 787-790, 2010.
6. N. Davis, W. Humphrey, S. T. Redwine, G. Zibulski, and G. McGraw, “Processes for Producing Secure Software: Summary of US National Cybersecurity Summit Subgroup Report,” IEEE Security and Privacy, vol. 2, pp. 18-25, 2004.
7. S. Velmourougan, P. Dhavachelvan, R. Baskaran, and B. Ravikumar, “Software Development Life Cycle Model to Improve Maintainability of Software Applications,” Advances in Computing and Communications (ICACC), Fourth International Conference on, pp. 270-273, 2014.
8. P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, and C. Seaman, “Reducing Friction in Software Development,” IEEE Software, vol. 33, pp. 66-73, 2016.
9. B. Boehm and V. R. Basili, “Software Defect Reduction Top 10 List,” Computer, vol. 34, pp. 135-137, 2001.
10. M. K. Gupta, M. C. Govil, and G. Singh, “Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey,” In Recent Advances and Innovations in Engineering (ICRAIE), pp. 1-5, 2014.
11. R. Chen, Y. Liu, Z. Jia, and J. Gao, “Isolating and Understanding Program Errors Using Probabilistic Dispute Model,” Computer Software and Applications Conference (COMPSAC), IEEE 37th Annual, pp. 633-638, 2013.
12. J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” 20th IEEE/ACM international Conference on Automated software engineering, CA, USA, 2005.
13. C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “SOBER: statistical model-based bug localization,” SIGSOFT Softw. Eng. Notes, vol. 30, pp. 286-295, 2005.
14. H. Cleve and A. Zeller, “Locating causes of program failures,” 27th international conference on Software engineering, USA, 2005.
15. G. K. Baah, A. Podgurski, and M. J. Harrold, “The probabilistic program dependence graph and its application to fault diagnosis,” International symposium on Software testing and analysis, USA, 2008.
16. جرجم، سلیمان، دهقانی، مهدی، امن‌سازی نرم‌افزار مبتنی بر کدنویسی امن، فصلنامه پدافند غیرعامل، شماره ۱۷، بهار ۱۳۹۳ ص ۴۱.
17. R. Kumar, S. K. Pandey, and S. I. Ahson, “Security in Coding Phase of SDLC,” Wireless Communication and Sensor Networks, WCSN '07. Third International Conference, pp. 118-120, 2007.
18. B. Taylor and S. Kaza, “Security Injections@Towson: Integrating Secure Coding into Introductory Computer Science Courses,” Journal of ACM Transactions on Computing Education (TOCE), vol. 16, Issue 4, Article no. 16, 2016.

19. M. Zaidman, “Teaching defensive programming in Java,” J. Comput. Sci. Coll., vol. 19, pp. 33-43, 2004.

۲۰. کشوری، سامان، جوادزاده، محمدعلی، عباسی، مصطفی، بررسی انواع آزمون نرم‌افزار جهت افزایش کیفیت و امنیت نرم‌افزار. دهمین کنفرانس رمز دانشگاه جامع امام حسین(ع)، ۱۳۹۴.

21. A. Khalili, A. Sami, M. Azimi, and ET. Al, “Employing secure coding practices into industrial applications: a case study,” Journal of Empirical Software Engineering, vol. 21, Issue 1, pp. 4-16, February 2016.

22. P. R. Domínguez, “10 Principles for Keeping Your Programming Code Clean,” Available: http://www.onextrapixel.com/2011/01/20/10-principles-for-keeping-your-programming-code-clean/ (Jan. 05, 2016).

23. D. M. Selfa, M. Carrillo, and M. D. R. Boone, “A Database and Web Application Based on MVC Architecture,” 16th International Conference on Electronics, Communications and Computers, 2006.

24. R. C. Martin, “The Clean Coder: A Code of Conduct for Professional Programmers: Prentice Hall Press,” Robert C. Martin Series, 1st Edition, 2011.

25. L. Hunt, “C# Coding Standards for .NET,” Document Version 1.15, http://www.lance-hunt.net, 2007.

26. R. C. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship: Prentice Hall PTR,” Prentice Hall; 1 Edition, 2008.

27. M. Aderhold, G. Alexander, and M. Heiko, “Choosing a Formalism for Secure Coding: FSM vs. LTL,” TU Darmstadt, Technical TUD-CS-2013-0180, 2013.

28. A. Hunt and D. Thomas, “Pragmatic Unit Testing in C# with NUnit: The Pragmatic Programmers,” Pragmatic Bookshelf; 2 Edition, 2004.

29. G. Meszaros, “xUnit Test Patterns: Refactoring Test Code Addison-Wesley,” Addison-Wesley Professional; 1 edition, 2007.

30. L. Williams, G. Kudrjavets, and N. Nagappan, “On the Effectiveness of Unit Test Automation at Microsoft,” 20th International Symposium on Software Reliability Engineering, pp. 81-89, 2009.

31. J. Shore, “Fail fast [software debugging],” IEEE Software, vol. 21, pp. 21-25, 2004.

Using Defensive Programming Technique to Increase Software Security in C# Language

S. Keshvari, M. R. Hasani Ahangar*, S. Keshvari

Abstract

Due to the importance of computer software security in military organizations, Programmers and designers of these systems are required to comply with security issues in programming. This article focuses on the coding level in the software development cycle. Defensive goals are to prevent accidents caused by improper coding. These events include software crash, access to source code, excessive use of hardware resources and increasing the required time to update program. Then, using named arguments in the function call is introduced to help increase readability and thus reduce time to update the program. To manage runtime errors, the fail fast method for exceptions and their appropriate management are provided. These actions cause the increased program security and non-crash software for improper coding, and also increase the number of functions and classes, and therefore increase the line number of the code. This issue may be the cause for concern about the increase in the runtime and cost of the process. This paper suggests implementing a program in both defense and non-defense modes. The result shows the average life of thread in defensive mode is 8.23 seconds while the amount for non-defense is 8.80 seconds. On the one hand, the percentages of CPU usage on defense and non-defense modes are 54.32 and 54.70, respectively, and on the other hand, for the processing cost of users are 24.23 and 24.41 percent, respectively. These numbers indicate the same processing cost and runtime of the program in both defense and non-defensive modes.

Key Words: *Software Security, Defense Programing, Secure Coding, C# Programing*

* Imam Hussein Comprehensive University (mrhasani@ihu.ac.ir)- Writer-in-Charge