نشریه علمی- ترویجی محاسبات نرم
شمارهٔ اول/ بهار و تابستان ۱۳۹۱/ صفحه ۵۶- ۶۹

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

ارائهٔ یک روش مبتنی بر افزونگی نرم‌افزاری سطح دستورالعمل جهت تشخیص خطاهای روند اجرای برنامهٔ درون و بین بلوکی

آتنا عبدی[۱]، سید امیر اصغری[۲]، حسین پدرام[۳]، حسن طاهری[٤]
[۱ و ۲] دانشجوی دکتری، مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران، ایران
[۳] دانشیار، دانشکدهٔ مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران، ایران
[٤] دانشیار، دانشکدهٔ مهندسی برق، دانشگاه صنعتی امیرکبیر، تهران، ایران
{atena_abdi, seyyed_asghari, pedram, htaheri}@aut.ac.ir

چکیده: تجهیزات الکترونیکی در کاربردهای فضایی، می‌توانند مقاوم یا غیرمقاوم در برابر تشعشعات باشند که به دلیل هزینه و عدم دسترسی در بسیاری از کاربردها، گزینهٔ مناسب، استفادهٔ تجهیزات تجاری رایج (غیرمقاوم) است. استفاده از تجهیزات تجاری رایج در کاربردهایی همچون کاربردهای فضایی به خودی خود، قابلیت تحمل‌پذیری را در برابر تهدیداتی همچون تابش یون‌های سنگین ندارند؛ لذا باید تمهیداتی را در نظر گرفت که بتوان این تجهیزات را در برابر تهدیدات احتمالی مقاوم کرد. در این مقاله، یک روش مبتنی بر افزونگی نرم‌افزاری سطح دستورالعمل جهت تشخیص خطاهای روند اجرای برنامهٔ درون و بین بلوکی ارائه شده است که در مقایسه با روش‌های پیشنهادشده تاکنون دارای سربارهای حافظهٔ کمتر، کارآیی بهتر و پوشش اشکال بیشتری است.

واژه‌های کلیدی: افزونگی نرم‌افزاری، تجهیزات تجاری رایج، تشخیص خطاهای روند اجرا، کاربردهای فضایی، قابلیت تحمل‌پذیری.

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

۱. مقدمه

استفاده از تجهیزات تجاری رایج، هم‌اکنون در بسیاری از کاربردها رایج است؛ از جملهٔ این کاربردها به کاربردهایی همانند کاربردهای صنعتی [۱ـ۵]، کاربردهای بی‌درنگ [۳ـ۷]، کاربردهای نظامی [۸ـ۹]، کاربردهای صنایع هوایی [۱۰ـ۱۲] و کاربردهای فضایی [۱۳ـ۳۱] می‌توان اشاره کرد. تجهیزات مذکور، قیمت مناسبی دارند و به علت طراحی ساده و همه‌منظوره، در برابر خطاهای احتمالی سیستم آسیب‌پذیرند. امروزه و در کاربردهای همه‌منظوره که هزینهٔ زیاد تجهیزات مقاوم در برابر اشعه‌ها تحمل‌نشدنی است، استفاده از قطعات تجاری رایج و به کار بردن روش‌های کشف خطاهای احتمالی بسیار پراستفاده است.

درصد زیادی از اشکالات گذرا در سیستم به اشکالات بی‌اثر و خوابیده در خروجی کل سیستم تبدیل می‌شوند؛ به عبارتی، اثر آن‌ها در سیستم حذف می‌شود و به خروجی نهایی انتقال نمی‌یابد. از میان درصد باقی‌مانده، ثابت شده است که حدود ۳۳٪ تا ۷۷٪ به خطاهای روند اجرا و درصد باقی‌مانده به خطاهای داده‌ای تبدیل می‌شوند [۳۲]؛ بنابراین می‌توان نتیجه گرفت که جایگزین کردن روش‌های جدید برای تشخیص خطاهای روند اجرا به جای روش‌های سنتی تشخیص اشکالات گذرا در لایهٔ معماری، می‌تواند هزینهٔ اضافی تشخیص اشکالات بی‌اثر در نتیجهٔ خروجی را از سیستم حذف کند و به این ترتیب، کارآیی سیستم را بهبود و هزینهٔ آن را کاهش دهد [۳۳].

اولین و مهم‌ترین گام برای تحمل‌پذیر کردن سیستم‌ها در برابر اشکالات گذرا، تشخیص آن‌هاست. موفقیت در این مرحله می‌تواند پوشش اشکال مناسبی را برای سیستم ایجاد کند. برای تشخیص اشکالات گذرا و خطاهای روند اجرا، روش‌هایی ارائه شده‌اند که می‌توان آن‌ها را به دو دستهٔ کلی افزونگی سخت‌افزاری و نرم‌افزاری دسته‌بندی کرد. روش‌های مبتنی بر افزونگی سخت‌افزاری، دارای پوشش اشکال بهتری هستند، ولی هزینه و سربار زیادی را به سیستم تحمیل می‌کنند که در کاربردهای عام منظوره مورد پسند کاربران نیست؛ از سوی دیگر، روش‌های نرم‌افزاری، پوشش اشکال کمتر و تأخیر اجرای بیشتری دارند، ولی هزینه و سربار بسیار کمی را به سیستم اعمال می‌کنند و به دلیل انعطاف‌پذیری، می‌توان آن‌ها در سیستم‌های گوناگون به کار برد.

برای کنترل روند اجرا، کد برنامهٔ در حال اجرا به تعدادی بلوک پایه تقسیم می‌گردد و اجرای کد داخل بلوک‌ها و انشعاب بین آن‌ها توسط پردازندهٔ مراقب بررسی می‌شود. هر بلوک پایه از تعدادی دستورالعمل تشکیل شده که مابین دستورات پرشی قرار گرفته‌اند. خطاهایی که در این روش‌ها باید مورد بررسی قرار گیرند، به سه دستهٔ کلی تقسیم می‌شوند:

۱. پرش‌های اشتباه رخ داده در داخل یک بلوک پایه؛
۲. پرش اشتباه رخ داده بین دو بلوک پایه؛
۳. پرش‌های اشتباه رخ داده از یک بلوک پایه به فضای استفاده نشده از حافظه.

روش‌های ارائه شده در این زمینه، چه سخت‌افزاری باشند و چه نرم‌افزاری، باید قابلیت برخورد با یکی یا تمام این خطاها را داشته باشند.

در این مقاله، روشی نرم‌افزاری برای تشخیص خطاهای روند اجرا پیشنهاد شده است که علاوه بر پوشش اشکال بهتر، دارای سربار کارآیی و حافظهٔ بسیار بهتری نسبت به روش‌های پیاده‌سازی شدهٔ دیگری است که در این زمینه ارائه شده‌اند.

۲. کارهای گذشته

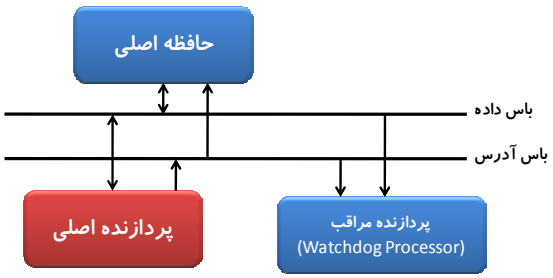
وقوع اشکالات گذرا در سیستم‌های کامپیوتری و در زمان اجرا، منجر به آسیب‌های قابل ملاحظه‌ای می‌گردد. برای رسیدن به قابلیت اطمینان در سیستم‌های کامپیوتری، یکی از اثربخش‌ترین روش‌های موجود، تشخیص خطاهای روند اجرا می‌باشد. برای تشخیص خطاهای روند اجرا از سال ۱۹۸۰ تاکنون، روش‌های بسیار زیادی ارائه شده است که می‌توان آن‌ها را به دو دسته عمدهٔ مبتنی بر سخت‌افزار و نرم‌افزار تقسیم کرد.

در روش‌های مبتنی بر افزونگی سخت‌افزاری، برای تشخیص خطای روند اجرا از یک پردازنده و زمان‌بند مراقب استفاده می‌شود. بدین نحو که اجرای عادی سیستم را با اجرای مورد انتظار مقایسه کرده و در صورت مغایرت بین آن‌ها، خطای روند اجرا تشخیص داده می‌شود. غالباً در این دسته از روش‌ها با انتساب یک امضا به هر بلوک پایه و ارسال امضاها در ابتدا و انتهای بلوک‌های پایه به پردازندهٔ مراقب، روند اجرای برنامه بررسی می‌شود [۳۳].

اولین مطالعات صورت‌گرفته در این زمینه، به دههٔ ۹۰ میلادی برمی‌گردد و تا به امروز همچنان ادامه دارد. یکی از ابتدایی‌ترین

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

روش‌های پیشنهاد شده برای تشخیص اشکالات گذرا در پردازنده‌ها استفاده از پردازنده‌های مراقب است که نوعی پردازشگر کمکی‌اند و با نظارت بر رفتار پردازندهٔ اصلی، اشکالات را در سیستم تشخیص می‌دهند. این ایده به نوعی، تعمیمی برای روش زمان‌بند مراقب است و در سطح سیستم پیاده‌سازی می‌گردد. ساختار کلی سیستمی که دارای پردازندهٔ مراقب است، در شکل (۱) آورده شده است [۳٤].



شکل (۱): ساختار سیستم دارای پردازندهٔ مراقب [۳٤]

روش‌های تشخیص اشکال مبتنی بر پردازندهٔ مراقب، دارای دو مرحله است: در فاز اول که مرحلهٔ تنظیمات نامیده می‌شود، اطلاعاتی دربارهٔ پردازنده و پردازش به پردازندهٔ مراقب داده می‌شود؛ در فاز دوم، پردازندهٔ کمکی، بر روی پردازندهٔ اصلی نظارت می‌کند و اطلاعات مربوط را استخراج می‌نماید. فرآیند تشخیص اشکال از مقایسهٔ اطلاعات جمع‌آوری شده و اطلاعات مرحلهٔ تنظیمات، تکمیل می‌گردد. اطلاعات ذکرشده شامل مکانیزم دسترسی به حافظه، جریان کنترلی (Control Flow)، سیگنال‌های کنترلی و یا نتایج منطقی است.

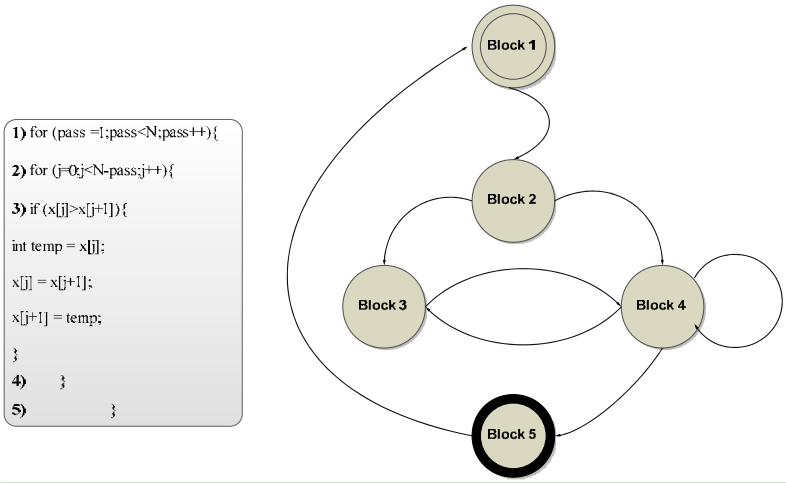
ساختار پردازندهٔ مراقب را می‌توان بدون تغییرات زیاد در سیستم، به آن اعمال کرد. هر برنامه را می‌توان به صورت گرافیکی نشان داد، به این ترتیب که گره‌ها نشانگر بخشی از برنامه است و یال‌ها، جریان کنترلی را نشان می‌دهند. هر گره می‌تواند یک عبارت تکی، بلوکی از دستورالعمل‌های بدون پرش، فاصلهٔ بدون پرش و یا یک روال تکی باشد. تمام طرح‌های بررسی کردن جریان کنترلی، بر اساس برچسبی است که به هر گره اختصاص داده می‌شود. مراقب به برچسب و روابط بین برچسب‌ها نسبت داده می‌شود. در طول اجرای یک برنامه، مراقب، جریان کنترلی برنامه را کنترل می‌کند؛ برچسب، گره‌ها را به صورت هم‌روند محاسبه می‌کند و با برچسب قبلی مقایسه می‌نماید. هرگونه تناقض بین دو برچسب ذکرشده به عنوان یک اشکال تلقی می‌گردد. روش‌های بسیار متفاوتی برای مقایسهٔ برچسب‌ها پیشنهاد شده است که در تعاریف گره‌ها و تفسیر انحراف با یکدیگر متفاوت‌اند.

در روش‌های نرم‌افزاری ارائه‌شده، روند کلی عملکرد، مشابه روش‌های قبلی است و بسته به کاربرد، امکان تشخیص سه نوع خطای روند اجرای بیان‌شده در آن‌ها وجود دارد. نکتهٔ اصلی و تفاوت آن‌ها با روش‌های سخت‌افزاری این است که عمل بررسی کردن روند اجرای کد برنامه توسط خودپردازندهٔ اصلی انجام می‌شود و به جای تحمیل افزونگی‌های سخت‌افزاری، از افزونگی‌های نرم‌افزاری در کد برنامه استفاده می‌شود. اساس کار شیوه‌های بررسی جریان کنترلی برنامه یا CFC[1] بر مقایسهٔ گراف کنترلی برنامه در حال اجرا با جریان کنترلی است که به طور استاتیک و در ابتدای کار برنامه برای آن پیش‌بینی شده، استوار می‌باشد. روش‌های نرم‌افزاری، معمولاً بر مبنای کپی‌برداری از کد یا داده انجام می‌گیرند و می‌توانند در سطح روال‌ها و یا عبارات (Statement) قرار بگیرند.

در این روش‌ها، تعدادی دستورالعمل افزونهٔ سطح ماشین به برنامه اضافه می‌شود. از سوی دیگر، برنامهٔ اجرایی بر روی پردازنده، به تعدادی بلوک پایه که دارای تعداد دستورالعمل‌های معینی می‌باشند و فاقد دستورات پرشی‌اند، تقسیم می‌گردد و برای هر بخش، یک امضا در نظر گرفته می‌شود. در زمان اجرا، توسط این امضاها جریان برنامه کنترل می‌گردد تا اجرا در نقاط صحیح به بلوک‌های برنامه، وارد و از آن‌ها خارج گردد. در شکل (۲)، نحوهٔ تقسیم کردن برنامه به بلوک‌های پایه و دستورات جداکننده نشان داده شده است. برنامه توسط یک گراف مستقیم نشان داده شده است که در آن، هر گره، یک دستورالعمل ماشین را نشان می‌دهد و یال‌ها همان جریان کنترلی می‌باشند. دستورات ماشین به صورت متوالی در حافظهٔ اصلی قرار داده می‌شوند. در شکل (۲)، دستورالعمل شمارهٔ ۲، یک فرمان شرطی است، پس به نوعی یک جداکننده بین بلوک‌های پایه محسوب می‌گردد. همچنین دستورالعمل شمارهٔ ۵ که مقصد یک پرش شرطی است، برای افراز بین بلوک‌ها به کار می‌رود. به این ترتیب و همان‌گونه که در شکل (۲) نشان داده شده، برنامهٔ اصلی به سه بلوک پایه تقسیم شده است [۳۵].

1. Control Flow Checking

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018



شکل (۲): نمایش نحوهٔ تقسیم‌بندی برنامهٔ نمونه به عده‌ای بلوک پایه [۳۵]

در برخی از کاربردهای خاص از جمله سیستم‌های فضایی، به علت تابش پرتوها و یون‌های سنگین، اشکالات گذرای زیادی منجر به اختلال در جریان کنترلی برنامه‌ها و سیستم‌های نرم‌افزاری و در نتیجه، رفتارهای غیرقابل پیش‌بینی می‌شوند. مطالعات زیادی بر روی اثر پروتون‌ها و یون‌های سنگین (اتم‌های سنگین‌تر از هلیم) بر روی مدارات الکترونیکی انجام شده است. یکی از آثار مهم این پروتون‌ها و یون‌ها، SEU[1] می‌باشد. تاکنون در کارهای زیادی در مورد برخورد نوترون‌ها، پروتون‌ها و الکترون‌هایی که به SEU منجر می‌شوند، صحبت شده است [۳۶ـ۴۲].

به دلیل خاص بودن محیط فضا و نیاز مبرم تجهیزات این محیط به قابلیت اطمینان بالا، کارهای زیادی برای افزایش قابلیت اطمینان تجهیزات فضایی انجام شده است؛ از جملهٔ این کارها RSCFC[2] [۴۳] است که در آن، برنامه به تعدادی بلوک پایه تقسیم می‌شود. در مرحلهٔ اول، وابستگی‌های بین بلوک‌ها استخراج می‌گردد، و سپس بر مبنای نوع وابستگی، به هر بلوک نشانی اختصاص داده می‌شود که در آن، وابستگی‌های موجود کد شده باشد. برای تشخیص اشکالات موجود در جریان کنترلی برنامه می‌توان از عملیات منطقی «و» مابین نشان‌های زمان اجرا با اطلاعات موجود در ابتدا و انتهای بلوک‌ها استفاده کرد.

این شیوه نسبت به کارهای پیشین، دارای پوشش اشکال و کارآیی بهتری است و حافظهٔ کمتری را مصرف می‌کند [۴۳]. در روش RSCFC، اگر برنامه را به N بلوک پایه تقسیم کنیم، آنگاه نشان تخصیص داده شده به هر بلوک، باید N+1 بیتی باشد که پرارزش‌ترین بیت آن باید مقدار ۱ داشته باشد. هر بیت این نشان، به جز پرارزش‌ترین بیت برای یک بلوک است [۴۴].

در مطالعهٔ دیگری که برای تشخیص اشکالات روند اجرا صورت پذیرفته CEDA[3] [۴۵]، سعی شده که دو پارامتر درصد پوشش اشکال و سربار اضافی را به نوعی متعادل کرده و هردوی آن‌ها در نظر گرفته شود. اساس کار در این روش بر این است که در زمان ترجمه، چندین دستورالعمل اضافه به صورت نهفته به برنامه افزوده می‌شود تا به صورت متناوب، امضاهای زمان اجرا را به‌روز و با مقادیر قبلی مقایسه کند. ارائهٔ یک روش جدید برای محاسبات زمان اجرا، منجر به کاهش چشم‌گیری در سربار کارآیی می‌شود و تشخیص اشکالات روند اجرا را ساده‌تر می‌کند. برای پیاده‌سازی این روش، مترجم GCC[4] تغییر داده شده و این روش با روش‌های گذشته توسط محک‌های SPEC 2000 مقایسه شده است. لازم به ذکر است که روش ارائه‌شده در این مقاله، از سایر روش‌های قبلی بهتر است و سربار کارآیی و پوشش اشکال را بهبود داده است [۴۶]. دستورالعمل‌های افزونه به صورت استاتیک به برنامه افزوده می‌شوند به گونه‌ای که در زمان اجرا، امضا متناوباً به‌روز می‌شود تا روند اجرای برنامه قابل کنترل باشد. برنامه به صورت یک گراف در نظر گرفته می‌شود و برای هر گره، امضای

1. Single Event Upset
2. Relationship Signatures for Control Flow Checking
3. Control Flow Error Detection through Assertion
4. CNU Compiler Collection

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

ابتدایی و امضای خروجی بر طبق عده‌ای قوانین نسبت داده می‌شوند. هر گره دارای دو پارامتر d1 و d2 است. این پارامترها برای به‌روزرسانی امضا استفاده می‌شوند و مقادیری دارند که امضا در هر نقطه با مقدار مورد انتظارش در زمانی که خطا وجود ندارد، برابر شود [۴۷]. هدف این روش، آن است که امضاهای ابتدا و انتهای هر بلوک به گونه‌ای نسبت داده شوند که خطاهای روند اجرا به صورت تفاوتی مابین مقادیر زمان اجرا و مقادیر مورد انتظار نشان داده شود و سپس تمام این تفاوت‌ها تشخیص شوند. این روش، تنها اشکالات پرش غیرمجاز مابین گره‌ها را تشخیص می‌دهد. پس از به‌روزرسانی انتهایی، مقدار امضای ابتدایی به امضای هنگام خروج تغییر می‌یابد. تغییرات ذکرشده را می‌توان در سطوح گوناگونی قرار داد. اگر ابزاری نوشته شود تا این تغییرات را به کد سطح بالا ببرد، قابلیت حمل در سطح معماری تضمین می‌شود، ولی این قابلیت در سطح دستورالعمل از بین خواهد رفت. از سوی دیگر، قرار دادن این تغییرات در سطوح پایین مانند اسمبلی، وضعیتی برعکس ایجاد می‌کند. با در نظر گرفتن این نکات در این روش، تغییرات به سطح میانی یعنی مترجم اعمال شده است. در این سطح، به هر دو قابلیت حمل ذکرشده خواهیم رسید، زیرا در این سطح عملیات مربوط به زبان تمام شده و عملیات وابسته به معماری به تازگی آغاز گردیده است. در نقاط معینی در برنامه که نقطهٔ بررسی (Checkpoint) نامیده می‌شوند، دستورهای کنترل قرار داده می‌شود. دستور کنترل، اشکالاتی را که قبلاً در اجرا رخ داده، تشخیص می‌دهد. در دستور کنترل، مقدار امضا با مقدار مورد انتظارش مقایسه می‌گردد و اگر برابر نبودند، اشکال رخ می‌دهد. در این روش، برنامه به صورت استاتیک تحلیل می‌شود و برای به‌روزرسانی و بررسی کردن امضای زمان اجرا، دستورهایی اضافه می‌کند. به این ترتیب به کارآیی بالا و تشخیص اشکال بسیار خوبی می‌رسد [۴۷].

CFCCB [۴۸]، روش دیگری در کنترل روند اجراست. در شیوهٔ CFCCB بر اساس دسته‌بندی بلوک‌های پایه به آن‌ها امضا تعلق می‌گیرد. برای دسته‌بندی بلوک‌های پایه تعداد پیش‌پردازنده(predecessor)های آن‌ها در نظر گرفته می‌شود که اگر یکی بود، دستهٔ شماره یک است وگرنه به دستهٔ شماره دو تعلق دارد. این دسته‌بندی در برخی مواقع درست نیست، زیرا گاهی ممکن است بین پیش‌پردازندهٔ دو گره اشتراک داشته باشیم. در این موارد، از روی گراف روند اجرا، یک گراف تقاطع (conflict) رسم می‌کنیم که در آن، دو گره در صورتی یال دارند که پیش‌پردازندهٔ مشترکی برای آن‌ها وجود داشته باشد. به این ترتیب، اگر در گراف تقاطع هیچ زیرگرافی از درجهٔ سه (درجهٔ گراف) موجود نباشد، می‌توان تمامی بلوک‌های پایه را در دو دسته قرار داد، در غیر این صورت، قابلیت دسته‌بندی در دو بخش وجود ندارد. برای حل مسئلهٔ تقاطع در CFCCB باید یک گره را از زیرگراف کامل به صورت اختیاری انتخاب کرد و سپس یک گره مابین گره انتخاب‌شده و پیش‌پردازندهٔ مشترک‌شده با گره دیگر قرار داد. گره افزوده شده هیچ دستورالعمل اضافه ندارد و یا فقط یک پرش مستقیم دارد و بلوک مجرد (abstract block) نامیده می‌شود. با افزودن بلوک مجرد در مکان‌های مورد نیاز می‌توان کلیهٔ بلوک‌های پایه را به دو بخش دسته‌بندی کرد. در روش CFCCB اگر اشکالی رخ بدهد، کنترل برنامه به ماژول مدیریت خطا (error handling) منتقل می‌گردد. در ابتدای هر بلوک پایه برحسب نوع دستهٔ آن، یک دستور قرار داده می‌شود تا امضا را به مقدار ورودی آن بلوک تنظیم کند و در خروجی نیز سه دستور قرار داده می‌شود و در آخرین آن‌ها مقدار امضا را با امضای خروجی این بلوک بررسی می‌کند که اگر برابر نباشد، خطایی رخ داده است.

در این مقاله برای تشخیص خطای بین بلوکی، روشی ارائه شده است که در آن، یک دستورالعمل در وسط بلوک پایه قرار داده می‌شود تا مقدار امضا را به صفر تغییر دهد. اگر در انتهای بلوک پایه، این دستور اجرا نشود، مقدار امضای این بلوک با مقدار مورد انتظار متفاوت خواهد بود؛ بنابراین، یک اشکال رخ داده است. در این الگوریتم، امکان تشخیص اشکالات پرشی مابین رویه‌ها (procedure) وجود دارد، به این معنا که زمانی که یک رویه چندین آدرس برگشت دارد، اگر به مقصد اشتباهی رفت، قابل تشخیص باشد. در این حالت، سه دستور دیگر به برنامه افزوده می‌شود که وظیفهٔ آن‌ها تشخیص این نوع از اشکالات می‌باشد. این روش، قابل پیکربندی است و از لحاظ سربار کارآیی، حافظه و همچنین پوشش اشکال، مقدار قابل قبولی دارد.

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

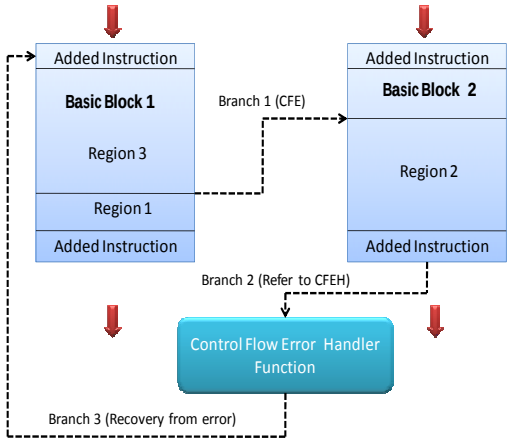
همان‌گونه که بیان شد، خطاهای کنترل روند اجرا به نوبهٔ خود می‌تواند منجر به خطاهای داده‌ای نیز شوند (داده‌های خروجی برنامه‌ها در این حالت نادرست خواهند شد). جهت تشخیص و تصحیح این خطاها نیز تاکنون روش‌های متعددی عنوان شده است (که البته این خطاها در دامنهٔ تحقیق این مقاله قرار نمی‌گیرند) که از جملهٔ آن‌ها می‌توان به روش ED^{4}I [۱] [۴۱] اشاره کرد. این روش، یک روش کشف اشکال نرم‌افزاری است که از طریق اجرای دو برنامهٔ مختلف اما با مجموعه داده‌های متفاوت (که یک عملکرد را پیاده‌سازی می‌کنند) و مقایسهٔ خروجی آن‌ها، می‌تواند هم اشکال‌های گذرا و هم اشکال‌های موقتی را تشخیص دهد. اشکال‌های گذرایی را که یکی از برنامه‌ها را دچار اشکال می‌کنند، می‌توان از طریق این روش تشخیص داد. از جملهٔ این اشکال‌های گذرا می‌توان به اشکال‌های گذرایی که در پردازنده‌ها و تغییر بیت‌هایی که در حافظه‌ها رخ می‌دهند، اشاره کرد. تغییر بیت، یک تغییر نامطلوب در حالت سلول‌های حافظه است که تحت تأثیر عوامل مختلفی ایجاد می‌شوند؛ SEUها، یکی از این عوامل‌اند؛ برای مثال، تغییر بیت‌هایی که در بخش کد برنامه در حین اجرای برنامه اتفاق می‌افتند، می‌توانند رفتار برنامه را تغییر دهند و منجر به تولید نتایج نادرست شوند. نیز می‌توان از طریق مقایسهٔ نتایج ناصحیح (حاصل از برنامهٔ دارای اشکال) و نتایج صحیح (حاصل از برنامهٔ بدون اشکال) به وجود اشکال (در این مورد، تغییر بیت) در یکی از برنامه‌ها پی برد. نکته‌ای که باید به آن توجه کرد، این است که این روش، توانایی تشخیص خطاهایی را که باعث می‌شوند تا برنامه در یک حلقهٔ نامحدود قرار گیرد و نتواند از آن خارج شود، ندارد (معمولاً این حالت در نتیجهٔ مواردی رخ می‌دهد که رجیسترهایی همانند IP دچار تغییر ناخواسته شده‌اند و اشاره‌گر دستورالعمل بعدی برنامه به محلی ناصحیح پرش می‌کند). این خطاها، معمولاً خطاهای روند اجرای برنامه نامیده می‌شوند (برنامه‌هایی که رویهٔ اجرای صحیح برنامه را با اختلال مواجه می‌کنند). برای تشخیص این اشکال‌ها می‌توان از روش‌های کشف خطاهای روند اجرای نرم‌افزاری و یا تایمرهای مراقب استفاده کرد.

1. Error Detection by Diverse Data and Duplicated Instruction

۳. روش پیشنهاد شده

در بخش قبلی، نمونه‌ای از روش‌های ارائه‌شده برای تشخیص خطاهای روند اجرا که مبتنی بر سخت‌افزار و نرم‌افزار بودند، بررسی شد. در ادامه، به بررسی روش ارائه‌شده در این مقاله پرداخته می‌شود. با توجه به محاسن برشمرده شده برای روش‌های مبتنی بر نرم‌افزار، روش ارائه‌شده، نرم‌افزاری می‌باشد و مانند روش‌های گذشته، کد برنامه را به عده‌ای بلوک پایه تقسیم می‌کند و به هر بلوک یک امضا تخصیص می‌دهد.

در شکل (۳)، یک خطای روند اجرا بین دو بلوک پایه نشان داده شده است که اجرای تکهٔ کد داخل بلوک پایهٔ شمارهٔ ۱ به دلیل رخ دادن یک خطای روند اجرا نیمه‌تمام باقی مانده و اجرا به اشتباه به بلوک پایهٔ شمارهٔ ۲ منتقل شده است. در انتهای بلوک پایهٔ شمارهٔ ۲، این خطا مشخص شده و کنترل به تابعی که جهت تصحیح این خطای روند اجرا رخ داده، انتقال یافته است. این تابع، قابلیت تشخیص بلوک پایه‌ای را که خطای روند اجرا در آن رخ داده است، دارد و کنترل را به ابتدای آن بلوک انتقال می‌دهد.



شکل (۳): نحوهٔ تشخیص و تصحیح خطاهای روند اجرا در پردازنده‌ها

برای تشخیص اشکالات روند اجرا در روش ارائه‌شده در این مقاله، به هر بلوک پایه، یک امضا نسبت داده می‌شود. این امضا، همان متغیر S_i است که بلوک‌های پیشرو بلوک فعلی را نشان می‌دهد.

برای کنترل روند اجرا در بلوک‌های پایه، سه دستورالعمل تعریف می‌کنیم. دستورالعملی به نام check که وظیفه‌اش، تأیید مقصد درست تعیین‌شده و بلوک فعلی یکی از پیشروهای بلوک

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

مبدأ بوده است. برای کنترل کردن این صحت دسترسی از رابطهٔ (۱) استفاده می‌کنیم:

err =S [sel] xor 1; (۱)

که در آن، S همان متغیر S_i است که با اجرای برنامه به‌روز شده است. در ابتدای هر بلوک پایه، اگر بیت شماره sel+1 (که این مقدار شمارهٔ بلوک پایه فعلی می‌باشد) برابر یک بود، یعنی مقصد به درستی انتخاب شده وگرنه سیگنال err که نشانگر خطاست، فعال می‌شود و کنترل برنامه به ابتدای دستور check گرهٔ قبلی بازگردانده شده و از آنجا دوباره اجرا می‌شود.

دستورالعمل دیگر به نام update به منظور به‌روزرسانی امضا که در متغیر S است، قرار داده می‌شود. برای به‌روزرسانی امضای روند اجرا از رابطهٔ (۲) استفاده می‌کنیم:

$S = S_i$ (۲)

به این ترتیب، متغیر S پس از خروج از هر بلوک پایه به‌روز می‌شود و برای رفتن به مقصد بعدی آماده می‌گردد. لازم به ذکر است که این مقدار برای اولین بار به مقدار ۱۰۰۰۰...۱ تنظیم می‌شود تا فقط بتواند به اولین بلوک پایه برود و سایر پرش‌ها برای آن غیرمجاز محسوب شود. برای ایجاد دستورالعمل‌های check, update سعی شده است تا حد ممکن از عملگر xor استفاده شود، زیرا صحت پردازشی بالایی را فراهم می‌کند و مشکلات نقاب‌بندی‌شدن‌های ناخواسته مانند and, or را ندارد.

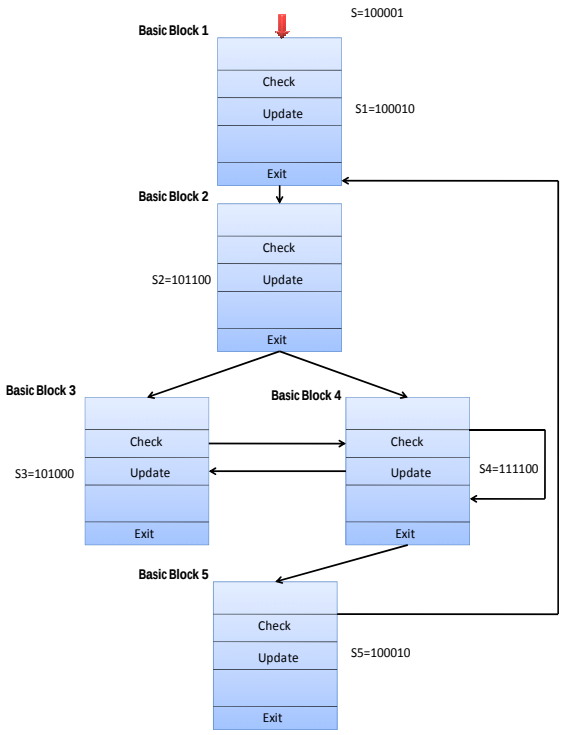
دستورالعمل بعدی، exit است که در هنگام خروج از بلوک پایه اجرا می‌شود و مقدار متغیر sel را به عددی که نشانگر بلوک پایه فعلی است، می‌رساند.

دو دستورالعمل ابتدایی را در وسط هر بلوک پایه قرار می‌دهیم، بدین ترتیب، درصدی از خطاهای ناشی از پرش داخلی غیرمجاز در یک بلوک پایه هم مشخص خواهند شد. پس از تشخیص خطا، سیگنال err برابر با یک می‌شود و کنترل برنامه به ابتدای دستورالعمل check در بلوک مبدأ بازگردانده می‌شود و اجرا دوباره از ابتدا آغاز می‌شود؛ لذا با احتمال زیادی بر اثر اجرای مجدد، اشکال تصحیح می‌شود. اگر عمل بازگرداندن به یک مکان مشخص از حد آستانه‌ای بیشتر شود، پیشنهاد می‌شود که کل سیستم بازنشانی گردد تا اشکال مذکور رفع شود. دستورالعمل exit را نیز در انتهای هر بلوک پایه قرار می‌دهیم. در شکل (٤) می‌توانیم پیاده‌سازی روش بیان‌شده را برای مرتب‌سازی حبابی مشاهده کنیم.

گراف کنترلی برنامهٔ مذکور نیز در شکل (۵) نشان داده شده است.

```
//basic block 1
for (int pass=1; pass< n; pass++) {
//basic block 2
        for (int i=0; i<n-pass; i++) {
                if (x[i]) > x[i+1]) {
//basic block 3
                int temp = x[i]; x[i]=x[i+1]; x[i+1] = temp;
                }
//basic block 4
        }
//basic block 5
```

شکل (٤): کد برنامهٔ نمونهٔ مرتب‌سازی حبابی و نحوهٔ تقسیم‌بندی آن به بلوک‌های پایه

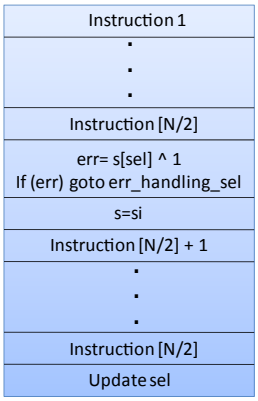


شکل (۵): گراف کنترلی برنامهٔ نمونهٔ مرتب‌سازی حبابی

در گراف کنترلی رسم‌شده در شکل (۵)، ارتباطات بین بلوک‌های پایه نشان داده شده است. طبق شکل، به هر بلوک پایه، یک امضا تخصیص داده شده که در آن، بلوک‌های پیشرو برای بلوک فعلی مشخص شده است. همچنین در این گراف،

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

ارتباطات میان بلوک‌های پایه و نحوهٔ تعامل آن‌ها مشخص شده است. ساختار داخلی یک بلوک پایه با افزودن دستورالعمل‌های اضافه‌شده، در شکل (٦) نشان داده شده است.



شکل (٦): ساختار داخلی یک بلوک پایه با افزودن دستورالعمل‌های اضافه‌شده

در ادامه، به بررسی قابلیت کشف خطای روش ارائه‌شده می‌پردازیم. با افزودن دستورالعمل‌های check, update می‌توان تمامی انواع اشکالات تکی ناشی از پرش‌های نادرست را کشف کرد. نیز اثبات‌هایی برای ادعای ذکرشده ارائه می‌شود.

۱. یک پرش از گره V_i به گره V_j ($V_j \in succ(V_i)$) که در شکل (۷) نشان داده شده است.

حالت اول: رخ دادن پرش غیر مجاز و ناخواسته از انتهای بلوک V_i به وسط بلوک V_j

در این حالت، زمانی که دستورالعمل check در بلوک V_j اجرا می‌شود، به دلیل به‌روز نشدن، مقدار متغیر sel در بلوک V_i، متغیر err مقدار یک را می‌گیرد و این خطا تشخیص داده می‌شود؛ برای مثال، فرض کنید در شکل (۷) از انتهای بلوک اول به وسط بلوک دوم پرشی ناخواسته داشته باشیم، در این حالت، به دلیل به‌روز نشدن، متغیر sel در انتهای بلوک اول، مقدار صفر را دارد. زمانی که به دستورهای check, update از بلوک پایه دوم می‌رسیم، بیت شمارهٔ صفر از متغیر S که در حال حاضر مقدار Signature-BB1 را دارد، با مقدار یک xor می‌شود که چون این بیت مقدار صفر را دارد، مقدار سیگنال err یک می‌گردد و خطا کشف می‌شود. در این حالت، خواهیم داشت:

S = Signature_BB1 = 100010
S [0] = 0
Err = S [0] ^ 1 = 0 ^ 1 = 1

حالت دوم: رخ دادن پرش غیرمجاز از وسط بلوک پایه V_i به ابتدای بلوک V_j

در این حالت، به دلیل اجرا نشدن دستورهای افزوده شده در بلوک V_i، متغیر sel به‌روز نمی‌شود و همانند حالت اول، خطا تشخیص داده خواهد شد؛ برای مثال فرض کنید در شکل (۷) از وسط بلوک اول به ابتدای بلوک دوم پرش ناخواسته‌ای داشته باشیم. مثال این وضعیت کاملاً شبیه حالت اول می‌باشد که متغیر sel در هنگام ورود به بلوک پایهٔ دوم، همان مقدار صفر را دارد و همانند مثال قبلی، اشکال در دستور check بلوک پایهٔ دوم تشخیص داده می‌شود.

حالت سوم: رخ دادن پرش غیرمجاز از وسط بلوک V_i به وسط بلوک V_j

در این حالت نیز مشابه حالت‌های گذشته به علت اجرا نشدن دستور exit، اشکال در دستورالعمل check بعدی تشخیص داده خواهد شد؛ برای مثال، فرض کنید در شکل (۷) از وسط بلوک اول به وسط بلوک دوم، پرش ناخواسته داشته باشیم، تشخیص خطا در این حالت کاملاً مشابه وضعیت‌های بیان‌شده در حالت اول و دوم است.

حالت چهارم: رخ دادن پرش غیرمجاز از انتهای بلوک V_i به ابتدا بلوک V_j

در این حالت هم مشابه حالت‌های گذشته به علت اجرا نشدن دستور exit، اشکال در دستورالعمل check بعدی تشخیص داده خواهد شد؛ برای مثال، فرض کنید در شکل (۷) از انتهای بلوک اول به ابتدای بلوک دوم پرش ناخواسته داشته باشیم، مثال این حالت نیز برای تشخیص خطا کاملاً مشابه حالت‌های پیشین است که توضیح داده شد.

۲. یک پرش از گره V_i به گره V_j که ($succ(V_i) \not\in V_j$) به صورت غیر مجاز رخ بدهد.

حالت پنجم: رخ دادن پرشی غیر مجاز و ناخواسته از انتهای بلوک V_i به ابتدای بلوک V_j؛ که شکل آن مشابه حالت چهارم است.

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

در این حالت هم مشابه سایر حالت‌ها اشکال به علت اجرا نشدن دستور exit تشخیص داده می‌شود؛ برای مثال، فرض کنید در شکل (۷) از انتهای بلوک اول به ابتدای بلوک سوم پرش داشته باشیم، آنگاه در دستور check بلوک سوم، بیت صفرم S بررسی می‌شود و چون sel به علت پرش ناخواسته به‌روز نشده و چون این بیت در S که همان S1 است، صفر می‌باشد، سیگنال err برابر یک می‌گردد و خطا تشخیص داده می‌شود.

حالت ششم: رخ دادن پرشی غیر مجاز و ناخواسته از انتهای بلوک V_i به وسط بلوک V_j؛ که شکل آن مشابه حالت اول است.

در این حالت هم مشابه سایر حالت‌ها به علت اجرا نشدن دستور exit، اشکال تشخیص داده می‌شود؛ برای مثال، فرض کنید در شکل (۷) از انتهای بلوک اول به وسط بلوک سوم پرش داشته باشیم، در این حالت کاملاً مشابه حالت ششم، سیگنال err در بلوک سوم برابر با یک می‌گردد و خطا تشخیص داده می‌شود.

حالت هفتم: رخ دادن پرشی غیر مجاز و ناخواسته از وسط بلوک V_i به ابتدای بلوک V_j؛ که شکل آن مشابه حالت دوم است.

در این حالت هم مشابه سایر حالت‌ها به علت اجرا نشدن دستور exit، اشکال تشخیص داده می‌شود؛ برای مثال، فرض کنید در شکل (۷) از وسط بلوک اول به ابتدای بلوک سوم پرش داشته باشیم، این حالت نیز کاملاً شبیه به دو حالت گذشته است که کاملاً توضیح داده شد.

حالت هشتم: رخ دادن پرشی غیر مجاز و ناخواسته از وسط بلوک V_i به وسط بلوک V_j؛ که شکل آن مشابه حالت سوم است.

در این حالت هم مشابه سایر حالت‌ها به علت اجرا نشدن دستور exit، اشکال تشخیص داده می‌شود؛ برای مثال، فرض کنید در شکل (۷) از وسط بلوک اول به وسط بلوک سوم پرش داشته باشیم، در دستور check بلوک سوم، بیت صفرم S بررسی می‌شود، چون sel به علت پرش ناخواسته به‌روز نشده و چون این بیت در S که همان S1 است، صفر می‌باشد، سیگنال err برابر یک می‌گردد و خطا تشخیص داده می‌شود.

علاوه بر حالت‌های شرح داده‌شده، روش ارائه‌شده قابلیت کشف برخی از خطاهایی را که به علت پرش غیرمجاز در داخل یک بلوک پایه رخ می‌دهند، نیز دارد.

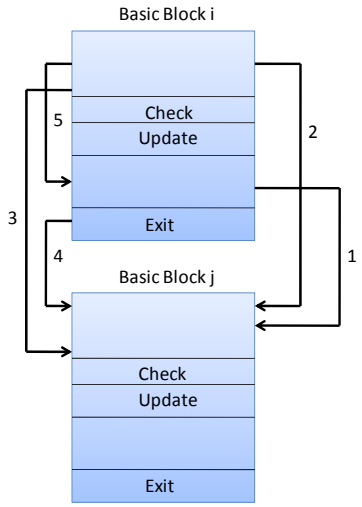
۳. زمانی یک پرش ناخواسته از یک دستورالعمل در بلوک پایه V_i به دستورالعمل دیگر در همان بلوک پایه رخ بدهد. شکل این حالت در شکل (۷) با فلش ۵ نشان داده شده است.

حالت نهم: رخ دادن پرش ناخواسته از دستورالعمل‌های قبل از check, update به دستورالعمل‌های بعد از آن‌ها

در این حالت، چون متغیر S در طی اجرا به‌روزرسانی نشده است، اشکال در دستورالعمل check بلوک بعدی تشخیص داده خواهد شد؛ برای مثال، فرض کنید پرشی ناخواسته از قبل از دستورالعمل check در بلوک اول به بعد از دستورالعمل update در همان بلوک رخ بدهد، در این حالت به علت اجرا نشدن دستور update، متغیر S مقدار اولیه خود را که همان ۱۰۰۰۰۱ است، دارد. در انتهای بلوک اول، متغیر sel به مقدار یک به‌روزرسانی می‌شود و برنامه وارد بلوک پایهٔ دوم می‌شود. در بلوک پایهٔ دوم، در دستورالعمل check بیت اول از متغیر S با مقدار sel در آن لحظه، xor می‌شود و حاصل برابر یک می‌شود که منجر به تشخیص خطای رخ داده شده می‌گردد:

$S = S_{initial} = 100001$

$err = S[sel] \wedge 1 = 0 \wedge 1 = 1$



شکل (۷): پرش‌های غیر مجاز قابل تشخیص توسط روش ارائه شده در حالت $V_j \in succ(V_i)$

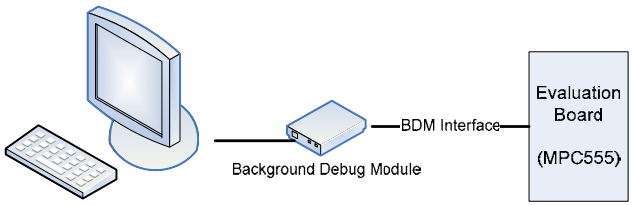
Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

٤. نتایج تجربی

٤ـ١. محیط تست

برای ارزیابی روش پیشنهادی از زیرساختار نشان داده شده در شکل (٨) استفاده شده که شامل المان‌های زیر می‌باشد:

- ✓ یک ماژول Background Debug Mode (BDM) که هم برای برنامه‌ریزی و هم برای اشکال‌زدایی استفاده می‌شود و می‌توان همانند [٤٩] از آن برای تزریق اشکال نیز استفاده کرد.
- ✓ بورد توسعهٔ phyCORE-MPC555
- ✓ یک کامپیوتر شخصی



شکل (٨): ساختار مکانیزم تزریق اشکال با استفاده از BDM

برای تزریق اشکال نیز از روش‌های مختلفی استفاده شده است که عبارت‌اند از:

- ✓ تزریق اشکال مستقیم بر روی رجیسترهای پردازنده با استفاده از ماژول BDM
- ✓ اعمال دستورات پرشی (JMP، JL، JG، JNE، JLE، JGE، CALL و RET)
- ✓ تغییر دستورات پرشی

عملیات تزریق اشکال بر روی چهار برنامهٔ محک Bubble Sort(BS)، Quick Sort(QS)، Matrix Multiplication(MM) 40×40 و Fast Fourier Transform(FFT) انجام می‌شود که برروی آن‌ها تعداد ۵۰۰۰ اشکال تزریق می‌شود.

از آنجا که در روش تزریق اشکال مستقیم بر روی رجیسترهای پردازنده با استفاده از BDM، می‌توان رجیسترهای پردازنده را به طور مستقیم دستکاری کرد، راه حلی است که سرعت و قابلیت مشاهدهٔ بسیار بیشتری دارد و به واقعیت بسیار نزدیک‌تر است؛ برای مثال، در این روش می‌توان به طور مستقیم رجیستر PC را دستکاری کرد. همان‌گونه که در [٤٩] نشان داده شده، با دستکاری رجیسترها، احتمال رخداد استثنا (Exception) بیش از ٧٦٪ است، لذا این روش، اگرچه به واقعیت بسیار نزدیک است، امکان تست دقیق روش تشخیص خطا را فراهم نمی‌کند؛ بنابراین، جهت ارزیابی روش تشخیص خطا از روش‌های دوم و سوم استفاده شده است.

٤ـ٢. نتایج آزمایشگاهی

در ارزیابی روش پیشنهادی از سه روش تزریق اشکال پیشنهاد شده در بخش ٤ـ١ استفاده شده است. نتایج حاصل از تزریق اشکال را می‌توان در جدول (١) مشاهده کرد. روش RSCFC دارای مشکل سربار کارآیی و حافظه نیز می‌باشد، زیرا همان‌گونه که شرح داده شد، در این روش دو متغیر n+1 بیتی که n تعداد بلوک‌های پایه است، تعریف می‌شود و از سوی دیگر برای کشف خطاهای روند اجرا، هفت دستور به هر بلوک پایه می‌افزاید. امروزه حجم هر بلوک پایه به طور متوسط در حدود ٣ تا ٥ دستورالعمل می‌باشد و افزودن ٧ دستور اضافه و دو متغیر n+1 بیتی به هر بلوک پایه، منطقی به نظر نمی‌رسد.

در ایدهٔ ارائه‌شده در این مقاله، تنها ٣ دستورالعمل به هر بلوک پایه افزوده می‌شود و یک متغیر n+1 بیتی هم به عنوان امضا و یک متغیر تک‌بیتی به نام sel برای انجام عملیات ذخیره می‌گردد. واضح است که سربار کارآیی و حافظهٔ روش ارائه‌شده در این مقاله، از روش RSCFC بسیار کمتر است. در [٣٦]، سربار کارآیی و حافظهٔ روش RSCFC با روش‌های CFCSS و ECCA مقایسه و ادعا شده که سربار کمتری در RSCFC مصرف می‌شود. به طور منطقی نتیجه می‌شود که سربار روش ارائه‌شده در این مقاله، از سه روش دیگر کمتر است.

توسط مقایسهٔ سایز و سرعت اجرای برنامه‌های مقاوم‌شده در مقایسه با حالت عادی، نتایج جدول (١) به دست آمده است که نشان می‌دهد روش ما دارای سربار کارآیی به میزان ٤٨.٢٪ و سربار حافظه در حدود ٥٢.٦٧٪ کمتر از روش‌های CFCSS، ECCA و RSCFC می‌باشد. پس از تزریق خطاهای روند اجرا، یکی از پنج حالت زیر رخ خواهد داد که در جدول (١) نیز نشان داده شده است.

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

- **خروجی صحیح**: برنامه با وجود اشکال تزریق‌شده به مشکلی برخورد نمی‌کند و خروجی نهایی برنامهٔ تولید شده صحیح می‌باشد.
- **مشکلات سیستم‌عاملی**: اشکال تزریق‌شده به برنامه به علت قابلیت‌ها و امکانات سیستم‌عامل برنامه مانند استثنا و... کشف شده است.
- **تمام شدن مهلت زمانی**: اشکال تزریق‌شده منجر به تشکیل یک حلقه با زمان اجرای زیاد در سیستم شده است و به علت طولانی‌تر شدن زمان اجرا از مقدار مورد انتظار، خطا کشف خواهد شد.
- **کشف خطا**: اشکال تزریق‌شده به علت مکانیزم کشف خطای اعمال‌شده به نرم‌افزار در حال اجرا کشف خواهد شد.
- **خروجی ناصحیح**: اشکال تزریق‌شده به برنامه کشف نشده و به خروجی برنامه انتشار یافته و منجر به تولید خروجی ناصحیح شده است.

جدول (۲) نشان می‌دهد که این روش، علاوه بر پوشش اشکال بهتر، دارای سربار کارآیی و حافظهٔ بسیار بهتری نسبت به روش‌های پیاده‌سازی شدهٔ دیگر است و به این علت، استفاده از آن بسیار مناسب‌تر است. به منظور محاسبهٔ سربار حافظهٔ مصرفی و کارآیی، حجم حافظهٔ برنامهٔ مقاوم‌شده در برابر خطاهای روند اجرا به حجم برنامهٔ اصلی تقسیم می‌شود و نسبت سربار محاسبه می‌گردد. از سوی دیگر، سربار کارآیی از طریق مقایسهٔ سرعت اجرای برنامه‌ها به دست می‌آید.

جدول (۱): نتایج تزریق اشکال در مقایسه با روش‌های CFCSS، ECCA و RSCFC

	خروجی صحیح	سیستم عامل	خروجی ناصحیح	تمام شدن مهلت	کشف خطا
BS-CFCSS	٦٥.٦٥٪	٥.٢٪	١.٨٪	٢.٨٧٪	٢٢.٨٪
QS-CFCSS	٥٣.٧٨٪	٤.٤٨٪	٦.٢٪	١.٥٨٪	٣٣.٣٢٪
MM-CFCSS	٣٥.٥٪	١٢.٨٧٪	١٣.١٤٪	٤.٢٣٪	٢٣.٣٤٪
FFT-CFCSS	٢٦٪	٨.٢٪	٥.٤٪	٧.٩٦٪	٤٧.٣١٪
BS-ECCA	٥٨.٧٪	١١٪	٨.٢٥٪	٨.١٪	١٩.٦٪
QS-ECCA	٥١.٤٣٪	٧.٦٪	٦.٢٪	٣.٥٦٪	٣٢.١٤٪
MM-ECCA	٤٢.٧٪	١١.٢٪	٩.٢٪	٥.٨٧٪	٢٩.١٤٪
FFT-ECCA	٤٣.٧٪	٦.٦٦٪	٥٪	٤.٣٪	٣٩.٨٪
BS-RSCFC	٦٢.٣٪	٥٪	٦.٦٪	٢.٥٪	٢٢.٥٪
QS-RSFCF	٤٢.٦٪	١٢.٧٦٪	٦.٧٦٪	٣.٦٩٪	٣٢.٣٪
MM-RSCFC	٤٦.٢٪	١٢.٣٤٪	٩.٧٪	٢.٥٪	٣٤.٥٪
FFT-RSCFC	٣٢.٧٥٪	١٧.٦٥٪	٩.٢٪	٣.٥٦٪	٢٤.٥٪
BS-I2BCFC [٥٠]	٣٦.٢٣٪	١٦.٩٪	١٣٪	٢.٧٪	٢٩.٥٥٪
QS-I2BCFC	٤٧.٨٪	٤٪	٣.٣٪	٦٪	٣٨.٨٪
MM-I2BCFC	٢٧.٢٪	٦.٧٪	٥٪	١.٣٤٪	٥٧.١٤٪
FFT-I2BCFC	٣٥.٢٪	٧.٢٪	١٠٪	٣٪	٤٤.٧٪

جدول (۲): مقایسهٔ سربار حافظه و کارآیی روش I2BCFC با روش‌های CFCSS، ECCA و RSCFC

	Memory Overhead				Performance Overhead			
Program	**CFCSS**	**ECCA**	**RSCFC**	**I2BCFC**	**CFCSS**	**ECCA**	**RSCFC**	**I2BCFC**
BS	١.٣٢	١.٥٤	١.٤٥	١.١٢	١.٨٦	٦.٨٧	١.٧٤	١.٥٤
QS	١.٣٣	١.٣٢	١.٣٨	١.١١	١.٣٣	١.٨٦	١.٢١	١.٠٣
MM	١.١٢	١.١٦	١.١٣	١.٠٥	١.٥٦	٣.٠٢	١.٤٣	١.٢٦
FFT	١.٢٣	١.٦	١.٣٤	١.١٩	١.٣٤	٢.٨٦	١.٤٣	١.١٥

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

۵. نتیجه‌گیری

استفاده از تجهیزات تجاری رایج، یکی از گزینه‌های مناسب برای استفاده در طیف وسیعی از کاربردها از جمله کاربردهای فضایی است. با وجود این، بدون در نظر گرفتن تمهیدات افزونگی مناسب در سطوح مختلف (سخت‌افزار، نرم‌افزار، زمان و اطلاعات) نمی‌توان از این تجهیزات در کاربردهای فضایی که با مخاطرات زیادی روبه‌رو هستند، استفاده کرد. در این مقاله، روشی بر مبنای افزونگی نرم‌افزاری در سطح دستورالعمل ارائه شد که نسبت به روش‌های ارائه شدۀ قبلی، دارای کارآیی بهتر و سربار حافظۀ کمتری است. با استفاده از این روش می‌توان درصد زیادی از اشکال‌های تکی گذرایی را که منجر به خطاهای روند اجرا می‌شوند، تشخیص داد.

■

مراجع

[1] Rajabzadeh, G. Miremadi and M. Mohandespour, *Error detection enhancement in COTS superscalar processors with performance monitoring features*, Journal of Electronic Testing: Theory Application (JETTA), 20(5), pp. 553–67, 2004.

[2] Rajabzadeh and G. Miremadi, *Transient detection in COTS processors using software approach*, Journal of Microelectron Reliability, 46(1), pp. 124-133, 2006.

[3] J. Srinivasan and K. Lundqvist, *Real-time architecture analysis: a COTS perspective*, In Proceedings of the 21th digital avionics systems, Vol. 1, pp. 5D4-1–9, 2002.

[4] Y. He and A. Avizienis, *Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study*, In Proceedings of the international conference on dependable systems and networks (DSN- 2000), pp. 81–6, June 2000.

[5] CD. Gill, RK. Cytron and DC. Schmidt, *Multiparadigm scheduling for distributed real-time embedded computing*, In Proceeding of IEEE; 91(1), pp.183–97, 2003.

[6] N. Oh, PP. Shirvani and EJ. McCluskey, *Error detection by duplicated instructions in super-scalar processors*, IEEE Transaction on Reliability; 51(1), pp. 63–75, 2002.

[7] P. Chevochot and I. Puaut, *Experimental evaluation of the failsilent behavior of a distributed real-time run-time support built from COTS components*, In Proceedings of the international conference on dependable systems and networks (DSN-2001), pp. 304–13, July 2001.

[8] SV. Pizzica, *Meeting military system test requirements with the usage of COTS products*, In Proceedings of the 17th AIAA/IEEE/SAE digital avionics systems conference (DASC), November 1998, Vol. 1, pp. B45/1–7.

[9] E. Trujillo, *Military requirements constrain COTS utilization*, In Proceedings of the 14th digital avionics systems conference (DASC), pp. 112–7, November 1995.

[10] P. Tso and P. Galaviz, *Improved aircraft readiness through COTS*, In IEEE systems readiness technology conference (AUTOTESTCON_99), pp. 451–6, September 1999.

[11] Newton, *Design assurance for airborne COTS hardware*, In IEE colloquium on COTS and safety critical systems (Digest No. 1997/013), pp. 4/1–3, January 1997.

[12] J. Profeta, N. Andrianos, Yu. Bing, B. Johnson, T. DeLong and D. Guaspart, Safety-critical systems built with COTS, Computer, 29(11), pp. 54–60, 1996.

[13] M. Pignol, *COTS-based Applications in Space Avionics*, 978-3-9810801-6-2/DATE10 © 2010 EDAA.

[14] Aicardi, P. Lay, A. Mouton, C. Revellat, D. Beauvallet and G. Lemarchand, et al., *Guidelines for commercial parts management*, In Proceeding of European Space Components Conference (ESCCON), pp. 185-188, 2002.

[15] S. Provost, M. Le Roy, B. Mamdy, G. Flandin and T. Paulsen, *GAIA video processing embedded algorithms: prototyping and validation activities*, In Proceeding of Eurospace DAta Systems in Aerospace Conf. (DASIA), 2007.

[16] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios and R. Heckmann, *Computing the worst case execution time of an avionics program by abstract interpretation*, In Proceeding of 5th International

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

Workshop on Worst- Case Execution Time (WCET) Analysis, pp. 21-24, 2005.

[17] K. Whisnant, R. Some and D.A. Rennels, et al., *An experimental evaluation of the REE SIFT environment for spaceborne applications*, In Proceeding of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 585-594, 2002.

[18] M. Rebaudengo, M. Sonza Reorda and M. Violante, *A new softwarebased technique for low-cost fault-tolerant application*, In Proceeding of IEEE Reliability and Maintainability Symposium, pp. 25-28, 2003.

[19] M.N. Lovelette, P.P. Shirvani and E.J. McCluskey, et al., *Strategies for fault-tolerant, space-based computing: lessons learned from the ARGOS testbed*, In Proceeding of IEEE Aerospace Conference, Vol. 5, pp. 2109-2119, 2002.

[20] Czajkowski and M. McCartha, *Ultra low-power space computer leveraging embedded SEU mitigation*, In Proceeding of IEEE Aerospace Conference, Vol. 5, pp. 2315-2328, 2003.

[21] P. Behr, W. Bärwald, K. Briess and S. Montenegro, *Fault tolerance and COTS: next generation of high performance satellite computers*, DAta Systems in Aerospace Conference, DASIA, session 14A, 2003.

[22] Guidal and P. David, *Development of a fault tolerant computer system for the HERMES space shuttle*, In Proceeeding of 23th IEEE Fault-Tolerant Computing Symposium (FTCS), 1993.

[23] H. Kanai, K. Hama, M. Akiyama and N. Natsuka, *Overview of SERVIS project toward application of commercial technology for space*, In Proceeding of 56th International Astronautical Congress, IAC-05-D1.2.09, 2005.

[24] H. Hihara, K. Yamada, M. Adachi, K. Mitani, M. Akiyama and K. Hama, *CRAFT: an experimental fault tolerant computer for SERVIS-2 satellite*, In Proceeding of 21th AIAA Int. Communications Satellite Systems Conference and Exhibit, AIAA 2003-2291, 2003.

[25] R. DeCoursey, R. Melton and R. Estes, *Non radiation hardened microprocessors in space-based remote sensing systems*, In Proceeding SPIE Sensors, Systems, and Next-Generation Satellites X, Vol. 6361, 2006.

[26] R. Hillman, G. Swift, P. Layton, M. Conrad, C. Thibodeau and F. Irom, *Space processor radiation mitigation and validation techniques for an 1,800 MIPS processor board*, In Proceeding of 12th RADECS Association / ESA / IEEE European Conf. on Radiations and its Effects on Components and Systems (RADECS), pp. 347-352, 2003.

[27] M. Pignol, *Methodology and tools developed for validation of COTSbased fault-tolerant spacecraft supercomputers*, In Proceeding of 13th IEEE International On- Line Testing Symp. (IOLTS), pp. 85-92, 2007.

[28] V. Pouget, P. Fouillat, D. Lewis, H. Lapuyade, L. Sarger, F.M. Roche, S. Duzellier and R. Ecoffet, *An overview of the applications of a pulsed laser system for SEU testing*, In Proceeding of 6th IEEE Int. On-Line Testing Symp. (IOLTS), pp. 52-57, 2000.

[29] B.S. Smith and J. Hengemihle, *The Small Explorer Data System, a data system based on standard interfaces*, In Proceeding of AIAA/NASA 2nd Int. Symposium on Space Information Systems, 1990.

[30] A. Kellner and H.-J. Kolinowitz, and G. Urban, *A novel approach to fault tolerant computing*, In Proceeding of IEEE Aerospace Conference, Vol. 3, pp. 1127- 1131, 2001.

[31] Space Math-III, S. Odenwald and C. James, NASA, *Hinode satellite program's Education and Public Outreach Project*, 2007.

[32] M.Jafari-Nodoushan, G.Miremadi and A.Ejlali, *Control-Flow Checking Using Branch Instructions*, In Proceeding of the 8th International Conference on Embedded and Ubiquitous Computing, 2008.

[33] Zhu and H. Aydin, *Reliability Effects of Process and Thread Redundancy on Chip Multiprocessors*, In Proceeding of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2006.

[34] P.Kongetira, K.Aingaran and K.Olukotun, *Niagara: A 32-Way Multithreaded Sparc Processor*, IEEE Micro, 2005.

[35] A. Rajabzadeh and G. Miremadi, *CFCET: A hardware based control flow checking technique in COTS processors using execution training*, Elsevier journal on Computer Microelectronics and Reliability, 46(5-6), pp. 959-972, May 2006.

[36] M. J. Gadlage, R. D. Schrimpf, B. Narasimham, J. A. Pellish, K. M. Warren, R. A. Reed, R. A. Weller, B. L. Bhuva, L. W. Massengill and X. Zhu, *Assessing Alpha Particle-Induced Single Event Transient Vulnerability in a 90-nm CMOS Technology*, IEEE ELECTRON DEVICE LETTERS, 29(6), pp. 638-640, June 2008.

[37] J. A. Felix, J. R. Schwank, M. R. Shaneyfelt, J. Baggio, P. Paillet, V. Ferlet-Cavrois, P. E. Dodd, S. Girard and

Downloaded from scj.kashanu.ac.ir at 2:26 +0330 on Sunday January 21st 2018

E. W. Blackmore, *Test Procedures for Proton-Induced Single Event Latchup in Space Environments*, IEEE TRANSACTIONS ON NUCLEAR SCIENCE, 55(4), pp. 2161-2165, August 2008.

[38] K. Kruckmeyer, R. L. Rennie and V. Ramachandran, *Use of Code Error and Beat Frequency Test Method to Identify Single Event Upset Sensitive Circuits in a 1 GHz Analog to Digital Converter*, IEEE TRANSACTIONS ON NUCLEAR SCIENCE, 55(4), pp. 2013-1018, August 2008.

[39] M. R. Shaneyfelt, J. R. Schwank, P. E. Dodd and J. A. Felix, *Total Ionizing Dose and Single Event Effects Hardness Assurance Qualification Issues for Microelectronics*, IEEE TRANSACTIONS ON NUCLEAR SCIENCE, 55(4), pp. 1926-1946, August 2008.

[40] J. A. Maestro and P. Reviriego, *Reliability of Single-Error Correction Protected Memories*, IEEE TRANSACTIONS ON RELIABILITY, 58(1), pp. 193-201, March 2009.

[41] A. M. Saleh, J. J. Serrano and J. H. Patel, *Reliability of scrubbing recovery- techniques for memory systems*, IEEE TRANSACTION ON RELIABILITY, 39(1), pp. 114–122, April 1990.

[42] L. Petersen, *Single-Event Data Analysis*, IEEE TRANSACTIONS ON NUCLEAR SCIENCE, 55(6), pp. 2819-2841, December 2008.

[43] Li and B. Hong, *On-line control flow error detection using relationship signatures among basic blocks*, Computers and Electrical Engineering, 36 (1), pp. 132–141, 2010.

[44] Nahmsuk Oh, Philip P. Shirvani, and Edvard J. McClusky, *Control Flow Checking By Software Signature*, IEEE TRANSACTION ON RELIABILITY, 55(6), pp. 111-122, 2002.

[45] Ramtilak Vemu and Jacob A. Abraham, *CEDA: Control-flow Error Detection through Assertions*, Proceedings of the 12th IEEE International On-Line Testing Symposium (IOLTS'06), 2006.

[46] H. Zarandi, M. Maghsoudloo and N. Khoshavi, *Two Efficient Techniques to Detect and Correct Control Flow errors*, In proceeding of 16th IEEE pacific Rim International Symposium on Dependable Computing (PRDC), 2010.

[47] Li. Jianli, T. Qingping and Xu. Jianjun, *A Software-Implemented Configurable Control Flow Checking Method*, *In Procedding of International Symposium on Parallel Archirectures*, Algorithmes and Programming (PAAP), 2010.

[48] N. Oh and S. Mitra, *ED4I: Error Detection by Diverse Data and Duplicated Instructions*, IEEE TRANSACTION ON COMPUTERS, 51(2), pp. 180-199, 2002.

[49] S. A. Asghari, H. Pedram, H. Taheri and M Khademi, *A New Background Debug Module Based Technique for Fault Injection in Embedded Systems*, International Review on Modeling and Simulation (IREMOS), 3(3), pp. 415-422, 2010.

[۵۰] اصغری، سید امیر، آتنا عبدی، حسن طاهری، حسین پدرام و سعادت پورمظفری، **ارائهٔ یک روش مبتنی بر افزونگی نرم‌افزاری سطح دستورالعمل جهت تشخیص خطاهای روند اجرای برنامهٔ درون و بین بلوکی**، بیستمین کنفرانس مهندسی برق، تهران، ۱۳۹۱.