JSST
فصلنامهٔ علمی - پژوهشی علوم و فناوری فضایی

جلد ۵ / شمارهٔ ۴ / زمستان ۱۳۹۱
ص. ص ۱۸- ۹

بررسی فنون نرم‌افزاری تحمل‌پذیری خطای گذرا در نرم‌افزارهای ماهواره

طاهره برومندنژاد[۱]، محمد عبداللهی ازگمی[۲*] و شاهرخ جلیلیان[۳]

۱ و ۲- دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت ایران

۳- پژوهشکده تحقیقات فضایی ایران

تهران، نارمک، خ فرجام

azgomi@iust.ac.ir

نرم‌افزار ماهواره از جمله کاربردهایی است که با توجه به محیط عملیاتی پرتشعشع، در معرض انوع خطاهای گذرا یا اشکال‌های نرم قرار دارد. این نوع خطاها باعث وقوع اشکال در اجرای نرم‌افزارهای ماهواره می‌شوند. طراحان ماهواره با استفاده از روش‌های حفاظت سخت‌افزاری (شیلدینگ) و طراحی مبتنی بر قطعات مقاوم، به مقابله با این خطاها می‌پردازند. اما عیب این روش‌ها به‌طور کلی افزایش هزینه، وزن، مصرف توان الکتریکی و کاهش کارایی است. تحقیقات اخیر نشان می‌دهد که می‌توان در ماهواره‌های LEO ضمن استفاده از قطعات تجاری (COTS) به‌جای قطعات فضایی خاص، با فنون نرم‌افزاری به مقابله با خطاهای گذرا پرداخت. در این مقاله، ضمن بررسی شاخه‌ای از این فنون، موسوم به فنون مبتنی بر جریان کنترل، روش کارآمدی انتخاب شده و نحوهٔ پیاده‌سازی، آزمون و ارزیابی آن ارائه داده شده است.

واژه‌های کلیدی: خطای گذرا (نرم)، جریان‌کنترل، تحمل‌پذیری خطا، ماهواره‌های LEO

علائم و اختصارات

ACFC	Assertion for Control Flow Checking
ASM	Autonomous Signature Monitoring
CFCSS	Control Flow Checking by Software Signature
CFE	Control Flow Error
CFEDA	Control-Flow Error Detection trough Assertions
COTS	Commercial off-the-Shelf
ECCA	Enhanced Control-flow Checking Assertion
EDAC	Error Detection and Correction
EDDI	Error Detection by Duplicated Instructions
ESM	Embedded Signature Monitoring
RSCFC	Relationship Signatures for Control Flow Checking
SIFT	Software Implemented Fault Tolerance
SIHFT	Software Implemented Hardware Fault Tolerance
SOR	Sphere of Redundancy
SWIFT	Software Implemented Fault Tolerance

مقدمه

نرم‌افزارهای توکار[۴] به‌طور گسترده‌ای در کاربردهای مختلف، مانند ماهواره‌ها، فرودگاه‌ها و سیستم‌های خودکار به‌کار گرفته می‌شوند. در چنین کاربردهایی، خرابی نرم‌افزار ممکن است زندگی انسان‌ها و سرمایه‌های مهم را به خطر بیندازد یا باعث از بین رفتن اطلاعات مهم شود [۲ و ۱]. در این سیستم‌ها خصوصیات غیرکارکردی، مانند قابلیت اطمینان، ایمنی و دسترس‌پذیری اهمیت زیادی دارند.

رفتار خطادار این سیستم‌ها، ممکن است باعث حوادث ناگوار شود. به همین دلیل این سیستم‌ها باید طوری طراحی شوند که قابلیت تشخیص خطا را در کمترین زمان ممکن داشته باشند. خطاها

۱. کارشناس ارشد

۲. استادیار (نویسنده مخاطب)

۳. کارشناس ارشد

4. Embedded software

دریافت مقاله: ۸۹/۰۸/۱۸ ، تأیید مقاله: ۹۱/۰۸/۳۰

Archive of SID
www.SID.ir

به سه دسته خطاهای دائم[۵]، گذرا[۶] و متناوب[۷] تقسیم می‌شوند. خطاهای دائم نتیجهٔ اشتباه طراح در طراحی سخت‌افزار و اجزای نرم‌افزاری است. اما منبع خطاهای گذرا و متناوب، اختلالات و تأثیرات محیطی است [۳].

خطاهای گذرا یا اشکال‌های نرم[۸] در نرم‌افزار به کرّات نه تنها در سیستم‌های الکترونیکی- فضایی بلکه روی زمین و سطح دریا هم رخ داده است. منشأ عمدهٔ این نوع خطاها، تشعشعات ناشی از ذرات آلفا و پرتوهای کیهانی است که باعث وقوع اشکال‌های گذرا و منجر به اجرای غلط دستورالعمل شده یا بیت‌هایی را در حافظه تغییر می‌دهند. این نوع خطاها مشابه خطاهای دائم، می‌توانند بر جریان اجرایی برنامه تأثیرگذار باشند، وضعیت سیستم را تغییر دهند یا باعث تغییر داده شوند. اشکال‌های نرم باعث خرابی‌های زیادی در سیستم‌های ایمنی- بحرانی می‌شوند. برای مثال در سال ۲۰۰۰، شرکت سان[۹] کشف کرد که پرتوهای کیهانی روی حافظه‌های نهان سرویس‌دهنده‌های این شرکت اختلال ایجاد کرده و باعث توقف سرویس‌های وب‌گاه‌هایی مانند ایبی[۱۰] و آن‌لاین امریکن[۱۱] شده است [۴].

روش‌های سخت‌افزاری، نرم‌افزاری و تلفیقی (هیبرید) فنونی هستند که برای تشخیص خطاهای گذرا استفاده می‌شوند. یک روش مقابله با خطاهای گذرا استفاده از قطعات مقاوم در برابر تشعشع و افزونگی‌های سخت‌افزاری است. اما عیب این روش کاهش کارایی، عدم امکان استفاده از قطعات با فناوری روز، افزایش وزن، مصرف توان و هزینه است [۶ و ۵].

روش دیگر، استفاده از یک نوع تحمل‌پذیری خطای نرم‌افزاری است که به SIHFT[۱۲] معروف است. فنونین روش دو دسته‌اند که با خطاهای حافظه و پردازنده مقابله می‌کنند. برای نمونه EDAC[۱۳] روشی نرم‌افزاری است که وقوع خطا در حافظه را تشخیص داده و آن را بازیابی می‌کند. روش دیگر EDDI[۱۴] است که برای مقابله با خطاهای تأثیرگذار بر پردازنده استفاده می‌شود [۷].

در روش‌های تلفیقی بهترین فنون نرم‌افزاری، مانند SWIFT[۱۵]، به‌منظور ایجاد توازن بین سربار و هزینه با فنون سخت‌افزاری ترکیب می‌شوند. CRAFT و مقاوم‌سازی برخط، نمونه‌هایی از این روش‌ها هستند [۹ و ۸].

احتمال رخداد خطای گذرا نسبت به خطای دائم در محیط‌های فضایی بیشتر است (۱۰ تا ۳۰ برابر [۱۰]) . اثر این خطا در اجزای سخت‌افزاری، مانند شمارندهٔ برنامه، مدارهای آدرس و عناصر حافظه، منجر به خطای جریان کنترل CFE[۱۶] می‌شود. طبق گزارش‌های ارائه شده، بیش از ۷۰ درصد خطاهای گذرا منجر به خطای جریان کنترل در اجرای برنامه می‌شوند. به همین دلیل، بررسی روش‌های مبتنی بر جریان کنترل و ارائهٔ فنونی به‌منظور بهینه‌سازی آنها از لحاظ کارآیی و حافظه اهمیت ویژه‌ای دارد [۶].

در این مقاله، ضمن بررسی فنون تحمل خطاهای گذرا مبتنی بر جریان کنترل، روش کارآمدی انتخاب شده و نحوهٔ پیاده‌سازی، آزمون و ارزیابی آن شرح داده می‌شود.

ساختار ادامهٔ این مقاله به این صورت است که در بخش دوم، کارهای مرتبط در زمینهٔ کشف خطا را معرفی می‌کنیم. در بخش سوم، به بررسی روش‌های مبتنی بر جریان کنترل و مقایسهٔ آنها می‌پردازیم. در بخش سوم، بهترین روش مبتنی بر جریان کنترل تحت عنوان روش بررسی جریان کنترل امضاء رابطه را توضیح داده و در بخش چهارم به پیاده‌سازی روش پرداخته می‌شود. بخش آخر نتیجه‌گیری حاصل از پژوهش است.

کارهای مرتبط

تاکنون روش‌های متنوعی براساس سخت‌افزار، نرم‌افزار و ترکیب آنها به‌منظور کشف خطا ارائه شده است که به بعضی از آنها در ادامه اشاره می‌نماییم.

دو روش مبتنی بر سخت‌افزار یکی Lock Stepping [۱۱] و دیگری پردازنده مراقب [۱۲] هستند که از ماجول‌های خاص سخت‌افزاری برای کشف خطا استفاده می‌کنند. Lock Stepping روشی است که در پردازنده Compaq Non-Stop Himalaya استفاده می‌شود و محاسبات یکسان را بر روی دو پردازنده انجام می‌دهد و در هر سیکل زمانی نتایج را با هم مقایسه می‌کند. این روش تمام خطاهای موجود در بخش داده را کشف می‌کند. همچنین، پردازنده مراقب [۱۲] بر جریان کنترل برنامه و نحوهٔ دستیابی به حافظه نظارت دارد. با به‌کارگیری سه نوع عملیات کنترل دستیابی به حافظه، بررسی سازگاری محتوای متغیرها و بررسی جریان کنترل برنامه، رفتار پردازندهٔ اصلی را که مجری کد برنامه است، تحت نظر قرار می‌گیرد.

5. Permanent
6. Transient
7. Intermittent
8. Soft Errors
9. Sun
10. eBay
11. American online
12.Software Implemented Hardware Fault Tolerance (SIHFT)
13Error Detection and Correction (EDAC)
14 Error Detection by Duplicated Instructions (EDDI)
15. Software Implemented Fault Tolerance
16. Control Flow Error (CFE)

Archive of SID
www.SID.ir

در *کنترل دستیابی به حافظه* [۱۳]، پردازندهٔ اصلی نظارت بر دست‌یابی به حافظهٔ اصلی را بر عهده دارد و زمانی‌که برنامه دسترسی غیرقابل انتظاری را انجام می‌دهد سیگنال خطا فعال می‌شود. بررسی سازگاری محتوای متغیرها، بر محتوا و مقدار متغیرها نظارت دارد و وجود خطا در محتوای آنها را گزارش می‌دهد. در بررسی جریان کنترل برنامه، بر اساس این‌که هر انشعاب مطابق با نمودار نرم‌افزار در حال اجرا بر روی پردازندهٔ اصلی است یا نه، جریان اجرایی برنامه کنترل می‌شود.

بر اساس بررسی جریان کنترل برنامه، دو نوع پردازندهٔ مراقب وجود دارد. پردازنده مراقب فعال[۱۷]، این پردازنده به موازات برنامه اصلی اجرا می‌شود و در طی اجرای برنامه، پردازندهٔ مراقب به طور مداوم جریان کنترلی را استنتاج می‌کند و آن را با جریان کنترلی برنامه در حال اجرا در پردازندهٔ اصلی مقایسه می‌کند. این روش کمترین میزان سربار را بر روی برنامهٔ اجرایی دارد. پردازندهٔ مراقب غیرفعال[۱۸]، برنامه‌ای را اجرا نمی‌کند بلکه امضاءهایی را با مشاهدهٔ گذرگاه پردازندهٔ اصلی محاسبه می‌کند تا هر زمان که برنامهٔ اصلی وارد بلاک پایه و خارج از آن می‌شود، بررسی سازگاری را انجام دهد. پردازنده مراقب غیرفعال ساده‌تر از پردازنده مراقب فعال است، اما به دلیل دستورات ارتباطاتی مورد نیاز، سربار ایجاد می‌کند [۱۲].

بعضی از انواع خطاهای گذرا وجود دارند که باعث خرابی داده به‌صورت پنهان می‌شوند که به این نوع خطا، SDC[۱۹] گویند. برای کشف این نوع از خطاها از فنون تلفیقی استفاده می‌شود. برای مثال SWIFT مستعد خطاهایی است که در فاصلهٔ زمانی بین معتبرسازی و استفاده از مقدار ثبات رخ می‌دهد. اگر این داده در یک دستور STORE استفاده شود، خطا در این دورهٔ زمانی باعث خطای SDC می‌شود. به‌منظور حفظ داده‌ای که در حافظه ذخیره می‌شود، روش CRAFT:CSB ارائه شده است که دستورات ذخیره‌سازی را تکرار می‌کند و یک بیت اضافی برای هر دستور ذخیره‌سازی در نظر می‌گیرد که مقدار آن نشان‌دهندهٔ نوع دستور ذخیره‌سازی اصلی یا تکراری است. در روش SWIFT دستورات بارگذاری نیز به منظور در امان بودن از خطاهای SDC نیاز به تکرار دارند. یک پنجره بین ارزیابی آدرس و مصرف آن به‌وسیلهٔ دستور بارگذاری وجود دارد. پنجرهٔ بعدی فاصلهٔ بین دستور بارگذاری و تکرار مقدار است. هر خطایی که در این پنجره اتفاق بیفتد به خرابی داده ختم می‌شود. CRAFT:LVQ مشابه با CRAFT:CBS این موضوعات را برای دستور بارگذاری پوشش می‌دهد و از ساختار سخت‌افزاری LOAD VALUE QUEUE برای دستیابی به تکرار دستور بارگذاری استفاده می‌کند [۹].

روش‌های نرم‌افزاری از نوعی تحمل‌پذیری خطای نرم‌افزاری[۲۰] و افزونگی نرم‌افزاری[۲۱] برای تشخیص خطاهای گذرا استفاده می‌کنند. این شیوه‌ها از افزونگی زمانی و اطلاعاتی برای کشف اشکال‌ها در طی اجرای برنامه استفاده می‌کنند. این فنون به دو گروه تقسیم می‌شوند:

۱- فنون Recovery Block [۱۴] و N-Version Programming [۱۵] که اجرای برنامه را تکرار نموده و بر اساس نتایج تولید شده به‌وسیلهٔ هر مؤلفه افزونه، رأی‌گیری انجام می‌دهد.

۲- نوع دیگری از این فنون مبتنی بر مقاوم‌سازی نرم‌افزار به‌وسیلهٔ افزودن کدهای کنترلی به برنامه است. این فنون SIHFT نامیده می‌شود که از افزونگی اطلاعات برای کشف خطا در داده و جریان کنترل برنامه استفاده می‌کند.

فنون مبتنی بر مقاوم‌سازی نرم‌افزار به دلیل کم‌هزینه بودن، سهولت به‌روزرسانی و قابلیت نگهداشت بهتر جذاب هستند. فنون نرم‌افزاری کشف خطا شامل تحمل خطا بر پایهٔ الگوریتم[۲۲]، ادعاها، بررسی جریان‌کنترل، EDAC، تکرار رویه و تبدیلات خودکار هستند.

روش تحمل‌پذیری خطا مبتنی بر الگوریتم روش خودشمول[۲۳] برای کشف، مکان‌یابی و تصحیح خطاهای درون رویه نرم‌افزاری است. این روش، از ساختار عددی عملیات استفاده می‌کند. این فن روی محاسبات ماتریسی و این‌که چگونه می‌توان خطاهای احتمالی در بردار ماتریسی و ضرب ماتریس در ماتریس را کشف و تصحیح کرد، تمرکز دارد. این روش، روش مؤثری است اما عمومیت ندارد و برای برنامه‌هایی که از ساختار منطقی استفاده می‌کنند، مناسب است [۱۶].

ادعاها، ابزار عمومی برای ارزیابی و کشف خطا در برنامه هستند. یک ادعای قابل اجرا، یک دستور است که بررسی می‌کند آیا شرایط معینی میان متغیرهای متنوع برنامه نگه‌داشته می‌شود یا نه؟ اگر شرایط مورد نظر وجود نداشت، کاری باید انجام گیرد. در این روش، دستورات منطقی یا ادعاها در نقاط مختلف برنامه درج می‌شوند که منعکس‌کنندهٔ روابط ثابت مابین متغیرهای برنامه هستند. چون این دستورات از دید برنامه‌نویس مخفی نبوده و تأثیر آنها به طبیعت برنامه و توانایی برنامه‌نویس بستگی دارد، باعث مشکلات فراوان می‌شوند [۱۷].

EDAC اصلی‌ترین راه حل برای جلوگیری از وقوع اشکال‌های گذرا در حافظه است. این نوع اشکال‌ها، موضوعات

17. Active Watchdog Processor
18. Passive Watchdog Processor
19. Silent Data Corruption

20. Software Fault Tolerance
21. Software Redundancy
22.Algorithm-Based Fault Tolerance
23. Self-Contained

Archive of SID

www.SID.ir

معروف و مورد توجه از نظر قابلیت اطمینان در سیستم‌های کامپیوتری هستند. سیستم‌های COTS، معمولاً شمای کدگذاری ساده EDAC را دارند یا فاقد چنین سیستمی هستند. این محدودیت به‌وسیلهٔ استفاده از EDAC نرم‌افزاری قابل حل است. EDAC نرم‌افزاری به سخت‌افزار اضافی برای پیاده‌سازی روش‌های محافظت از حافظه نیاز ندارد و هدف آن طراحی شمایی برای محافظت از داده مقیم در حافظه است. برنامه EDAC به‌صورت یک وظیفه در پس زمینه اجرا می‌شود و از دید برنامه‌های دیگر که در پردازنده اجرا می‌شوند، پنهان است [۱۹].

در روش تکرار رویه، برنامه‌نویس تصمیم می‌گیرد که اکثر رویه‌های بحرانی را تکرار کند و نتایج به‌دست آمده از اجرای رویه‌ها را با هم مقایسه کند. این روش نیاز به تصمیم‌گیری برنامه‌نویس در مورد رویه‌های تکراری و تعریف نقاط مناسب برای بررسی نتایج دارد. این تغییر کد به‌صورت دستی انجام می‌گیرد و باعث ایجاد خطا می‌شود [۱۸].

روش تبدیلات خودکار، روشی است مبتنی بر تولید داده و افزونگی که طبق مجموعه‌ای از تبدیلات در سطح کد منبع انجام می‌گیرد. کد تبدیل شده، توانایی کشف خطای مؤثر بر داده و کد را دارد. هدف اول با تکرار هر متغیر و اضافه‌کردن بررسی‌های سازگاری بعد از هر عمل خواندن به‌دست می‌آید. اما تبدیلاتی که تمرکز بر کد دارند، دستورات برنامه را تکرار می‌کند و بعد از اجرای عملیات اصلی و تکرار برنامه، بررسی سازگاری دستورات اجرا شده صورت می‌گیرد [۲۰-۱۶].

وظیفهٔ اصلی، روش بررسی جریان کنترل، بخش‌بندی برنامه در قالب بلاک‌های پایه یا بخش‌های آزاد از انشعاب کد است. امضاء معینی به هر بلاک نسبت داده می‌شود و خطا با مقایسهٔ امضای زمان اجرا و امضای محاسبه شده کشف می‌شود.

روش‌های مبتنی بر جریان کنترل و مقایسهٔ آنها

فنون بررسی جریان کنترل اکثراً بر قانون پایش امضاء بنا نهاده شده‌اند. در این فنون، شمایی از جریان اجرای برنامه استخراج می‌شود و امضاءهایی که نمایش‌دهندهٔ جریان اجرای درست برنامه هستند قبل از اجرای سیستم به آن نسبت داده می‌شوند. در طی اجرای برنامه، امضاءها دوباره محاسبه و با مقدار ذخیره شده مقایسه می‌شوند. در صورت عدم مطابقت دو مقدار، یک خطای گذرا گزارش داده می‌شود.

مهم‌ترین تفاوت شیوه‌های مختلف در روشی است که برای محاسبهٔ بررسی امضاء استفاده می‌کنند. دو روش عمده برای بررسی جریان کنترل برنامه عبارتند از:

۱- Embedded Signature Monitoring (ESM): در این شیوه، امضاءها در درون برنامه کاربردی قرار دارند و برنامه بررسی‌های لازم را انجام می‌دهد [۲۲، ۲۱].

۲- Autonomous Signature Monitoring (ASM): در این شیوه امضاءها در یک حافظهٔ ناظر اختصاصی ذخیره می‌شود و برنامهٔ ناظر جریان کنترل را بررسی می‌کند [۲۲، ۲۱].

مهم‌ترین فنون بررسی جریان کنترل شامل ECCA[۲۴]، CFCSS[۲۵]، ACFC[۲۶]، CEDA[۲۷]، SWIFT، RSCFC[۲۸] است که در ادامه به معرفی آنها می‌پردازیم:

روش ECCA

به هر بلاک یک شناسه یکتای بزرگ‌تر از دو، که شناسهٔ بلاک نامیده می‌شود، نسبت داده می‌شود. دو خط کد به هر بلاک نسبت داده می‌شود. اولین ادعا، طبق معادلهٔ (۱)، یک نسبت‌دهی ساده است که در زمان ورود به بلاک اجرا می‌شود و دومین ادعا طبق معادلهٔ (۲) نیز یک مقداردهی است. این دستور بلافاصله قبل از خروج از بلاک اجرا می‌شود و در انتهای بلاک قرار دارد.

$$Id=BID/!(Id \bmod BID)*(id \bmod 2) \quad (۱)$$

$$Id=NEXT+!!(Id- BID) \quad (۲)$$

که در آن، Id یک متغیر سراسری است که در طی اجرا، هنگام ورود و خروج به هر بلاک تغییر می‌کند و BID شناسه بلاک، عدد فرد یکتایی است که به هر بلاک نسبت داده می‌شود و در زمان کامپایل مشخص می‌شود. NEXT طبق معادلهٔ (۳) عددی است که در زمان کامپایل مشخص می‌شود و برابر مضرب مشترک BID بلاک‌های قابل دسترسی از بلاک جاری یعنی ساکسسورهای[۲۹] بلاک جاری می‌شود.

$$NEXT=\prod BID \quad (۳)$$

این روش قادر به کشف خطاهای تکی بین بلاک‌هاست اما نمی‌تواند خطاهای جریان کنترل درون بلاک‌ها و خطاهایی که باعث تصمیم‌گیری نادرست در انشعاب شرطی می‌شود، کشف کند [۲۳].

روش CFCSS

این روش با به‌کارگیری یک ثبات اختصاصی (GSR) که شامل امضای زمان اجرای گرهٔ V_n است، جریان کنترل برنامه را بررسی می‌کند. این ثبات امضای زمان اجرا را با امضای تولید شده در زمان

23. Enhanced Control flow Checking Assertion
24. Control Flow Checking by Software Signature
25. Assertion for Control Flow Checking
26. Control-Flow Error Detection trough Assertions
27. Relationship Signatures for Control Flow Checking
29. Successors

Archive of SID
www.SID.ir

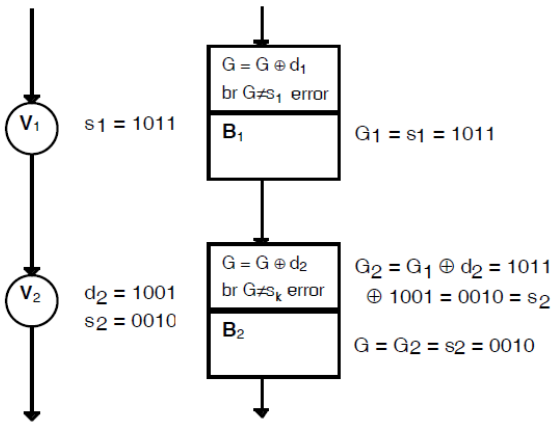
کامپایل مقایسه می‌کند و درستی انشعاب بر اساس این مقایسه مشخص می‌شود. در زمان کامپایل، شماره یکتای S_i به هر بلاک پایه اختصاص داده می‌شود. در زمان اجرا اگر ثبات G حاوی مقداری متفاوت با مقدار S_i تخصیص داده شده به گره جاری باشد، خطایی در برنامه رخ می‌دهد. G زمان انتقال از یک بلاک پایه به بلاک پایهٔ دیگر تولید می‌شود و تابع امضاء در گرهٔ مقصد انشعاب، G جدید را تولید می‌کند. تابع f با استفاده از امضای گره قبلی و گره جدید طبق معادلهٔ زیر G را محاسبه می‌کند:

$$f=f(G,d_i)=G \text{ xor } d_i \qquad (۴)$$

در این معادله، G، شامل امضای گره قبلی است. چون گره‌های مبدأ و مقصد انشعاب به‌طور یکتا در E مشخص می‌شود از این دو مقدار استفاده می‌شود. بر اساس انشعاب br_{sd} که s گره مبدأ و d گره مقصد است، معادله به صورت زیر درمی‌آید:

$$d_d=S_s \text{ xor } S_d \qquad (۵)$$

S_s و S_d در زمان کامپایل محاسبه می‌شود و d_d تفاوت امضاءها درگره مقصد است. روش CFCSS در شکل (۱) نشان داده شده است.

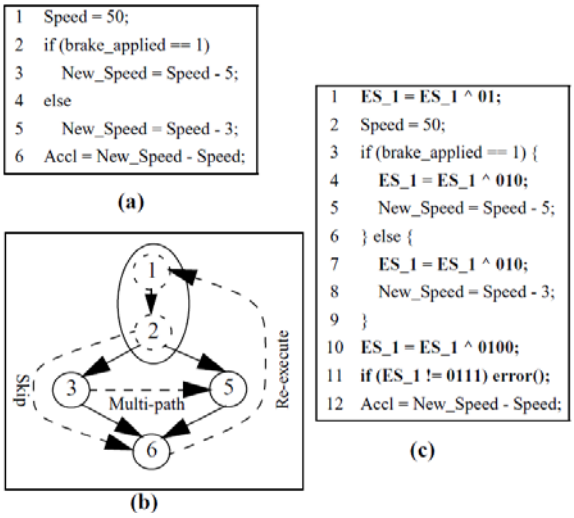


شکل ۱- روش CFCSS [۲۴]

این روش نمی‌تواند خطاهایی که چندین گره، چندین گره مقصد را که خاصیت *branch-fan-in* دارند و به اشتراک می‌گذارند، پوشش دهد. گره *branch-fan-in* گره‌ای است که $Pred(V)>1$ باشد. همچنین در آن به خطاهای مؤثر بر داده برنامه توجه نشده است [۲۴].

روش ACFC

روش جدیدی است که به‌منظور کشف خطا برخلاف روش‌های دیگر از تطابق اجرایی استفاده می‌کند و تنها خطای بین بلاکی را کشف می‌کند. تفاوت اصلی این روش با روش‌های قبلی در این است که شمای خطاهای جریان کنترل برنامه را طبقه‌بندی می‌کند و روش جدیدی برای بررسی کنترل جریان برنامه که وابستگی به ساکسسور و پردسسور بلاک‌های پایه ندارند، ارائه می‌کند. این روش از اجرای توازن روی بلاک پایه استفاده می‌کند و متغیر خاصی را به رویهٔ بررسی جریان کنترل اضافه می‌کند که کلمات وضعیت اجرایی[۳۰] نام دارد. هر بلاک پایه موجود در نمودار برنامه یک بیت تطابق در کلمهٔ *ES* دارد. یک روتین با تعداد زیاد بلاک‌های پایه، ممکن است چند کلمهٔ *ES* نیاز داشته باشد که باعث سربار حافظه می‌شود [۲۵]. این روش در شکل (۲) نشان داده شده است.



شکل ۲- روش ACFC [۲۵]

روش SWIFT

روش EDDI+ECC+CFE را به‌عنوان پایه استفاده می‌کند و بهینه‌سازی‌هایی را بر روی آن انجام می‌دهد که در ادامه به شرح آن می‌پردازیم:

۱- بررسی جریان کنترل در بلاک‌هایی با دستور *Store*: بررسی جریان کنترل در این بلاک‌ها، به دلیل این‌که تنها دستورات *Store* داده را به خارج از ناحیهٔ SOR[۳۱] می‌فرستد، انجام می‌شود.

۲- افزونگی در بررسی جریان کنترل/انشعاب: روش SWIFT به‌منظور بهینه‌سازی، دستورات افزونه انشعاب را حذف می‌کند.

SWIFT با ارائه سازوکار ECC باعث افزایش کارآیی شده است. این روش فرض‌کرده که با محافظت حافظه به‌طور سخت‌افزاری، نیاز به افزونگی دستورات ذخیره‌سازی نیست[۲۶].

سه معیار مهمی که در این روش‌های نرم‌افزاری باید مورد توجه قرار گیرد *پوشش خطا، سربار حافظه و سربار کارایی* است که با بررسی‌های انجام شده روش RSCFC به دلیل توجه به هر سه

28. Execution Status
29. Sphere Of Redundancy

Archive of SID
www.SID.ir

مورد و ارائه راه‌حلی جامع برای اعمال در سیستم‌های فضایی به‌منظور جلوگیری از وقوع مشکلات غیرمترقبه و توجه به تمام جنبه‌های اثر خطای گذرا از روش‌های دیگر بهتر است. جدول (۱) نشان‌دهندهٔ پوشش خطای روش‌های موجود است. از بین روش‌های ارائه شده، RSCFC سربار حافظه و سربار کارایی کمتری دارد.

جدول ۱- مقایسهٔ روش‌های نرم‌افزاری کشف خطا

	خطای درون بلاکی	خطای بین‌بلاکی	خطای بخش‌داده
ECCA		✓	
CFCSS		✓	
ACFC		✓	
SWIFT		✓	✓
RSCFC	✓	✓	✓

معرفی روش RSCFC

روش RSCFC جریان کنترل برنامه را با به‌کارگیری متغیر سراسری اختصاصی S که شامل امضای زمان اجرای مربوط به گره جاری در نمودار جریان کنترل است، بررسی می‌کند [۲۷].

یک دستور *test* در ابتدای هر بلاک پایه تولید می‌شود و یک دستور *set* در انتهای هر بلاک پایه قرار می‌گیرد. زمانی که کنترل از بلاک v_j به v_i انتقال می‌یابد، دستور *test* با استفاده از دو مقدار بررسی می‌کند که آیا این گره مقصد v_i قابل قبول است یا نه، که در ذیل به توضیح آن می‌پردازیم.

ایجاد امضاء در بلاک‌های پایه

روش RSCFC نیز مانند روش‌های دیگر مبتنی بر بخش‌بندی کد برنامه به بلاک‌های پایه بنا نهاده شده است. بنابراین بر روی قطعه کدی از Bubble Sort برای توضیح روش که در شکل (۳) نشان داده شده و نمودار جریان کنترل متناظر با آن تمرکز می‌شود.

```
1 top=elem;
  while(top>1){
2   i=1;
    while(i<top){
      if(sortlist[i]>sortlist[i+1]){
3       j=sortlist[i];
        sortlist[i]=sortlist[i+1];
        sortlist[i+1]=j;
      }
4   i=i+1;
   }
5  top=top-1;
  }
```

شکل ۳- قطعه کد Bubble Sort [۲۷]

با توجه به نمودار، جریان کنترل $P=\{V, E\}$، $Succ(v_i)$ را به عنوان مجموعه گره‌های بعدی v_i و $pred(v_i)$ را به عنوان مجموعه گره‌های قبلی v_i تعریف می‌کنیم. گره $v_j \in Succ(v_i)$ است اگر و تنها اگر $br_{i,j} \in E$ باشد.

به طور مشابه، گره $v_j \in Pred(v_i)$ است اگر $br_{j,i} \in E$ باشد. انشعاب $br_{i,j}$، در طی اجرای برنامهٔ P در صورتی‌که عضو مجموعه E نباشد غیرقانونی است و منتهی به خطای جریان کنترل می‌شود.

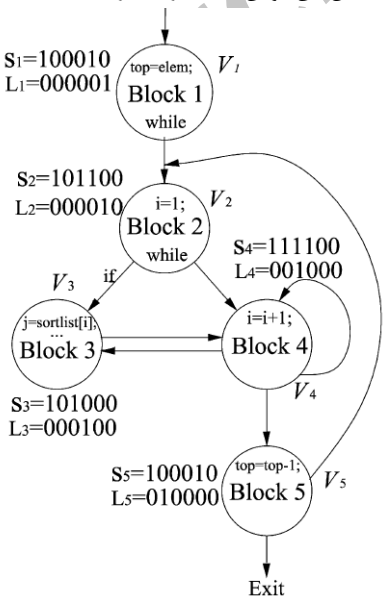
همان‌طورکه در شکل (۴) نشان داده شده، $Pred(v_2) = \{v_1 , v_5\}$ و $Succ(v_2) = \{v_3 , v_4\}$ است. اگر پرشی از v_2 به v_5 داشته باشیم، که جزء $Succ(v_2)$ نباشد، باعث خطا می‌شود. فرض کنید یک برنامه n بلاک پایه دارد با نام‌های $\{v_1 , v_2 , \dots , v_n\}$ و برای هر بلاک v_i اگر معادله (۵) را داشته باشیم:

$$Succ(v_i) = \{v_f , \dots , v_k , \dots v_m\} \qquad (۵)$$

بر این اساس امضاء S_i از بلاک v_i به صورت زیر مقداردهی می‌شود:

$$S_i = \underbrace{l^{n+1} o l^m \dots . l^k \dots . l^f \dots . o^l}_{n+1} \qquad (۶)$$

به‌طوری‌که $n+1, m, k, f, 1$ نشان‌دهندهٔ اولین، f امین، k امین، mامین و $n+1$امین بیت در S_i است. بیت $n+1$ام همیشه یک خواهد بود. هر بیت دیگر در S_i به غیر از $n+1$امین بیت نشان‌دهندهٔ گرهٔ مطابق آن در نمودار جریان برنامه است. به عنوان مثال بیت ۱ نشان‌دهندهٔ بلاک v_1 است، بیت ۲ نشان‌دهندهٔ بلاک v_2 است و بیت n نشان‌دهندهٔ بلاک nام است. درصورتی‌که گرهی در ساکسسور گره L_i باشد بیت مطابق با آن در S_i برابر یک می‌شود در غیراین‌صورت برابر صفر می‌شود. طبق این روش امضاء برای هر بلاک ساخته می‌شود.



شکل۴- نمودار جریان کنترل برنامه Bubble Sort[۲۷]

Archive of SID

www.SID.ir

کشف جریان کنترل بین بلاکی

RSCFC جریان‌کنترل برنامه را با به‌کارگیری متغیر سراسری اختصاصی S که شامل امضاء زمان اجرای مربوط به گره جاری در نمودار جریان‌کنترل است، بررسی می‌کند.

یک دستور *test* در ابتدای هر بلاک پایه تولید می‌شود و یک دستور *set* در انتهای هر بلاک پایه قرار می‌گیرد. زمانی که کنترل از بلاک v_j به v_i انتقال می‌یابد، دستور *test* با استفاده از دو مقدار بررسی می‌کند که آیا این گره مقصد v_i قابل قبول است یا نه، که این دو مقدار عبارتند از:

- امضای گرهٔ قبلی (گره منبع انشعاب v_j)
- اطلاعات مکانی L_i گره جاری در گراف جریان کنترل

$$L_i = \underbrace{00\ldots10\ldots0}_{n+1}^{i} \qquad (۷)$$

ایجاد L_i مشابه با امضاء بلاک S_i است. تمایز بین آنها تنها در بیتی است که به ۱ ست می‌شود و نشان‌دهندهٔ این است که شماره بلاک v_i در نمودار جریان کنترل i است همان‌طورکه در شکل (۴) می‌بینید L_1=*000001* و L_5=*010000* است. دستور *test* و *set* هر بلاک پایه v_i به صورت زیر تولید می‌شود:

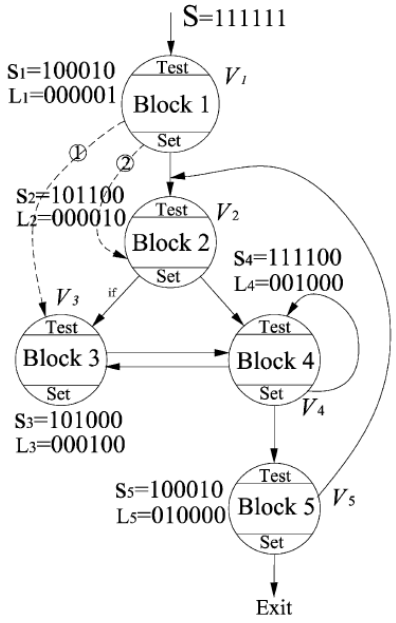
$$if\ (S = S\ \&\ Li) \equiv 0\ error \qquad (۸)$$

$$S = S_i \&\ (-!\ (S \oplus L_i)) \qquad (۹)$$

قبل از انشعاب S، $br_{j,i}$ مقدار S_j را دارد که شامل اطلاعات مربوط به ساکسسورهای گره v_j است. بعد از انشعاب کنترل به v_i انتقال می‌یابد. درغیاب خطا $S\&L_i$ برابر L_i می‌شود. بنابراین اگر مقدار S غیرصفر باشد نشان‌دهندهٔ انشعاب درست است. صفر شدن S نشان‌دهندهٔ ایجاد خطا در جریان کنترل برنامه است. زمان مقداردهی set، نتیجهٔ XOR دو مقدار S و L_i مقدار صفر می‌شود درحالی‌که دستور Not آن را به یک تبدیل می‌کند و نتیجهٔ نهایی ۱- می‌شود و در نهایت S با مقدار جدید امضاء زمان اجرا مقداردهی می‌شود.

$$S_i \&\ (-1) = S_i \qquad (۱۰)$$

به عبارت دیگر اگر v_i، ساکسسور گره v_j نباشد، مقدار $S\&L_i$ صفر می‌شود و خطای جریان کنترل بعد از اجرای دستور *test* گره v_i کشف می‌شود. کد تغییریافته در شکل (۵) نشان داده شده است.



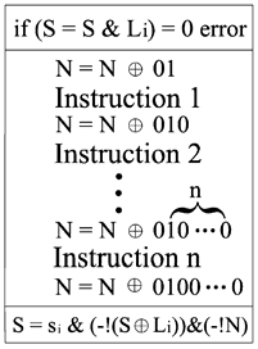
شکل ۵- کد تغییریافته[۲۷]

کشف جریان کنترل درون بلاکی

برای کشف خطاهایی از این نوع، امضای محلی N به‌منظور حفظ و بررسی درستی اجرای متوالی دستورات در یک بلاک پایه تولید شده است. برای هربلاک پایه B_i، یک شماره n_i از دستورات در غیاب خطا اجرا می‌شود. در ابتدای هر بلاک پایه، امضای N با یک بیت تطابق *XOR* می‌شود. در غیاب خطا، در انتهای هر بلاک پایه N مقدار صفر را می‌گیرد. در انتهای هر بلاک تأثیر امضای N به امضای زمان اجرای S از طریق معادلهٔ زیر اضافه می‌شود:

$$S = S\ \&\ (-!\ N) \qquad (۱۱)$$

نحوهٔ پیاده‌سازی آن در بلاک پایه شکل (۶) دیده می‌شود. با توجه به اینکه احتمال خطای درون بلاکی بسیار پایین است می‌توان از پیاده‌سازی این بخش به‌منظور رسیدن به کارآیی بالاتر در حافظه و زمان اجرا صرف‌نظر کرد.



شکل ۶- بلاک پایه با سازوکار کشف خطای درون بلاکی [۲۷]

Archive of SID

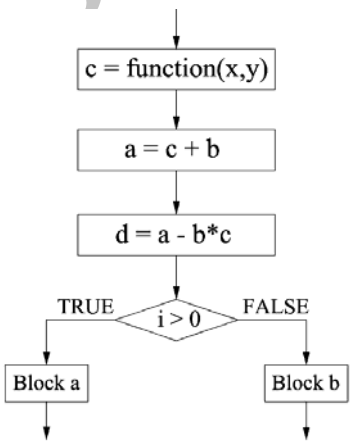
www.SID.ir

ساز و کار کشف خطا در جریان داده

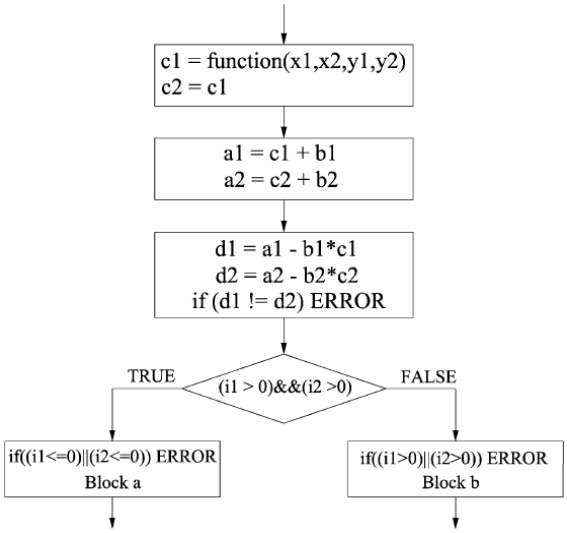
این روش به کشف خطاهای مؤثر بر داده کمک می‌کند. ایدهٔ اصلی تعریف، رابطهٔ وابستگی داخلی بین متغیرهای برنامه و طبقه‌بندی آنها در دو طبقه بر طبق نقش آنها در برنامه است.

- **متغیرهای میانی:** متغیرهایی هستند که برای محاسبهٔ مقدار متغیرهای دیگر استفاده می‌شود.
- **متغیرهای نهایی:** متغیرهایی هستند که در محاسبهٔ متغیر دیگری استفاده نمی‌شود.

بعد از تعیین روابط متغیرها برنامه تکرار می‌شود. برای هر عمل که روی متغیر اصلی انجام می‌شود، عملی برای افزونهٔ آن تکرار می‌شود که آن را تکرار متغیر می‌نامیم. بعد از هر عمل نوشتن روی متغیرهای نهایی، دستورات بررسی یکسان‌بودن مقادیر دو متغیر تولید می‌شود. در صورت تمایز بین این دو مقدار خطا کشف می‌شود. نحوهٔ پیاده‌سازی این روش در شکل (۷) و (۸) دیده می‌شود.



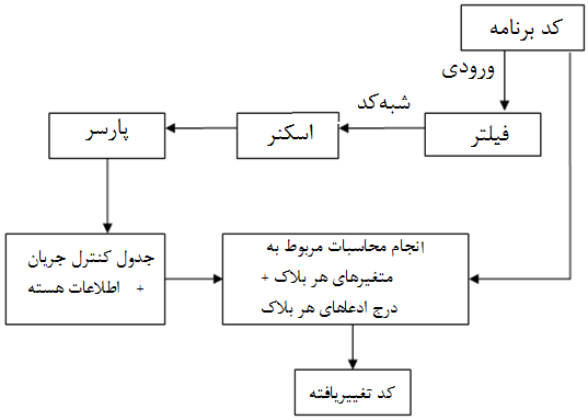
شکل ۷- کد اصلی [۲۷]



شکل ۸- کد تغییریافته با پیاده‌سازی ساز وکار کشف خطا[۲۷]

پیاده‌سازی روش RSCFC

مراحل تولید کد تغییر یافته پیشنهادی در شکل (۹) آمده است. این ابزار به زبان C++ نوشته شده است و به عنوان ورودی، برنامه به زبان C را گرفته و آن را با درج دستورات به کدی که قابلیت تشخیص خطای گذرا را دارد تبدیل می‌کند.



شکل ۹- رویهٔ پیاده‌سازی RSCFC

چون پارسر[۳۲] تنها به پیمایش دستورات مؤثر بر جریان کنترل برنامه نیاز دارد، برای کاهش دستورات برنامهٔ عملیات فیلتر انجام می‌شود. در عملیات فیلتر، دستوراتی که جریان کنترل برنامه را تغییر نمی‌دهند به دستورات تهی تبدیل می‌شوند. به عنوان مثال کد زیر را ببینید:

```
for(i=0;i<10;i++){
X++;
Y--;
}
```

به صورت زیر تبدیل می‌شود:

```
for(){
;
 ;
}
```

بعد از ساخت نمودار برنامه در خروجی اسکنر، توسط پارسر درخت تحت تسلط و پس تسلط استخراج شده و هستهٔ برنامه مشخص می‌شود. با داشتن این اطلاعات، روش RSCFC بر برنامه اعمال می‌شود و خروجی این فاز، برنامه تغییر یافته است.

این ابزار به زبان C++ نوشته شده است و به عنوان ورودی، برنامه مورد نظر برای مقاوم‌سازی به زبان C را گرفته و آن را با افزودن دستوراتی به کدی که قابلیت تشخیص خطای گذرا را دارد تبدیل می‌کند. تزریق خطا در سطح کد منبع انجام شد و نتایج آزمون آن نشانگر کارایی بالای این روش است.

32. Parser

Archive of SID
www.SID.ir

برای بررسی کارایی و میزان حافظهٔ مصرفی روش پیشنهاد شده، سه برنامهٔ محک، مرتب‌سازی درجی، مرتب‌سازی سریع و ضرب ماتریس‌ها را انتخاب کردیم:

1. Insertion Sort (IN)
2. Quick Sort (QS)
3. Matrix Multiplication (MM)

این برنامه‌ها بر روی سخت‌افزار پنتیوم ۴ با حافظهٔ ۵۱۲ مگابایت و تحت ویندوز XP اجرا شده است. برای تخمین کارایی و حافظه از نرم‌افزار پروفایلینگ AQTIME6 استفاده شده است که نتایج به‌دست آمده در جدول (۱) ارائه شده است

جدول ۱- نسبت سربار حافظه و کارایی برنامه‌ها

برنامه	سربار حافظه	سربار کارایی *(زمان اجرا)*	اندازه کد
IN	۱/۵	۱/۱۵	۱/۳۵
QS	۱/۳	۱/۱۷	۱/۲۱
MM	۱/۶۵	۱/۰۹	۲/۱

در جدول (۱)، نسبت سربار حافظه و کارایی برنامه‌ها نشان داده شده است که نتایج به‌دست آمده حاکی از آن است که سرعت برنامه‌ها بعد از اعمال روش نسبت به برنامهٔ اصلی و خام کاهش چندانی ندارد، اما سربار حافظه و اندازهٔ کد افزایش یافته است.

نتیجه‌گیری

به دلیل احتمال بالای تأثیر خطای گذرا بر جریان کنترل برنامه، بررسی این روش‌ها اهمیت بالایی دارد. روش‌های متنوعی برای بررسی جریان کنترل ارائه شده است که از بین آنها، RSCFC با توجه به کارایی بیشتر انتخاب و پیاده‌سازی شد. سپس برای تخمین کارایی و حافظه از نرم‌افزار پروفایلینگ AQTIME6 استفاده شد که نشان‌دهندهٔ سربار قابل قبولی است. این فناوری می‌تواند در گسترهٔ وسیعی از نرم‌افزارهای اتکاءپذیر به خصوص نرم‌افزار مورد استفاده در ماهواره مورد استفاده قرار گیرد. کاربرد آن در نرم‌افزارهای ماهواره می‌تواند به‌عنوان ادامهٔ کار مطرح شود. همچنین، از آنجاکه در اکثر روش‌های مبتنی بر جریان کنترل، بزرگ‌ترین مشکل، تنظیم دانه‌بندی آزمون و تعیین نقاط بررسی است، ارائهٔ فنونی به‌منظور بهینه‌سازی روش‌های جریان کنترل از لحاظ کارآیی و مصرف حافظه نیز می‌تواند به‌عنوان کار بعدی درنظرگرفته شود.

مراجع

[1] Croll, P. and Nixon, P., "Developing Safety-Critical Software within a CASE Environment," *Proceedings of the IEEE Colloquium on Computer Aided Software Engineering Tools for Real-Time Control*, 1991, p. 8.

[2] Ignat, N., Nicolescu, B., Savaria, Y. and Nicolescu, G., "Soft-Error Classification and Impact Analysis on Real-Time Operating Systems," *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*, Germany, 2006, pp.182-187.

[3] Reis, G., Chang, J., Vachharajani N., Rangan R., August I., "SWIFT: Software Implemented Fault Tolerance", *Proceedings of the CGO'05*, 2005, pp. 243-254.

[4] Baumann, R., "Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends," *Proceedings of the IEEE Reliability Physics Tutorial Notes*, Reliability Fundamentals, 2002, pp. 121-01.1–121-04.

[5] Shirvani, P.P., Oh, N., McCluskey, E.J., and Wood, D.L., "Software-Implemented Hardware Fault Tolerance Experiments COTS in Space," *Proceedings of the International Conference on Dependable Systems and Network*, New York, NY, 25-28, 2000

[6] Yenier, U., *Fault Tolerant Computing in Space Environment and Software Implemented Hardware Fault Tolerance Techniques*, Technical Report, Department of Computer Engineering, Bosphorus University, Istanbul, 2003.

[7] Rebaudengo, M., Sonza Reorda, M., Torchiano, M., and Violante, M., "Soft-error Detection Through Software Fault-Tolerance Techniques", *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Albuquerque, NM, USA, Nov 1999, pp. 210-218.

[8] Lisboa, C.A.L, Carro, L., Reorda, M., Violante, M., "Online Hardening of Programs Against SEUs and SETs," *Proceedings of the 21st International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2006.

[9] Reis, G., Chang, J., Vachharajani, N., Rangan, R., August and Shubhendu, S., "Design and Evaluation of Hybrid Fault-Detection System," *Proceedings of the 32nd IEEE International Symposium on Computer Architecture*, Albuquerque, NM, USA, pp. 148-159, 2005.

[10] Jing, Y., Garzaran, M. J., and Snir, M., "Efficient Software Checking for Fault Tolerance," *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, April 14-18, 2008, pp. 1-5.

[11] Horst, R.W., Harris, R. L. and Jardine, R. L., "Multiple Instruction Issue in the NonStop Cyclone Processor," *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 216–226.

[12] Mahmood, A. and McCluskey, E. J., "Concurrent Error Detection Using Watchdog Processors-A Survey," *IEEE Transactions on Computers*, Vol. 37, No. 2, 1998, pp. 160-174.

[13] Namjaoo, M. and McCluskey, E. J., "Watchdog Processors and Capability Checking," *Proceedings of the 12th International Symposium on Fault-Tolerant Computing (FTCS-12)*, 1982, pp. 245-248.

[14] Randell, B., "System Structure for Software Fault Tolerant," *IEEE Transaction on Software Engineering*, Vol. 1, No. 2, 1975, pp. 220-232.

Archive of SID

www.SID.ir

[15] Avizienis A., "The N-Version Approach to Fault-Tolerant Software," *IEEE Transaction on Software Engineering*, Vol. 11, No. 12, 1985, pp. 1491-1501.

[16] Stefanidis, V. K., and Margarits, K. J., "Algorithm Based Fault Tolerance: Review and Study," *Proceedings of the 2004 International Conference of Numerical Analysis and Applied Mathematics (ICNAAM'04)*, 2004, pp. 1-8

[17] Rebaudengo, M., Sonza Reorda, M., Torchiano, M. and Violante, M., "Soft-error Detection Through Software Fault-Tolerance Techniques", *Proceedingsof the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Albuquerque*, NM, USA, Nov 1999, pp. 210-218.

[18] Oh, N. and McCluskey, E. J., "Low Energy Error Detection Technique using Procedure Call Duplication," *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*, 2001.

[19] Shirvani, P. P., Saxena, N. and McCluskey, E. J., "Software Implemented EDAC Protection against SEUs," *IEEE Transactions on Reliability*, Vol. 49, No. 3, 2000, pp. 273-284.

[20] Rebaudengo, M., Sonza Reorda, M. and Violante, M., "A Source-to-Source Compiler for Generating Dependable Software," *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001, pp. 33-42.

[21] Benso, A., Di Stefano, C., Natale, G., Prinnetto, P. and Tagliaferri, L., "Control Flow Checking via Regular Expression," *Proceedings of the 10th Asian Test Symposium, Dipt. di Automatica e Informatica*, Politecnico di Torino, 2001, pp. 299-303.

[22] Ziener, D. and Teich, J., "Concepts for Autonomous Control Flow Checking for Embedded CPUs," *Proceedings of the 5th International Conference on Autonomic and Trusted Computing, Lecture Notes in Computer Science*, Vol. 5060, Springer-Verlag, 2008, pp.234-248.

[23] Alkhalifa, Z., Nair, V.S.S., Krishnamurthy, N. and Abraham, J. A., "Design and Evaluation of System-Level Checks for on-Line Control Flow Error Detection," *IEEE Transactionon Parallel and Distributed Systems*, Vol. 10, No.6, 1999, pp. 627-641.

[24] Oh, N., Shirvani, P. P. and McCluskey, E. J., "Control-Flow Checking by Software Signatures," *IEEE Transactions on Reliability*, Vol. 51, No. 1, 2002, pp. 111-122.

[25] Venkatasubramanian, R., Hayes, J. P. and Murray, B. T., "Low-Cost On-Line Fault Detection Using Control Flow Assertions," *Proceedingsof the 9th IEEE International On-Line Testing Symposium*, July 2003, pp.137-143.

[26] Reis, G., Chang, J., Vachharajani, N., Rangan, R. and August I., "SWIFT: Software Implemented Fault Tolerance", *Proceeding of the CGO'05*, 2005, pp. 243-254.

[27] Li, A. and Hong, B., "Software Implemented Transient Fault Detection in Space Computer," *Aerospace Science and Technology*, Vol. 11, No. 2-3, 2007, pp. 245-252.

Archive of SID

www.SID.ir