دانشگاه صنعتی امیر کبیر
(پلی تکنیک تهران)

گزارش کتبی درس روش تحقیق و گزارش‌نویسی

ارائه یک روش جدید برای تحلیل ایستا برای پیدا کردن استفاده ناامن از متغیرهای برنامه‌نویسی در برنامه‌های جاوا

استاد راهنما: جناب آقای دکتر شیری

نگارش: سید محمدمهدی احمدپناه

شماره دانشجویی: ۹۰۳۱۸۰۶

خرداد ۱۳۹۳

عنوان:

ارائه یک روش جدید برای تحلیل ایستا برای پیدا کردن استفاده ناامن از متغیرهای برنامه‌نویسی در برنامه‌های جاوا

A New Method to Static Analysis of Unsafe Use of Variables in Java Program

چکیده:

معرفی زمینه: زبان برنامه‌نویسی جاوا به عنوان یکی از مشهورترین و کامل‌ترین زبان‌های برنامه‌نویسی، در سال ۱۹۹۵ پدید آمد. این زبان از جمله زبان‌های type safe به شمار می‌رود. به دلیل کاربرد زیاد این زبان در نوشتن برنامه‌های گوناگون، با ایجاد یک ابزار برای تحلیل ایستا در خصوص یافتن خطاهای برنامه‌نویسی از جمله استفاده غیرمطمئن از متغیرها و Exception Handling نادرست می‌توان از ایجاد اینگونه خطاها در زمان اجرا (Runtime) جلوگیری کرد. مزیت این کار این است که یافتن و اصلاح خطاهای برنامه نویسی در زمان اجرا هزینه‌بر و مشکل‌تر از قبل از زمان اجرا است. در موارد حساس مانند برنامه‌های نظامی و فضایی یا برنامه‌هایی که با جان و مال مردم مرتبط است، نمی‌توان خطایابی برنامه را در زمان اجرا انجام داد. در این گونه موارد است که اهمیت تحلیل ایستا بیش از پیش مشخص می‌شود.

سوال تحقیق: یک راه حل بهتر برای تحلیل ایستا در خصوص یافتن استفاده غیرمطمئن از متغیرهای برنامه‌نویسی در جاوا چیست؟

روش‌های کنونی: ابزار و برنامه‌های نسبتاً زیادی برای حل این مسئله ساخته شده‌اند که هر کدام مشکلاتی دارند که باعث می‌شود که برنامه‌نویس نتواند این اطمینان را پیدا کند که قبل از اجرای برنامه، همه خطاهای احتمالیِ کد نوشته شده کشف می‌شود. از مهم‌ترین خطاهایی که ابزارهای کنونی قادر به کشف آنها نیستند، می توان به خطاهای ناشی از Exception Handling نادرست اشاره کرد.

روش مورد نظر و دلایل انتخاب آن: در این روش می‌خواهیم با استفاده از مفاهیم صوری (Formal) نظیر استفاده از گراف و الگوریتم‌های مرتبط با آن و بررسی جریان داده‌ها و به خصوص بررسی خطاهای محتمل رایج در برنامه نوشته شده، به تشخیص خطاهای ناشی از استفاده غیرمطمئن از متغیرها به خصوص خطاهای مرتبط با Exception Handling نامناسب بپردازیم که قبل از زمان اجرا بتوان نشان داد که برنامه نوشته شده توسط برنامه‌نویس عاری از این گونه خطاها است.

کلمات کلیدی: کنترل جریان اطلاعات، تحلیل ایستا، مدیریت استثناها در جاوا، استفاده ناامن از متغیرها

Keywords: Information Flow Control, Static Analysis, Exception Handling in Java, Unsafe Use of Variables

مقدمه:

تولید نرم‌افزار مقاوم یک چالش همیشگی است. برای سادگی تولید نرم‌افزار مقاوم، زبان‌های برنامه‌نویسی، مانند c++ و جاوا، ساختار Exception Handling را فراهم کرده‌اند که به برنامه‌نویس این اجازه را می‌دهد که خطاهای احتمالی برنامه را راحت‌تر و سریع‌تر مدیریت کند. گرچه ساختار Exception تشخیص خطاها و شکست‌های برنامه را راحت‌تر و ساده‌تر می‌کند، ولی کنترل جریان تلویحی با استفاده از Exception را دشوارتر می‌کند.[۱]

مطالعات بیانگر کیفیت پایین کدهای Exception Handling در پروژه‌های صنعتی است. سایر تمرکزها روی فاکتورهای انسانی می‌تواند منجر به خطای Exception Handling شود. همه این‌ها بیانگر اهمیت بهبود کیفیت برنامه‌هایی است که از ساختار Exception Handling پشتیبانی می‌کند. [۲]

در جاوا وقتی یک محدودیت معناشناختی نقض می‌شود یا خطای Exceptional رخ می‌دهد، یک Exception توسط یک عملیات پرتاب می‌شود و ممکن است به فراخوان‌کننده آن عملیات منتشر شود. وقتی یک عملیات شکست می‌خورد، قطعات کدی که بستگی به کامل انجام شدن موفقیت‌آمیز آن عملیات دارند، ممکن است اجرا نشوند. این به ویژگی امن‌بودن وابستگی که توسط B.Jacobs و F.Piessens ارائه شده، برمی‌گردد. [۳]

وابستگی کدها از طریق متغیرها، مشهود است. یک assignment را در نظر بگیرید، اگر Exception رخ دهد و assign کردن به متغیر انجام نشود، استفاده‌های بعدی بدون هرگونه بررسی از آن متغیر، به احتمال زیاد منجر به ایجاد bug می‌شود. [۴] bug pattern ها، اصطلاح‌های کد هستند که غالباً منجر به تولید خطاها می‌شوند.

زبان برنامه‌نویسی جاوا از جمله زبان‌های type safe به شمار می‌رود. type safe بودن یک زبان به این معناست که یک زبان خاصیت progress و preservation داشته باشد. یعنی یک عبارت خوش‌نوع (well-typed) توقف نمی‌کند (تعریف Progress) و اگر یک عبارت خوش‌نوع یک مرحله ارزیابی شود، سپس نتیجه آن عبارت نیز خوش‌نوع است. (تعریف Preservation) هر دوی این تعاریف باعث می‌شود که دریابیم که یک عبارت خوش‌نوع هیچگاه در طول ارزیابی، متوقف نمی‌شود. [۵] طبق این تعاریف و با توجه به خواص زبان جاوا درمی‌یابیم که جاوا یک زبان type safe و دارای یک type system قوی است.

به دلیل کاربرد زیاد این زبان در نوشتن برنامه‌های گوناگون، با ایجاد یک ابزار برای تحلیل ایستا در خصوص یافتن خطاهای برنامه‌نویسی از جمله استفاده غیرمطمئن از متغیرها و Exception Handling نادرست می‌توان از ایجاد این‌گونه خطاها در زمان اجرا (Runtime) جلوگیری کرد. مزیت این کار این است که یافتن و اصلاح خطاهای برنامه‌نویسی در زمان اجرا هزینه‌بر و مشکل‌تر از قبل از زمان اجرا است. در موارد حساس مانند برنامه‌های

نظامی و فضایی یا برنامه‌هایی که با جان و مال مردم مرتبط است، نمی‌توان خطایابی برنامه را در زمان اجرا انجام داد. در این گونه موارد است که اهمیت تحلیل ایستا بیش از پیش مشخص می‌شود.

در حالت کلی، برای یافتن و برطرف کردن خطاهای برنامه‌نویسی در زمان اجرا پرهزینه‌تر از همین‌کار در حالت ایستا است. چرا که برای این کار در زمان اجرا، باید به ازای هربار اجرای برنامه، برطرف کردن خطاها انجام شود که یعنی هزینه یکبار عیب‌یابی و برطرف کردن آن در تعداد بار اجرای برنامه ضرب می‌شود که در برنامه‌های پرکاربرد عدد بسیار بزرگی می‌شود؛ در حالی که در حالت ایستا، فقط یکبار این عیب‌یابی و بهبود آن انجام می‌شود و در مرحله اجرا نیازی به debug کردن نیست.

به طور کلی این موضوع پذیرفته شده است که هزینه می‌تواند کاهش یابد اگر اشکالات بیشتری از نرم‌افزار، زودتر در چرخه توسعه نرم‌افزار تشخیص داده شوند.[۶]

در سال‌های اخیر، ابزارها و تکنیک‌های زیادی برای یافتن خودکار bugها به‌وسیله تحلیل source code یا تحلیل استاتیک کد میانی ساخته شده است. خیلی از این تکنیک‌ها هم‌اکنون با کامپایلرها تجمیع شده‌اند. از جمله این ابزارها برای جاوا می‌توان به JLint[۷] و FindBugs[۸] اشاره کرد. به عنوان مثال، FindBugs از تکنیک‌های ad-hoc برای تعادل دقت، کارایی و قابلیت‌استفاده بهره می‌برد. اکثر این ابزارها bugهای معرفی شده برای ساختار Exception Handling مثل استفاده ناامن از متغیرها را در نظر نمی‌گیرد.[۹]

تکنیک‌های تحلیل مانند کنترل جریان، جریان داده و وابستگی کنترل برای گستره زیادی از کارهای مهندسی نرم‌افزار شامل آزمون رگرسیون و ساختاری، پروفایل اجرای پویا، برش ایستا و پویا و فهم برنامه استفاده می‌شود. برای کاربرد برنامه‌ها در زبان‌های برنامه‌نویسی مانند جاوا و C++، این تکنیک‌های تحلیل باید تأثیر رخدادهای Exception و ساختارهای Exception Handling لحاظ شوند. در صورتی‌که این‌ها لحاظ نشوند، تکینک‌های تحلیل به درستی و کامل انجام نخواهند شد و نتایج آن‌ها محدود و غیرقابل ارائه برای کاربردهای مختلف می‌شود.[۱۰]

unsafe use of variables به معنای استفاده نامطمئن متغیرها است؛ یعنی برنامه‌نویس در طول برنامه ممکن است در قسمت‌های مختلف کد، از متغیری به نادرستی یا به‌طور ناامن استفاده کند که این ممکن است باعث پایان ناگهانی و غیرمنتظره برنامه در زمان اجرا شود. ناامنی می‌تواند به دلیل عدم رعایت قوانین type safety باشد. در مسیر اجرای برنامه، در صورتی که گزاره‌ای منجر به گرفتن یک مقدار از متغیر شود و قبل از آن حالت آن متغیر به دلیل رخداد Exception به درستی و با موفقیت تغییر نکند، استفاده ناامن تلقی می‌شود که می‌توان این حالت را با ارائه تعاریفی به طور فرمال تعریف کرد. [۱۱]

هدف این است که به طور خودکار، مغایرت‌ها با امن‌بودن وابستگی در برنامه‌های جاوا در حالت متغیر تشخیص داده شود. بعضی از شکست‌های برنامه، یافتن و ردیابی آن‌ها بسیار سخت است، چون احتمال دارد فقط در زمانی که واقعاً رخ می‌دهند (زمان اجرا) قابل تشخیص باشند.

حال در اینجا است که اهمیت سوال تحقیق مشخص می‌گردد و اینکه در صورت یافتن پاسخ مناسب به این سوال که آیا روشی برای تشخیص و تعیین ایستای استفاده ناامن از متغیرها در زبان جاوا (unsafe use of variables in java) می‌توان ارائه کرد که از طرفی باعث افزایش کیفیت برنامه‌های نوشته شده با جاوا شود و از طرفی باعث سردرگمی و ناراحتی برنامه‌نویس و باعث کندشدن روند تولید نرم‌افزار نشود؟ می‌توان راه‌حلی برای این سوال داد که فقط افزایش کیفیت و امن‌شدن نرم‌افزار را تضمین کند ولی درباره اینکه باعث ایجاد مزاحمت گروه تولیدکننده نرم‌افزار می‌شود یا نه، هیچ تضمینی ارائه نمی‌کند. پس به دنبال ارائه روش جدیدی برای این سوال باید گشت.

کارهای مرتبط:

با توجه به سوال این تحقیق، و آنچه در قبل گفته شد؛ برتری و مزایای تحلیل ایستا نسبت به تحلیل پویا یا دینامیک مشخص است اما تحلیل ایستا بسیار سخت‌تر از تحلیل پویا است. در تحلیل پویا در هنگام اجرای برنامه می‌توان به راحتی همه اشکال‌های برنامه را یافت. در تحلیل ایستا باید با استفاده از ابزارها و مفاهیم فرمال، ارائه الگوریتم و بررسی جریان داده در برنامه راهکاری برای یافتن این‌گونه خطاها یافت. روش‌های ایستا از روش‌های ملموس‌تر گرفته تا روش‌های پیچیده‌تر که روش‌های تحلیل براساس معناشناخت[۱] است، وجود دارد. البته دو تئوری معروف Rice [۱۲] و Halting Problem [۱۳] محدودیت روش‌های تحلیل را بیان می‌کنند. در اینجا [۱۴] به تحلیل جریان داده[۲]، تفسیر انتزاعی[۳] و تحلیل نمادین[۴] اشاره خواهد شد.

تحلیل جریان داده یک فرایند برای جمع‌آوری اطلاعات زمان اجرا درباره داده‌های در برنامه‌ها است بدون اینکه واقعا آن‌ها را اجرا کند. گرچه تحلیل جریان داده از معناشناخت عملگرها استفاده نمی‌کند.

بلوک پایه[۵] دنباله‌ای از دستورالعمل‌های متوالی است که:

۱. کنترل فقط در ابتدای آن بلوک وارد می‌شود؛

۲. کنترل فقط در پایان آن بلوک (تحت اجرای عادی) خارج می‌شود؛

۳. کنترل نمی‌تواند متوقف شود مگر اینکه آن بلوک پایان یابد.

یعنی اگر هر دستورالعمل در یک بلوک پایه اجرا شود، سپس همه دستورالعمل‌ها در آن بلوک اجرا شده‌اند.

گراف کنترل جریان[۶] یک نمایش انتزاعی از یک رویه یا برنامه است. هر گره بیانگر یک بلوک پایه است. یال‌های جهت‌دار برای نمایش حرکت جریان کنترل است. اگر یک بلوک هیچ یال واردشونده‌ای نداشته باشد، بلوک ورودی، و اگر هیچ یال خارج‌شونده‌ای نداشته باشد، بلوک خروجی نامیده می‌شود.

مسیر کنترل جریان[۷] یک مسیر در گراف کنترل جریان است که از یک بلوک ورودی آغاز و به یک بلوک خروجی ختم می‌شود. ممکن است تعداد مسیرهای کنترل جریان ممکن به دلیل وجود حلقه در گراف، بی‌شمار باشد. یکی از کارهای اصلی در تحلیل بدترین اجرا، پیدا کردن طولانی‌ترین مسیر کنترل جریان ممکن است. تحلیل کنترل

[1] semantics
[2] data flow analysis
[3] abstract interpretation
[4] symbolic analysis
[5] basic block
[6] control flow graph
[7] control flow path

داده در دو مرحله انجام می‌شود. در مرحله اول واقعیت‌های مطلوب جمع‌آوری می‌شود. نتیجه این مرحله، n مجموعه gen و n مجموعه kill است که n بیانگر تعداد گزاره[8]های برنامه است. این مجموعه‌ها در مجموعه بعدی تحلیل جریان داده برای تشکیل و حل معادلات استفاده می‌شوند.

در ادامه، در تحلیل رو به جلو[9]، که در آن معادلات در طول انتقال اطلاعات از گزاره‌های آغازین به گزاره‌های پایانی تشکیل می‌شوند و در تحلیل رو به عقب[10]، که در آن معادلات در طول انتقال اطلاعات از گزاره‌های پایانی به گزاره‌های ابتدایی تشکیل می‌شوند؛ اصطلاحاتی مطرح می‌شود. اصطلاح‌های تحلیل عبارت موجود[11]و تحلیل تعاریف دستیابی[12] در تحلیل رو به جلو و اصطلاح‌های تحلیل عبارت خیلی‌مشغول[13] و تحلیل متغیر زنده[14] در تحلیل رو به عقب مطرح می‌شوند. با استفاده از این مفاهیم و مفاهیم دیگر، می‌توان تحلیل جریان داده را به کار برد. تحلیل جریان داده روشی بسیار کارا و عملی برای تحلیل برنامه است و این تحلیل برای تولید کد بهینه‌شده در کامپایلرها استفاده می‌شود. گرچه برای یافتن خطاهای ممکن در زمان اجرا یا محاسبه بدترین زمان اجرای یک برنامه به اندازه کافی قدرت‌مند نیست. در این روش، خیلی از قسمت‌های مهم اطلاعات جمع‌آوری نمی‌شود زیرا تحلیل کنترل داده از معناشناخت عملگرهای زبان برنامه‌نویسی بهره نمی‌برد.

تفسیر انتزاعی یک نظریه درباره تقریب معناشناخت است. ایده تفسیر انتزاعی، ایجاد یک معنای جدید از زبان برنامه‌نویسی است که آن معنا همیشه پایان می‌دهد و انبار نقطه‌های هر برنامه شامل یک ابرمجموعه[15] از مقادیر ممکن در معنای واقعی است، برای هر ورودی ممکن. در حالی که در این معنای جدید یک انبار شامل یک مقدار واحد برای یک متغیر نمی‌شود، بلکه شامل مجموعه‌ای از مقادیر ممکن است؛ ارزیابی بولین و عبارت‌های ریاضی باید دوباره تعریف شوند.

تحلیل نمادین یک روش تحلیل ایستا برای استدلال درباره مقادیر برنامه است که ممکن نیست ثابت باشند. این یعنی رسیدن به یک ویژگی‌های دقیق ریاضیاتی محاسبات و می‌توان به عنوان یک کامپایلر که یک برنامه را به یک زبان دیگر ترجمه می‌کند که این زبان شامل عبارات نمادین و بازگشتی‌های نمادین می‌شود. سیستم‌های

[8] statement
[9] forward
[10] backward
[11] Available Expression Analysis
[12] Reaching Definitions Analysis
[13] Very Busy Expression Analysis
[14] Live Variable Analysis
[15] superset

جبری کامپیوتری مانند اصل‌موضوعی[۱۶]، اشتقاق[۱۷]و کاهش[۱۸]؛ نقش مهمی در پشتیبانی از این روش را بازی می‌کنند چون کیفیت نتایج نهایی بستگی قابل توجهی به روش‌های ساده‌سازی هوشمند جبری دارد.

پس تحلیل جریان داده یک نوع خاصی از تحلیل برنامه است که معمولا برپایه یک معناشناخت انتزاعی آن برنامه قرار می‌گیرد. الگوریتم‌های تحلیل جریان داده برای ساختن نسبتا ساده هستند و معمولا از نظر محاسباتی امکان‌پذیر هستند.

تفسیر انتزاعی یک چارچوب کلی برای ساخت تکنیک‌های تحلیل برنامه است. هر نوعی از تحلیل برنامه می‌تواند به عنوان یک نمونه از تفسیر انتزاعی قلمداد کرد.

تحلیل نمادین یک روش برای رسیدن به ویژگی‌های دقیق خصوصیات برنامه طبق یک مسیر پارامتری است.

حال به مقایسه ابزارهای یافتن اشکال برای زبان برنامه‌نویسی جاوا می‌پردازیم. [۱۵]

در سال‌های اخیر، تعداد زیادی ابزار و تکنیک برای یافتن اشکال‌ها به طور خودکار با استفاده از تحلیل ایستای (در هنگام کامپایل) کد برنامه یا کد میانی تولید شده است. ابزارها و تکنیک‌های متفاوت، ویژگی‌های متفاوتی دارند، اما تاثیر عملی آن‌ها به خوبی درک نشده است. در اینجا پنج ابزار یافتن اشکال شامل Bandera [۱۶]، ESC/Java 2 [۱۷]، FindBugs [۱۸]، JLint [۱۹] و PMD [۲۰]روی برنامه‌های گوناگون جاوا بررسی می‌شود. آزمایش‌ها نشان می‌دهد که هیچ یک از این ابزارها موکدا نسبت به دیگری برتری ندارد و اغلب این ابزارها اشکال‌هایی را می‌یابند که با اشکال‌های یافته‌شده توسط دیگری هم‌پوشانی ندارد.

ابزارهای زیادی برای یافتن اشکال‌ها به طور خودکار از روی کد برنامه با استفاده از تکنیک‌هایی مانند تطبیق الگوی نحوی[۱۹]، تحلیل جریان داده، type systemها، بررسی مدل[۲۰] و اثبات نظریه[۲۱] توسعه یافته است. بسیاری از این ابزارها انواع مشابه‌ای از اشتباه‌های برنامه‌نویسی را بررسی می‌کنند.

با توجه به آزمایش‌های انجام شده، به طور مشخص، هیچ کدام از این ابزارهای یافتن اشکال، به تنهایی بهترین نیستند. اکثر این ابزارها تعداد زیادی هشدار[۲۲] گزارش می‌دهند که این باعث سخت شدن یافتن اشکال مهم‌تر می‌شود. گرچه این ابزارها هشدارهای دارای هم‌پوشانی را مشخص نمی‌کنند، مجموع هشدارهای گزارش‌شده را

[16] Axiom
[17] Derive
[18] Reduce
[19] syntactic pattern matching
[20] model checking
[21] theorem proving
[22] warning

مرتبط در نظر می‌گیریم. به عنوان مثال اگر یک ابزار هشدارهای زیادی برای یک کلاس[۲۳] تولید کرد، احتمالا ابزار دیگر نیز مانند این ابزار تولید خواهد کرد. گرچه آزمایش‌ها نشان می‌دهد که این عبارت به طور کلی درست نیست. هیچ ارتباطی بین تعداد هشدارها در ابزارهای مختلف وجود ندارد. همچنین، تعداد هشدارها به تعداد خطوط کد نیز وابستگی چندانی ندارد.

مطالب مطرح شده بیانگر لزوم مقایسه بین ابزارهای مختلف یافتن اشکال است. مطالعات این مقاله بر روی ابزارهای PMD، FindBugs و JLint که از تشخیص تطابق الگوی نحوی استفاده می‌کنند، است. ضمنا JLint و FindBugs از جزء جریان داده نیز بهره می‌برد. ضمنا بر روی ابزارهای ESC/Java که از اثبات نظریه و Bandera که از بررسی مدل بهره می برد، مطالعه انجام شده است.

برنامه‌های با اندازه‌های متفاوت با دامنه‌های متفاوت مورد بررسی توسط این ابزارها قرار گرفته‌اند. این یک نتیجه غیرقابل تصمیم‌گیری است که هیچ ابزار یافتن اشکالی نمی‌تواند همیشه نتایج صحیحی گزارش دهد. به عبارت دیگر، همه ابزارها باید تعادلی بین یافتن اشکال‌های درست با تولید مثبت‌های غلط[۲۴] (هشدارهایی در خصوص کد درست) و منفی‌های غلط[۲۵] (عدم هشدار در خصوص کد نادرست) ایجاد کنند.

تهدیدهای مختلفی برای اعتبار این مطالعه مطرح است. مهم‌ترین آن سادگی حوزه محدود این مطالعه است؛ هم در اندازه برنامه‌های تست و هم در انتخاب ابزارها. محدودیت دیگر در زبان برنامه‌نویسی است که تنها به جاوا پرداخته می‌شود، گرچه برای زبان‌هایی مثل C و C++ می‌توان از تکنیک‌های مشابهی بهره برد اما در حیطه مطالعات نیست.

تهدید دیگری برای اعتبار این است که دسته‌بندی دقیقی برای همه مثبت‌های غلط و منفی‌های غلط از ابزارها نشده است. انجام این کار بسیار سخت و پیچیده است. البته در ادامه نتایج مقایسه‌ای کلی برای بیان نحوه تاثیر پیاده‌سازی تکنیک‌ها بر روی هشدارها مطرح شده است.

تهدید بعدی برای اعتبار این است که تمایزی بین شدت یک اشکال در برابر دیگری قائل نشده است. بررسی شدت اشکال‌ها نیز یک مسئله مشکل دیگری است و مورد مطالعه این مقاله نبوده است. به عنوان مثال قطعه کد زیر را در نظر بگیرید:

[23] class
[24] false positives
[25] false negetives

```
int x = 2;
int y = 3;
if (x == y)
        if (y == 3)
                x = 3;
else
        x = 4;
```

در مثال اخیر، دندانه‌گذاری کد تلقین می‌کند که else مربوط به اولین if است، ولی گرامر زبان چیز دیگری را می‌گوید. این نتیجه بیشتر خطای منطقی است. چون یک برنامه‌نویس ممکن است فکر کند نتیجه این قطعه کد x=4 خواهد بود در حالی که نتیجه x=2 است. بسته به استفاده‌های بعدی از x، این مسئله ممکن است منجر به خطا شود. به دلیل استفاده از مجموعه قوانین برای اطمینان از صحت کد، که به استفاده از آکلادها برای مشخص کردن بدنه if اشاره دارد، PMD این برنامه را مشکوک می‌پندارد.

خطای مشخص‌تر زیر توسط JLint، FindBugs و ESC/Java تشخیص داده شده است:

```
String s = new String (“I’m not null yet!”);
s = null;
System.out.println(s.length());
```

این قطعه کد، قطعا منجر به تولید یک Exception در زمان اجرا می‌شود، که مطلوب نیست؛ اما باعث توقف برنامه به محض رخداد این خطا می‌شود (در شرایطی که این Exception لحاظ نشده باشد). علاوه بر این، اگر این قطعه کد در یک مسیر عمومی برنامه باشد، این خطا در زمان اجرای برنامه بیشتر پدید می‌آید و این Exception به محل دقیق روی دادن این خطا اشاره خواهد کرد.

بسیاری از برنامه‌نویس‌ها ممکن است ارجاع به null را بدتر از عدم استفاده از آکلادها در گزاره‌های if بدانند. گرچه این خطای منطقی به دلیل عدم وجود آکلادها ممکن است به نسبت ارجاع به null، شدیدتر و سخت‌تر قابل ردیابی باشد.

این مثال‌ها نشان می‌دهد که برای هر اشکال برنامه مشخص، شدت آن خطا نمی‌تواند از متن مورد استفاده در برنامه آن جدا باشد. جدول زیر به طور خلاصه مقایسه‌ای بین ابزارها ارائه می‌کند:

Name	Version	Input	Interface	Technology
Bandera	0.3b2 (2003)	Source	Command Line, GUI	Model Checking
ESC/Java	2.0a7 (2004)	Source[۲۶]	Command Line, GUI	Theorm Proving
FindBugs	0.8.2 (2004)	Bytecode	Command Line, GUI, IDE, Ant	Syntax, Dataflow
JLint	3.0 (2004)	Bytecode	Command Line	Syntax, Dataflow
PMD	1.9 (2004)	Source	Command Line, GUI, IDE, Ant	Syntax

جدول ۱. ابزارهای یافتن اشکال و ویژگی‌های اولیه آن‌ها

FindBugs یک تشخیص‌دهنده الگوی اشکال برای جاوا است. FindBugs از مجموعه‌ای از تکنیک‌های خاص طراحی‌شده برای توازن صحت، کارایی و قابلیت استفاده بهره می‌برد. یکی از تکنیک‌های اصلی که FindBugs استفاده می‌کند، تطابق نحوی کد برنامه برای مشکوک دانستن برنامه نوشته‌شده است که به گونه‌ای به ASTLog شبیه است.

به عنوان مثال، FindBugs بررسی می‌کند که فراخوانی‌های ()wait، که در برنامه‌های چندنخی[۲۷] جاوا استفاده می‌شوند، همیشه در داخل یک حلقه باشد، که در بیشتر موارد شکل استفاده صحیح است.

FindBugs همچنین در بعضی موارد، از تحلیل جریان داده برای بررسی اشکال‌ها استفاده می‌کند. به عنوان مثال، FindBugs از یک تحلیل ساده و درون‌رویه‌ای (در داخل یک متد) کنترل داده برای بررسی ارجاع‌های اشاره‌گر null استفاده می‌کند.

FindBugs می‌تواند با نوشتن تشخیص‌دهنده اشکال شخصی در جاوا گسترش یابد. FindBugs را می‌توان به عنوان یک ابزار با اولویت هشدارهای متوسط گزارش داد.

[26] may require bytecode or some files
[27] multi-threaded

JLint مانند FindBugs، Java Bytecode را تحلیل می‌کند، برنامه را از لحاظ نحوی بررسی می‌کند و از تحلیل کنترل داده استفاده می‌کند. JLint همچنین شامل یک جزء درون‌رویه‌ای و بین‌فایلی است، برای پیدا کردن بن‌بست‌ها به‌وسیله ساختن یک گراف قفل و اطمینان حاصل کردن از این‌که هیچ دوری در این گراف وجود ندارد. JLint 3.0، شامل پیوست‌های بررسی برنامه چندنخی است که توسط Artho تعریف شده است. [۲۱] JLint به سادگی قابل گسترش نیست.

PMD مانند FindBugs و JLint ، از بررسی نحوی روی متن کد برنامه استفاده می‌کند ولی جزء کنترل داده را ندارد. علاوه بر تشخیص بعضی از کدهای مشخصا اشتباه، بسیاری از اشکال‌هایی که PMD به دنبال سبکِ قراردادهایی[۲۸] است که تخطی از آن‌ها در بعضی موقعیت‌ها، مشکوک است.

به عنوان مثال، داشتن یک گزاره try با یک بلوک خالی catch، ممکن است بیانگر این باشد که از خطای لحاظ‌شده به نادرستی دست کشیده شده است. زیرا PMD شامل بسیاری از تشخیص‌دهنده‌های اشکال است که به سبک برنامه‌نویسی بستگی دارد. در مطالعات این مقاله، PMD با مجموعه قوانین پیشنهادی به‌وسیله مستنداتِ unusedcode.xml، basic.xml، import.xml و favorits.xml اجرا شده است.

تعداد هشدارها بسته به مجموعه قوانین مورد استفاده، می‌تواند افزایش یا کاهش یابد. PMD به سادگی توسط برنامه‌نویسانی که می‌توانند با Java یا XPath تشخیص‌دهنده‌های الگوی اشکال جدید بنویسند، قابل گسترش است.

Bandera یک ابزار تصدیق مبتنی بر بررسی مدل و انتزاع است. برای استفاده از Bandera، برنامه‌نویس باید کد برنامه خود را با مشخصاتی که باید بررسی شود، بنویسد؛ یا اگر برنامه‌نویس فقط می‌خواهد بعضی از مشخصات همگام‌سازی‌شده استاندارد تصدیق شود نیازی به نوشتن مشخصات دیگر نیست. به طور خاص، بدون نوشتن مشخصات توسط برنامه‌نویس، Bandera عدم وجود بن‌بست را تصدیق می‌کند. Bandera شامل برش‌زنی اختیاری و فازهای انتزاعی است که ناشی از بررسی مدل است. Bandera می‌تواند از انواع مختلفی از بررسی‌کننده‌های مدل مانند SPIN[۲۲] و Java PathFinder [۲۳] استفاده کند.

توسعه‌دهندگان Bandera در وب‌سایت خود نوشته‌اند که Bandera نمی‌تواند فراخوانی‌های کتابخانه‌های استاندارد جاوا را تحلیل کند. که این مسئله باعث کاهش قابلیت استفاده و کاربرد Bandera شده است.

[28] conventions

ESC/Java، سیستم بررسی ایستای توسعه‌یافته برای جاوا[۲۹]، بر پایه اثبات نظریه است که عملیات تصدیق فرمال[۳۰] ویژگی‌های کد برنامه به زبان جاوا را انجام می‌دهد. برای استفاده از ESC/Java، برنامه‌نویس پیش‌شرط‌ها، پس‌شرط‌ها و حلقه‌های ثابتی را به متن کد برنامه به صورت مجموعه‌ای از توضیح‌های[۳۱] خاص اضافه می‌کند. ESC/Java از یک اثبات‌کننده نظریه برای تصدیق برنامه‌ای که با مشخصات تطابق دارد، استفاده می‌کند.

ESC/Java می‌تواند حتی بدون تعریف مشخصات، خروجی قابل‌استفاده‌ای تولید کند. در این حالت، ESC/Java به دنبال خطاهای ارجاع اشاره‌گر null، خارج از محدوده بودن آرایه و مانند این‌ها می‌گردد. توضیحات بیشتر می‌تواند برای حذف مثبت‌های غلط و منفی‌های غلط یا اضافه کردن مشخصاتی برای بررسی، مورد استفاده قرار بگیرد.

ESC/Java به این دلیل در مطالعات این مقاله قرار گرفته است که روش آن برای یافتن اشکال‌ها به طور قابل توجهی با سایر ابزارها متفاوت است. البته عدم وجود توضیحات بیشتر در برنامه باعث تولید گروهی از هشدارها می‌شود. Houdini[۲۴] می‌تواند در ESC/Java به طور خودکار به برنامه‌ها توضیحات بیشتر را اضافه کند، ولی آن روی ESC/Java 2 کار نمی‌کند.

می‌توان همه اشکال‌هایی را که این ابزارها پیدا می‌کنند را به گروه‌هایی تقسیم‌بندی کرد که در جدول ۲ آمده است. ستون اول یک کلاس کلی و ستون دوم یک مثال مرسوم از آن کلاس را بیان می‌کند. ستون آخر بیانگر این است که کدام ابزار یافتن اشکال در آن دسته قرار می‌گیرد. به این دلیل Bandera در این جدول قرار نگرفته است که بدون افزودن توضیحات بیشتر، بررسی‌های آن محدود به ویژگی‌های همگام‌سازی است.

با توجه به این جدول، بیشترین هم‌پوشانی بین FindBugs و PMD است که شش دسته مشترک دارند. دسته عمومی برای بررسی‌هایی است که در هیچ یک از دسته‌های دیگر قرار نمی‌گیرد. بنابراین همه ابزارها چیزی در این دسته را پیدا می‌کنند. همه این ابزارها نیز به دنبال خطاهایی همروندی می‌گردند. در مجموع، دسته‌های مشترک زیادی در بین این ابزارها وجود دارد دسته‌های زیادی نیز در این ابزارها متمایز هستند.

[29] the Extended Static Checking system for Java
[30] formal verification
[31] comments

دو دسته‌بندیِ Orthogonal Defect Classification و IEEE Standard Classification for Software Anomalies بر روی کل فازهای چرخه حیات نرم‌افزار تاکید دارند که در این مطالعه مورد نظر قرار نگرفته است. به عنوان مثال، آن‌ها امکان این را دارند که تشخیص دهند که خطا ناشی از یک مشکل منطقی است، اما نمی‌توانند بیان کنند که این مشکل منجر به چه خطایی می‌شود یا از چه خطایی منجر شده است، مانند همگام‌سازی نادرست.

Bug Category	Example	ESC/Java	FindBugs	JLint	PMD
General	Null dereference	√*	√*	√*	√
Concurrency	Possible deadlock	√*	√	√*	√
Exceptions	Possible unexpected exception	√*			
Array	Length may be less than zero	√		√*	
Mathematics	Division by zero	√*		√	
Conditional, loop	Unreachable code due to constant guard		√		√*
String	Checking equality using == or !=		√	√*	√
Object overriding	Equal objects must have equal hashcodes		√*	√*	√*
I/O stream	Stream not closed on all paths		√*		
Unused or duplicate statement	Unused local variable		√		√*
Design	Should be a static inner class		√*		
Unnecessary statement	Unnecessary return statement				√*

√ - tool checks for bugs in this category * - tool checks for this specific example

جدول ۲. انواع اشکال‌های یافته‌شده توسط ابزارها

در جدول ۳، اندازه هر benchmark بر اساس گزاره‌های کد برنامه بدون توضیحات[۳۲] برحسب تعداد کاراکترهای ';' و '{' در برنامه و تعداد فایل‌های کلاس لیست شده است. سایر ستون‌های جدول ۳ زمان‌های اجرا و مجموع تعداد هشدارهای تولیدشده توسط هر ابزار را نشان می‌دهد. برای محاسبه زمان‌های اجرا، همه برنامه‌ها از خط دستور اجرا شده‌اند. برای هر ابزار تعداد دقیق هشدارهای تولیدشده، بدون هیچ نرمال‌سازی و تلاش برای کاهش تعداد هشدارهای تولیدشده، گزارش داده شده است.

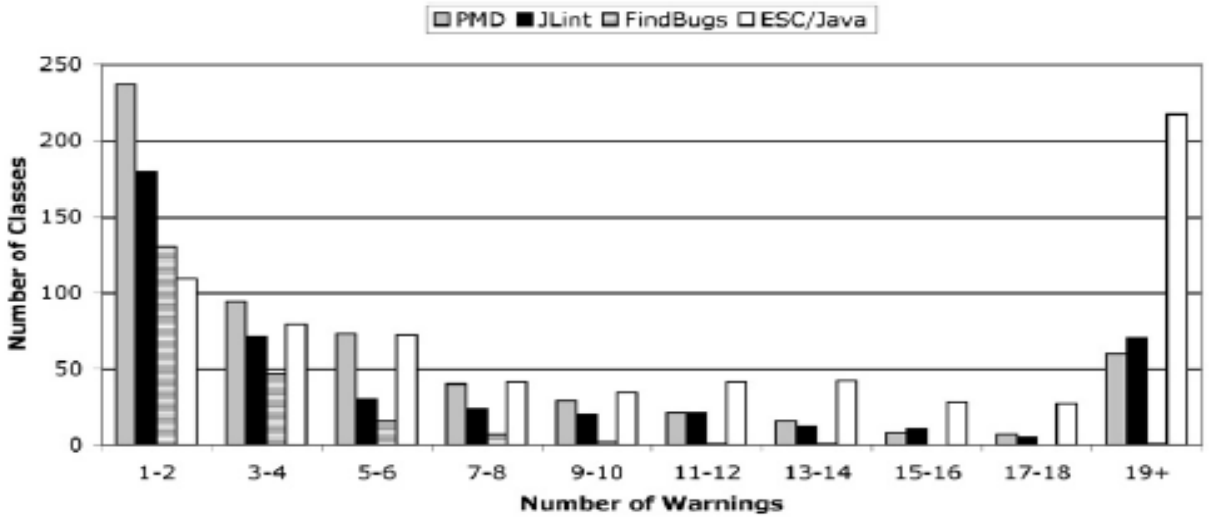
برای ESC/Java، تعداد هشدارهای تولیدشده در بعضی اوقات به‌شدت زیاد است. در بین سایر ابزارها، JLint و پس از آن، PMD، تمایل به گزارش بیشترین تعداد هشدار را دارند. FindBugs معمولا تعداد هشدار کمتری به نسبت سایر ابزارها گزارش می‌کند. به طور کلی، می‌توان گفت که FindBugs برای استفاده راحت‌تر است؛ به دلیل آن‌که تعداد نتایج کمتری باید بررسی شوند.

[32] Non Commented Source Statements (NCSS)

Name	NCSS (Lines)	Class Files	Time (min:sec.csec)				Warning Count			
			ESC/Java	FindBugs	JLint	PMD	ESC/Java	FindBugs	JLint	PMD
Azureus 2.0.7	35,549	1053	211:09.00	01:26.14	00:06.87	19:39.00	5474	360	1584	1371
Art of Illusion 1.7	55,249	676	361:56.00	02:55.02	00:06.32	20:03.00	12813	481	1637	1992
Tomcat 5.019	34,425	290	90:25.00	01:03.62	00:08.71	14:28.00	1241	245	3247	1236
JBoss 3.2.3	8,354	274	84:01.00	00:17.56	00:03.12	09:11.00	1539	79	317	153
Megamek 0.29	37,255	270	23:39.00	02:27.21	00:06.25	11:12.00	6402	223	4353	536

جدول ۳. زمان اجرا و تعداد هشدار تولیدشده برای هر ابزار

نمودار ۱ یک نمودار هیستوگرام[۳۳] از تعداد هشدارها بر کلاس ارائه می‌کند. مشخص است که در بسیاری از موارد، زمانی‌که این ابزارها اشکال‌های احتمالی را می‌یابند، آن‌ها فقط تعداد کمی را پیدا می‌کنند و تعداد کلاس‌هایی که چندین هشدار دارند به‌سرعت کاهش می‌یابد. برای PMD و JLint، تعداد نسبتا کمی کلاس وجود دارند که دارای ۱۹ هشدار یا بیشتر هستند؛ در حالی‌که برای FindBugs بسیار کمتر است. برای ESC/Java، بسیاری از کلاس‌ها دارای ۱۹ هشدار یا بیشتر هستند.



نمودار ۱. نمودار تعداد هشدار برحسب کلاس

واضح است که این ابزارها تعداد بسیار زیادی از هشدارها را تولید می‌کنند که باید به طور دستی بررسی شوند. حال می‌خواهیم به بررسی کارایی این ابزارها بر روی سه وظیفه‌ی بررسی[۳۴] که تعدادی از این ابزارها با یکدیگر اشتراک دارند، بپردازیم: همروندی، ارجاع null و خطاهای محدودیت آرایه. گرچه برای وظیفه یکسانی انواع

[33] histogram
[34] checking tasks

مختلف هشدار توسط ابزارهای مختلف گزارش شده است. جدول شماره ۴، شامل یک شکست تعداد هشدارها می‌باشد.

	ESC/ Java	Find Bugs	JLint	PMD
Concurrency Warnings	126	122	8883	0
Null Dereferencing	9120	18	449	0
Null Assignment	0	0	0	594
Index out of Bounds	1810	0	264	0
Prefer Zero Length Array	0	36	0	0

جدول ۴. تعداد هشدارها برای انواع دسته‌بندی‌های وظایف بررسی

همه ابزارها حداقل یک نوع از خطای همروندی را بررسی می‌کنند. ESC/Java شامل پشتیبان بررسی خودکار برای شرایط مسابقه و بن‌بست‌های احتمالی است. ولی ESC/Java هیچ شرایط مسابقه‌ای پیدا نکرد و ۱۲۶ هشدار بن‌بست برای benchmark تولید کرده است. پس از بررسی تعدادی از این هشدارها، مشخص شد که مقداری از آن‌ها مثبت‌های غلط هستند. بررسی‌های بیشتر دشوار هستند، چون‌که ESC/Java بلوک‌های synchronized که احتمال وقوع بن‌بست دارند را گزارش می‌دهد، ولی این مجموعه از قفل‌ها در بن‌بست خاصی قرار نداند.

PMD حاوی بررسی‌هایی برای بعضی الگوهای رایج اشکال، مانند اشکال well-known double-checked locking در جاوا است. FindBugs نیز مانند PMD، اشکال well-known double-checked locking را بررسی می‌کند.

FindBugs همچینین درباره رخداد سایر الگوهای اشکال همروندی، مانند قرار ندادن یک ناظر[۳۵] فراخوانی wait() در یک حلقه while، هشدار می‌دهد. مطالعات نشان می‌دهد که هشدارهای به‌دست‌آمده از گزارش‌های FindBugs، معمولا به‌درستی رخداد یک الگوی اشکال در کد را بیان می‌کند.

JLint نیز هشدارهای زیادی را درباره بن‌بست‌های احتمالی می‌دهد که در بیشتر موارد، هشدارهایی برای یک اشکال مشابه هستند.

[35] monitor

در بین این چهار ابزار، ESC/Java، FindBugs و JLint به بررسی ارجاعات null می‌پردازند. به طور قابل‌توجهی، هم‌پوشانی زیادی بین هشدارهای گزارش‌شده ابزارهای مختلف وجود ندارد.

در جاوا، فهرست کردن[۳۶] خارج از محدوده یک آرایه، یک Exception زمان اجرا است. در حالی‌که یک خطای محدوده در جاوا، ممکن است خطای صدمه‌انگیزی در C و C++ نباشد.

JLint و ESC/Java، بررسی‌هایی برای خطاهای محدودیت آرایه انجام می‌دهند مانند هشدارهای ارجاعات null، JLint و ESC/Java، همیشه هشدارهای مشابهی برای مکان‌های ثابتی از برنامه نمی‌دهند.

FindBugs و PMD بررسی‌ای بر روی خطاهای محدودیت‌های آرایه نمی‌کنند، گرچه FindBugs درباره برگرداندن null از یک متد که یک آرایه را برمی‌گرداند، هشدار می‌دهد.

تا این‌جا نتایج حاصل از به‌کار بستن پنج ابزار یافتن اشکال به برنامه‌های جاوا بررسی شد. گرچه هم‌پوشانی‌هایی بین انواع اشکال‌های پیداشده توسط ابزارها وجود داشت، ولی بیشتر هشدارها متمایز بودند. می‌توان با ارائه یک meta-tool تعداد هشدارهای غیرمرتبط را تا حدی کاهش داد. با اجرای ابزارها و مقایسه خروجی‌ها، به نظر می‌رسد که به طور کلی چیزهای کمی می‌تواند مفید باشد. مشکل اصلی در استفاده از این ابزارها به اندازه تعداد خروجی‌ها است. بهتر است برنامه‌نویس بتواند برای مانع شدن از وجود مثبت‌های غلط، که ممکن است منجر به ایجاد خطاهایی در آینده شود، تفسیرها یا توضیحاتی خاص را به کد اضافه کند.

در پایان این مقاله آمده است که یادآوری می‌شود که این ابزارها به نحو نادرستی استفاده شده‌اند. بنابراین عدم وجود هشدارها در هر یک از ابزارها به معنای عدم وجود خطا نیست. این قطعا یک تعادل است، زیرا در این‌جا تنها تعداد هشدارهای تولیدشده توسط یک ابزار مورد بررسی قرار گرفته است.

بدون افزودن توضیحات کاربر به ابزاری مثل ESC/Java، آن ابزار بهتر می‌تواند هشدارهای برای تصدیق بیشتری به نسبت JLint، PMD و FindBugs تولید کند. البته هنوز یافتن یک تناسب درست بین ابزارهای یافتن اشکال قابل تحقیق و بررسی است.

[36] indexing

روش پیشنهادی:

در این مقاله، یک الگوی اشکال جدید از استفاده ناامن از متغیرها مطرح می‌شود که منجر به روی دادن Exception می‌شود. کد کردن Exception Handling با بی‌دقتی، باعث ایجاد اشکال‌هایی می‌شود و معمولا به شکل یک نوع الگوی اشکال است. ضمنا یک روش برای تشخیص استفاده ناامن از متغیرها که می‌تواند اشکال‌های احتمالی در برنامه‌های جاوا را معرفی کند، ارائه شده است. این روش می‌تواند با ابزارهای تشخیص اشکال فعلی، که در قبل به تعدادی از آن‌ها اشاره شد، برای کمک به برنامه‌نویس برای تولید برنامه با کیفیت به زبان جاوا، تجمیع شود.

هدف این است که به طور خودکار، مغایرت‌ها با امنیت وابستگی در برنامه‌های جاوا در حالت[۳۷] متغیر تشخیص داده شود. بعضی از شکست‌های برنامه، یافتن و ردیابی آن‌ها بسیار سخت است، چون ممکن است فقط در زمانی که واقعا رخ می‌دهند، قابل تشخیص باشند.

ابتدا یک تعریف فرمال برای استفاده ناامن[۳۸] بر پایه گسترش عملگرها بر روی متغیرهای استفاده‌شده در تحلیل جریان داده مرسوم ارائه می‌شود.

در تحلیل جریان داده مرسوم، عملگرها روی متغیرها به دو دسته تقسیم می‌شوند: define، use و kill

عملگرهای define، در گزاره‌هایی حالت متغیر را تغییر می‌دهند، رخ می‌دهند؛ مثل assignment.

عملگرهای use در گزاره‌هایی رخ می‌دهد که منجر به گرفتن یک مقدار از یک متغیر می‌شود.

عملگرهای kill وقتی رخ می‌دهد که متغیر آزاد شده است و دیگر قابل دسترسی نیست.

حالت متغیر ممکن است در هنگام رخ دادن Exception تغییر کند.

عملگرهای define به دو دسته زیر تقسیم می‌شوند:

۱) sDef؛ نوعی از عملگرهای define است که زمانی رخ می‌دهد که حالت متغیر به‌درستی و با موفقیت تغییر کند.

۲) eDef؛ نوعی از عملگرهای define است که زمانی رخ می‌دهد که حالت متغیر به دلیل وقوع Exception، با موفقیت و به درستی تغییر نکند.

[37] state
[38] unsafe use

در طول اجرا، عملگرهای define یا sDef یا eDef هستند.

استفاده ناامن، نوعی از عملگرهای use در مسیر اجرا است که با یک عملگر eDef در قبل ظاهر می‌شود و هیچ عملگر define دیگری بین این دو عملگر رخ نمی‌دهد. عملگر استفاده ناامن، باعث نقض ویژگی امنیت وابستگی[۳۹] می‌شود.

برای یافتن زوج‌های eDef در برنامه‌ها مانند تحلیل زوج‌های define-use مرسوم، ابتدا برای برنامه جاوا با ساختار Exception Handling، گراف کنترل جریان می‌سازیم و سپس زوج‌های eDef را با الگوریتم تحلیل جریان داده‌ی سلسله‌مراتبی تشخیص می‌دهیم.

الف) ساختن گراف کنترل جریان

در برنامه‌ی به زبان جاوا، Exceptionها از دو طریق برمی‌آیند[۴۰]:

مستقیما با گزاره‌های throw، برمی‌آیند یا غیرمستقیم از طریق فراخوانی‌های متد، برمی‌آیند. وقتی یک Exception در یک بلوک try برمی‌آید، کنترل به قسمت catchای که پارامتر آن با نوعِ Exception برآمده، هم‌خوانی داشته‌باشد، منتقل می‌شود. سپس کد در قسمت catch مذکور اجرا می‌شود. اگر با پارامتر هیچ بلوک catchای هم‌خوانی نداشته‌باشد، Exception، خارج از این تابع منتشر می‌شود یا باعث خروج ناگهانی برنامه می‌شود. بعد از اجرا شدن کد handler، اجرای عادی در اولین گزاره دقیقا بعد از یک بلوک try-catch که Exception برآمده بود؛ ادامه پیدا می‌کند. قبل از این‌که کنترل از گزاره try خارج شود، بلوک finally مربوط به گزاره try، در صورت وجود، اجرا می‌شود.

کنترل جریان آگاه‌به‌متنِ statement-level با مسیرهای اضافی که شرایط Exception را توصیف می‌کند، ساخته می‌شود.

در ابتدا، یک گراف کنترل جریان ساخته می‌شود که بیانگر جریان کنترل در طول اجرای عادی و سپس گراف کنترل جریان با یال‌های اضافی که بیانگر جریان کنترل، بعد از رخ دادن Exceptionها است، اضافه می‌شود. به این یال‌های اضافی عنوان یال‌های Exception و به سایر یال‌ها، یال عادی نسبت داده می‌شود که یال‌های Exception به‌صورت نقطه‌چین مشخص می‌شوند.

گره‌های Exceptional-exit برای بیان انتشار Exceptionها استفاده می‌شود و با نوع Exception پرتاب‌شده، برچسب‌گذاری می‌شود.

[39] dependency safety property
[40] raise

ب) تحلیل جریان داده‌ی سلسله‌مراتبی:

طبق تعاریف، تعریف عملگر بستگی به حالات متغیرها دارد. نوع‌های داده‌ی متغیرها در برنامه جاوا، به دو دسته تقسیم می‌شود:

اول primitive data type و دوم abstract data type. برای primitiveها، مانند int، حالت متغیر، مقدار آن است. برای abstractها مانند Object، حالت متغیرها مجموعه‌ای از حالت‌های همه صفت‌های آن است.

در برنامه به زبان جاوا، object توسط مرجعِ[41] آن کنترل می‌شود. در این مقاله، جداگانه درباره مرجع و objectای که رجوع می‌کند، بحث می‌شود.

برای یک متغیر با نوع اولیه[42]، عملگر define در گزاره‌هایی رخ می‌دهد که مقادیر آن‌ها می‌تواند تغییر کند. برای یک متغیر abstract data type، define زمانی رخ می‌دهد که حداقل یکی از حالت صفت‌های آن تغییر کند. مخصوصا ساختن یک object و نسبت دادن به مرجع آن، به‌عنوان عملگر define آن object رفتار می‌شود.

برای یک متغیر با نوع اولیه، use در گزاره‌هایی رخ می‌دهد که بدون تغییر مقدارش، به آن ارجاع داده می‌شود. برای یک متغیر abstract data type، use در گزاره‌هایی رخ می‌دهد که بدون هیچ تغییر در حالت هر صفت آن، نسبت داده می‌شود.

سپس یک الگوریتم برای تشخیص unsafe use ارائه شده است. این الگوریتم یک گسترش از تحلیل جریان داده است که عملگرهای جدید sDef و eDef معرفی شده‌اند.

گام‌های ابتدایی شرح داده می‌شود و الگوریتم به طور کامل در شکل شماره ۱ است:

- تولید مجموعه متغیرها برای هر گزاره:

براساس گراف کنترل جریان، ابتدا عملگرهای روی هر گره را با توجه به تعریف‌ها تحلیل می‌شوند. اگر حالت متغیر در آن گره تغییر نکرد، آن به مجموعه useها اضافه می‌شود. در غیر این‌صورت، آن به مجموعه sDef اضافه می‌شود. اگر Exceptionها ممکن است رخ دهد، متغیرهای مربوط به مجموعه eDef اضافه می‌شود.

برای تشخیص دادن sDef و eDef، بر روی یال خروجی از آن گره، برچسب‌گذاری می‌شود.

- تولید ردهای[43] عملگر برای هر متغیر:

[41] reference
[42] primary
[43] traces

رد عملگر برای یک متغیر، یک دنباله‌ای از عملگرها در طول مسیر اجرا است. می‌تواند از طریق پیمایش گراف کنترل جریان تولید شود. مسیر اجرا توسط گراف کنترل جریان تعیین می‌شود. هر متغیر یک رد عملگر بر روی هر مسیر اجرا دارد.

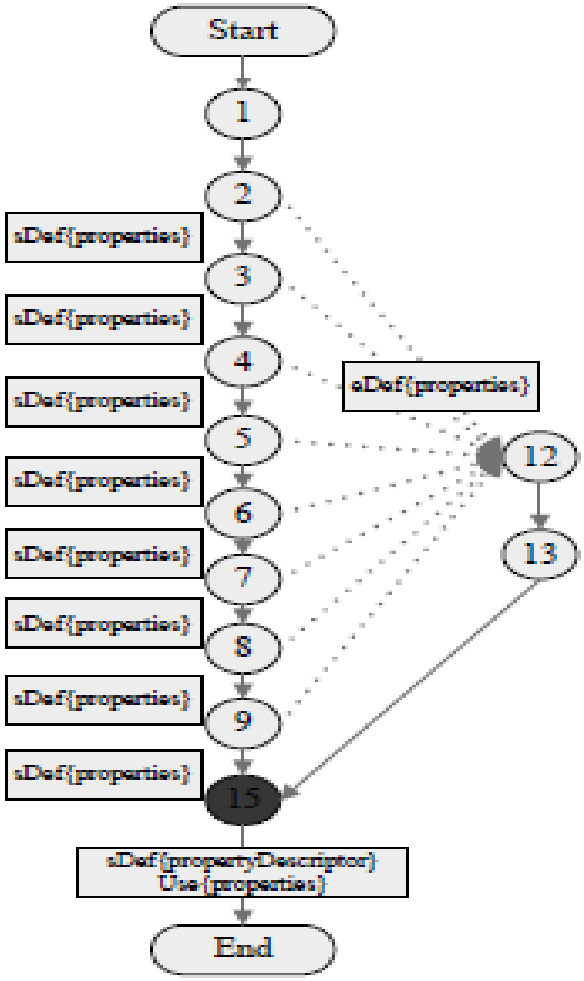
- تشخیص جفت‌های eDef-use در هر رد و تعیین کردن unsafe use:

این قسمت از الگوریتم با یک مثال در مقاله توضیح داده شده است.

```
Algorithm: detect unsafe use of variables for method M
Input:     Control flow graph of M
Output:    Nodes where unsafe use of variables occur

Begin
/* Step1: Generate variable set for the operator of sDef,
eDef, Use and Kill */
      For (each node in control flow graph){
            Divide each variable into the set of sDef, eDef,
        Use and Kill.
      }
/* Step2: Generate operation traces for each variable */
      Traverse the control flow graph to generate the
  operation traces for each variable;
/* Step3: Detect unsafe use on each trace */
      For (each trace) {
            Detect all appearance of EU pairs.
            For (each appearance of EU in the trace) {
                  Locate unsafe use node in program;
      }}
End
```

شکل شماره ۱. الگوریتم تشخیص unsafe use



(a)

Variable	Path	Trace
properties	1,2,3,4,5,6,7,8,9,15	SSSSSSSSU
	1,2,12,13,15	SSU
	1,2,3,12,13,15	SSSU
	1,2,3,4,12,13,15	SSSSU
	1,2,3,4,5,12,13,15	SSSSSU
	1,2,3,4,5,6,12,13,15	SSSSSSU
	1,2,3,4,5,6,7,12,13,15	SSSSSSSU
	1,2,3,4,5,6,7,8,12,13,15	SSSSSSSSU
	1,2,3,4,5,6,7,8,9,12,13,15	SSSSSSSSSU

(b)

شکل ۲. مثالی از یک گراف کنترل جریان پردازش‌شده

آزمایش‌ها و نتایج:

در این مقاله، روش مورد نظر تحت مثال‌هایی توضیح داده شده است. در واقع، این مثال‌ها به عنوان آزمایش‌های این مقاله هستند. آزمایش‌ها بر روی قسمت‌هایی از کد موجود در پکیج org.hsqldb.util.QueryTool و بخش‌هایی از کد پکیج com.daffodilwoods.daffodildb.server.datadictionarysystem.information و متد add(E e) از پکیج java.util.Vector صورت گرفته است.

در ادامه داده‌های مورد آزمایش این مقاله آمده است.

```
1 package com.daffodilwoods.daffodildb.server.datadictionarysystem.information;
2
3
4 /**
5  * Title:
6  * Description:
7  * Copyright: Copyright (c) 2002
8  * Company:
9  * @author
10  * @version 1.0
11  */
12 import java.util.*;
13 import com.daffodilwoods.database.resource.*;
14 import com.daffodilwoods.daffodildb.server.datadictionarysystem.information.*;
15 import java.beans.*;
16 import com.daffodilwoods.daffodildb.server.sql99.utils._Reference;
17 public class _TriggerInfoBeanInfo extends SimpleBeanInfo{
18
19   public _TriggerInfoBeanInfo() {
20   }
21   public PropertyDescriptor[] getPropertyDescriptors() {
22     Vector properties = new Vector();
23     try{
24        properties.add(new PropertyDescriptor("Name",_TriggerInfo.class,"getName",null));
25        properties.add(new PropertyDescriptor("ActionTime",_TriggerInfo.class,"getActionTime",null));
26        properties.add(new PropertyDescriptor("TriggerEvent",_TriggerInfo.class,"getTriggerEvent",null)
);
27        properties.add(new PropertyDescriptor("ActionOrientation",_TriggerInfo.class,"getActionOrient
```

```
ation",null));
28      properties.add(new PropertyDescriptor("WhenCondition", _TriggerInfo.class,"getWhenCondition
",null));
29      properties.add(new PropertyDescriptor("TriggerStatements", _TriggerInfo.class,"getTriggerState
ments",null));
30      properties.add(new PropertyDescriptor("OldAlias", _TriggerInfo.class,"getOldAlias",null));
31      properties.add(new PropertyDescriptor("NewAlias", _TriggerInfo.class,"getNewAlias",null));
32
33    }
34    catch(Exception  JavaDoc  e){
35      e.printStackTrace();
36    }
37    PropertyDescriptor []propertyDescriptor = new PropertyDescriptor[properties.size()];
38    properties.toArray(propertyDescriptor);
39    return propertyDescriptor;
40  }
41
42
43 }
```

```
1  try {
2     cConn = DriverManager.getConnection(url + database, user, password);
3  } catch (Exception e) {
4     System.out.println("QueryTool.init: " + e.getMessage());
5     e.printStackTrace();
6  }
7  sRecent = new String[iMaxRecent];
8  iRecent = 0;
9
10 try {
11    sStatement = cConn.createStatement();
12 } catch (SQLException e) {
13    System.out.println("Exception: " + e);
14 }
```

```
1  public synchronized boolean add(E e) {
2      modCount++;
3      ensureCapacityHelper(elementCount + 1);
4      elementData[elementCount++] = e;
5      return true;
6  }
```

این روش بر پایه چارچوب تحلیل Soot پیاده‌سازی شده است.

به این ترتیب که گراف کنترل جریان برای هر متد ساخته می‌شود. تحلیلگر Exception در چارجوب تحلیل Soot، Exceptionها را پیدا می‌کند و مشخص می‌کند که Exceptionهای رخ داده شده، در کدام گزاره و از چه نوعی هستند.

برای رسیدن به نتیجه آزمایش باید سوال‌های زیر پاسخ داده شوند:

- آیا استفاده ناامن از متغیرها در کد واقعی وجود دارد؟

- آیا هشدارها بیانگر اشکال‌های واقعی هستند؟

- آیا این اشکال‌ها، قابل گزارش توسط این ابزار هستند؟

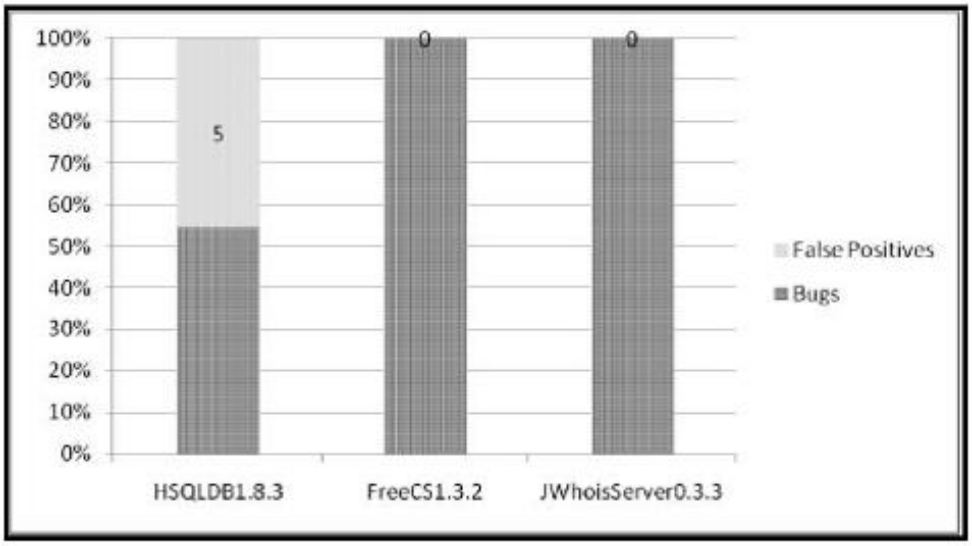
با انجام آزمایش بر روی HSQLDB مشاهده شد که ۱۱ هشدار گزارش شده و برای FreeCS و JWhoisServer، هر کدام فقط یک هشدار گزارش شده است.

Project Name	Packages	Classes	Lines of code
HSQLDB1.8.3	14	258	143291
FreeCS1.3.2	13	140	29949
JWhoisServer0.3.3	4	28	7603

جدول ۵ - برنامه‌های مورد استفاده در آزمایش

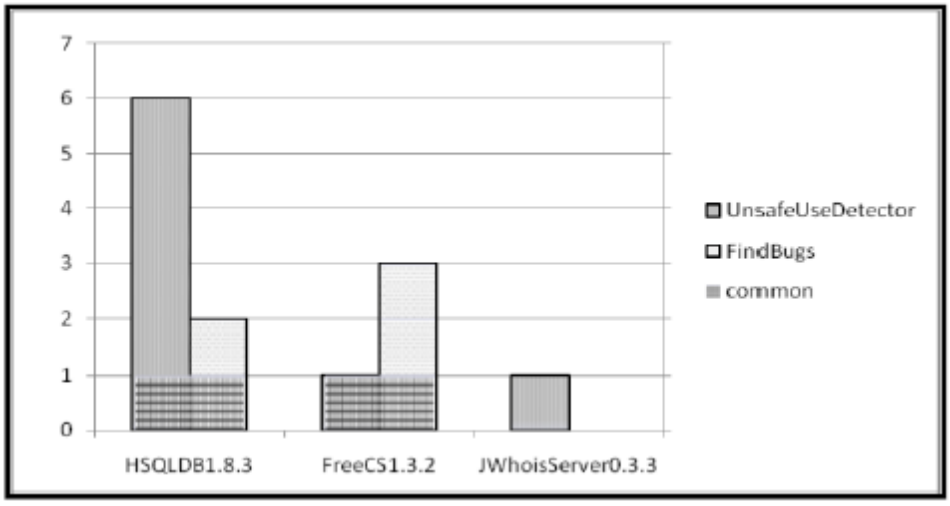
Project Name	Warnings	Bugs	False Positives
HSQLDB1.8.3	11	6	5
FreeCS1.3.2	1	1	0
JWhoisServer0.3.3	1	1	0

جدول ۶ - تعداد اشکال‌ها و مثبت‌های غلط در آزمایش‌ها



نمودار ۲ - درصد اشکال‌ها و مثبت‌ها غلط

در نمودار ۳، تفاوت بین ابزار معرفی شده و ابزار FindBugs مشخص می‌شود. FindBugs برای HSQLDB دو هشدار را گزارش کرده است که هر دوی آن‌ها اشکال هستند که یکی از آن‌ها توسط ابزار ما گزارش شده در حالی‌که دیگری گزارش نشده است. پنج هشدار دیگری که ابزار ما گزارش داده، توسط FindBugs یافت نشده است. برای FreeCS، FindBugs سه هشدار را گزارش داده که یکی از آن‌ها اشکال بوده است که آن نیز توسط ابزار ما گزارش شده است. پس FindBugs، دو مثبت غلط داشته است.



نمودار ۳ - نتایج حاصل از تحلیل ابزار معرفی شده با FindBugs

FindBugs نمی‌تواند اثرات مقادیر خطادار به وسیله Exceptionها در مسیرهای Exceptionای، به درستی تشخیص و در برنامه انتشار دهد. بخاطر همین تعدادی از این نوع باگ‌ها را پیدا نمی‌کند در حالیکه ابزار ما می‌تواند.

```
try {
        pwDsv = new PrintWriter((charset == null)
                ? (new OutputStreamWriter(new
                FileOutputStream(dsvFile)))
                : (new OutputStreamWriter(new
                FileOutputStream(dsvFile),charset)));
} catch (FileNotFoundException e) {
        throw new
        BadSpecial(rb.getString(SqltoolRB.FILE_WRITEFAIL,
                other), e);
} catch (UnsupportedEncodingException e) {
        throw ne
        w BadSpecial(rb.getString(SqltoolRB.FILE_WRITEFAIL,
                other), e);
} finally {
        if (pwDsv != null) {
                pwDsv.close();
        }
}
```

شکل ۳- قسمتی از کد آزمایش‌شده HSQLDB

نتیجه‌گیری و کارهای آینده:

در تحلیل کنترل جریان، Exception typeهایی که غیرمستقیم برمی‌آیند، به‌وسیله اینترفیسِ[۴۴] متد، که تکنیک استفاده آن مثل استنتاج نوع[۴۵] برای پالایش تحلیل نیست، تعیین می‌شود؛ زیرا توسعه‌دهندگان معمولا برنامه‌هایشان را با اینترفیس می‌نویسند. این باعث مشکل‌شدن برنامه می‌شود و این‌جا از تحلیل آگاه‌به‌متن برای ساختن صحیح‌تر کنترل جریان استفاده شده است. در این مقاله، Exceptionهای بررسی‌نشده[۴۶]، نام مستعار[۴۷] متغیرها و محدود کردن پویا[۴۸] در نظر گرفته نشده است. در این مقاله، قرار بود یک الگوی اشکال جدید برای استفاده ناامن از متغیرها ارائه شود که ممکن است ویژگی امنیت وابستگی برنامه‌ها را نقش کند. یک روش برای تشخیص خودکار آن در برنامه‌های جاوا با تحلیل ایستا، داده شد. این الگوریتم می‌تواند به ابزارهای یافتن اشکال مانند FindBugs اضافه شود تا به برنامه‌نویس‌ها برای افزایش کیفیت برنامه‌هایشان، کمک کند.آژیر کاذب[۴۹]، یکی از عوامل کلیدی برای تاثیر عملی‌بودن این روش است. به دلیل محافظه‌کارانه بودن روش ارائه شده، امکان وقوع آژیر کاذب و غلط‌های مثبت بیشتر می‌شود. برای بررسی استفاده ناامن از متغیرها در برنامه‌های متن‌باز[۵۰] و یافتن راه‌هایی برای کاهش آژیر کاذب باید تحقیق و بررسی شود.

[44] interface
[45] type inference
[46] unchecked
[47] alias
[48] dynamic binding
[49] False Alarm
[50] open source

تشکر و قدردانی:

از پدر و مادر و خانواده خود بسیار سپاسگزارم که شرایط تهیه این گزارش را فراهم آوردند و بدون کمک آن‌ها تهیه این گزارش امکان‌پذیر نبود.

از استاد گرانقدرم، جناب آقای دکتر شیری کمال تشکر را دارم، چرا که بدون راهنمایی‌های ایشان، تهیه این گزارش و یادگیری شیوه نگارش یک گزارش فنی، بسیار مشکل بود.

همچنین از دوستان و دانشجویان دانشکده مهندسی کامپیوتر و فناوری اطلاعات دانشگاه صنعتی امیرکبیر نیز کمال سپاس را دارم که با کمک و یاری آن‌ها و نظراتشان، به روند ارائه این گزارش سرعت بخشیدند.

مراجع:

[1] B. Ryder, et al., "A Static Study of Java Exceptions Using JESP," in Lecture Notes In Computer Science. vol. 1781, ed Heidelberg: Springer, 2000, pp. 67-81.
[2] B. Cabral and P. Marques, "Exception Handling: A Field Study in Java and .NET," in ECOOP 2007 – Object-Oriented Programming. vol. 4609/2007, ed Heidelberg: Springer 2007, pp. 151-175.
[3] B. Jacobs and F. Piessens, "Failboxes: Provably safe exception handling," ECOOP 2009¨CObject-Oriented Programming, pp. 470-494, 2009.
[4] D. Hovemeyer and W. Pugh, "Finding bugs is easy," SIGPLAN Not., vol. 39, pp. 92-106, 2004.
[5] Pierce, Benjamin C., Types and Programming Languages, MIT Press, Massachusetts, 2002.

[6] Pressman, Roger S., Software Engineering A Practitioner's Approach, The McGraw-Hill Companies, Inc, 2010.

[7] C. Artho and A. Biere, "Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs," in Proceedings of the 13th Australian Conference on Software Engineering, 2001, p. 68.
[8] Fingbugs. Available: http://findbugs.sourceforge.net/

[9] N. Ayewah, et al., "Using Static Analysis to Find Bugs," Software, IEEE, vol. 25, pp. 22-29, 2008.
[10] S. Sinha ,and M.J.Harrold, "Analysis and Testing of Programs with Exception-Handling Constructs, Software Engineering, IEEE Transactions on (Volume:26 , Issue: 9), 2000.

[11] Static Detection of Unsafe Use of Variables in Java, Ubiquitous Intelligence & Computing and 7th International Conference on Autonomic & Trusted Computing (UIC/ATC), pp. 439-443, 2010.

[12] M. Bernard, "The theory of computation", Addison-Wesley, 1998. ISBN 0-201-25828-5
[13] A.M. Turing, "On Computable Numbers, with an Application to the Entscheidungs problem", Proceedings of the London Mathematical Society, Series 2, 42 (1936-37), pp.230-265.
[14] W. Wosgerer, "A Survey of Static Program Analysis Techniques", Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , Volume:27, Issue: 7 , 2005.

[15] N. Rutar, et al., "A Comparison of Bug Tools for Java", Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, 2004, pp. 245-256.

[16] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code", In

Proceedings of the 22nd International Conference on Software Engineering, 2000, pp. 439–448.
[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java", In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 234–245.
[18] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy", available on: http://www.cs.umd.edu/˜pugh/java/bugs/docs/findbugsPaper.pdf, 2003.
[19] JLint, http://artho.com/jlint.
[20] PMD/Java, http://pmd.sourceforge.net.
[21] C. Artho, "Finding faults in multi-threaded programs", Master's thesis, Institute of Computer Systems, Federal Institute of Technology, 2001.
[22] G. J. Holzmann, "The model checker SPIN", *Software Engineering*, 23(5), 1997, pp.279–295.
[23] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2000, pp. 366–381.
[24] C. Flanagan and K. R. M. Leino. Houdini, "an Annotation Assistant for ESC/Java", *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods*, number 2021 in Lecture Notes in Computer Science, 2001, pp. 500–517.
[25] X.Wu, et al., "Static Detection of Bugs Caused by Incorrect Exception Handling in Java Programs", 11th International Conference On Quality Software, 2011.