Mở rộng khả năng phát hiện lỗ hổng bảo mật SedSVD cho ngôn ngữ Java

ACM Reference Format:
. 2025. Mở rộng khả năng phát hiện lỗ hổng bảo mật SedSVD cho ngôn ngữ Java. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Motivation

Phát hiện lỗ hổng bảo mật trong mã nguồn là một thách thức quan trọng trong phát triển phần mềm.

Công trình **SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding** đề xuất phương pháp hiệu quả để phát hiện lỗ hổng bảo mật ở mức độ câu lệnh, đạt chỉ số F1-measure 95,15%. Tuy nhiên, mô hình này hiện chỉ được áp dụng cho C/C++, như đã được chỉ ra trong phần `Threats to validity`: *"Vì mô hình chủ yếu nhắm vào các chương trình C/C++, kết quả phát hiện có thể bị hạn chế."*

Dự án này tập trung vào việc mở rộng SedSVD cho ngôn ngữ Java - một trong những ngôn ngữ phổ biến nhất trong phát triển phần mềm hiện nay. Chúng em sẽ tận dụng khả năng mở rộng sẵn có để tăng tính tổng quát của mô hình mà không phải phát triển công cụ phân tích mới, như tác giả đã đề cập: *"vì mô hình chúng em thiết kế không phụ thuộc vào ngôn ngữ lập trình, nên có thể mở rộng và linh hoạt".*

2 Timeline

Hình 1 thể hiện timeline dự án của chúng em. Tại thời điểm viết báo cáo này, chúng em đã hoàn thành quá trình Thu thập Dữ liệu (Data Collection) và Xây dựng Code Property Graph (CPG) và đang thực hiện chọn node trung tâm cho đồ thị con, tạo đồ thị con và nhúng các đồ thị con. Trong tương lai gần, chúng em sẽ thực hiện cùng các quá trình đã đề cập với ngôn ngữ C và C++ và cuối cùng là huấn luyện mô hình máy học.

3 Method (Data Collection & Preprocessing)

3.1 Source Code

Nguồn dữ liệu từ SARD[1], thu thập qua API. Tập trung vào ba ngôn ngữ là C, C++ (từ bài báo chính) và Java (ngôn ngữ nhóm đề xuất mở rộng), với số lượng mẫu từ 3.000 đến 4.000 mẫu cho mỗi ngôn ngữ. Các truy vấn API được lọc theo:

- Ngôn ngữ lập trình (`language`)
- Loại mẫu (`state`): *good* (không chứa lỗi), *bad* (có chứa lỗ hổng bảo mật), và *mixed* (chứa cả đoạn mã lỗi và mã đã vá).

Mỗi mẫu dữ liệu trong SARD là một file zip chứa, trong đó bao gồm:(1) các *tệp mã nguồn* của ngôn ngữ tương ứng, (2) tệp *manifest.sarif* chứa metadata (CWE ID, mô tả lỗi, dòng mã lỗi, và loại mẫu). Sau khi tải về, dữ liệu được xử lý như sau:

- Trích xuất mã nguồn chính từ mẫu dữ liệu, dựa trên các tệp **.c**, **.cpp**, **.java**.
- Đọc tệp *manifest.sarif* để xác định số hiệu dòng chứa lỗ hổng dựa trên thẻ **<region>** nằm trong thẻ **<locations>** để phục vụ huấn luyện mô hình phát hiện lỗi ở cấp độ dòng.

3.2 Graph Construction

Sau khi đã thu thập xong dữ liệu từ các nguồn của các ngôn ngữ C/C++ và Java, sử dụng công cụ **Joern** để tiến hành phân tích cú pháp. Mỗi mẫu thư mục thu thập được chứa các tập tin mã nguồn C/C++ hay Java, coi nó như một dự án (Project), sau đó phân tích cú pháp các tập tin có trong thư mục Project đó và dùng Joern để xuất **Control Property Graph (CPG)** dưới dạng một tập tin **.dot**.

Joern tạo đồ thị dựa trên khái niệm Code Property Graphs. Code Property Graphs là đồ thị, và cụ thể hơn là property graphs. Một property graph được cấu tạo từ các thành phần cơ bản sau:

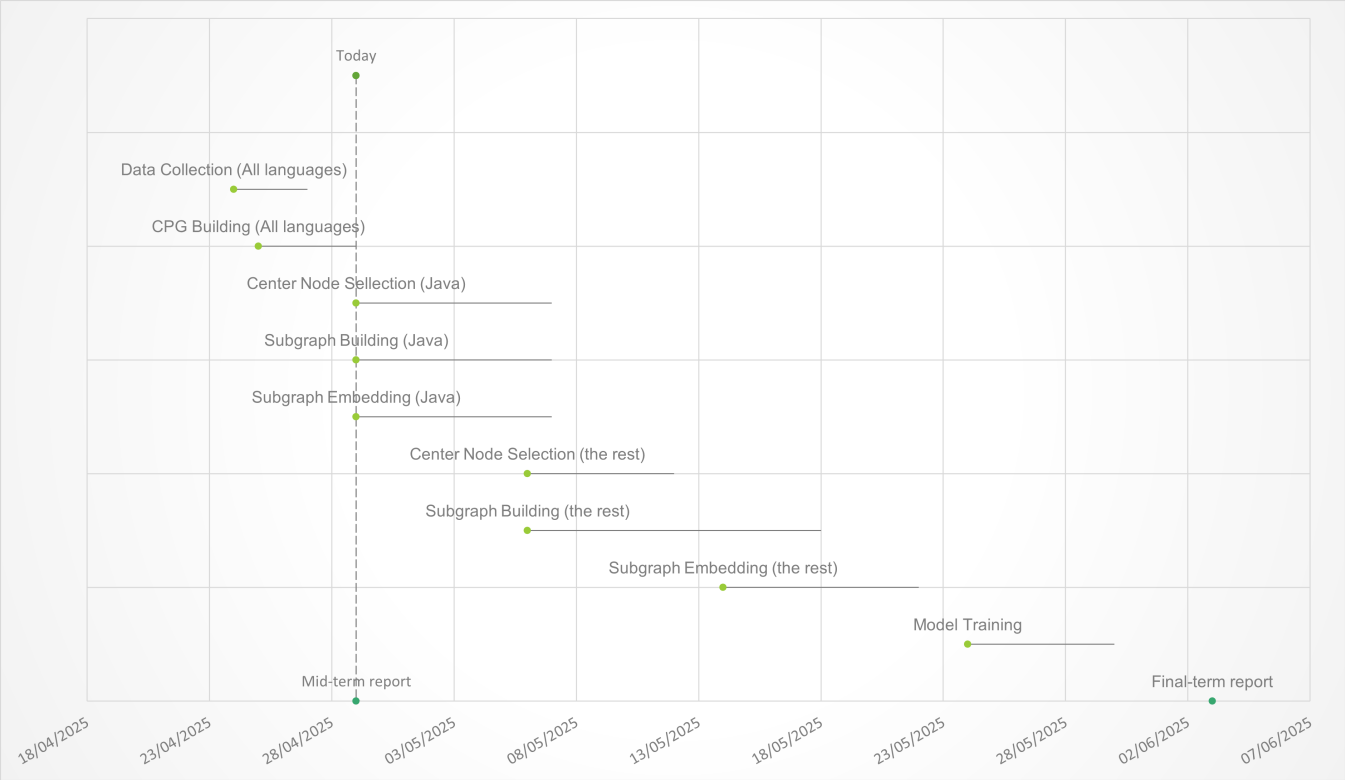
Các nút (node) và loại của chúng: Các nút đại diện cho các cấu trúc chương trình. Điều này bao gồm các cấu trúc ngôn ngữ cấp thấp như phương thức (method), biến (variable) và cấu trúc điều khiển (control structures), nhưng cũng có thể bao gồm các cấu trúc cấp cao hơn như điểm cuối HTTP (HTTP endpoints) hoặc phát hiện (findings). Mỗi nút có một loại (type). Loại nút biểu thị loại cấu trúc chương trình mà nút đó đại diện, ví dụ, một nút có loại METHOD đại diện cho một phương thức trong khi một nút có loại LOCAL đại diện cho việc khai báo một biến cục bộ.

Các cạnh(egde) có hướng và có nhãn(label): Các mối quan hệ giữa các cấu trúc chương trình được biểu diễn thông qua các cạnh giữa các nút tương ứng của chúng. Ví dụ, để thể hiện rằng một phương thức chứa một biến cục bộ, chúng ta có thể tạo một cạnh có nhãn CONTAINS từ nút của phương thức đến nút của biến cục bộ. Bằng cách sử dụng các cạnh có nhãn, chúng ta có thể biểu diễn nhiều loại mối quan hệ trong cùng một đồ thị. Hơn nữa, các cạnh có hướng để thể hiện, ví dụ, rằng phương thức chứa biến cục bộ chứ không phải ngược lại. Nhiều cạnh có thể tồn tại giữa cùng hai nút.

[1] https://samate.nist.gov/SARD/

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn



Hình 1: Timeline dự án

Các cặp khóa-giá trị: Các nút mang các cặp khóa-giá trị (thuộc tính - attributes), trong đó các khóa hợp lệ phụ thuộc vào loại nút. Ví dụ, một phương thức có ít nhất một tên (name) và một chữ ký (signature) trong khi một khai báo biến cục bộ có ít nhất tên và kiểu dữ liệu (type) của biến được khai báo.

3.3 Center Node Selection

Từ các dòng trong tập tin dot có được sau khi Joern phân tích cú pháp, chúng em tạo một Bảng tiêu chí (Bảng 2) như sau:

- Từ file manifest.sarif chứa các thông tin về Project, có cả thông tin về file có lỗ hổng và dòng mã nguồn gây ra lỗ hổng, chúng em ghi nhận lại các dòng ấy từ các Project Java khác nhau.
- Trong file dot, có các node có thể có thuộc tính `LINE_NUMBER` (số hiệu dòng) và `CODE` (mã nguồn tại dòng ấy), chúng em tìm và lọc các node có số hiệu dòng từ bước tổng hợp trước kèm nhãn (label) và code của các node ấy.
- Thống kê và sắp xếp số lần xuất hiện của các thuộc tính như label, `METHOD_NAME` của node theo thứ tự giảm dần và lọc thủ công các loại label không có ý nghĩa cụ thể (Ví dụ: label `BLOCK` đại diện cho các khối mã không chứa logic cụ thể hoặc thông tin quan trọng, hay giá trị code `<empty>` node không chứa code)

Sau khi đã tạo được **Bảng tiêu chí**, chúng em tiến hành thực hiện chọn node trung tâm theo cách như sau:

Đầu tiên, chúng em tải Bảng tiêu chí từ tệp CSV đã được tạo. Bảng này chứa thông tin về các loại nút (Node Type) được coi là hợp lệ để phân tích. Từ cột **Node Type** của bảng, chúng em trích xuất tập hợp các loại nút hợp lệ này, bao gồm cả các nhãn (labels) và tên phương thức (methods).

Tiếp theo, chúng em xử lý từng tệp DOT (biểu diễn CPG) một cách riêng biệt. Đối với mỗi tệp DOT, chúng em sử dụng công cụ phân tích đồ thị (chẳng hạn như PyDot kết hợp với NetworkX) để đọc và biểu diễn cấu trúc đồ thị của CPG. Quá trình phân tích này bao gồm việc trích xuất thông tin về các nút và các cạnh của đồ thị.

Khi phân tích các nút, chúng em kiểm tra xem mỗi nút có thuộc loại nút hợp lệ đã được xác định từ Bảng tiêu chí hay không. Một nút được coi là **mạnh** (strong) nếu nó có cả nhãn *và* tên phương thức thuộc tập hợp các loại nút hợp lệ. Một nút được coi là **yếu** (weak) nếu nó chỉ có nhãn *hoặc* tên phương thức thuộc tập hợp các loại nút hợp lệ, hoặc không có thông tin phù hợp.

Sau đó, chúng em nhóm các nút lại với nhau. Ban đầu, các nút `mạnh` được xem xét là các ứng viên trung tâm. Chúng em tìm các nút `yếu` là hàng xóm trực tiếp của các nút `mạnh` này trong đồ thị. Nếu một nút `mạnh` có các nút `yếu` lân cận, chúng em tạo một nhóm(group) bao gồm nút `mạnh` đó và các

nút `yếu` lân cận của nó. Các nút `yếu` đã được gán vào một nhóm sẽ không được xem xét lại.

Các nút `yếu` còn lại chưa được gán vào bất kỳ nhóm nào sau bước trên sẽ được xử lý tiếp. Chúng em tạo một đồ thị con chỉ chứa các nút `yếu` này và xác định các thành phần liên thông trong đồ thị con đó. Mỗi thành phần liên thông được xem là một nhóm mới.

Cuối cùng, từ mỗi nhóm đã được hình thành, chúng em chọn nút đầu tiên trong nhóm đó làm nút trung tâm đại diện cho nhóm. Tập hợp các nút trung tâm (center nodes) này sau đó được sử dụng cho các phân tích hoặc xử lý tiếp theo. Chúng em cũng ghi nhận tổng số nút có trong đồ thị gốc và tổng số nút trung tâm để thống kê và tham khảo.

3.4 Subgraph Extraction

Sau khi đã có một tập hợp các node trung tâm được xác định từ bước trước, chúng em tiến hành tạo các đồ thị con tương ứng với từng node trung tâm này bằng cách tiếp cận sau:

Đối với mỗi thư mục dự án ban đầu (tương ứng với một tệp DOT), và với mỗi node đã được xác định là node trung tâm trong thư mục đó, chúng em xây dựng một đồ thị con. Đồ thị con này bao gồm:

- Chính nút trung tâm đó.
- Tất cả các nút là hàng xóm trực tiếp (1-hop neighbors) của nút trung tâm trong đồ thị CPG đầy đủ.
- Các cạnh nối nút trung tâm với các hàng xóm trực tiếp của nó.

Tuy nhiên, không phải tất cả các hàng xóm đều được đưa vào đồ thị con. Chúng em chỉ giữ lại những hàng xóm có nhãn (label) thuộc một tập hợp được định nghĩa trước các nhãn *được cho phép*. Tập hợp này bao gồm các loại nút cấu trúc chương trình cụ thể mà chúng em quan tâm, chẳng hạn như: `arrayInitializer`, `CatchClause`, `assignment`, `fieldAccess`, `CALL`, `CONTROL_STRUCTURE`, v.v. Điều này giúp tập trung vào các cấu trúc mã nguồn có khả năng liên quan đến lỗ hổng bảo mật. Công thức toán học biểu diễn việc chọn các nút lân cận có thể được mô tả như sau:

Cho đồ thị CPG đầy đủ là $G = V, E$ trong đó V là tập hợp các nút và E là tập hợp các cạnh.

Cho Nc là tập hợp các nút lân cận trực tiếp của c trong G, tức là:

$$Nc = \{v \in V \mid c, v \in E \text{ hoặc } v, c \in E\}$$

Cho Lv là nhãn của nút v.

Cho A là tập hợp các nhãn được cho phép.

Tập hợp các nút lân cận được giữ lại trong đồ thị con, ký hiệu là $N_{allowed}c$, được tính bằng:

$$N_{allowed}c = \{v \in Nc \mid Lv \in A\}$$

Đồ thị con $G_c = V_c, E_c$ cho nút trung tâm c sẽ có:

$$V_c = \{c\} \cup N_{allowed}c$$

$$E_c = \{u, v \in E \mid u \in V_c \text{ và } v \in V_c \text{ và } u = c \text{ hoặc } v = c\}$$

Quá trình này được thực hiện cho mỗi node trung tâm trong từng tệp DOT. Kết quả là một tập hợp các đồ thị con, mỗi đồ thị đại diện cho ngữ cảnh xung quanh một node trung tâm cụ thể, tập trung vào các cấu trúc chương trình liên quan theo tiêu chí đã định.

Hình 2 thể hiện các đồ thị con được chọn, các node đậm màu và có khung màu đen là node trung tâm, node nhạt màu hơn tượng trưng cho node hàng xóm của node trung tâm đang xét.

3.5 Feature Extraction

Sau khi trích xuất các đoạn mã nguồn từ các thuộc tính `CODE` của các nút trong đồ thị con, chúng em tiến hành chuẩn hóa và tổng quát hóa (standardization and generalization) các đoạn mã này. Mục tiêu của bước này là giảm thiểu sự phụ thuộc vào tên biến, tên hàm cụ thể và các giá trị cố định, tập trung vào cấu trúc và luồng điều khiển của mã.

Quá trình chuẩn hóa và tổng quát hóa được thực hiện theo các bước sau:

- **Xử lý ký hiệu:** Chúng em thêm khoảng trắng xung quanh các ký hiệu đặc biệt trong mã nguồn (như `()` `;` `,` `.`) để đảm bảo rằng chúng được coi là các mã thông báo (tokens) riêng biệt khi phân tách. Điều này giúp tách rõ các thành phần của mã, không bị dính liền vào nhau.
- **Phân tách mã thông báo:** Đoạn mã sau khi được xử lý ký hiệu sẽ được phân tách thành một danh sách các mã thông báo riêng lẻ dựa trên khoảng trắng.
- **Tổng quát hóa mã thông báo:** Chúng em duyệt qua từng mã thông báo trong danh sách đã phân tách:

(1) **Từ khóa Java** Các từ khóa dành riêng của ngôn ngữ Java (như `if`, `else`, `for`, `while`, `class`, `public`, `private`, v.v.) được giữ nguyên.

(2) **Biến:** Các mã thông báo trông giống như tên biến (bắt đầu bằng chữ cái hoặc dấu gạch dưới, theo sau là chữ cái, chữ số hoặc dấu gạch dưới) và không phải là từ khóa Java hoặc tên hàm đã được nhận dạng sẽ được thay thế bằng một mã thông báo tổng quát hóa, ví dụ `VAR1`, `VAR2`,... Các lần xuất hiện tiếp theo của cùng một tên biến sẽ được thay thế bằng cùng mã thông báo tổng quát hóa tương ứng.

(3) **Hàm/Phương thức:** Các mã thông báo trông giống như tên biến và theo sau là dấu mở ngoặc đơn `(` (biểu thị một lời gọi hàm hoặc phương thức) và không phải là từ khóa điều khiển luồng (như `if`, `for`, `while`, v.v.) sẽ được thay thế bằng một mã thông báo tổng quát hóa cho hàm, ví dụ `FUN1`, `FUN2`,... Tương tự như biến, các lần gọi tiếp theo đến cùng một hàm sẽ được thay thế bằng cùng mã thông báo tổng quát hóa tương ứng.

Quá trình này giúp chúng em có được biểu diễn mã nguồn độc lập với tên gọi cụ thể của biến và hàm, làm nổi bật cấu trúc thuật toán và logic của đoạn mã. Điều này đặc biệt quan trọng trong việc phát hiện các mẫu mã liên quan đến lỗ hổng bảo mật, vốn thường phụ thuộc vào cấu trúc hơn là tên gọi cụ thể. Kết quả là một chuỗi các mã thông báo đã được chuẩn hóa và tổng quát hóa, sẵn sàng cho việc sử dụng trong các mô hình học máy hoặc phân tích văn bản.

Hình 2: Mô tả đồ thị con

4 Preliminary Experiments

4.1 Data Statistics

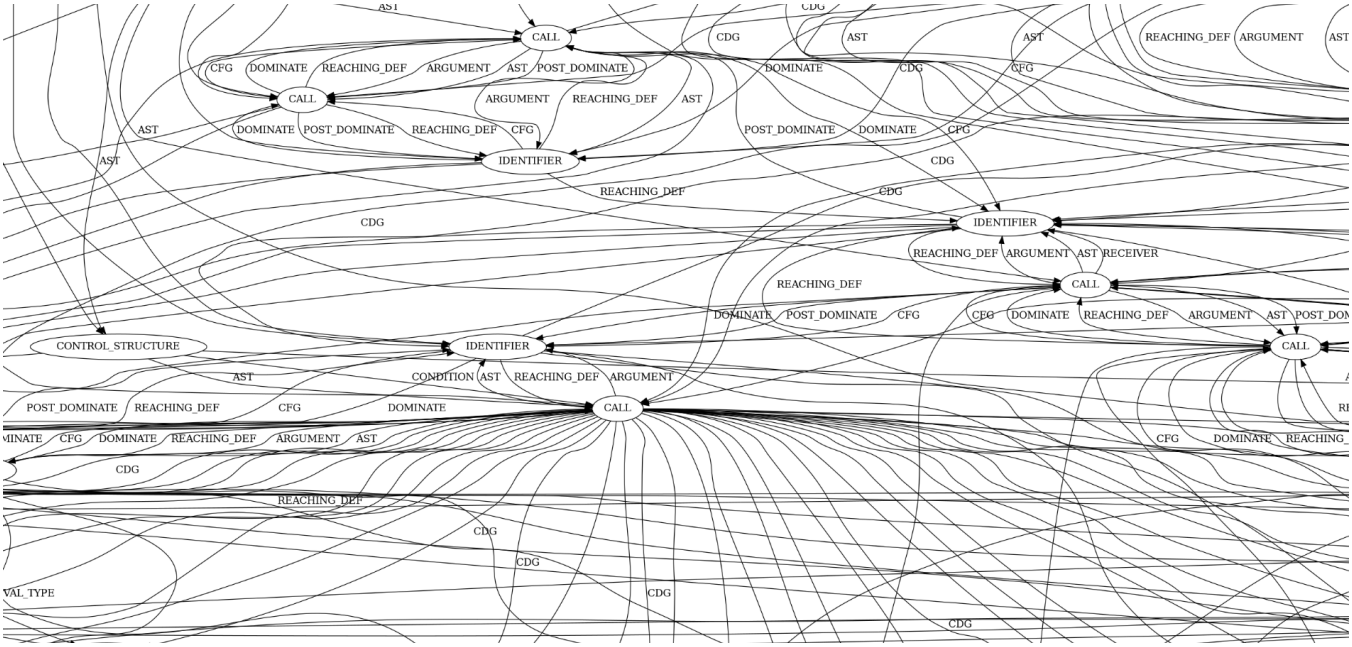
Bảng 1 thể hiện số lượng các trạng thái mã nguồn của từng ngôn ngữ. Trong quá trình thu thập, có một vài mã nguồn không thể lấy được do không còn tồn tại (mã phản hồi 404).

Bảng 1: Bảng thống kê chung về số lượng mẫu dữ liệu theo trạng thái và ngôn ngữ lập trình.

Trạng thái\Ngôn ngữ	Java	C	C++
Good	108	544	54
Bad	1930	1894	438
Mixed	1800	1454	3493
Total	**3838**	**3892**	**3985**

4.2 CPG Graph

Hình 3 cho thấy một phần đồ thị CPG được dựng lên bằng Graphviz, có thể thấy các node với các nhãn như `CALL`, `IDENTIFIER`,... và các loại cạnh mà các node kết nối với nhau



Hình 3: Một phần của đồ thị CPG

như AST (Abstract Syntax Tree), CDG (Control Dependency Graph)...

4.3 Vulnerable Characteristics Table

Bảng đặt trưng hay Bảng tiêu chí được lập trước khi chọn nút trung tâm, được thể hiện ở Bảng 2. Các đặc trưng này sẽ là cơ sở để đánh giá và lựa chọn tập nút trung tâm của các đồ thị, để từ các nút trung tâm này xây dựng các subgraph.

5 Next Steps

Kế hoạch tiếp theo để hoàn thiện dự án gồm:

- **Bổ sung hỗ trợ cho ngôn ngữ C và C++:** Tiếp tục thực hiện các bước xử lí dữ liệu cho hai ngôn ngữ này, hướng đến thực hiện lại bài báo để có kết quả so sánh với kết quả mở rộng trên ngôn ngữ Java.
- **Thực hiện Embedding Subgraph:**
  - Gán nhãn **Node type** theo bảng tiêu chí đã xây dựng.
  - Huấn luyện **Word2Vec** từ danh sách các tokens, rồi ánh xạ code thành chuỗi vector đại diện cho Node code. **Node code**.
  - Xử lý **Edge type** bao gồm các loại cạnh như AST, CFG và các loại cạnh khác.

  Sau đó, ghép các vector lại để tạo thành dữ liệu đầu vào cho mô hình học máy.
- **Chuẩn bị cho Modeling:** Tiến hành thiết kế, huấn luyện và kiểm tra mô hình dựa trên dữ liệu đã xử lý.

Bảng 2: Bảng tiêu chí

	Vulnerability Characteristics	Total Count	Node Type	Example Code
0	Function calls	46035	['CALL']	int factor = (1 « 31) % random;
1	Decide the type of the variable	39122	['IDENTIFIER']	factor; random; Tracer; factor; counter;
2	Access a field of an object of aggreg	13827	['FIELD_IDENTIFIER', 'fieldAccess', 'indexAcces']	vowlessInferentialist; length;
3	Assign values to variables	5559	['assignment']	int factor = (1 « 31) % random;
4	Conduct an arithmetic calculation	2606	['addition']	e.getClass().getName() +
5	Open or discard a memory space	1307	['alloc']	new int[size];
6	Exception handling	1112	['CatchClause']	catch; catch; catch; catch;
7	Relate to control flow and code struct	1055	['CONTROL_STRUCTURE']	else; catch; else; catch; else; catch;
8	Use an array	1037	['arrayInitializer', 'stonesoup_array']	<operator>.arrayInitializer; <operato
9	Conduct a boolean/logical/comparis	631	['logicalAnd', 'logicalNot']	(stonesoup_counter + stonesoup_offet > 0)
10	Type casting and downcasting	496	['cast']	(Integer) supercongestionMiliarium.g