joern Documentation

Release 0.2.5

Fabian Yamaguchi

December 26, 2016

Contents

Joern is a platform for robust analysis of C/C++ code developed by Fabian Yamaguchi at the Computer Security Group of the University of Goettingen. It generates *code property graphs*, a novel graph representation that exposes the code's syntax, control-flow, data-flow and type information in a joint data structure. Code property graphs are stored in a Neo4J graph database. This allows code to be mined using search queries formulated in the graph traversal language Gremlin.

- **Fuzzy Parsing.** Joern employs a fuzzy parser, allowing code to be imported even if a working build environment cannot be supplied.
- **Code Property Graphs.** Joern creates code property graphs from the fuzzy parser output and makes and stores them in a Neo4J graph database. For background information on code property graphs, we strongly encourage you to read our paper on the topic.
- **Extensible Query Language.** Based on the graph traversal language Gremlin, Joern offers an extensible query language based on user-defined Gremlin steps that encode common traversals in the code property graph. These can be combined to create search queries easily.

This is joern's official documentation. It covers its installation and configuration, discusses how code can be imported and retrieved from the database and gives an overview of the database contents.

Contents:

CHAPTER 1

Installation

Joern currently consists of the following components:

- joern(-core) parses source code using a robust parser, creates code property graphs and finally, imports these graphs into a Neo4j graph database.
- python-joern is a (minimal) python interface to the Joern database. It offers a variety of utility traversals (so called *steps*) for common operations on the code property graph (think of these are stored procedures).
- joern-tools is a collection of command line tools employing python-joern to allow simple analysis tasks to be performed directly on the shell.

Both python-joern and joern-tools are optional, however, installing python-joern is highly recommended for easy access to the database. While it is possible to access Neo4J from many other languages, you will need to write some extra code to do so and therefore, it is currently not recommended.

1.1 System Requirements and Dependencies

Joern is a Java Application and should work on systems offering a Java virtual machine, e.g., Microsoft Windows, Mac OS X or GNU/Linux. We have tested Joern on Arch Linux as well as Mac OS X Lion. If you plan to work with large code bases such as the Linux Kernel, you should have at least 30GB of free disk space to store the database and 8GB of RAM to experience acceptable performance. In addition, the following software should be installed:

- **A Java Virtual Machine 1.7.** Joern is written in Java 7 and does not build with Java 6. It has been tested with OpenJDK-7 but should also work fine with Oracle's JVM.
- **Neo4J 2.1.5 Community Edition.** The graph database Neo4J provides access to the imported code.
- **Gremlin for Neo4J 2.X.** The Gremlin plugion for Neo4J 2.X allows traversals written in the programming language Gremlin to be run on the Neo4J database.

Build Dependencies. A tarball containing all necessary build-dependencies is available for download here . This contains files from the following projects.

- The ANTLRv4 Parser Generator
- Apache Commons CLI Command Line Parser 1.2
- Neo4J 2.1.5 Community Edition
- The Apache Ant build tool (tested with version 1.9.2).

The following sections offer a step-by-step guide to the installation of Joern, including all of its dependencies.

1.2 Building joern

Begin by downloading the latest stable version of joern at http://mlsec.org/joern/download.shtml. This will create the directory `joern` in your current working directory.

```
wget https://github.com/fabsx00/joern/archive/0.3.1.tar.gz
tar xfzv 0.3.1.tar.gz
```

Change to the directory `joern-0.3.1/`. Next, download build dependencies at http://mlsec.org/joern/lib/lib.tar.gz and extract the tarball.

```
cd joern-0.3.1
wget http://mlsec.org/joern/lib/lib.tar.gz
tar xfzv lib.tar.gz
```

The JAR-files necessary to build joern should now be located in `joern-0.3.1/lib/`.

Note: If you want to build the development version, you need to download the build dependencies from http://mlsec.org/joern/lib/lib-dev.tar.gz .

Build the project using `ant` by issuing the following command.

```
ant
```

Create symlinks (optional). The executable JAR file will be located in `joern-0.3.1/bin/joern.jar`. Simply place this JAR file somewhere on your disk and you are done. If you are using bash, you can optionally create the following alias in your `.bashrc`:

```
# ~/.bashrc
alias joern='java -jar $JOERN/bin/joern.jar'
```

where `$JOERN` is the directory you installed joern into.

Build additional tools (optional). Tools such as the code{argumentTainter} can be built by issuing the following command.

```
ant tools
```

Upon successfully building the code, you can start importing C/C++ code you would like to analyze. To interact with the database using python and the shell, it is also highly recommended to install `python-joern` and `joern-tools` as outlined in the following sections.

1.3 Installing python-joern

`python-joern` is a thin python access layer for joern and a set of utility traversals. It depends on the following python modules:

- py2neo 2.0 (http://py2neo.org/)

To install `python-joern`, first make sure python setuptools are correctly installed. On Debian/Ubuntu, issuing the following command on the shell should be sufficient.

```
sudo apt-get install python-setuptools python-dev
```

`python-joern` and all its dependencies can then be installed as follows:

```
wget https://github.com/fabsx00/python-joern/archive/0.3.1.tar.gz
tar xfzv 0.3.1.tar.gz
```

```
cd python-joern-0.3.1
sudo python2 setup.py install
```

1.4 Installing joern-tools

`joern-tools` is a set of shell utilities for code analysis based on joern. It is at a very early stage of development and has not been labeled for release. However, it can be installed from github.

`joern-tools` depends on `python-joern` for database communication and graphviz/pygraphviz for graph visualization. To install it, make sure graphviz is installed. On Debian/Ubuntu, the following command will install graphviz:

```
sudo apt-get install graphviz libgraphviz-dev
```

Just like `python-joern`, `joern-tools` is installed using python-setuptools as follows:

```
git clone https://github.com/fabsx00/joern-tools
cd joern-tools
sudo python2 setup.py install
```

After installation, type `joern-lookup` to verify correct installation.

CHAPTER 2

Importing Code

2.1 Populating the Database

Once joern has been installed, you can begin to import code into the database by simply pointing `joern.jar` to the directory containing the source code:

```
java -jar $JOERN/bin/joern.jar $CodeDirectory
```

or, if you want to ensure that the JVM has access to your heap memory

```
java -Xmx$SIZEg -jar $JOERN/bin/joern.jar $CodeDirectory
```

where `$SIZE` is the maximum size of the Java Heap in GB. As an example, you can import `$JOERN/testCode`.

This will create a Neo4J database directory `.joernIndex` in your current working directory. Note that if the directory already exists and contains a Neo4J database, `joern.jar` will add the code to the existing database. You can thus import additional code at any time. If however, you want to create a new database, make sure to delete `.joernIndex` prior to running `joern.jar`.

2.2 Tainting Arguments (Optional)

Many times, an argument to a library function (e.g., the first argument to `recv`) is tainted by the library function. There is no way to statically determine this when the code of the library function is not available. Also, Joern does not perform inter-procedural taint-analysis and therefore, by default, symbols passed to functions as arguments are considered *used* but not *defined*.

To instruct Joern to consider arguments of a function to be tainted by calls to that function, you can use the tool `argumentTainter`. For example, by executing

```
java -jar ./bin/argumentTainter.jar recv 0
```

from the Joern root directory, all first arguments to `recv` will be considered tainted and dependency graphs will be recalculated accordingly.

2.3 Starting the Database Server

It is possible to access the graph database directly from your scripts by loading the database into memory on script startup. However, it is highly recommended to access data via the Neo4J server instead. The advantage of doing so is

that the data is loaded only once for all scripts you may want to execute allowing you to benefit from Neo4J's caching for increased speed.

To install the neo4j server, download version textbf{1.9.7} from http://www.neo4j.org/download/other_versions.

Once downloaded, unpack the archive into a directory of your choice, which we will call `$Neo4jDir` in the following.

Next, specificy the location of the database created by joern in your Neo4J server configuration file in `$Neo4jDir/conf/neo4j-server.properties`:

```
# neo4j-server.properties
org.neo4j.server.database.location=/$path_to_index/.joernIndex/
```

For example, if your `.joernIndex` is located in `/home/user/joern/.joernIndex`, your configuration file should contain the line:

```
# neo4j-server.properties
org.neo4j.server.database.location=/home/user/joern/.joernIndex/
```

Please also make sure that `org.neo4j.server.database.location` is set only once.

You can now start the database server by issuing the following command:

```
$Neo4jDir/bin/neo4j console
```

If your installation of Neo4J is more recent than the libraries bundled with joern, the database might fail to start and request an upgrade of the stored data. This upgrade can be performed on the fly by enabling `allow_store_upgrade` in `neo4j.properties` as follows:

```
# neo4j.properties
allow_store_upgrade=true
```

The Neo4J server offers a web interface and a web-based API (REST API) to explore and query the database. Once your database server has been launched, point your browser to http://localhost:7474/ .

Next, visit http://localhost:7474/db/data/node/0 . This is the *reference node*, which is the root node of the graph database. Starting from this node, the entire database contents can be accessed. In particular, you can get an overview of all existing edge types as well as the properties attached to nodes and edges.

Of course, in practice, you will not want to use your browser to query the database. Instead, you can use python-joern to access the REST API using Python as described in the following section.

CHAPTER 3

Accessing Code with python-joern

Once code has been imported into a Neo4j database, it can be accessed using a number of different interfaces and programming languages. One of the simplest possibilities is to create a standalone Neo4J server instance as described in the previous section and connect to this server using `python-joern`, the python interface to joern.

General Note: It is highly recommended to test your installation on a small code base first. The same is true for early attempts of creating search queries, as erroneous queries will often run for a very long time on large code bases, making a trial-and-error approach unfeasible.

3.1 Basic Usage

Python-joern currently provides a single class, JoernSteps, that allows to connect to the database server and run queries. The following is a simple sample script that employs JoernSteps to configure the database connection, connect to the server and run a Gremlin query.

```
from joern.all import JoernSteps

j = JoernSteps()

j.setGraphDbURL('http://localhost:7474/db/data/')

# j.addStepsDir('Use this to inject utility traversals')

j.connectToDatabase()

res =  j.runGremlinQuery('getFunctionsByName("main")')
# res =  j.runCypherQuery('...')

for r in res: print r
```

3.2 python-joern API

The sample script described in the previous section employs all methods offered by JoernSteps. We now discuss each of these methods in detail.

3.2.1 setGraphDbURL(url)

Sets the URL of the graph database server. The REST API of the Neo4J Database server is exposed on port 7474 by default. If your server runs on a different port or server, you can use setGraphDbURL to specify the alternate URL.

3.2.2 addStepsDir(dirname)

Add a source directory for utility traversals. By default, python-joern will inject all utility traversals contained in any of the source files in joern/joernsteps into the database before running scripts. Additional traversals specific to your application or analysis are best placed in a separate directory. python-joern can be instructed to honor this additional directory using addStepsDir.

3.2.3 connectToDatabase()

Connect to the database. Call this method once the connection has been configured to connect to the database server. A connection is required before queries can be executed.

3.2.4 runGremlinQuery(query)

Run the specified Gremlin query. The supplied query is executed and the result is returned. Depending on the query, the result may have a different data type, however, it is typically an iterable containing nodes that match the query.

3.2.5 runCypherQuery(query)

Run the specified Cypher query. The supplied query is executed and the result is returned. Depending on the query, the result may have a different data type, however, it is typically an iterable containing nodes that match the query.

CHAPTER 4

Performance Tuning

4.1 Optimizing Code Importing

Joern uses the Neo4J Batch Inserter for code importing (see Chapter 35 of the Neo4J documentation). Therefore, the performance you will experience mainly depends on the amount of heap memory you can make available for the importer and how you assign it to the different caches used by the Neo4J Batch Inserter. You can find a detailed discussion of this topic at https://github.com/jexp/batch-import .

By default, Joern will use a configuration based on the maximum size of the Java heap. For sizes below 4GB, the following configuration is used:

```
cache_type = none
use_memory_mapped_buffers = true
neostore.nodestore.db.mapped_memory = 200M
neostore.relationshipstore.db.mapped_memory = 2G
neostore.propertystore.db.mapped_memory = 200M
neostore.propertystore.db.strings.mapped_memory = 200M
neostore.propertystore.db.index.keys.mapped_memory = 5M
neostore.propertystore.db.index.mapped_memory = 5M
```

The following configuration is used for heap-sizes larger than 4GB:

```
cache_type = none
use_memory_mapped_buffers = true
neostore.nodestore.db.mapped_memory = 1G
neostore.relationshipstore.db.mapped_memory = 3G
neostore.propertystore.db.mapped_memory = 1G
neostore.propertystore.db.strings.mapped_memory = 500M
neostore.propertystore.db.index.keys.mapped_memory = 5M
neostore.propertystore.db.index.mapped_memory = 5M
```

The Neo4J Batch Inserter configuration is currently not exported. If you are running Joern on a machine where these values are too low, you can adjust the values in `src/neo4j/batchinserter/ConfigurationGenerator.java`.

For the `argumentTainter`, the same default configurations are used. The corresponding values reside in `src/neo4j/readWriteDb/ConfigurationGenerator.java`.

4.2 Optimizing Traversal Speed

To experience acceptable performance, it is crucial to configure your Neo4J server correctly. To achieve this, it is highly recommended to review Chapter 22 of the Neo4J documentation on Configuration and Performance. In particular, the following settings are important to obtain good performance.

Size of the Java heap. Make sure the maximum size of the java heap is high enough to benefit from the amount of memory in your machine. One possibility to ensure this, is to add the `-Xmx$SIZEg` flag to the variable `JAVA_OPTS` in `$Neo4JDir/bin/neo4j` where `$Neo4JDir` is the directory of the Neo4J installation. You can also configure the maximum heap size globally by appending the `-Xmxx` flag to the environment variable `_JAVA\_OPTIONS`.

Maximum number of open file descriptors. If, when starting the Neo4J server, you see the message

```
WARNING: Max 1024 open files allowed, minimum of 40 000 recommended.
```

you need to raise the maximum number of open file descriptors for the user running Neo4J (see the Neo4J Linux Performance Guide).

Memory Mapped I/O Settings. Performance of graph database traversals increases significantly when large parts of the graph database can be kept in RAM and do not have to be loaded from disk (see http://docs.neo4j.org/chunked/stable/configuration-io-examples.html). For example, for a machine with 8GB RAM the following `neo4j.conf` configuration has been tested to work well:

```
# conf/neo4j.conf
use_memory_mapped_buffers=true
cache_type=soft
neostore.nodestore.db.mapped_memory=500M
neostore.relationshipstore.db.mapped_memory=4G
neostore.propertystore.db.mapped_memory=1G
neostore.propertystore.db.strings.mapped_memory=1300M
neostore.propertystore.db.arrays.mapped_memory=130M
neostore.propertystore.db.index.keys.mapped_memory=200M
neostore.propertystore.db.index.mapped_memory=200M
keep_logical_logs=true
```

4.3 Chunking Traversals

Running the same traversal on a large set of start nodes often leads to unacceptable performance as all nodes and edges touched by the traversal are kept in server memory before returning results. For example, the query:

```
getAllStatements().astNodes().id
```

which retrieves all astNodes that are part of statements, can already completely exhaust memory.

If traversals are independent, the query can be chunked to gain high performance. The following example code shows how this works:

```
from joern.all import JoernSteps

j = JoernSteps()
j.connectToDatabase()

ids =  j.runGremlinQuery('getAllStatements.id')

CHUNK_SIZE = 256
for chunk in j.chunks(ids, CHUNK_SIZE):
```

```
query = """ idListToNodes(%s).astNodes().id """ % (chunk)

for r in j.runGremlinQuery(query): print r
```

This will execute the query in batches of 256 start nodes each.

CHAPTER 5

Database Overview

In this section, we give an overview of the database layout created by Joern, i.e., the nodes, properties and edges that make up the code property graph. The code property graph created by Joern matches that of the code property graph as described in the paper and merely introduces some additional nodes and edges that have turned out to be convenient in practice.

5.1 Code Property Graphs

For each function of the code base, the database stores a code property graph consisting of the following nodes.

Function nodes (type: Function). A node for each function (i.e. procedure) is created. The function-node itself only holds the function name and signature, however, it can be used to obtain the respective Abstract Syntax Tree and Control Flow Graph of the function.

Abstract Syntax Tree Nodes (type:various). Abstract syntax trees represent the syntactical structure of the code. They are the representation of choice when language constructs such as function calls, assignments or cast-expressions need to be located. Moreover, this hierarchical representation exposes how language constructs are composed to form larger constructs. For example, a statement may consist of an assignment expression, which itself consists of a left- and right value where the right value may contain a multiplicative expression (see Wikepedia: Abstract Syntax Tree for more information). Abstract syntax tree nodes are connected to their child nodes with `IS_AST_PARENT_OF` edges. Moreover, the corresponding function node is connected to the AST root node by a `IS_FUNCTION_OF_AST` edge.

Statement Nodes (type:various). This is a sub-set of the Abstract Syntax Tree Nodes. Statement nodes are marked using the property `isCFGNode` with value `true`. Statement nodes are connected to other statement nodes via `FLOWS_TO` and `REACHES` edges to indicate control and data flow respectively.

Symbol nodes (type:Symbol). Data flow analysis is always performed with respect to a variable. Since our fuzzy parser needs to work even if declarations contained in header-files are missing, we will often encounter the situation where a *symbol* is used, which has not previously been declared. We approach this problem by creating *symbol* nodes for each identifier we encounter regardless of whether a declaration for this symbol is known or not. We also introduce symbols for postfix expressions such as `a->b` to allow us to track the use of fields of structures. Symbol nodes are connected to all statement nodes using the symbol by `USE`-edges and to all statement nodes assigning to the symbol (i.e., *defining* the symbol}) by `DEF`-edges.

Code property graphs of individual functions are linked in various ways to allow transition from one function to another as discussed in the next section.

5.2 Global Code Structure

In addition, to the nodes created for functions, the source file hierarchy, as well as global type and variable declarations are represented as follows.

File and Directory Nodes (type:File/Directory). The directory hierarchy is exposed by creating a node for each file and directory and connecting these nodes using `IS_PARENT_DIR_OF` and `IS_FILE_OF` edges. This *source tree* allows code to be located by its location in the filesystem directory hierarchy, for example, this allows you to limit your analysis to functions contained in a specified sub-directory.

Struct/Class declaration nodes (type: Class). A Class-node is created for each structure/class identified and connected to file-nodes by `IS_FILE_OF` edges. The members of the class, i.e., attribute and method declarations are connected to class-nodes by `IS_CLASS_OF` edges.

Variable declaration nodes (type: DeclStmt). Finally, declarations of global variables are saved in declaration statement nodes and connected to the source file they are contained in using `IS_FILE_OF` edges.

5.3 The Node Index

In addition to the graphs stored in the Neo4J database, Joern makes an Apache Lucene Index available that allows nodes to be quickly retrieved based on their properties. This is particularly useful to select start nodes for graph database traversals. For examples of node index usage, refer to Querying the Database.

CHAPTER 6

Querying the Database

This chapter discusses how the database contents generated by Joern can be queried to locate interesting code. We begin by reviewing the basics of the graph traversal language Gremlin and proceed to discuss how to select start nodes from the node index. The remainder of this chapter deals with code retrieval based on syntaxtaint-style queries in and finally, traversals in the function symbol graph.

The user-defined traversals presented throughout this chapter are all located in the directory `joernsteps` of `python-joern`. It may be worth studying their implementation to understand how to design your own custom steps for Gremlin.

6.1 Gremlin Basics

In this section, we will give a brief overview of the most basic functionality offered by the graph traversal language Gremlin developed by Marko A. Rodriguez. For detailed documentation of language features, please refer to http://gremlindocs.com and the Gremlin website.

Gremlin is a language designed to describe walks in property graphs. A property graph is simply a graph where key-value pairs can be attached to nodes and edges. (From a programmatic point of view, you can simply think of it as a graph that has hash tables attached to nodes and edges.) In addition, each edge has a type, and that's all you need to know about property graphs for now.

Graph traversals proceed in two steps to uncover to search a database for sub-graphs of interest:

1. **Start node selection.** All traversals begin by selecting a set of nodes from the database that serve as starting points for walks in the graph.

2. **Walking the graph.** Starting at the nodes selected in the previous step, the traversal walks along the graph edges to reach adjacent nodes according to properties and types of nodes and edges. The final goal of the traversal is to determine all nodes that can be reached by the traversal. You can think of a graph traversal as a sub-graph description that must be fulfilled in order for a node to be returned.

The simplest way to select start nodes is to perform a lookup based on the unique node id as in the following query:

```
// Lookup node with given nodeId
g.v(nodeId)
```

Walking the graph can now be achieved by attaching so called emph{Gremlin steps} to the start node. Each of these steps processes all nodes returned by the previous step, similar to the way Unix pipelines connect shell programs. While learning Gremlin, it can thus be a good idea to think of the dot-operator as an alias for the unix pipe operator `|`. The following is a list of examples.

```
// Traverse to nodes connected to start node by outgoing edges
g.v(nodeId).out()

// Traverse to nodes two hops away.
g.v(nodeId).out().out()

// Traverse to nodes connected to start node by incoming edges
g.v(nodeId).in()

// All nodes connected by outgoing AST edges (filtering based
// on edge types)
g.v(nodeId).out(AST_EDGE)

// Filtering based on properties:
g.v(nodeId).out().filter{ it.type == typeOfInterest}

// Filtering based on edge properties
g.v.outE(AST_EDGE).filter{ it.propKey == propValue }.inV()
```

The last two examples deserve some explanation as they both employ the elementary step `filter` defined by the language Gremlin. `filter` expects a so called *closure*, an anonymous function wrapped in curly brackets (see Groovy - Closures). This function is applied to each incoming node and is expected to return `true` or `false` to indicate whether the node should pass the filter operation or not. The first parameter given to a closure is named `it` by convention and thus `it.type` denotes the property `type` of the incoming node in the first of the two examples. In the second example, edges are handed to the filter routine and thus `it.propKey` refers to the property `propKey` of the incoming edge.

6.2 Start Node Selection

In practice, the ids of interesting start nodes are rarely known. Instead, start nodes are selected based on node properties, for example, one may want to select all calls to function `memcpy` as start nodes. To allow efficient start node selection based on node properties, we keep an (Apache Lucene) node index (see Neo4J legacy indices). This index can be queried using Apache Lucene queries (see Lucene Query Language for details on the syntax). For example, to retrieve all AST nodes representing callees with a name containing the substring code{cpy}, one may issue the following query:

```
queryNodeIndex("type:Callee AND name:*cpy*") \end{verbatim}
```

The Gremlin step `queryNodeIndex` is defined in `joernsteps/lookup.groovy` of `python-joern`. In addition to `queryNodeIndex`, `lookup.groovy` defines various functions for common lookups. The example just given could have also been formulated as:

```
getCallsTo("*cpy*")
```

Please do not hesitate to contribute short-hands for common lookup operations to include in `joernsteps/lookup.groovy`.

6.3 Traversing Syntax Trees

In the previous section, we outlined how nodes can be selected based on their properties. As outline in Section *Gremlin Basics*, these selected nodes can now be used as starting points for walks in the property graph.

As an example, consider the task of finding all multiplications in first arguments of calls to the function `malloc`. To solve this problem, we can first determine all call expressions to `malloc` and then traverse from the call to its first argument in the syntax tree. We then determine all multiplicative expressions that are child nodes of the first argument.

In principle, all of these tasks could be solved using the elementary Gremlin traversals presented in Section *Gremlin Basics*. However, traversals can be greatly simplified by introducing the following user-defined gremlin-steps (see `joernsteps/ast.py`).

```
// Traverse to parent nodes in the AST
parents()

// Traverse to child nodes in the AST
children()

// Traverse to i'th children in the AST
ithChildren()

// Traverse to enclosing statement node
statements()

// Traverse to all nodes of the AST
// rooted at the input node
astNodes()
```

Additionally, `joernsteps/calls.groovy` introduces user-defined steps for traversing calls, and in particular the step `ithArguments` that traverses to i'th arguments of a given a call node. Using these steps, the exemplary traversal for multiplicative expressions inside first arguments to `malloc` simply becomes:

```
getCallsTo('malloc').ithArguments('0')
.astNodes().filter{ it.type == 'MultiplicativeExpression'}
```

6.4 Syntax-Only Descriptions

The file `joernsteps/composition.groovy` offers a number of elementary functions to combine other traversals and lookup functions. These composition functions allow arbitrary syntax-only descriptions to be constructed (see Modeling and Discovering Vulnerabilities with Code Property Graphs). For example, to select all functions that contain a call to `foo` AND a call to `bar`, lookup functions can simply be chained, e.g.,

```
getCallsTo('foo').getCallsTo('bar')
```

returns functions calling both `foo` and `bar`. Similarly, functions calling code{foo} OR code{bar} can be selected as follows:

```
OR( getCallsTo('foo'), getCallsTo('bar') )
```

Finally, the `not`-traversal allows all nodes to be selected that do NOT match a traversal. For example, to select all functions calling *foo* but not *bar*, use the following traversal:

```
getCallsTo('foo').not{ getCallsTo('bar') }
```

6.5 Traversing the Symbol Graph

As outlined in Section Database Overview, the symbols used and defined by statements are made explicit in the graph database by adding symbol nodes to functions (see Appendix D of Modeling and Discovering Vulnerabil-

ities with Code Property Graphs). We provide utility traversals to make use of this in order to determine symbols defining variables, and thus simple access to types used by statements and expressions. In particular, the file `joernsteps/symbolGraph.groovy` contains the following steps:

```
// traverse from statement to the symbols it uses
uses()

// traverse from statement to the symbols it defines
defines()

// traverse from statement to the definitions
// that it is affected by (assignments and
// declarations)
definitions()
```

As an example, consider the task of finding all third arguments to `memcpy` that are defined as parameters of a function. This can be achieved using the traversal

```
getArguments('memcpy', '2').definitions()
.filter{it.type == TYPE_PARAMETER}
```

where `getArguments` is a lookup-function defined in `joernsteps/lookup.py`.

As a second example, we can traverse to all functions that use a symbol named `len` in a third argument to `memcpy` that is not used by any condition in the function, and hence, may not be checked.

```
getArguments('memcpy', '2').uses()
.filter{it.code == 'len'}
.filter{
        it.in('USES')
        .filter{it.type == 'Condition'}.toList() == []
}
```

This example also shows that traversals can be performed inside filter-expressions and that at any point, a list of nodes that the traversal reaches can be obtained using the function `toList` defined on all Gremlin steps.

6.6 Taint-Style Descriptions

The last example already gave a taste of the power you get when you can actually track where identifiers are used and defined. However, using only the augmented function symbol graph, you cannot be sure the definitions made by one statement actually *reach* another statement. To ensure this, the classical *reaching definitions* problem needs to be solved. In addition, you cannot track whether variables are sanitized on the way from a definition to a statement.

Fortunately, joern allows you to solve both problems using the traversal `unsanitized`. As an example, consider the case where you want to find all functions where a third argument to `memcpy` is named `len` and is passed as a parameter to the function and a control flow path exists satisfying the following two conditions:

begin{itemize} item The variable code{len} is not re-defined on the way. item The variable is not used inside a relational or equality expression on the way, i.e., its numerical value is not ‘‘checked” against some other variable. end{itemize}

You can use the following traversal to achieve this:

```
getArguments('memcpy', '2')
.sideEffect{ paramName = '.*len.*' }
.filter{ it.code.matches(paramName) }
.unsanitized{ it.isCheck( paramName ) }
.params( paramName )
```

where `isCheck` is a traversal defined in `joerntools/misc.groovy` to check if a symbol occurs inside an equality or relational expression and code{params} traverses to parameters matching its first parameter.

Note, that in the above example, we are using a regular expression to determine arguments containing the sub-string `len` and that one may want to be a little more exact here. Also, we use the Gremlin step `sideEffect` to save the regular expression in a variable, simply so that we do not have to re-type the regular expression over and over.

CHAPTER 7

Joern-tools

7.1 joern-slice

7.1.1 SYNOPSIS

joern-slice [options]

7.1.2 DESCRIPTION

Creates program slices for all nodes passed to the program via standard input or a supplied file. Input is expected to be a list of node ids separated by newlines. Both forward and backward slices can be calculated. For each node, the tool generates a line of output of the following format:

```
label TAB NodeId_1 ... NodeId_m TAB EdgeId_1 .... EdgeId_n
```

where label is the id of the reference node, NodeId_1 ... NodeId_m is the list of nodes of the slice and EdgeId_1 .... EdgeId_n is the list of edges.

Forward Slices

The exact behavior depends on the node type:

- **Statements and Conditions:** For statement and condition nodes (i.e., nodes where `isCFGNode``is 'True`), the slice is calculated for all symbols defined by the statement.
- **Arguments:** The slice is calculated for all symbols defined by the

argument.

- **Callee:** The slice is calculated for all symbols occurring on the

left hand side of the assignment (*return values*).

Backward Slices

The exact behavior depends on the node type:

- **Statements, Conditions and Callees:** For statement and condition nodes (i.e., nodes where `isCFGNode``is 'True`), the slice is calculated for all symbols used by the statement.
- **Arguments.** The slice is calculated for all symbols used inside the

argument.

7.2 joern-apiembedder

7.2.1 SYNOPSIS

joern-apiembedder [options]

joern-stream-apiembedder [options]

7.2.2 DESCRIPTION

embedder.py creates an embedding of functions based on the API symbols they employ as well as a corresponding distance matrix. These embeddings are used by knn.py to identify similar functions but may also serve as a basis for other tools that require a vectorial representation of code.

-d –dirname <dirname>

Output directory of the embedding. By default, output will be written to the directory 'embedding'.

Note: Please use `joern-stream-apiembedder` in new code, `joern-apiembedder` is only kept around because it is still being used by legacy code.

7.3 joern-knn

7.3.1 SYNOPSIS

joern-knn [options]

7.3.2 DESCRIPTION

knn.py is a tool to identify similar functions/program slices. It does not deal with the extraction of functions from code nor their characterization (see embedder.py), however, given a representation of each function/program-slice by a set of strings, it allows similar functions to be identified.

7.3.3 OPTIONS

-l –limit <file>

Limit possible neighbours to those specified in the provided file. The file is expected to contain ids of possible neighbors separated by newlines.

-d –dirname <dirname>

The name of the directory containing the embedding, as for example, created by apiEmbed.py. In summary, the directory must contain the following files.

/TOC

A line containing labels for each data point where the i'th line contains the label for the i'th data point.

/data/$i

The i'th data point where $i is an ordinal. Lines of the file correspond to elementary features. For example, if the function is represented by API symbols and it contains the symbol 'int' twice, the corresponding file will be contain the lines:

int int

Write-access to this directory is required as knn.py will cache distance matrices in this directory.

CHAPTER 8

Development

8.1 Accessing the GIT Repository

We use the revision control system git to develop Joern. If you want to participate in development or test the development version, you can clone the git repository by issuing the following command:

```
git clone https://github.com/fabsx00/joern.git
```

Optionally, change to the branch of interest. For example, to test the development version, issue the following:

```
git checkout dev
```

If you want to report issues or suggest new features, please do so via https://github.com/fabsx00/joern . For fixes, please fork the repository and issue a pull request or alternatively send a diff to the developers by mail.

8.2 Modifying Grammar Definitions

When building Joern, pre-generated versions of the parsers will be used by default. This is fine in most cases, however, if you want to make changes to the grammar definition files, you will need to regenerate parsers using the antlr4 tool. For this purpose, it is highly recommended to use the optimized version of ANTLR4 to gain maximum performance.

To build the optimized version of ANTLR4, do the following:

```
git clone https://github.com/sharwell/antlr4/
cd antlr4
mvn -N install
mvn -DskipTests=true -Dgpg.skip=true -Psonatype-oss-release -Djava6.home=$PATH_TO_JRE install
```

If the last step gives you an error, try building without `-Psonatype-oss-release`.

```
mvn -DskipTests=true -Dgpg.skip=true -Djava6.home=$PATH_TO_JRE install
```

Next, copy the antlr4 tool and runtime to the following locations:

```
cp tool/target/antlr4-$VERSION-complete.jar $JOERN/
cp runtime/Java/target/antlr4-runtime-$VERSION-SNAPSHOT.jar $JOERN/lib
```

where $JOERN is the directory containing the $JOERN installation.

Parsers can then be regenerated by executing the script `$JOERN/genParsers.sh`.

CHAPTER 9

Tutorials

9.1 Code Analysis with joern-tools (Work in progress)

This tutorial shows how the command line utilities `joern-tools` can be used for code analysis on the shell. These tools have been created to enable fast programmatic code analysis, in particular to hunt for bugs and vulnerabilities. Consider them a possible addition to your GUI-based code browsing tools and not so much as a replacement. That being said, you may find yourself doing more and more of your code browsing on the shell with these tools.

This tutorial offers both short and concise commands that *get a job done* as well as more lengthly queries that illustrate the inner workings of the code analysis platform `joern`. The later have been provided to enable you to quickly extend `joern-tools` to suit your specific needs.

Note: If you end up writing tools that may be useful to others, please don't hesitate to send a pull-request to get them included in `joern-tools`.

9.1.1 Importing the Code

As an example, we will analyze the VLC media player, a medium sized code base containing code for both Windows and Linux/BSD. It is assumed that you have successfully installed joern into the directory `$JOERN` and Neo4J into `$NEO4J` as described in Installation. To begin, you can download and import the code as follows:

```
cd $JOERN
mkdir tutorial; cd tutorial
wget http://download.videolan.org/pub/videolan/vlc/2.1.4/vlc-2.1.4.tar.xz
tar xfJ vlc-2.1.4.tar.xz
cd ..
./joern tutorial/vlc-2.1.4/
```

Next, you need to point Neo4J to the generated data in `.joernIndex`. You can do this by editing the configuration file `org.neo4j.server.database.location` in the directory `$NEO4J/conf` as follows

```
# neo4j-server.properties
org.neo4j.server.database.location=$JOERN/.joernIndex/
```

Finally, please start the database server in a second terminal:

```
$NEO4J/bin/neo4j console
```

We will now take a brief look at how the code base has been stored in the database and then move on to joern-tools.

9.1.2 Exploring Database Contents

The Neo4J Rest API

Before we start using `joern-tools`, let's take a quick look at the way the code base has been stored in the database and how it can be accessed. `joern-tools` uses the web-based API to Neo4J (REST API) via the library `python-joern` that in turn wraps `py2neo`. When working with `joern-tools`, this will typically not be visible to you. However, to get an idea of what happens underneath, point your browser to:

```
http://localhost:7474/db/data/node/0
```

This is the *reference node*, which is the root node of the graph database. Starting from this node, the entire database contents can be accessed using your browser. In particular, you can get an overview of all existing edge types as well as the properties attached to nodes and edges. Of course, in practice, even for custom database queries, you will not want to use your browser to query the database. Instead, you can use the utility `joern-lookup` as illustrated in the next section.

Inspecting node and edge properties

To send custom queries to thedatabase, you can use the tool `joern-lookup`. By default, `joern-lookup` will perform node index lookups (see *Fast lookups using the Node Index*). For Gremlin queries, the `-g` flag can be specified. Let's begin by retrieving all nodes directly connected to the root node using a Gremlin query:

```
echo 'g.v(0).out()' | joern-lookup -g

(1 {"type":"Directory","filepath":"tutorial/vlc-2.1.4"})
```

If this works, you have successfully injected a Gremlin script into the Neo4J database using the REST API via `joern-tools` . Congratulations, btw. As you can see from the output, the reference node has a single child node. This node has two *attributes*: "type" and "filepath". In the joern database, each node has a "type" attribute, in this case "Directory". Directory nodes in particular have a second attribute, "filepath", which stores the complete path to the directory represented by this node.

Let's see where we can get by expanding outgoing edges:

```
# Syntax
# .outE(): outgoing Edges

echo 'g.v(0).out().outE()' | joern-lookup -g | sort | uniq -c

14 IS_PARENT_DIR_OF
```

This shows that, while the directory node only contains its path in the *filepath* attribute, it is connected to its subdirectories by edges of type *IS_PARENT_DIR_OF*, and thus its position in the directory hierarchy is encoded in the graph structure.

Filtering. Starting from a directory node, we can recursively enumerate all files it contains and filter them by name. For example, the following query returns all files in the directory 'demux':

```
# Syntax
# .filter(closure): allows you to filter incoming objects using the
# supplied closure, e.g., the anonymous function { it.type ==
# 'File'}. 'it' is the incoming pipe, which means you can treat it
# just like you would treat the return-value of out().
# loop(1){true}{true}: perform the preceeding traversal
# exhaustively and emit each node visited
```

```
echo 'g.v(0).out("IS_PARENT_DIR_OF").loop(1){true}{true}.filter{ it.filepath.contains("/demux/") }'
```

File nodes are linked to all definitions they contain, i.e., type, variable and function definitions. Before we look into functions, let's quickly take a look at the *node index*.

Fast lookups using the Node Index

Before we discuss function definitions, let's quickly take a look at the node index, which you will probably need to make use of in all but the most basic queries. Instead of walking the graph database from its root node, you can lookup nodes by their properties. Under the hood, this index is implemented as an Apache Lucene Index and thus you can make use of the full Lucene query language to retrieve nodes. Let's see some examples.

```
echo "type:File AND filepath:*demux*" | joern-lookup -c
```

```
echo 'queryNodeIndex("type:File AND filepath:*demux*")' | joern-lookup -g
```

Advantage:

```
echo 'queryNodeIndex("type:File AND filepath:*demux*").out().filter{it.type == "Function"}.name' | jo
```

9.1.3 Plotting Database Content

To enable users to familarize themselves with the database contents quickly, `joern-tools` offers utilities to retrieve graphs from the database and visualize them using *graphviz*.

Retrieve functions by name

```
echo 'getFunctionsByName("GetAoutBuffer").id' | joern-lookup -g | joern-location

/home/fabs/targets/vlc-2.1.4/modules/codec/mpeg_audio.c:526:0:19045:19685
/home/fabs/targets/vlc-2.1.4/modules/codec/dts.c:400:0:13847:14459
/home/fabs/targets/vlc-2.1.4/modules/codec/a52.c:381:0:12882:13297
```

Usage of the shorthand getFunctionsByName. Reference to python-joern.

```
echo 'getFunctionsByName("GetAoutBuffer").id' | joern-lookup -g | tail -n 1 | joern-plot-ast > foo.do
```

Plot abstract syntax tree

Take the first one, use joern-plot-ast to generate .dot-file of AST.

```
dot -Tsvg foo.dot -o ast.svg; eog ast.svg
```

Plot control flow graph

```
 echo 'getFunctionsByName("GetAoutBuffer").id' | joern-lookup -g | tail -n 1 | joern-plot-proggraph -
dot -Tsvg cfg.dot -o cfg.svg; eog cfg.svg
```

Show data flow edges

```
 echo 'getFunctionsByName("GetAoutBuffer").id' | joern-lookup -g | tail -n 1 | joern-plot-proggraph -
dot -Tsvg ddgAndCfg.dot -o ddgAndCfg.svg; eog ddgAndCfg.svg
```

Mark nodes of a program slice

```
echo 'getFunctionsByName("GetAoutBuffer").id' | joern-lookup -g | tail -n 1 | joern-plot-proggraph -c
dot -Tsvg slice.dot -o slice.svg;
```

Note: You may need to exchange the id: 1856423.

9.1.4 Selecting Functions by Name

Lookup functions by name

```
echo 'type:Function AND name:main' | joern-lookup
```

Use Wildcards:

```
echo 'type:Function AND name:*write*' | joern-lookup
```

Output all fields:

```
echo 'type:Function AND name:*write*' | joern-lookup -c
```

Output specific fields:

```
echo 'type:Function AND name:*write*' | joern-lookup -a name
```

Shorthand to list all functions:

```
joern-list-funcs
```

Shorthand to list all functions matching pattern:

```
joern-list-funcs -p '*write*
```

List signatures

echo "getFunctionASTsByName('*write*').code" | joern-lookup -g

9.1.5 Lookup by Function Content

Lookup functions by parameters:

```
echo "queryNodeIndex('type:Parameter AND code:*len*').functions().id" | joern-lookup -g
```

Shorthand:

```
echo "getFunctionsByParameter('*len*').id" | joern-lookup -g
```

From function-ids to locations: joern-location

```
echo "getFunctionsByParameter('*len*').id" | joern-lookup -g | joern-location
```

Dumping code to text-files:

```
echo "getFunctionsByParameter('*len*').id" | joern-lookup -g | joern-location | joern-code > dump.c
```

Zapping through locations in an editor:

```
echo "getFunctionsByParameter('*len*').id" | joern-lookup -g | joern-location | tail -n 2 | joern-edi
```

Need to be in the directory where code was imported or import using full paths.

Lookup functions by callees:

```
echo "getCallsTo('memcpy').functions().id" | joern-lookup -g
```

You can also use wildcards here. Of course, joern-location, joern-code and joern-editor can be used on function ids again to view the code.

List calls expressions:

```
echo "getCallsTo('memcpy').code" | joern-lookup -g
```

List arguments:

```
echo "getCallsTo('memcpy').ithArguments('2').code" | joern-lookup -g
```

9.1.6 Analyzing Function Syntax

- Plot of AST
- locate sub-trees and traverse to statements

9.1.7 Analyzing Statement Interaction

- some very basic traversals in the data flow graph

9.2 Finding Similar Functions with joern-tools

Embed functions in vector space.

- Represents functions by the API symbols used
- Applies TF-IDF weighting
- Dumps data in libsvm format

```
joern-stream-apiembedder
```

To allow this to scale to large code bases:

- database requests are chunked to not keep all results in memory at any point in time
- data is streamed onto disk

Determine nearest neighbors.

Get a list of available functions first:

```
joern-list-funcs
```

Get id of function by name:

```
joern-list-funcs -p VLCEyeTVPluginInitialize | awk -F "\t" '{print $2}'
```

where `VLCEyeTVPluginInitialize` is the name of the function in this example.

Lookup nearest neighbors.

```
joern-list-funcs -p VLCEyeTVPluginInitialize | awk -F "\t" '{print $2}' | joern-knn
```

Show location name or location.

```
joern-list-funcs -p VLCEyeTVPluginInitialize | awk -F "\t" '{print $2}' | joern-knn

joern-list-funcs -p VLCEyeTVPluginInitialize | awk -F "\t" '{print $2}' | joern-knn | joern-location
```

Dump code or open in editor.

```
joern-list-funcs -p VLCEyeTVPluginInitialize | awk -F "\t" '{print $2}' | joern-knn | joern-location

joern-list-funcs -p VLCEyeTVPluginInitialize | awk -F "\t" '{print $2}' | joern-knn | joern-location
```

CHAPTER 10

Articles

Why You Should Add Joern to Your Source Code Audit Toolkit (by Kelby Ludwig)