
joern Documentation

Release 0.2.5

Fabian Yamaguchi

Apr 12, 2017

Contents

1	Installation	3
1.1	System Requirements and Dependencies	3
1.2	Testing the server	4
1.3	Testing client scripts	4
2	Parsing and Importing Code	5
2.1	Importing code	5
2.2	Parsing code without importing	5
3	Accessing Code via Python	7
3.1	Basic Usage	7
3.2	Chunking Traversals	7
4	Database Overview	9
4.1	Code Property Graphs	9
4.2	Global Code Structure	10
5	Querying the Database	11
5.1	Gremlin Basics	11
5.2	Start Node Selection	12
5.3	Traversing Syntax Trees	12
5.4	Syntax-Only Descriptions	13
5.5	Traversing the Symbol Graph	13
5.6	Taint-Style Descriptions	14
6	Joern-tools	17
6.1	joern-slice	17
6.2	joern-apiembedder	18
6.3	joern-knn	18
6.4	joern-plot-prograph	19
7	Development	23
7.1	Accessing the GIT Repository	23
7.2	Build system and IDEs	23
8	Tutorials	25
8.1	Code Analysis with joern-tools (Work in progress)	25

8.2	Finding Similar Functions with joern-tools	28
9	Articles	31

Joern is a platform for robust analysis of C/C++ code developed by [Fabian Yamaguchi](#) and [Alwin Maier](#) at the [Institute of Systems Security](#) of the Technische Universitaet Braunschweig. It is part of the octopus project for graph-based program analysis tools. Joern 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 an OrientDB graph database. This allows code to be mined using search queries formulated in the graph traversal language Gremlin. In addition, long-running analysis tasks can be implemented as plugins for the platform.

- **Fuzzy Parsing.** Joern employs a fuzzy parser. This allows 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:

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 Debian Jessie, where OpenJDK-8 and Gradle 2 have been installed from jessie backports. 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 1.8.** Joern is written in Java 8 and does not build with Java 6 or 7. It has been tested with OpenJDK-8 but should also work fine with Oracle's JVM.
- **Python 3.** Joern implements a client/server architecture where client scripts are written in Python 3. Please note that these scripts are **not compatible with Python2**.
- **Python3-setuptools and python3-dev.** Client scripts are installed using setuptools. Moreover, some of the python libraries client tools depend on are written in C and require header files from python3-dev to be present.
- **Graphviz-dev.** Plotting tools require Graphviz and its development files to be installed.
- **Gradle 2.x.** Joern uses the gradle build tool, and some features specific to Gradle 2.0 and above.

If you are on a Debian-based system, try the following to download the necessary dependencies:

```
sudo apt-get install openjdk-8-jdk gradle python3 python3-setuptools python3-dev_
↪graphviz graphviz-dev
```

Please note, however, that Debian stable ("Jessie") currently does not include openjdk8 nor gradle 2 by default, so for Joern to work on Debian stable, please make use of Debian backports.

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

Building joern

Please make sure Gradle 2.x is installed. Then clone the repository and invoke the build script as follows. The build script will automatically download and install dependencies.

```
git clone https://github.com/octopus-platform/joern
cd joern
./build.sh
```

Testing the server

In the joern root directory, invoke the script

```
./joern-server.sh
```

to start the server.

Testing client scripts

Client scripts are installed into the user script directory, which is typically *~/.local/bin*. Please make sure this directory is in your path, e.g., by adding the line

```
export PATH="$PATH:~/.local/bin"
```

to your *~/.bashrc*, and restarting the shell. You can execute the script

```
joern-import
```

without parameters to verify that scripts are installed correctly.

Parsing and Importing Code

Importing code

Once joern has been installed, you can start the server and begin to import code into the database by executing *joern-import*. In one terminal, execute the joern server:

```
cd $JOERN
./joern-server
```

where *\$JOERN* is the joern root directory. In a second terminal, import the code as follows

```
`cd $JOERN`
`tar -cvf testCode.tar.gz testCode`
`joern-import testCode.tar.gz`
```

This will upload the tarball to the server, unpack it, parse the code and create a new project and corresponding graph database. The project name corresponds to the name of the tarball.

Parsing code without importing

In addition to offering a tool to automatically parse and import code into a graph database (*joern-import*), joern provides a tool to parse code and store its intermediate graph representation in a text file. The fuzzy parser can thus be used without the graph database backend, e.g., to provide input for other standalone code analysis tools.

To parse source code in the directory *\$codeDir*, simply invoke *joern-parse* as follows.

```
./joern-parse $codeDir
```

This will create a directory named *parsed*, which contains two files for each source file: a node file (*nodes.csv*) and an edge file (*edges.csv*).

Accessing Code via Python

Once code has been imported into the platform it can be accessed via the joern shell *josh*, a Java plugin API, and a python scripting API. In this section, we explain how the Python scripting API can be accessed.

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.

Basic Usage

Joern currently provides a single python class, 'DBInterface', that allows to connect to the database server and run queries. The following is a simple sample script that employs this interface to select a project, connect to the server and run a Gremlin query.

```
#!/usr/bin/env python3

from octopus.server.DBInterface import DBInterface

projectName = 'testCode.tar.gz'
query = "g.V.has('type', 'Function').code"

db = DBInterface()
db.connectToDatabase(projectName)

result = db.runGremlinQuery(query)
for x in result: print(x)
```

Chunking Traversals

Note: It is not clear whether this optimization is still necessary now that the code has been ported to Tinkerpop3.

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:

```
g.V.has('type', 'FunctionDef').astNodes().id
```

which retrieves all astNodes that are part of functions, 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:

```
#!/usr/bin/env python3

from octopus.server.DBInterface import DBInterface

projectName = 'testCode.tar.gz'
query = "g.V.has('type', 'FunctionDef').id"

db = DBInterface()
db.connectToDatabase(projectName)

ids = db.runGremlinQuery(query)

CHUNK_SIZE = 256
for chunk in db.chunks(ids, CHUNK_SIZE):
    query = """ idListToNodes(%s).astNodes().id """ % (chunk)
    for r in db.runGremlinQuery(query):
        print(r)
```

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

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.

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 [Wikipedia: 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.

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.

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. The remainder of this chapter deals with code retrieval based on syntax, taint-style queries and finally, traversals in the function symbol graph.

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://tinkerpop.apache.org/docs/3.0.1-incubating/>.

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 key.

```
// Lookup node with given key
g.V.has('_key', key)
```

Walking the graph can now be achieved by attaching so called *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.has('_key', key).out()

// Traverse to nodes two hops away.
g.V.has('_key', key).out().out()

// Traverse to nodes connected to start node by incoming edges
g.V.has('_key', key).in()

// All nodes connected by outgoing AST edges (filtering based
// on edge types)
g.V.has('_key', key).out(AST_EDGE)

// Filtering based on properties:
g.V.has('_key', key).out().has('type', typeOfInterest)

// Filtering based on edge properties
g.V.has('_key', key).outE(AST_EDGE).has(propKey, propValue).inV()
```

Start Node Selection

In practice, the ids or keys 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. For example, to retrieve all AST nodes representing callees with a name containing the substring `cpy`, one may issue the following query:

```
g.V.has('type', 'Callee').has('code', textRegex('.*cpy.*')).code
```

This is quite lengthy. Fortunately, once you identify a common operation, you can create a custom step to perform this operation in the future. We have collected common and generic steps of this type in the query library *joern-lang*, which you find in *projects/joern-lang*. In particular, we have defined the step *getCallsToRegex* in *lookup.groovy*, so the previous query can also be written as:

```
getCallsToRegex(".*cpy.*")
```

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

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, `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()
.hasRegex(NODE_TYPE, '.*Mul.*')
```

Syntax-Only Descriptions

The file `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 `foo` OR `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') }
```

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 Vulnerabilities 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 `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 `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.

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:

- The variable `len` is not re-defined on the way.
- 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.

You can use the following traversal to achieve this:

```
getArguments('memcpy', '2')
  .sideEffect{ paramName = '.*len.*' }
  .unsanitized({ it, s -> it.isCheck(paramName) })
  .match{ it.type == "Parameter" && it.code.matches(paramName) }.code
```

where `isCheck` is a traversal defined in `misc.groovy` to check if a symbol occurs inside an equality or relational expression and `match` looks for nodes matches the closure passed to it in the given syntax tree.

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.

joern-slice

SYNOPSIS

joern-slice [options]

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.

joern-apiembedder

SYNOPSIS

joern-apiembedder [options]

joern-stream-apiembedder [options]

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>

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.

joern-knn

SYNOPSIS

joern-knn [options]

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.

OPTIONS

-l <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>

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 contain the lines:

int int

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

joern-plot-proggraph

SYNOPSIS

```
joern-plot-proggraph [ -h ] [ -f FILE ] [ -o OUT ] [ -ast ] [ -cfg ] [ -dfg ] [ -ddg ] [ -cdg ] [ -dom ]
[ -all ] [ -P ] [ -c PLOT_CONFIG ] project
```

DESCRIPTION

`joern-plot-proggraph` lets you plot various program graphs, but restricted to one plot per function. Retrieves a graph representation of a function with the given id. The default output format is graphviz's 'dot'.

OPTIONS

positional arguments:

project, the name of the Joern project.

optional arguments

- h, --help** show this help message and exit
- f FILE, --file FILE** read input from the provided file
- o OUT, --out OUT** write output to provided file
- ast, --show-ast** Show AST in CFG nodes.
- cfg, --show-control-flow** Show control flow.
- dfg, --show-data-flow** Show data flow.
- ddg, --show-data-dependences** Show data dependences.
- cdg, --show-control-dependences** Show control dependences.
- dom, --show-dominance-edges** Show dominance edges.
- all, --show-all** Show all edge types

- P, --id-property use functionId property value to identify function
- c PLOT_CONFIG, --plot-config PLOT_CONFIG use plot configuration from file

PLOT CONFIGURATION

The default plot configuration can be found in the directory `scripts/data/plotconfig.cfg`. The config consists of lines of the following format:

element_type . rule_type = pattern : [+] { [& | -] property [, ...] | param [, ...] }

element_type: graph element, can be node or edge.

rule_type: display to tell which properties are shown in the graph, layout to determine graphviz layout options.

pattern:

***** : matches any element

prop . val : matches if the property prop has value val. If value is * then any value of the field results in a match.

if the property or parameter list starts with a +, the result will be added to the result of previous matching rules. If + is omitted, the current result will be replaced.

fields can start with &, in which case the property label will be displayed. fields can start with -, in which case the property will be removed from the current results.

Lines that start with optional whitespace followed by # are comments and not processed.

Example

```
# comment lines are possible
# for all nodes, show childNum, id, type and code properties, without property keys
node.display=*:childNum,id,type,code
# if we wanted property keys, use
# node.display=*&childNum,&id,&type,&code
node.layout=*:shape=rectangle,style=filled,fillcolor=white
node.layout=isCFGNode.True:++fillcolor=lightcyan
# for CFGEntryNode and CFGExitNode, do not show childNum and type, but keep the rest
node.display=type.CFGEntryNode:+-childNum,-type
node.display=type.CFGExitNode:+-childNum,-type
# keep the layout parameters from earlier matches (+), change fillcolor and add
↪fontcolor
node.layout=type.CFGEntryNode:++fillcolor=slategray,fontcolor=white
node.layout=type.CFGExitNode:++fillcolor=black,fontcolor=white
node.layout=type.Symbol:++shape=ellipse,fillcolor=wheat
# this overrides the display options for Symbol nodes
node.display=type.Symbol:code
edge.display=*:label
edge.display=label.IS_AST_PARENT:
edge.layout=label.IS_AST_PARENT:color=gray
# this clears all display properties for FLOWS_TO edges
edge.display=label.FLOWS_TO:
edge.layout=label.FLOWS_TO:color=black
edge.layout=label.USE:color=lightpink,fontcolor=lightpink
edge.layout=label.DEF:color=deeppink,fontcolor=deeppink
edge.layout=label.DOM:color=navy,fontcolor=navy
edge.layout=label.POST_DOM:color=deepskyblue,fontcolor=deepskyblue
```



```
edge.layout=label.CONTROLS:color=seagreen,fontcolor=seagreen
edge.display=label.REACHES:+var
edge.layout=label.REACHES:color=darkolivegreen,fontcolor=darkolivegreen
```


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/octopus-platform/joern.git
```

If you want to report issues or suggest new features, please do so via <https://github.com/octopus-platform/joern>. For fixes, please fork the repository and issue a pull request.

Build system and IDEs

Joern is a multi-module Gradle project where Java/Groovy sub-projects are located in the directory 'projects', while python projects reside in the directory 'python'.

To start hacking on Joern, make sure you can build it using the supplied build script *build.sh*. For small modifications, it may be sufficient to edit the source files using a simple text editor, and subsequently invoking this script.

For larger changes, please consider using a JAVA IDE such as IntelliJ or Eclipse. We use both of these IDEs for Joern-development on a regular basis, so the import should hopefully be possible without trouble using the corresponding Gradle plugin. IntelliJ typically comes with a gradle plugin pre-installed, and Eclipse offers a plugin in its "Marketplace".

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 lengthy 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`.

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` 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
tar zcf vlc-2.1.4.tar.gz vlc-2.1.4/
cd ..
```

Next, start the `joern-server`:

```
./joern-server
```

Open a new terminal and import the code:

```
cd $JOERN
joern-import tutorial/vlc-2.1.4.tar.gz
```

Exploring Database Contents

Inspecting node and edge properties

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 'g.V().has("type", "File").hasRegex("code", ".*demux.*").code' | joern-lookup_
↪ vlc-2.1.4.tar.gz
```

Advantage:

```
echo 'g.V().has("type", "File").hasRegex("code", ".*demux.*").out().has("type",
↪ "Function").code' | joern-lookup vlc-2.1.4.tar.gz
```

Plotting Database Content

To enable users to familiarize 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 vlc-2.1.4.tar.gz | 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.dot
```

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 -cfg > cfg.dot;
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 -ddg -cfg > ddgAndCfg.dot;
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 -ddg -cfg | joern-plot-slice 1856423 'p_buf' > slice.dot;
dot -Tsvg slice.dot -o slice.svg;
```

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

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
```

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-editor
```

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
```

Analyzing Function Syntax

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

Analyzing Statement Interaction

- some very basic traversals in the data flow graph

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-code
```

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


CHAPTER 9

Articles

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