bjoern Documentation

Release 0.1

Fabian Yamaguchi

June 24, 2016

Bjoern is a platform for the analysis of binary code. The platform ports, extends, and reimplements concepts of the source code analysis platform joern. It generates a graph representation of native code directly from output provided by the open-source reverse engineering toolchain radare2, and stores it in an OrientDB graph database for subsequent mining with graph database queries.

Contents:

CHAPTER 1

Overview

The Bjoern platform consists of a server component, and a collection of client utilities which can be used to query and update the database contents. All heavy computation is performed in threads on the server side, which are spawned in response to client requests issued via a HTTP REST API. These threads can directly access the database using a lightweight binary protocol ("plocal"), a protocol that introduces only low overhead when compared to standard HTTP-based access protocols.

Access to the database is synchronized to allow multiple users and long-running analysis tasks to be executed in parallel. For multiple user access, bjoern implements a groovy shell server. This makes it possible for users to execute commands on the server side, disconnect, and come back later to view results, similar in style to the way GNU screen sessions are used on shared servers.

At heart, the Bjoern is an OrientDB server instance, extended with plugins that are loaded at startup. These plugins expose the platforms functionality via a REST API, which makes it possible to invoke server functionality via HTTP requests. Bjoern-server extends the language-agnostic server component Octopus with plugins for binary analysis and provides a language for binary code analysis. In the following, we describe the main components Bjoern is composed of.

1.1 Octopus

Octopus is a server component that provides shell access to an OrientDB graph database. It allows arbitrary property graphs to be imported from CSV files that describe nodes and edges. This is achieved using the OrientDBImporter, a generic library for batch importing large property graphs into OrientDB.

In summary, octopus offers the following two primary features.

- **Import of CSV files.** The CSV importer enables the user to perform batch imports of graphs given in CSV files. It is a generic component implemented to allow fast import of any property graph into an OrientDB graph database. The importer runs in a thread on the server-side to access the database without the overhead introduced by OrientDB's access protocols. Several databases can be created in parallel using the importer plugin.
- **Shell Access.** Octopus offers a server-side shell that provides database access via the general purpose scripting language Groovy and the traversal language Gremlin. Like importers, shells run as threads inside the server, giving them access to the database with low overhead. Multiple users can spawn shells on the server side to work on the database in parallel.
- **Execution of plugins.** Octopus offers a plugin interface that can be used to extend functionality at runtime. In particular, this allows language-dependent analysis algorithms to be executed on the graph database contents.

1.2 Bjoern-Radare

Bjoern-Radare generates graph-based program representations from binaries and outputs them in a CSV format. The resulting files can be imported into the octopus server. Under the hood, bjoern-radare uses radare2 to perform an initial automatic analysis of a binary and extract symbol information, control flow graphs, and call graphs. Moreover, it translates machine code into radare's intermediate language ESIL to allow platform independent analysis of code.

1.3 Bjoern-plugins

Bjoern-plugins are a set of plugins that turn Octopus into a platform for binary code analysis. On the one hand, these plugins allow structures such as control flow graphs of functions to be exported, on the other, they perform active computations on the database contents to generate new nodes and edges.

1.4 Bjoern-lang and Octopus-lang

Bjoern-lang and Octopus-lang provide a domain specific language for binary code analysis and generic traversal of property graphs respectively. These languages are realized as so called *steps* for the graph-traversal language Gremlin.

1.5 Bjoern-Shell

Bjoern-shell provides user-friendly access to the database via a command line shell. It features a working environment with convenient features like code completion, reverse search, and in-shell documentation. It also features an awesome banner. Multiple bjoern-shells can be used in parallel. Moreover, it is possible to detach from a bjoern-shell and re-attach to it later on.

CHAPTER 2

Installation

2.1 System Requirements and Dependencies

- **A Java Virtual Machine 1.8.** Bjoern is written in Java 8 and does not build with Java 7. It has been tested with OpenJDK 8 but should also work with Oracle's JVM.
- **Radare2** The primitives provided by the radare2 reverse engineering framework are employed to dissect and analyze binary files to obtain graph-based program representations from them.
- **OrientDB 2.1.5 Community Edition.** The bjoern-server is based on OrientDB version *2.1.5* and has not been tested with any other version. You can download the correct version here .
- **Bjoern-shell [Optional.]** The bjoern-shell is a convenient tool to query the database contents and develop new query-primitives (so-called steps) that can be re-used in subsequent queries.

2.2 Installing radare2

Please follow the instructions here to install radare2, and make sure the programs *radare2* and *r2* are in the path.

2.3 Building bjoern (step-by-step)

```
git clone https://github.com/fabsx00/bjoern
cd bjoern
gradle deploy
```

2.4 Installing the bjoern-shell

```
git clone https://github.com/a0x77n/bjoern-shell
cd bjoern-shell
python3 setup.py install
bjosh
```

First steps (Tutorial)

The following tutorial illustrates basic usage of bjoern. You will learn how to start the server, import code, spawn a bjoern-shell and run queries against the database.

1. Begin by starting the bjoern-server:

```
./bjoern-server.sh
```

This starts an orientDB server instance, along with the OrientDB Studio on port 2480. Studio provides a useful interface to explore the database contents (see http://orientdb.com/docs/last/Home-page.html).

2. In another shell, import some code

```
./bjoern-import.sh /bin/ls
```

This will start a thread inside the server process which performs the import. You will see an 'Import finished' message in the server log upon completion.

3. Create a shell thread using bjosh

```
bjosh create
```

4. Connect to the shell process using bjosh

```
bjosh connect
```

5. Get names of all functions (sample query)

```
 _     _           _
| |__ (_) ___  ___| |__
| '_ \| |/ _ \/ __| '_ \
| |_) | | (_) \__ \ | | |
|_.__// |\___/|___/_| |_|
    |__/      bjoern shell


bjoern> queryNodeIndex('nodeType:Func').repr
```

6. Get all calls

```
getCallsTo('').map
```

7. Get basic blocks containing calls to 'malloc'

```
getCallsTo('malloc').in('IS_BB_OF').repr
```

8. Walk to first instruction of each function

```
getFunctions('').funcToInstr().repr
```

CHAPTER 4

Database Contents

CHAPTER 5

CSV Import

The CSV importer allows property graphs to be imported into the graph database. It requires nodes and edges to be specified in a CSV format.

5.1 Usage

The importer can be invoked via an HTTP GET request or via the utility 'octopus-csvimport.sh' in *projects/octopus*.

The HTTP GET request can be issued with curl as follows

```
curl http://localhost:2480/importcsv/<nodeFilename>/<edgeFilename><dbname>/
```

where nodeFilename is a CSV file containing nodes, edgeFilename is a CSV file containing edges, and dbname is the name of the database to import into.

Alternatively, the script 'octopus-csvimport.sh' can be invoked as follows

```
projects/octopus/octopus-csvimport.sh [dbname]
```

where dbname is the name of the database. The tool will automatically impor the files 'nodes.csv' and 'edges.csv' if present in the current working directory.

5.2 Configuration

none.

5.3 Input Format for Nodes

Nodes are described using a CSV file format, where the first line describes the row format (CSV header), and the remaining lines contain the actual nodes. The tabular character is used as a deliminator. Double-quotes can be used to enclose values of fields that contain newlines or tabs.

The CSV header has two mandatory fields: *command*, and *key*.

The *command* field specifies the action to perform for this node. The following commands are currently supported:

Name	Description
ANR	Add node, replacing any existing node with the same key.
A	Add node, creating an alternative key if a node with this key already exists. The alternative key is generated by adding an underscore followed by a number to the key.

The *key* field contains an identifier for the node. The key can be an arbitrary string. The strategy to follow when a node with this key already exists depends on the command.

The remaining fields specify the names of node properties. As an example, please take a look at nodes.csv as generated by the *bjoern-radare.sh* tool.

5.4 Input Format for Edges

Edges between nodes are described in a CSV file, where nodes are referred to by their keys. The first line of the CSV file (CSV header) describes the row format.

The CSV header has three mandatory fields: *sourcekey*, *destkey*, and *edgeType*.

The *sourcekey* and *destkey* fields specify the key of the edge's source and destination node respectively, while *edgeType* specifies the type of the edge as an arbitrary string.

The remaining fields are the names of edge properties. As an example, please take a look at edges.csb as generated by *bjoern-radare.sh*.

CHAPTER 6

Shell Access (Gremlin)

Octopus provides access to Gremlin shells that can be used to query the database. Shells run inside the server process and can therefore make use of the ‘plocal’ binary protocol for efficient access.

6.1 Usage

The shells currently running inside the server process can be listed using the *listshells* command as follows:

```
curl http://localhost:2480/listshells
```

A new shell can be created using the *shellcreate* command as follows.

```
curl http://localhost:2480/shellcreate/[dbname]
```

where *dbname* is the name of the database to connect to. By default, a shell for *bjoernDB* is created.

6.2 Configuration

none

CHAPTER 7

Plugins

Plugins are the best way to extend the Octopus server with custom functionality. You can write own plugins or to use the existing ones.

7.1 Execution of Plugins

Plugins are invoked via HTTP POST requests. The body of the request message contains the plugin's configuration in JSON format. It includes information about how to execute the plugin (required by all plugins) and additional settings for the plugin (dependent on the specific plugin). For example

```
{
    "plugin": <name of the plugin>
    "class": <class implementing the IPlugin interface>,
    "settings": <JSON object containing plugin specific settings>
}
```

The configuration file is used by the *executeplugin* command of the Octopus server, which loads and executes the plugin. The POST request can be issued using curl

```
cat plugin.json | curl -d @- http://localhost:2480/executeplugin/
```

where *plugin.json* contains the configuration.

7.2 The Function Export Plugin

The function export plugin can be used to export database content at function level. Functions consist of a function node, basic blocks and instruction nodes along with edges between those nodes. It is possible to export functions as a whole or only parts of a function, e.g., the control flow graph.

7.2.1 Configuration

The plugins configuration file contains the following data:

```
{
    "plugin": "functionexport.jar",
    "class": "bjoern.plugins.functionexporter.FunctionExportPlugin",
    "settings": {
        "database": <database name>,
        "format": "dot"|"graphml"|"gml",
        "destination": "<output directory>,
        "threads": <number of threads to use>,
        "nodes": <JSON array of node types>,
        "edges": <JSON array of edge types>
    }
}
```

Note: Edges are only exported if the head and the tail is exported as well.

7.2.2 Example

To extract the control flow graphs of all functions of a database named *ls* you can start with the settings below:

```
{
    "plugin": "functionexport.jar",
    "class": "bjoern.plugins.functionexporter.FunctionExportPlugin",
    "settings": {
        "database": "ls",
        "format": "dot",
        "destination": "some/path/you/like",
        "threads": "4",
        "nodes": ["BB"],
        "edges": ["CFLOW_ALWAYS", "CFLOW_TRUE", "CFLOW_FALSE"]
    }
}
```

7.3 The Instruction Linker Plugin

The instruction linker plugin connects the instructions of a function accordingly to the execution order and the flow of control. This is useful to obtain control flow information at the level of instructions as opposed to basic blocks.

7.3.1 Configuration

The plugins configuration file contains the following data:

```
{
    "plugin": "instructionlinker.jar",
    "class": "bjoern.plugins.instructionlinker.InstructionLinkerPlugin",
    "settings": {
        "database": <database name>,
    }
}
```

7.4 Writing Plugins

All plugins must implement the *IPlugin* interface (*octopus.server.components.pluginInterface.IPlugin*). The interface specifies the following four methods:

Method	Description
configure	This method is used to configure the plugin. The only argument passed to this method is the JSON object specified by the settings attribute of the configuration file.
execute	This method contains the main code of the plugin.
beforeExecution	This method is called before the execution of the plugin.
afterExecution	This method is called after the execution of the plugin.

The methods are invoked in the following order: *configure*, *beforeExecution*, *execute*, *afterExecution*.

Most plugins will require access to some database. The class *OrientGraphConnectionPlugin* (*bjoern.pluginlib.OrientGraphConnectionPlugin*) implements the *IPlugin* interface and opens a connection to a graph database in *beforeExecution*. The connection is closed in *afterExecution*. The name of the database is read in *configure*, the corresponding attribute must be named *database*. The class *OrientGraphConnectionPlugin provides two additional methods to acquire a graph instance: ‘getGraphInstance* and *getNoTxGraphInstance* for non-transactional graphs and transactional graphs, respectively.

Note: If you override any other method of the *IPlugin* interface, make sure you don’t forget to call *super*.