

Pigaios: A Tool for Diffing Source Codes against Binaries

Joxean Koret
Hacktivity 2018

Diffing Source Codes to Binaries

- Motivation
- The Tool
- Heuristics
- Problems
- Demo
- Future

Motivation

Motivation

Often, while doing reverse engineering projects, I have had the need to:

1. Apply symbols from multiple Open Source projects that are used in my target.
2. Apply symbols from partial or uncompileable source codes to my target.

Random examples: leaked source code or old source that cannot be easily compiled today.

What I have always done was to try to build a binary, open it in IDA, export symbols with Diaphora (or BinDiff) and, then, port the symbols to my target.

Often, however, it's simply not possible (partial or uncompileable source codes).

Motivation

What can we do whenever we have partial and/or non compilable source code to port symbols from that codebase to our (binary) target?

- Either we build small parts of the source code to object files and use a binary diffing tool like Diaphora with each object file...
- Or “someone” writes a tool to directly port symbols from non compilable or partially compilable source codes to binaries.

As I’m masochistic, I decided to take option #2.

Motivation

I had this idea when I was reverse engineering some unnamed security product and I realized there was a big source code leak happened ~10 years ago.

I tried to build the software so I could use Diaphora to port symbols from binaries compiled with symbols to my IDA databases. Alas, it wasn't possible:

- The compilers used for building that code base dated from more than 10 years ago.
- There were various headers and libraries that were required for compilation or linking phases that I didn't have, as the leak was partial.
- Even if I could somehow compile portions of the old source code, I would be able to only port symbols from those files that I managed to compile.
 - Of course, these source files were the less complex and less interesting ones.

Motivation

After a while, I gave up trying to build the source code and started “porting” symbols by hand:

- Finding this or that string constant and the functions where it was used in the source code and in the binaries.
- Finding nearby functions.
- Finding callers and callees of some functions that I already discovered.

At some point I gave up too and said to myself “this is ridiculous, I should automate it somehow”.

I thought about creating such a tool in 2015. I finally started working on it in 2017.

The Tool

The Tool: Pigaio

As previously explained, I started working on this tool, named Pigaio (Greek for “source” as in “source code”) in 2017.

The first, rudimentary, version of the tool took me ~1 month to write. Mainly because I have had to write everything around “working with source codes” as I only had such tools for binaries.

After I had a basic and rudimentary framework to extract “things” from source codes, I was able to write my first prototype as it was on my mind.

But, how does such a tool work?

The Tool

The process of porting symbols and/or diffing source codes against binaries, summarized, is the following:

1. Parse the binaries and the source codes.
2. Extract artifacts for each function found in each source code.
3. Extract the same artifacts for each function found in the binaries.
4. Find matches between source code and binary functions based on the extracted artifacts and assign an accuracy ratio to each match.

And, basically, it's all what the tool does. However, it's a bit more complex than that and many problems appear at each step.

Parsing

We can easily extract information from binaries using IDA for each function, basic block, etc...

Unfortunately, there isn't an IDA like tool for source codes. The closer can be "SciTools Understand", but is not even remotely close.

So, for parsing source codes, first we need to build some tool or library to be able to do this task.

Parsing Source Codes

This process is far from being an already solved process.

- Parsing one dialect of one programming language is *easy*.
- Parsing any and all dialects of any and all programming languages that you want to support in your tool is a nightmare.

There are so many examples that come to mind by just thinking about the C language...

Problems Parsing C Source Codes

- Compiler extensions that are incompatible between different compilers.
 - `#pragmas` are some of the best examples.
 - Pre-compiled headers (Microsoft).
 - Omitting the middle operand of a ternary expression `-> X ? : Y` (GCC).
 - Embedded assembler differences (GCC, CLang, Microsoft).
 - Switch case ranges (case 1..9).
 - Out of class/struct definition of static const integral or enum members (Microsoft).
 - ...
- Ugly workarounds for specific compilers and specific compiler bugs. Not as rare as you might think. Try the following searches:
 - `site:github.com workaround for gcc`
 - `site:github.com workaround for clang`
 - `site:github.com "workaround for" "visual c"`

Parsing Source Codes

In my case I *just* want to parse C source codes. There is a number of options:

- Use Pycparser or any other similarly limited parser. I don't recommend doing so unless it's for a quick prototype or a "use and throw" tool.
- Use Flex, Bison, ANTLR, Ragel, YourOwnOne TM... You will discover soon that writing a parser is a nightmare, you will never be done with it and you will hate yourself for years to come.
- Use a fuzzy parser like Joern, from Fabian Yamaguchi, or Scitools Understand. They have less rules and are easier to write but, again, you will never be done with the parser and you will be focusing in writing a parser, not the tool you want.
- ...

Parsing Source Codes

- Use GCCXML or any other old and unmaintained tool based on GCC. Not recommended for several reasons, being one of them that GCC is terribly complex to understand.
- If you have a rich uncle in America or if you're in a University, you can be lucky enough to get a license of the de-facto C/C++ compiler front-end: Edison Design Group. They support pretty much any computer dialect by even supporting specific compiler + versions bugs.
- Otherwise, the only remaining choice is CLang and its bindings or the C/C++ APIs. Using CLang you will have support for whatever CLang supports and you will not have support for whatever CLang does not support.
 - That was my choice, for obvious reasons.

The CLang Python Bindings (or APIs)

Pros:

- Both Linux, MacOSX, Windows and FreeBSD are officially supported.
- You get an Abstract Syntax Tree (AST) even for files or functions with errors.
- You get support for anything the CLang front-end(s) support, like compilers extensions.

Cons:

- The CLang developers don't really care about them.
- You often need to resort to parsing tokens instead of using “documented” APIs because the developers didn't expose this or that “thing”.
 - For example, you don't even have a legal way to get the value for an `INTEGER_LITERAL`!

Extracting Artifacts

Using the Clang Python bindings to parse the source code, artifacts are extracted from each functions AST.

Using IDA Python, binaries “are parsed” and the same artifacts are extracted.

- I have used IDA, but one could use Binary Ninja or Radare2 or whatever other tool that can extract the same artifacts properly.

But... what “artifacts” are extracted?

Extracting artifacts

In the current version of Pigaios, only the following artifacts are extracted:

- String constants.
- Number of loops, conditions, function calls, externals and globals.
- Number of switches and their cases.
- If the function is recursive or not.
- List of callees.

Finding Matches

Once we “parsed” both source codes and binaries, we can just compare the extracted artifacts to find matches.

This is how I find the initial matches:

```
348 def find_initial_rows(self):
349     cur = self.db.cursor()
350     sql = """ select bin.ea, src.name, src.id, bin.id
351                from functions bin,
352                     src.functions src
353                where (bin.conditions between src.conditions and src.conditions + 3
354                      or bin.name = src.name)
355                      and bin.constants = src.constants
356                      and bin.constants_json = src.constants_json
357                      and (select count(*) from src.functions x where x.constants_json = src.constants_json) < %d
358                      and src.constants_json != '[]'
359                      and src.constants > 0
360                      and src.conditions > 1
361                      and bin.loops = src.loops """
```

Finding Matches

The previous query tries to find almost 1 to 1 matches between a source code based SQLite database and a binary based SQLite database. These initial results, naturally, are pretty scarce:



Joxean Koret

@matalaz



First quick & dirty proof-of-concept comparing C source codes to binaries. 90 matches out of 4817 functions with 0 false positives. Not that bad.

20:34 - 2017 aza. 30

Finding matches

Matching just 90 functions out of 4,817 might look like bad results, but considering that these matches have zero false positives, it's actually good **initial** results.

Starting with near zero false positive results we can then use some heuristics to find more matches.

Naturally, the matches based on heuristics will be *less good* than the ones we found with the previous SQL queries.

It means that we have another problem: calculating a similarity/accuracy ratio for matches. But let's first focus on how to develop heuristics for finding more matches.

Heuristics

Heuristics

All heuristics (but one) are based on the initial good results previously explained. Currently, the following heuristics are being used in Pigaio:

- Caller/callee match.
- Nearby function.
- Specific callee search.
- Same rare constant.

These heuristics are applied iteratively for any new results until no more new results are found (i.e.: If we find 1 new function with each of the previous heuristics, it will apply the heuristics using the newly discovered functions).

Heuristic: Call-graph match (caller/callee, iteration \$i)

Starting from a 1 to 1 match between a codebase and a binary, build the list of callers and callees for the function in both the source code and the binary.

Try to match each caller/callee from both and calculate some 'score' to determine how good or bad the match.

For “good enough” matches, add them to a temporary list.

After all the callers and callees are processed, remove the bad matches and only consider the good ones.

If there are multi-matches, only consider the best matches (or the first match if both are equally good).

Heuristic: Nearby Function

If a function Y in the codebase is in the file foo.c and is between functions X and Z, chances are that function Y' in the binary is between functions X' and Z'.

As simple as it sounds. The current implementation of this algorithm is too conservative and it stops searching for nearby functions when the first match going up or down seems to be a bad match.

It can be improved by a lot.

Heuristic: Specific Callee Search

The callers of a function in a codebase and the callers in a binary can differ a lot.

The callees of a function in a codebase and the callees in a binary, however, tend not to differ too much.

As so, if we have a good match between X (source) and X' (binary), chances are that source callees Y and Z are the same as the binary callees Y' and Z'.

Simple, isn't it?

Heuristic: Same rare constant

This is the only heuristic that doesn't actually rely in previous matches.

For uncommon string constants (less than 3 appearances in the source code or binary), find those that reference it in both the source codes and the binaries.

These are usually very-very good matches.

Heuristics

And these are, so far, all the heuristic that I'm using in the current version of Pigaio.

These heuristics find many matches between source codes and binaries with a greatly varying quality.

And here is where another problem (that I already briefly mentioned before) appears again: how can we rate how accurate or the opposite are matches?

We will talk about it at some point in the next section...

Problems

Problems

Porting symbols from codebases to binaries is “problematic”. Surprise. Some of the many problems I have discovered:

- CFGs based heuristics (used in Diaphora, BinDiff, Darun Grim, ...) cannot be used to compare source codes (especially from human written ones) to binaries.
- ASTs based heuristics (used in Diaphora) cannot be used to compare source codes to binaries.
- Blind Diffing: How can I graphically diff when I don't have pseudo-code?
- It's almost impossible to determine how much a function “changed” from the source to the binary, thus, *it isn't easy* to determine how good or bad a match is.

Control Flow Graphs

One of my first ideas was to port various graph based heuristics from Diaphora to Pigaio.

I quickly realized that it was a bad idea most of the time. Why?

Because CFGs obtained from source codes, especially from human written ones, are totally different to CFGs obtained after optimizing compilers process sources.

Control Flow Graphs

Easier to see with (an extreme) graphical example of compilers optimization:

Source code:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    if ( getuid() != 0 && geteuid == 0 )
    {
        printf("Only root!\n");
        exit(1);
    }
    printf("OK!\n");
}
```

Pseudo-code:

```
1 int __cdecl main()
2 {
3     getuid();
4     return puts("OK!");
5 }
```

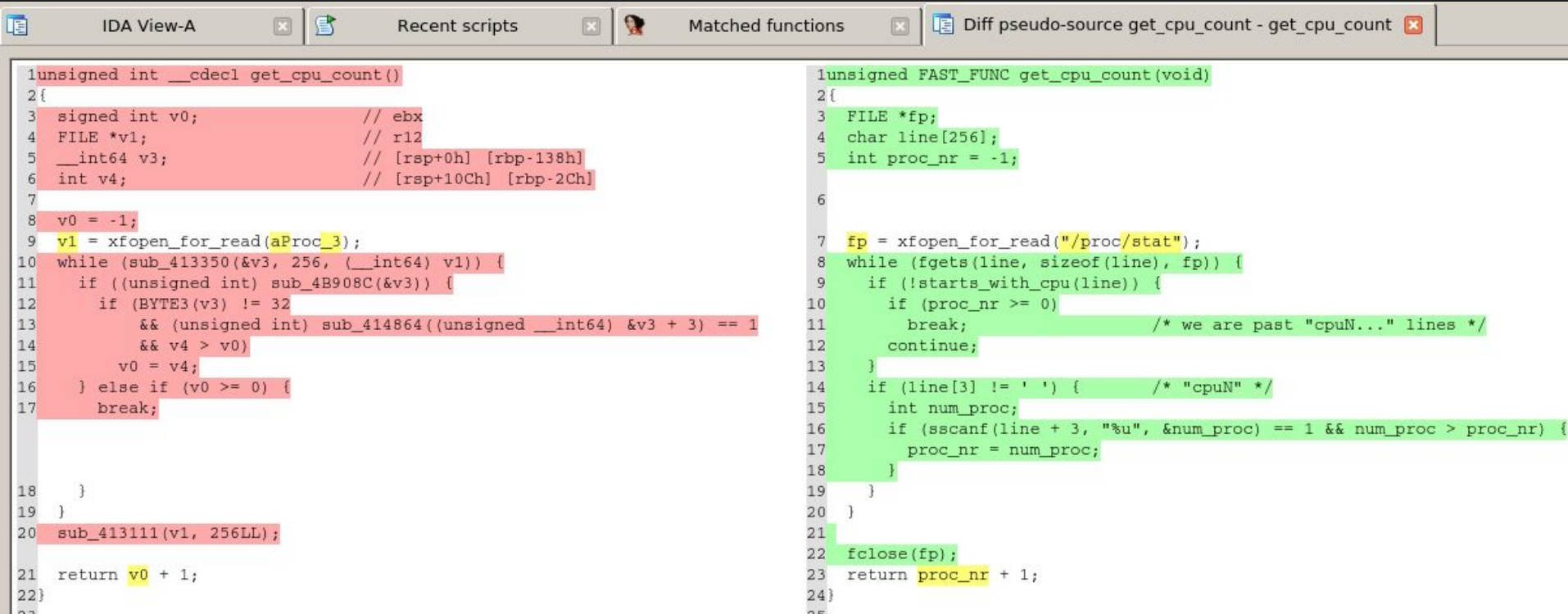

Abstract Syntax Trees

ASTs from source codes, especially from human written codes, are very different to ASTs obtained with an optimizing decompiler, like the Hex-Rays one, that takes as input machine code already optimized by a compiler.

Actually, even for trivial functions, trying to ‘diff’ source codes to pseudo-codes proves to be ‘complicated’...

Abstract Syntax Trees

A non extreme graphical example:



```
IDA View-A  Recent scripts  Matched functions  Diff pseudo-source get_cpu_count - get_cpu_count

1 unsigned int __cdecl get_cpu_count()
2 {
3     signed int v0;           // ebx
4     FILE *v1;                // r12
5     __int64 v3;               // [rsp+0h] [rbp-138h]
6     int v4;                   // [rsp+10Ch] [rbp-2Ch]
7
8     v0 = -1;
9     v1 = xfopen_for_read(aProc_3);
10    while (sub_413350(&v3, 256, (__int64) v1)) {
11        if ((unsigned int) sub_4B908C(&v3)) {
12            if (BYTE3(v3) != 32
13                && (unsigned int) sub_414864((unsigned __int64) &v3 + 3) == 1
14                && v4 > v0)
15                v0 = v4;
16        } else if (v0 >= 0) {
17            break;
18        }
19    }
20    sub_413111(v1, 256LL);
21    return v0 + 1;
22}

1 unsigned FAST_FUNC get_cpu_count(void)
2 {
3     FILE *fp;
4     char line[256];
5     int proc_nr = -1;
6
7     fp = xfopen_for_read("/proc/stat");
8     while (fgets(line, sizeof(line), fp)) {
9         if (!starts_with_cpu(line)) {
10             if (proc_nr >= 0)
11                 break; /* we are past "cpuN..." lines */
12             continue;
13         }
14         if (line[3] != ' ') { /* "cpuN" */
15             int num_proc;
16             if (sscanf(line + 3, "%u", &num_proc) == 1 && num_proc > proc_nr) {
17                 proc_nr = num_proc;
18             }
19         }
20     }
21
22     fclose(fp);
23     return proc_nr + 1;
24}
25
```

Abstract Syntax Trees

Another non extreme graphical example:

Diff pseudo-source gzerror - sub_140001F90

```
1 const char * __fastcall sub_140001F90( __int64 a1, _DWORD * a2)
2 {
3     const char *result;           // rax
4     int v3;                       // eax
5     const char *v4;               // rcx
6
7     if (!a1)
8         return 0i 64;
9     v3 = *(_DWORD *) (a1 + 24);
10    if (v3 != 7247 && v3 != 31153)
11        return 0i 64;
12    if (a2)
13        *a2 = *(_DWORD *) (a1 + 108);
14    if (*(_DWORD *) (a1 + 108) == -4)
15        return "out of memory";
16    v4 = *(const char **) (a1 + 112);
17    result = (const char *) &unk_14000C3BE;
18    if (v4)
19        result = v4;
20    return result;
21 }
22
```

```
1 const char *ZEXPORT gzerror(file, errnum)
2 gzFile file;
3 int *errnum;
4 {
5     gz_statep state;
6
7     /* get internal structure and check integrity */
8     if (file == NULL)
9         return NULL;
10    state = (gz_statep) file;
11    if (state->mode != GZ_READ && state->mode != GZ_WRITE)
12        return NULL;
13
14    /* return error information */
15    if (errnum != NULL)
16        *errnum = state->err;
17    return state->err == Z_MEM_ERROR ? "out of memory" :
18        (state->msg == NULL ? "" : state->msg);
19 }
20
```

Abstract Syntax Trees

Only in some rare cases the ASTs look roughly the same:

Diff pseudo-source gz_look - sub_1400023D0

```
1 signed __int64 __fastcall sub_1400023D0(__int64 a1)
2 {
3     __int64 v1;           // rbx
4     void *v2;             // rax
5     signed __int64 result; // rax
6     unsigned int v4;       // eax
7     _BYTE *v5;            // rax
8     void *v6;            // rcx
9     unsigned int v7;       // eax
10
11     v1 = a1;
12     if (!(*_DWORD *) (a1 + 40)) {
13         *(_QWORD *) (a1 + 48) = malloc(*(unsigned int *) (a1 + 44));
14         v2 = malloc((unsigned int) (2 * *(_DWORD *) (v1 + 44)));
15         *(_QWORD *) (v1 + 56) = v2;
16         if (!(*_QWORD *) (v1 + 48) || !v2) {
17             free(v2);
18             free(*(void **) (v1 + 48));
19             sub_140001A80(v1, 4294967292i 64, "out of memory");
20             return 0xFFFFFFFFi 64;
21         }
22         *(_DWORD *) (v1 + 40) = *(_DWORD *) (v1 + 44);
23         *(_QWORD *) (v1 + 168) = 0i 64;
24         *(_QWORD *) (v1 + 176) = 0i 64;
25         *(_QWORD *) (v1 + 184) = 0i 64;
26         *(_QWORD *) (v1 + 128) = 0;
27         *(_QWORD *) (v1 + 120) = 0i 64;
28         if ((unsigned int) sub_140004940(v1 + 120, 31i 64, "1.2.11", 88i 64)) {
29             free(*(void **) (v1 + 56));
30             free(*(void **) (v1 + 48));
31             *(_DWORD *) (v1 + 40) = 0;
32             sub_140001A80(v1, 4294967292i 64, "out of memory");
33             return 0xFFFFFFFFi 64;
34         }
35     }
```

```
1 local int gz_look(state)
2 gz_statep state;
3 {
4     z_streamp strm = &(state->strm);
5
6     /* allocate read buffers and inflate memory */
7     if (state->size == 0) {
8         /* allocate buffers */
9         state->in = (unsigned char *) malloc(state->want);
10        state->out = (unsigned char *) malloc(state->want << 1);
11        if (state->in == NULL || state->out == NULL) {
12            free(state->out);
13            free(state->in);
14            gz_error(state, Z_MEM_ERROR, "out of memory");
15            return -1;
16        }
17        state->size = state->want;
18
19        /* allocate inflate memory */
20        state->strm.zalloc = Z_NULL;
21        state->strm.zfree = Z_NULL;
22        state->strm.opaque = Z_NULL;
23        state->strm.avail_in = 0;
24        state->strm.next_in = Z_NULL;
25        if (inflateInit2(&(state->strm), 15 + 16) != Z_OK) { /* gunzip */
26            free(state->out);
27            free(state->in);
28            state->size = 0;
29            gz_error(state, Z_MEM_ERROR, "out of memory");
30            return -1;
31        }
32    }
```

Blind Diffing

For visually diffing, in IDA, I use the Hex-Rays decompiler generated pseudo-code.

Both the decompiler pseudo-code and the true source code are indented to the same format using either GNU Indent or Clang-format so there aren't differences because of the programming style.

However, what can I do to diff when the target architecture (say, MIPS or SH4 or whatever else) is not supported by IDA?

- We simply cannot visually diff.
- But we can show what caused both functions to match.

Blind Diffing

IDA View-A		Recent scripts		Matched functions				
Line	Id	Source Function	Local Address	Local Name	Ratio	ML	AVG	Heuristic
027	00245	file_compress	0x140001050	sub_140001050	1.0	1.0	1.0	Same rare constant
023	00231	file_uncompress	0x1400013a0	sub_1400013A0	1.0	1.0	1.0	Same rare constant
004	00136	gz_init	0x140002b10	sub_140002B10	1.0	1.0	1.0	Specific callee search
022	00144	gz_look	0x1400023d0	sub_1400023D0	1.0	1.0	1.0	Callgraph match (callee, iteration 1)
028	00250	gz_uncompress	0x1400014c0	sub_1400014C0	1.0	1.0	1.0	Attributes matching
011	00180	gzdopen	0x140001f00	sub_140001F00	1.0	1.0	1.0	Attributes matching
010	00161	gzerror	0x140001f90	sub_140001F90	1.0	1.0	1.0	Specific callee search
002	00134	gzread	0x1400028c0	sub_1400028C0	1.0	1.0	1.0	Specific callee search
029	00123	gzwrite						Attributes matching
018	00026	inflateBack						Same rare constant
026	00111	inflate_fast						Callgraph match (callee, iteration 1)
013	00319	gzcopy						9690... Callgraph match (caller, iteration 1)
016	00183	gz_open						8634... Callgraph match (callee, iteration 1)
019	00220	main						8089... Same rare constant
003	00135	gz_read						7025 Specific callee search
030	00125	gz_write						7025 Specific callee search
012	00182	gzopen						6940... Specific callee search
006	00012	inflateSetDictionary						1638... Callgraph match (caller, iteration 2)
017	00237	adler32	0x1400089d0	sub_1400089D0	0.3181...	0	0.1590...	Specific callee search
001	00132	gz_comp	0x140002d80	sub_140002D80	0.3	1.0	0.65	Same rare constant
009	00157	gz_error	0x140002f90	sub_140002F90	0.2545...	0	0.1272...	Specific callee search
020	00228	error	0x140001350	sub_140001350	0.2439...	0	0.1219...	Callgraph match (callee, iteration 1)
007	00142	gz_fetch	0x140002700	sub_140002700	0.2333...	0	0.1166...	Callgraph match (callee, iteration 1)



Information



Similar number of conditions (3, 4) -> 0.375000

Similar JSON constants_json (set([u'': filename too long\n', u'': can't gzopen %s\n"])))

Same number of switches (0)

Similar number of calls (10, 5) -> 0.250000

Same number of externals (12)

☐ Don't display this message again

OK

Calculating Similarity/Accuracy Ratios

Once we have a match, we need to determine how “good” or “bad” it is.

For this, we need to create some function that takes as input a source code function and a binary function and outputs a ratio (0.0 to 1.0).

I started writing, by hand, a function `compare_functions()` that assigned weights to each artifact matching as I thought it was “right” for me.

Decided that it was un-scientific and started looking for a more scientific way.

- What happened next will shock you...

**I HAVE NO
IDEA WHAT
I'M DOING**



Machine Learning?

I started learning a bit about Machine Learning, Deep Learning and all this stuff that many companies use as buzzwords for selling snake-oil applications.

After various back and forths with different classifier algorithms, it turned out that no single algorithm was good enough as to replace my horribly and hand-written function.

Machine Learning?

My final choice after many probes is a multi-classifier based on the following ML algorithms:

```
58 ML_CLASSIFIERS = [  
59     (tree.DecisionTreeClassifier, "Decision Tree Classifier", "gini"),  
60     (naive_bayes.BernoulliNB, "Bernoulli Naive Bayes", 1.0),  
61     (ensemble.GradientBoostingClassifier, "Gradient Boosting Classifier", "deviance"),  
62     (ensemble.RandomForestClassifier, "Random Forest Classifier", 10),  
63 ]
```

Machine Learning?

It outputs pretty good results over all once I fed to my trainer a good enough dataset:

```
$ ml/pigaios_ml.py -multi -v
[Tue Sep 25 11:54:06 2018] Using the Pigaios Multi Classifier
[Tue Sep 25 11:54:06 2018] Loading model and data...
[Tue Sep 25 11:54:07 2018] Predicting...
[Tue Sep 25 11:54:22 2018] Correctly predicted 5140 out of 6989 (true positives 1849 -> 73.544141%, false positives 161 -> 0.161000%)
[Tue Sep 25 11:54:22 2018] Total right matches 104979 -> 98.121302%
```

But, guess what? My horrible-super-quick-and-dirty-hand-written function beats in pretty much every case these ML based classifiers. My idea was to use ML as a replacement for that function but I ended up using both methods.

- *“I had one problem, so I used Machine Learning. Now I have 2 problems. And I don’t really understand one.” - Me*

Machine Learning

In any case, I finally added an ML column to the IDA plugin and, when both indicators are “good”, it turns out the results are very-very good:

Matched functions									
Line	Id	Source Function	Local Address	Local Name	Ratio [^]	ML	AVG	Heuristic	
027	00245	file_compress	0x140001050	sub_140001050	1.0	1.0	1.0	Same rare constant	
023	00231	file_uncompress	0x1400013a0	sub_1400013A0	1.0	1.0	1.0	Same rare constant	
004	00136	gz_init	0x140002b10	sub_140002B10	1.0	1.0	1.0	Specific callee search	
022	00144	gz_look	0x1400023d0	sub_1400023D0	1.0	1.0	1.0	Callgraph match (callee, iteration 1)	
028	00250	gz_uncompress	0x1400014c0	sub_1400014C0	1.0	1.0	1.0	Attributes matching	
011	00180	gzdopen	0x140001f00	sub_140001F00	1.0	1.0	1.0	Attributes matching	
010	00161	gzerror	0x140001f90	sub_140001F90	1.0	1.0	1.0	Specific callee search	
002	00134	gzread	0x1400028c0	sub_1400028C0	1.0	1.0	1.0	Specific callee search	
029	00123	gzwrite	0x140003060	sub_140003060	1.0	1.0	1.0	Attributes matching	
018	00026	inflateBack	0x1400030d0	sub_1400030D0	1.0	1.0	1.0	Same rare constant	
026	00111	inflate_fast	0x140008130	sub_140008130	1.0	1.0	1.0	Callgraph match (callee, iteration 1)	
013	00319	gzcopy	0x140002080	sub_140002080	0.9380...	1.0	0.9690...	Callgraph match (caller, iteration 1)	
016	00183	gz_open	0x140001b80	sub_140001B80	0.7269...	1.0	0.8634...	Callgraph match (callee, iteration 1)	
019	00220	main	0x1400011a0	sub_1400011A0	0.6179...	1.0	0.8089...	Same rare constant	
003	00135	gz_read	0x1400025a0	sub_1400025A0	0.405	1.0	0.7025	Specific callee search	
030	00125	gz_write	0x140002c40	sub_140002C40	0.405	1.0	0.7025	Specific callee search	
012	00182	gzopen	0x140001fe0	sub_140001FE0	0.3881...	1.0	0.6940...	Specific callee search	
006	00012	inflateSetDictionary	0x140007550	sub_140007550	0.3277...	0	0.1638...	Callgraph match (caller, iteration 2)	
017	00237	adler32	0x1400089d0	sub_1400089D0	0.3181...	0	0.1590...	Specific callee search	
001	00132	gz_comp	0x140002d80	sub_140002D80	0.3	1.0	0.65	Same rare constant	
009	00157	gz_error	0x140002f90	sub_140002F90	0.2545...	0	0.1272...	Specific callee search	
020	00228	error	0x140001350	sub_140001350	0.2439...	0	0.1219...	Callgraph match (callee, iteration 1)	
007	00142	gz_fetch	0x140002700	sub_140002700	0.2333...	0	0.1166...	Callgraph match (callee, iteration 1)	

Machine Learning

In any case, I finally added an ML column to the IDA plugin and, when both indicators are “good”, it turns out the results are very-very good:

Matched functions									
Line	Id	Source Function	Local Address	Local Name	Ratio [^]	ML	AVG	Heuristic	
027	00245	file_compress	0x140001050	sub_140001050	1.0	1.0	1.0	Same rare constant	
023	00231	file_uncompress	0x1400013a0	sub_1400013A0	1.0	1.0	1.0	Same rare constant	
004	00136	gz_init	0x140002b10	sub_140002B10	1.0	1.0	1.0	Specific callee search	
022	00144	gz_look	0x1400023d0	sub_1400023D0	1.0	1.0	1.0	Callgraph match (callee, iteration 1)	
028	00250	gz_uncompress	0x1400014c0	sub_1400014C0	1.0	1.0	1.0	Attributes matching	
011	00180	gzdopen	0x140001f00	sub_140001F00	1.0	1.0	1.0	Attributes matching	
010	00161	gzerror	0x140001f90	sub_140001F90	1.0	1.0	1.0	Specific callee search	
002	00134	gzread	0x1400028c0	sub_1400028C0	1.0	1.0	1.0	Specific callee search	
029	00123	gzwrite	0x140003060	sub_140003060	1.0	1.0	1.0	Attributes matching	
018	00026	inflateBack	0x1400030d0	sub_1400030D0	1.0	1.0	1.0	Same rare constant	
026	00111	inflate_fast	0x140008130	sub_140008130	1.0	1.0	1.0	Callgraph match (callee, iteration 1)	
013	00319	gzcopy	0x140002080	sub_140002080	0.9380...	1.0	0.9690...	Callgraph match (caller, iteration 1)	
016	00183	gz_open	0x140001b80	sub_140001B80	0.7269...	1.0	0.8634...	Callgraph match (callee, iteration 1)	
019	00220	main	0x1400011a0	sub_1400011A0	0.6179...	1.0	0.8089...	Same rare constant	
003	00135	gz_read	0x1400025a0	sub_1400025A0	0.405	1.0	0.7025	Specific callee search	
030	00125	gz_write	0x140002c40	sub_140002C40	0.405	1.0	0.7025	Specific callee search	
012	00182	gzopen	0x140001fe0	sub_140001FE0	0.3881...	1.0	0.6940...	Specific callee search	
006	00012	inflateSetDictionary	0x140007550	sub_140007550	0.3277...	0	0.1638...	Callgraph match (caller, iteration 2)	
017	00237	adler32	0x1400089d0	sub_1400089D0	0.3181...	0	0.1590...	Specific callee search	
001	00132	gz_comp	0x140002d80	sub_140002D80	0.3	1.0	0.65	Same rare constant	
009	00157	gz_error	0x140002f90	sub_140002F90	0.2545...	0	0.1272...	Specific callee search	
020	00228	error	0x140001350	sub_140001350	0.2439...	0	0.1219...	Callgraph match (callee, iteration 1)	
007	00142	gz_fetch	0x140002700	sub_140002700	0.2333...	0	0.1166...	Callgraph match (callee, iteration 1)	

DEMO

The Future

The Future?

The current status of this project is “production ready” but with a low number of results. My main focus was to get results with near zero false positives.

The next steps of this project are the following:

- Importing structures and enumerations. It looks easy but turns out to be pretty-pretty hard to do.
- Objective-C and *partial* C++ support.
- Integrating with Diaphora, as an optional component of it.
 - Actually, it shares too much code with Diaphora.
- Exporting to SQLite databases Open Source codebases and putting them for download somewhere, so you don't need to do.

And that's all!

The project is Open Source, you can get the source code from there:

<https://github.com/joxeankoret/pigaios>

Thanks for you time! Any questions?