

Diaphora – An IDA Python BinDiffing plugin

Index

Introduction.....	2
Files distributed with the diaphora distribution.....	2
Running Diaphora.....	2
Diaphora quick start.....	4
Finding differences in new versions (Patch diffing).....	4
Ignoring small differences (Finding new functionalities).....	9
Porting symbols.....	11
Diffing huge databases (or exporting smaller .SQLite databases).....	14
Heuristics.....	14
Best matches.....	14
Partial and unreliable matches (according to the confidence's ratio):.....	14
Unreliable matches.....	15
Experimental (and likely to be removed or moved or changed in the future):.....	16

Introduction

Diaphora is a plugin for IDA Pro that aims to help in the typical BinDiffing tasks. It's similar to other competitor products and open source projects like Zynamics BinDiff, DarunGrim or TurboDiff. However, it's able to perform more actions than any of the previous IDA plugins or projects.

In the next paragraphs, I will describe how to use it in different scenarios.

Files distributed with the diaphora distribution

Diaphora is distributed as a compressed file with various files and folders inside it. The structure is similar to the following one:

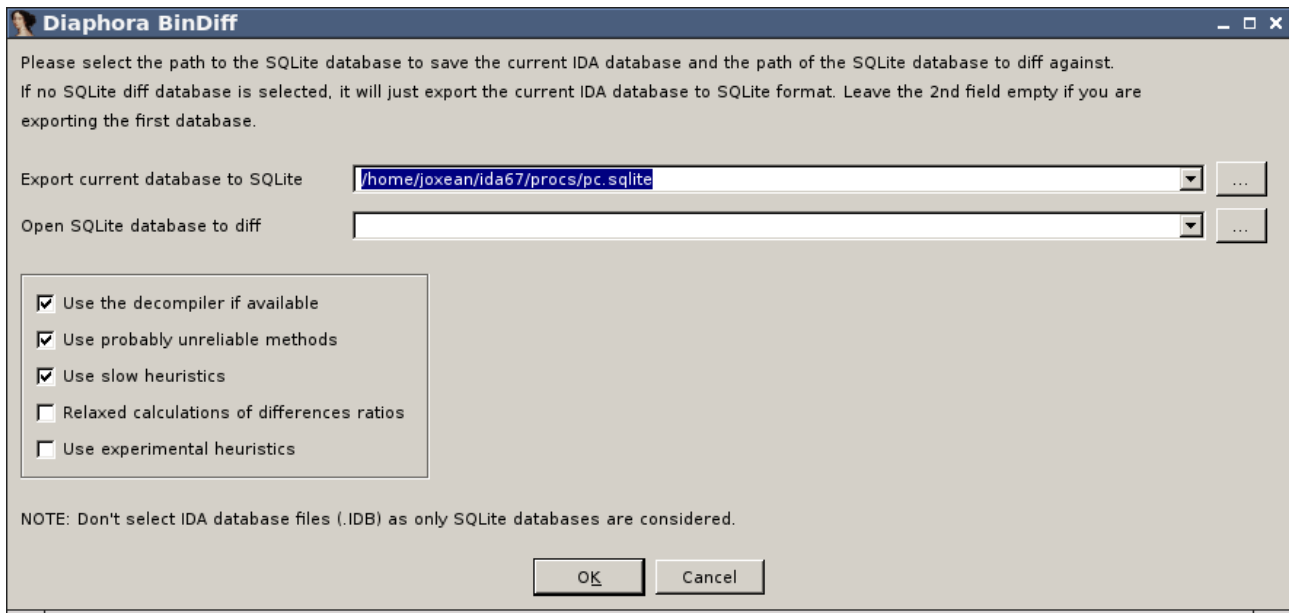
1. diaphora.py: The main IDAPython plugin. It contains all the code of the heuristics, graphs displaying, export interface, etc...
2. jkutils/kfuzzy.py: This is an unmodified version of the kfuzzy.py library, part of the DeepToad project, a tool and a library for performing fuzzy hashing of binary files. It's included because fuzzy hashes of pseudo-codes are used as part of the various heuristics implemented.
3. jkutils/factor.py: This is a modified version of a private malware clusterization toolkit based on graphs theory. This library offers the ability to factor numbers quickly in Python and, also, to compare arrays of prime factors. Diaphora uses it to compare fuzzy AST hashes and call graph fuzzy hashes based on small-primes-products (an idea coined and implemented by Thomas Dullien and Rolf Rolles first, authors or former authors of the Zynamics BinDiff commercial product, in their [“Graph-based comparison of Executable Objects – Zynamics”](#) paper).
4. Pygments/: This directory contains an unmodified distribution of the Python pygments library, a “generic syntax highlighter suitable for use in code hosting, forums, wikis or other applications that need to prettify source code”.

Running Diaphora

Diaphora can only be used by running the script, as of March 2015. Initially, during the BETA phases, there was support for installing it as a true IDA plugin. However, it causes a lot of maintenance problems, like finding workarounds for known IDA problems and bugs. As so, and because during the beta phase more time was expended in finding workarounds to different IDA bugs and problems with many different versions of IDA than actually fixing bugs on Diaphora, the support have been dropped. It may be added back again at some point in the future, but is unlikely.

So, in order to run Diaphora, simply, unpack the compressed distribution file wherever you prefer

and directly execute “diaphora.py” from the IDA Pro menu File → Script file. Once the script diaphora.py is executed, a dialog like the following one will be opened:



This dialog, although can be a bit confusing at first, is used for both exporting the current IDA database to SQLite format as well for performing diffing against another SQLite exported format database.

The first field, is the path of the SQLite file format database that will be created with all the information extracted from the current database. The 2nd field is the other SQLite format database to diff the current database against. If this field is left empty, Diaphora will just export the current database to SQLite format. If the 2nd field is not empty, it will diff both databases.

The other fields, the check-boxes, are explained bellow:

1. **Use the decompiler if available.** If the Hex-Rays decompiler is installed with IDA and IDA Python bindings are available, Diaphora will use the decompiler to get many interesting information that will help during the bindiffing process.
2. **Use probably unreliable methods.** Diaphora uses many heuristics to try to match functions in both databases being compared. However, some heuristics are not really reliable or the ratio of similarity is very low. Check this box if you want to see also the likely unreliable matches Diaphora may find. Unreliable results are shown in a specific list, it doesn't mix the “Best results” (results with a ratio of 1.00) with the “Partial results” (results with a ratio of 0.50 or higher) or “Unreliable results”.
3. **Use slow heuristics.** Some heuristics can be quite expensive and take long. For medium to big databases, it's disabled by default and is recommended to left unchecked unless the results from a execution with this option disabled are not good enough. It will likely find more better matches than the normal, not that slow, heuristics, but it will take significantly longer.
4. **Relaxed calculations of difference ratios.** Diaphora uses, by default, a kind of aggressive method to calculate difference ratios between matches. It's possible to relax that

aggressiveness level by checking this option. Under the hood, the function [SequenceMatcher.quick_ratio](#) is used when this option is unchecked and [SequenceMatcher.real_quick_ratio](#) when this option is checked. Also, when the option is checked, Diaphora will use too the difference ratio of the primes numbers calculated from the AST of the pseudo-code of the 2 functions, calculating the highest ratio from the AST, assembly and pseudo-code comparisons.

5. **Use experimental heuristics.** It says it all: experimental heuristics are enabled only if this check-box is marked. Disabled by default as they are likely not useful.

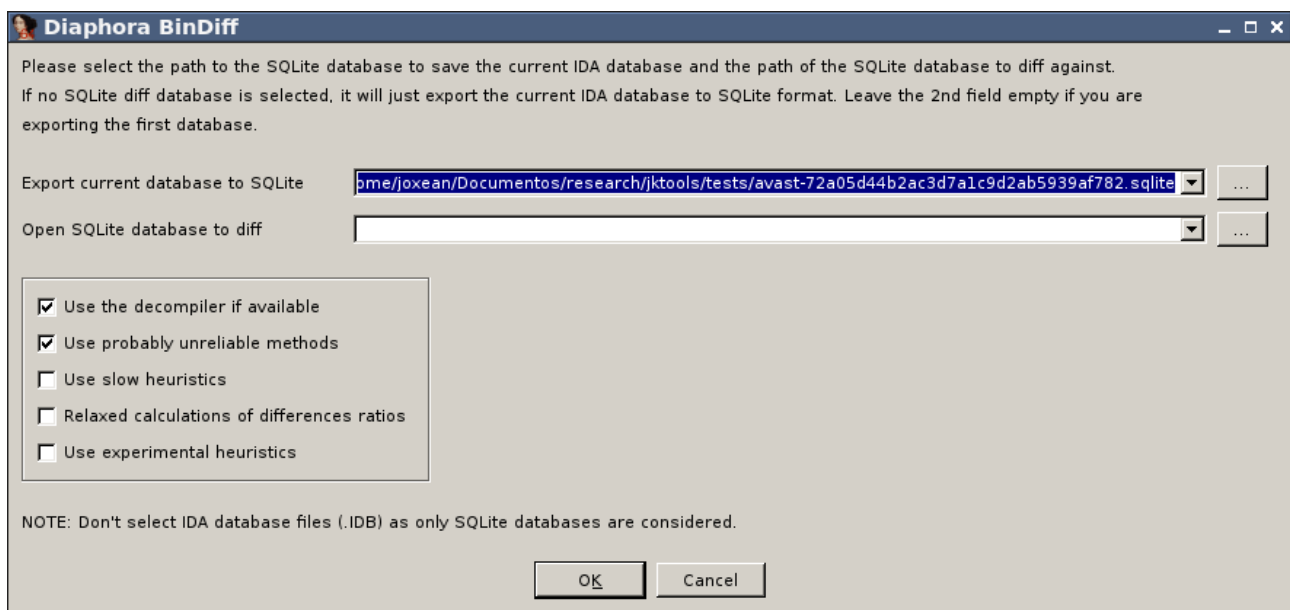
Diaphora quick start

Finding differences in new versions (Patch diffing)

In order to use Diaphora we need at least two binary files to compare. I will take as example 2 different versions of the “avast” binary from Avast for Linux x86_64. The files has the following hashes:

1. ed5bdbbfaf25b065cf9ee50730334998 avast
2. 72a05d44b2ac3d7a1c9d2ab5939af782 avast-72a05d44b2ac3d7a1c9d2ab5939af782

The file “avast-<hash>” is the previous ones and the binary “avast” is the latest version. Launch IDA Pro for 64 bits (ida64) and open the file “avast-72a05d44b2ac3d7a1c9d2ab5939af782”. Once the initial auto-analysis finishes launch Diaphora by either running the script “diaphora.py”. The following dialog will open:

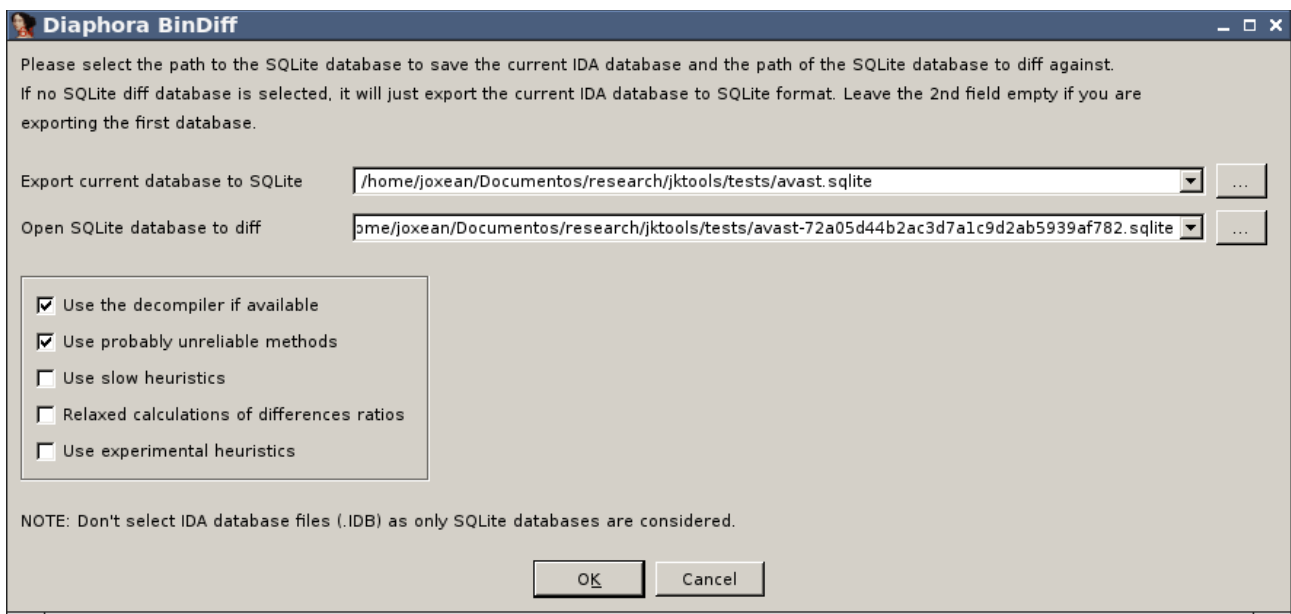


We only need to care about 2 things:

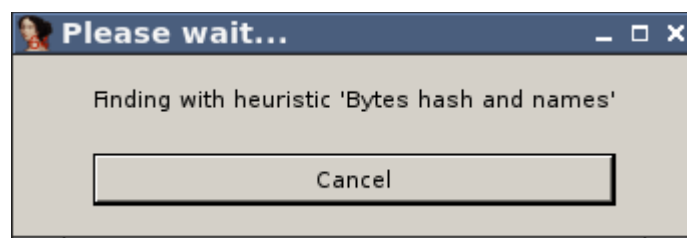
1. Field “**Export current database to SQLite**”. This is the path to the SQLite database that will be created with all the information extracted from the IDA database of this avast binary.
2. Field “**Use the decompiler if available**”. If the Hex-Rays decompiler is available and we want to use it, we will leave this check-box marked, otherwise uncheck it.

After correctly selecting the appropriate values, press OK. It will start exporting all the data from the IDA database. When export process finishes the message “Database exported.” will appear in the IDA's Output Window.

Now, close this database, save the changes and open the “avast” binary. Wait until the IDA's auto-analysis finishes and, after it, run Diaphora like with the previous binary file. This time, we will select in the 2nd field, the one name “SQLite database to diff”, the path to the .sqlite file we just exported in the previous step, as shown in the next figure:










After this, press the OK button. It will first export the current IDA database to the SQLite format as understood by Diaphora and, then, right after finishing, compare both databases. It will show an IDA's wait box dialog with the current heuristic being applied to match functions in both databases as shown in the next figure:
















After a while a set of lists (choosers, in the HexRays workers language) will appear:












There is one more list that is not shown for this database, named “Unreliable matches”. This list holds all the matches that aren't considered reliable. However, in the case of this binary with symbols, there isn't even a single unreliable result. There are, however, unmatched functions in both the primary (the latest version) and the secondary database (the previous version):

Line	Address	Name
 00000000	0040eb40	strup_enable.part.1
 00000001	ffffffffffffff	UnixUnlockDefinitionsFolder(void...
 00000002	ffffffffffffff	UnixLockDefinitionsFolder(char c...
 00000003	ffffffffffffff	asw::root::CGenericFile::writesa...
 00000004	0043d470	dep_fsUnlockFile
 00000005	0043d510	dep_fsLockFile
 00000006	ffffffffffffff	google::protobuf::io::ZeroCopy...





















The previous image shows the functions not matched in the secondary database, that is: the functions removed in the latest version. The second figure shows the functions not matched in the previous database, the new functions added:

Line	Address	Name
 00000000	0040c040	handler
 00000001	0040d140	context
 00000002	004574c0	cpu_check_avx_os
 00000003	004574cf	cpu_check_avx2
 00000004	0045750d	cpu_check_bmi1
 00000005	0045752f	cpu_check_bmi2
 00000006	0045755e	cpu_check_avx
 00000007	004575d9	cpu_check_pclmulqdq
 00000008	0045760e	cpu_check_popcnt
 00000009	0045763b	cpu_check_sse42
 00000010	00457651	cpu_check_sse41
 00000011	0045769c	cpu_check_ssse3
 00000012	004576d5	cpu_check_sse2

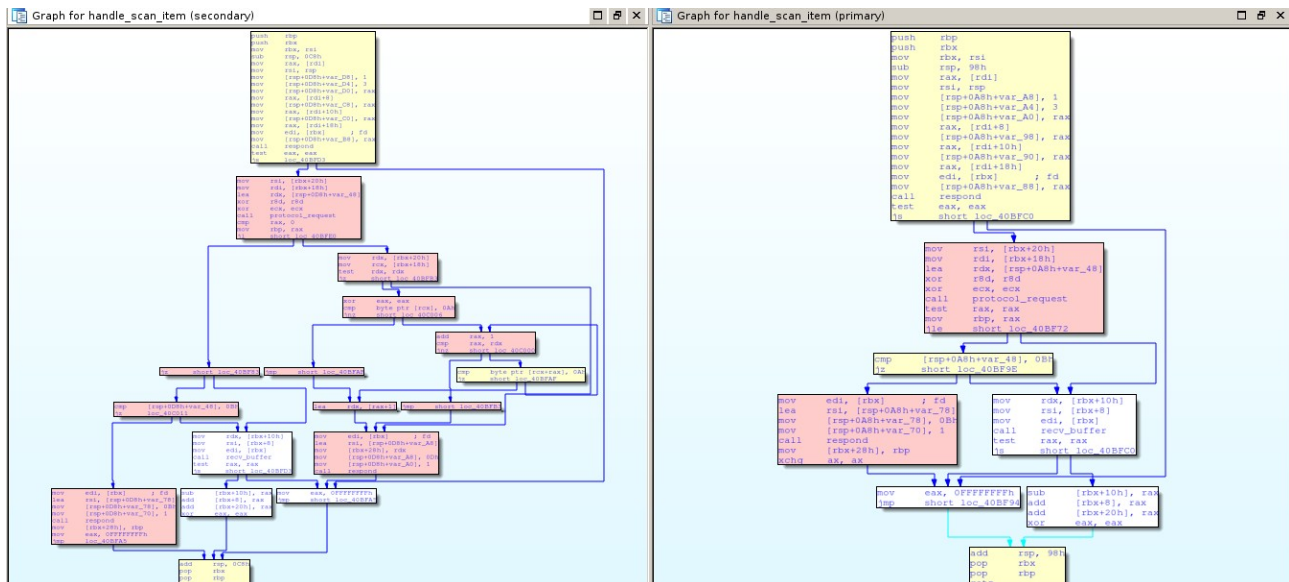
It seems they added various functions to check for SSE, AVX, etc... Intel instructions. Also, they added 2 new functions called handler and context. Let's take a look now to the “Best matches” tab opened:

Line	Address	Name	Address 2	Name 2	Description
 00000265	0040b9c0	int_set.isra.3	0040b9c0	int_set.isra.3	100% equal
 00000266	0040ba40	ini_callback	0040ba40	ini_callback	100% equal
 00000267	0040bc00	config_free	0040bc00	config_free	100% equal
 00000268	0040bc50	config_read	0040bc50	config_read	100% equal
 00000269	0040bea0	recv_buffer	0040bea0	recv_buffer	100% equal
 00000270	0041b680	StreamingUpdateClient::GetLastUpdateTim...	0041b520	StreamingUpdateClient::GetLast...	Equal pseudo-code
 00000271	00458450	CryptoPP::Integer::operator-(void)const	0045bb10	CryptoPP::Integer::operator-(voi...	Equal pseudo-code
 00000272	0043fd00	dep_strAnsiToOem	0043fe20	dep_strAnsiToOem	Equal pseudo-code
 00000273	0043fd20	dep_strOemToAnsi	0043fe40	dep_strOemToAnsi	Equal pseudo-code

There are many functions in the “Best matches” tab, 2556 functions, and in the primary database there are 2659 functions. The results shown in the “Best matches” tab are these functions matched with some heuristic (like “100% equal”, where all attributes are equal, or “Equal pseudo-code”, where the pseudo-code generated by the decompiler is equal) that, apparently, doesn't have any difference at all. If you're diffing these binaries to find vulnerabilities fixed, just skip this tab, you will be more interested in the “Partial matches” one ;) In this tab we have many results:

Line	Address	Name	Address 2	Name 2	Description
 00000000	0040e880	load_vps	0040eab0	load_vps	Bytes hash and names (ratio 0.825000)
 00000001	0040ec80	setup_ini_callback	0040ed70	setup_ini_callback	Bytes hash and names (ratio 0.923077)
 00000002	0040ecc0	engine_init	0040edb0	engine_init	Bytes hash and names (ratio 0.753846)
 00000003	0040f0d0	engine_scan	0040f270	engine_scan	Bytes hash and names (ratio 0.980769)
 00000004	0040f3e0	engine_exclude_path	0040f5b0	engine_exclude_path	Bytes hash and names (ratio 0.854545)
 00000005	0040f520	engine_set_packers	0040f6f0	engine_set_packers	Bytes hash and names (ratio 0.913043)
 00000006	0040f5c0	engine_set_flags	0040f790	engine_set_flags	Bytes hash and names (ratio 0.913043)
 00000007	0040f660	engine_set_sensitivity	0040f830	engine_set_sensitivity	Bytes hash and names (ratio 0.913043)
 00000008	0040fc60	engine_verify_vps	0040fe30	engine_verify_vps	Bytes hash and names (ratio 0.818653)
 00000009	00413c20	avldrGetEngineInformation	00413db0	avldrGetEngineInformation	Bytes hash and names (ratio 0.961749)
 00000010	00413e70	aswLoaderDllMain	00414000	aswLoaderDllMain	Bytes hash and names (ratio 0.750000)
 00000011	0041b610	InitializeStrUpdate	0041b4b0	InitializeStrUpdate	Bytes hash and names (ratio 0.727273)
 00000012	0042b150	aswcmnbsDllMain	0042afc0	aswcmnbsDllMain	Bytes hash and names (ratio 0.652174)
 00000013	0042b2c0	cmnblnit	0042b130	cmnblnit	Bytes hash and names (ratio 0.746479)
 00000014	0042b6f0	aswcmnosDllMain	0042b560	aswcmnosDllMain	Bytes hash and names (ratio 0.788991)
 00000015	0042b880	cmnosinit	0042b6f0	cmnosinit	Bytes hash and names (ratio 0.724490)
 00000016	0044a950	DSA_FileVerifyWithSigCompare	0044ddc0	DSA_FileVerifyWithSigCom...	Bytes hash and names (ratio 0.978495)
 00000017	0044b130	DSA_BlockVerify	0044e5a0	DSA_BlockVerify	Bytes hash and names (ratio 0.975610)
 00000018	0040a200	main	0040a200	main	Same address, nodes, edges and mnemonics (ratio 0.995290)
 00000019	0040bf00	handle_scan_item	0040bf00	handle_scan_item	Perfect match, same name (ratio 0.434211)

It shows the functions matched between both databases and, in the description field, it says which heuristic matched and the ratio of differences. If you're looking for functions where a vulnerability was likely fixed, this is where you want to look at. It seems that the function “handle_scan_item”, for example, was heavily modified: the ratio is 0.49, so it means that more than the 49% of the function differs between both databases. Let's see the differences: we can see then in an assembly graph, in plain assembly or we can diff pseudo-code too. Right click on the result and select “Diff assembly in a graph”, the following graph will appear:



The nodes in yellow colour, are these with only minor changes; pink ones, are these that are either new or heavily modified and the blank ones, the basic blocks that were not modified at all. Let's diff now the assembly in plain text: go back to the “Partial matches” tab, right click on the function “handle_scan_item” and select “Diff assembly”:

<pre> f1 handle_scan_item proc near 2 push rbp 3 push rbx 4 mov rbx, rsi n5 sub rsp, 98h 6 mov rax, [rdi] 7 mov rsi, rsp n8 mov [rsp+0A8h+var_A8], 1 9 mov [rsp+0A8h+var_A4], 3 10 mov [rsp+0A8h+var_A0], rax 11 mov rax, [rdi+8] n12 mov [rsp+0A8h+var_98], rax 13 mov rax, [rdi+10h] n14 mov [rsp+0A8h+var_90], rax 15 mov rax, [rdi+18h] 16 mov edi, [rbx] ; fd n17 mov [rsp+0A8h+var_88], rax 18 call respond 19 test eax, eax n20 js short loc_40BFCD 21loc_40bf4c: 22 mov rsi, [rbx+20h] 23 mov rdi, [rbx+18h] n24 lea rdx, [rsp+0A8h+var_48] 25 xor r8d, r8d 26 xor ecx, ecx 27 call protocol_request n28 test rax, rax 29 mov rbp, rax n30 jle short loc_40BF72 31loc_40bf6b: 32 cmp [rsp+0A8h+var_48], 0Bh 33 jz short loc_40BF9E 34loc_40bf72: </pre>	<pre> f1 handle_scan_item proc near 2 push rbp 3 push rbx 4 mov rbx, rsi n5 sub rsp, 0C8h 6 mov rax, [rdi] 7 mov rsi, rsp n8 mov [rsp+0D8h+var_D8], 1 9 mov [rsp+0D8h+var_D4], 3 10 mov [rsp+0D8h+var_D0], rax 11 mov rax, [rdi+8] n12 mov [rsp+0D8h+var_C8], rax 13 mov rax, [rdi+10h] n14 mov [rsp+0D8h+var_C0], rax 15 mov rax, [rdi+18h] 16 mov edi, [rbx] ; fd n17 mov [rsp+0D8h+var_B8], rax 18 call respond 19 test eax, eax n20 js loc_40BFD3 21loc_40bf50: 22 mov rsi, [rbx+20h] 23 mov rdi, [rbx+18h] n24 lea rdx, [rsp+0D8h+var_48] 25 xor r8d, r8d 26 xor ecx, ecx 27 call protocol_request n28 cmp rax, 0 29 mov rbp, rax n30 jl short loc_40BFED 31loc_40bf73: 32 jz short loc_40BF83 33loc_40bf75: 34 cmp [rsp+0D8h+var_48], 0Bh 35 jz loc_40C011 36loc_40bf83: </pre>
--	---

It shows the differences, in plain assembly, that one would see by using a tool like the Unix command “diff”. We can also dif the pseudo-code: go back to the “Partial matches” tab, right click in the function and select “Graph pseudo-code”:

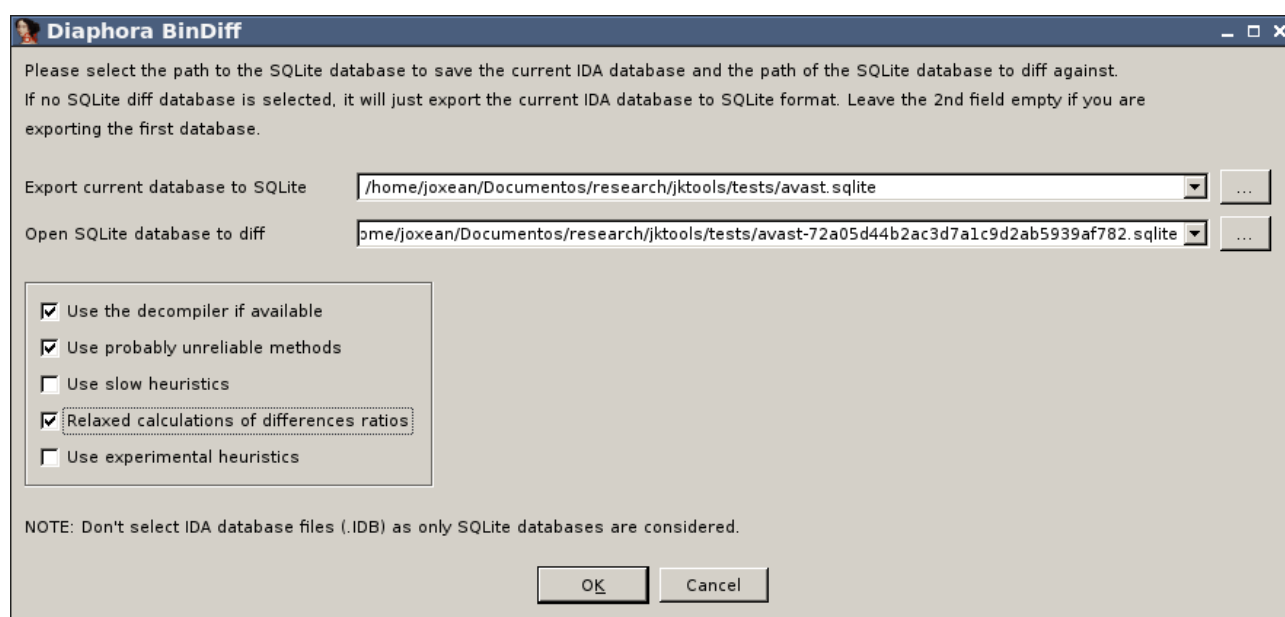
```

1 signed __int64 __fastcall handle_scan_item(__int64 a1, __int64 a2)
2 {
3     __int64 v2; // rax@1
4     __int64 v3; // rax@1
5     int v4; // edi@1
6     __int64 v5; // rbp@2
7     ssize_t v6; // rax@4
8
9     int v7; // edi@7
10    int v8; // [sp+0h] [bp-8h]@1
11    int v10; // [sp+4h] [bp-4h]@1
12    __int64 v11; // [sp+8h] [bp-0h]@1
13    __int64 v12; // [sp+10h] [bp-8h]@1
14    __int64 v13; // [sp+18h] [bp-0h]@1
15    __int64 v14; // [sp+20h] [bp-8h]@1
16    int v15; // [sp+30h] [bp-7h]@7
17    int v16; // [sp+38h] [bp-0h]@7
18
19    int v17; // [sp+0h] [bp-48h]@2
20
21    v2 = *(_QWORD *)a1;
22    v3 = 1;
23    v10 = 3;
24    v11 = v2;
25    v12 = *(_QWORD *)a1 + 8;
26    v13 = *(_QWORD *)a1 + 16;
27    v4 = *(_QWORD *)a1 + 24;
28    v5 = *(_QWORD *)a2;
29    if ( (signed int)respond(v4, (__int64)v5) >= 0 )
30    {
31        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)
32        {a2 + 32}, (__int64)v17, 0LL, 0LL);
33        if ( v5 > 0 && v17 == 11 )
34        {
35            v8 = *(_QWORD *)a2;
36            v15 = 11;
37            v16 = 1;
38            respond(v8, (__int64)v15);
39        }
40    }
41
42    v2 = *(_QWORD *)a1;
43    v3 = 1;
44    v10 = 3;
45    v11 = v2;
46    v12 = *(_QWORD *)a1 + 8;
47    v13 = *(_QWORD *)a1 + 16;
48    v4 = *(_QWORD *)a1 + 24;
49    v5 = *(_QWORD *)a2;
50    if ( (signed int)respond(v4, (__int64)v5) < 0 )
51    {
52        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
53        v6 = v5;
54        if ( v6 < 0 )
55        {
56            v9 = *(_QWORD *)a2 + 32;
57            v11 = *(_BYTE *)a2 + 24;
58            if ( v9 )
59            {
60                v12 = 0LL;
61                if ( *v11 == 10 )
62                {
63                    v12 = 1;
64                }
65            }
66        }
67    }
68    v12 = 0LL;
69    if ( *v11 == 10 )
70    {
71        v12 = 1;
72    }
73    v10 = 3;
74    v11 = v2;
75    v12 = *(_QWORD *)a1 + 8;
76    v13 = *(_QWORD *)a1 + 16;
77    v4 = *(_QWORD *)a1 + 24;
78    v5 = *(_QWORD *)a2;
79    if ( (signed int)respond(v4, (__int64)v5) < 0 )
80    {
81        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
82        v6 = v5;
83        if ( v6 < 0 )
84        {
85            v9 = *(_QWORD *)a2 + 32;
86            v11 = *(_BYTE *)a2 + 24;
87            if ( v9 )
88            {
89                v12 = 0LL;
90                if ( *v11 == 10 )
91                {
92                    v12 = 1;
93                }
94            }
95        }
96    }
97    v12 = 0LL;
98    if ( *v11 == 10 )
99    {
100    v12 = 1;
101    }
102    v10 = 3;
103    v11 = v2;
104    v12 = *(_QWORD *)a1 + 8;
105    v13 = *(_QWORD *)a1 + 16;
106    v4 = *(_QWORD *)a1 + 24;
107    v5 = *(_QWORD *)a2;
108    if ( (signed int)respond(v4, (__int64)v5) < 0 )
109    {
110        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
111        v6 = v5;
112        if ( v6 < 0 )
113        {
114            v9 = *(_QWORD *)a2 + 32;
115            v11 = *(_BYTE *)a2 + 24;
116            if ( v9 )
117            {
118                v12 = 0LL;
119                if ( *v11 == 10 )
120                {
121                    v12 = 1;
122                }
123            }
124        }
125    }
126    v12 = 0LL;
127    if ( *v11 == 10 )
128    {
129        v12 = 1;
130    }
131    v10 = 3;
132    v11 = v2;
133    v12 = *(_QWORD *)a1 + 8;
134    v13 = *(_QWORD *)a1 + 16;
135    v4 = *(_QWORD *)a1 + 24;
136    v5 = *(_QWORD *)a2;
137    if ( (signed int)respond(v4, (__int64)v5) < 0 )
138    {
139        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
140        v6 = v5;
141        if ( v6 < 0 )
142        {
143            v9 = *(_QWORD *)a2 + 32;
144            v11 = *(_BYTE *)a2 + 24;
145            if ( v9 )
146            {
147                v12 = 0LL;
148                if ( *v11 == 10 )
149                {
150                    v12 = 1;
151                }
152            }
153        }
154    }
155    v12 = 0LL;
156    if ( *v11 == 10 )
157    {
158        v12 = 1;
159    }
160    v10 = 3;
161    v11 = v2;
162    v12 = *(_QWORD *)a1 + 8;
163    v13 = *(_QWORD *)a1 + 16;
164    v4 = *(_QWORD *)a1 + 24;
165    v5 = *(_QWORD *)a2;
166    if ( (signed int)respond(v4, (__int64)v5) < 0 )
167    {
168        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
169        v6 = v5;
170        if ( v6 < 0 )
171        {
172            v9 = *(_QWORD *)a2 + 32;
173            v11 = *(_BYTE *)a2 + 24;
174            if ( v9 )
175            {
176                v12 = 0LL;
177                if ( *v11 == 10 )
178                {
179                    v12 = 1;
180                }
181            }
182        }
183    }
184    v12 = 0LL;
185    if ( *v11 == 10 )
186    {
187        v12 = 1;
188    }
189    v10 = 3;
190    v11 = v2;
191    v12 = *(_QWORD *)a1 + 8;
192    v13 = *(_QWORD *)a1 + 16;
193    v4 = *(_QWORD *)a1 + 24;
194    v5 = *(_QWORD *)a2;
195    if ( (signed int)respond(v4, (__int64)v5) < 0 )
196    {
197        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
198        v6 = v5;
199        if ( v6 < 0 )
200        {
201            v9 = *(_QWORD *)a2 + 32;
202            v11 = *(_BYTE *)a2 + 24;
203            if ( v9 )
204            {
205                v12 = 0LL;
206                if ( *v11 == 10 )
207                {
208                    v12 = 1;
209                }
210            }
211        }
212    }
213    v12 = 0LL;
214    if ( *v11 == 10 )
215    {
216        v12 = 1;
217    }
218    v10 = 3;
219    v11 = v2;
220    v12 = *(_QWORD *)a1 + 8;
221    v13 = *(_QWORD *)a1 + 16;
222    v4 = *(_QWORD *)a1 + 24;
223    v5 = *(_QWORD *)a2;
224    if ( (signed int)respond(v4, (__int64)v5) < 0 )
225    {
226        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
227        v6 = v5;
228        if ( v6 < 0 )
229        {
230            v9 = *(_QWORD *)a2 + 32;
231            v11 = *(_BYTE *)a2 + 24;
232            if ( v9 )
233            {
234                v12 = 0LL;
235                if ( *v11 == 10 )
236                {
237                    v12 = 1;
238                }
239            }
240        }
241    }
242    v12 = 0LL;
243    if ( *v11 == 10 )
244    {
245        v12 = 1;
246    }
247    v10 = 3;
248    v11 = v2;
249    v12 = *(_QWORD *)a1 + 8;
250    v13 = *(_QWORD *)a1 + 16;
251    v4 = *(_QWORD *)a1 + 24;
252    v5 = *(_QWORD *)a2;
253    if ( (signed int)respond(v4, (__int64)v5) < 0 )
254    {
255        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
256        v6 = v5;
257        if ( v6 < 0 )
258        {
259            v9 = *(_QWORD *)a2 + 32;
260            v11 = *(_BYTE *)a2 + 24;
261            if ( v9 )
262            {
263                v12 = 0LL;
264                if ( *v11 == 10 )
265                {
266                    v12 = 1;
267                }
268            }
269        }
270    }
271    v12 = 0LL;
272    if ( *v11 == 10 )
273    {
274        v12 = 1;
275    }
276    v10 = 3;
277    v11 = v2;
278    v12 = *(_QWORD *)a1 + 8;
279    v13 = *(_QWORD *)a1 + 16;
280    v4 = *(_QWORD *)a1 + 24;
281    v5 = *(_QWORD *)a2;
282    if ( (signed int)respond(v4, (__int64)v5) < 0 )
283    {
284        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
285        v6 = v5;
286        if ( v6 < 0 )
287        {
288            v9 = *(_QWORD *)a2 + 32;
289            v11 = *(_BYTE *)a2 + 24;
290            if ( v9 )
291            {
292                v12 = 0LL;
293                if ( *v11 == 10 )
294                {
295                    v12 = 1;
296                }
297            }
298        }
299    }
300    v12 = 0LL;
301    if ( *v11 == 10 )
302    {
303        v12 = 1;
304    }
305    v10 = 3;
306    v11 = v2;
307    v12 = *(_QWORD *)a1 + 8;
308    v13 = *(_QWORD *)a1 + 16;
309    v4 = *(_QWORD *)a1 + 24;
310    v5 = *(_QWORD *)a2;
311    if ( (signed int)respond(v4, (__int64)v5) < 0 )
312    {
313        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
314        v6 = v5;
315        if ( v6 < 0 )
316        {
317            v9 = *(_QWORD *)a2 + 32;
318            v11 = *(_BYTE *)a2 + 24;
319            if ( v9 )
320            {
321                v12 = 0LL;
322                if ( *v11 == 10 )
323                {
324                    v12 = 1;
325                }
326            }
327        }
328    }
329    v12 = 0LL;
330    if ( *v11 == 10 )
331    {
332        v12 = 1;
333    }
334    v10 = 3;
335    v11 = v2;
336    v12 = *(_QWORD *)a1 + 8;
337    v13 = *(_QWORD *)a1 + 16;
338    v4 = *(_QWORD *)a1 + 24;
339    v5 = *(_QWORD *)a2;
340    if ( (signed int)respond(v4, (__int64)v5) < 0 )
341    {
342        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
343        v6 = v5;
344        if ( v6 < 0 )
345        {
346            v9 = *(_QWORD *)a2 + 32;
347            v11 = *(_BYTE *)a2 + 24;
348            if ( v9 )
349            {
350                v12 = 0LL;
351                if ( *v11 == 10 )
352                {
353                    v12 = 1;
354                }
355            }
356        }
357    }
358    v12 = 0LL;
359    if ( *v11 == 10 )
360    {
361        v12 = 1;
362    }
363    v10 = 3;
364    v11 = v2;
365    v12 = *(_QWORD *)a1 + 8;
366    v13 = *(_QWORD *)a1 + 16;
367    v4 = *(_QWORD *)a1 + 24;
368    v5 = *(_QWORD *)a2;
369    if ( (signed int)respond(v4, (__int64)v5) < 0 )
370    {
371        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
372        v6 = v5;
373        if ( v6 < 0 )
374        {
375            v9 = *(_QWORD *)a2 + 32;
376            v11 = *(_BYTE *)a2 + 24;
377            if ( v9 )
378            {
379                v12 = 0LL;
380                if ( *v11 == 10 )
381                {
382                    v12 = 1;
383                }
384            }
385        }
386    }
387    v12 = 0LL;
388    if ( *v11 == 10 )
389    {
390        v12 = 1;
391    }
392    v10 = 3;
393    v11 = v2;
394    v12 = *(_QWORD *)a1 + 8;
395    v13 = *(_QWORD *)a1 + 16;
396    v4 = *(_QWORD *)a1 + 24;
397    v5 = *(_QWORD *)a2;
398    if ( (signed int)respond(v4, (__int64)v5) < 0 )
399    {
400        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
401        v6 = v5;
402        if ( v6 < 0 )
403        {
404            v9 = *(_QWORD *)a2 + 32;
405            v11 = *(_BYTE *)a2 + 24;
406            if ( v9 )
407            {
408                v12 = 0LL;
409                if ( *v11 == 10 )
410                {
411                    v12 = 1;
412                }
413            }
414        }
415    }
416    v12 = 0LL;
417    if ( *v11 == 10 )
418    {
419        v12 = 1;
420    }
421    v10 = 3;
422    v11 = v2;
423    v12 = *(_QWORD *)a1 + 8;
424    v13 = *(_QWORD *)a1 + 16;
425    v4 = *(_QWORD *)a1 + 24;
426    v5 = *(_QWORD *)a2;
427    if ( (signed int)respond(v4, (__int64)v5) < 0 )
428    {
429        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
430        v6 = v5;
431        if ( v6 < 0 )
432        {
433            v9 = *(_QWORD *)a2 + 32;
434            v11 = *(_BYTE *)a2 + 24;
435            if ( v9 )
436            {
437                v12 = 0LL;
438                if ( *v11 == 10 )
439                {
440                    v12 = 1;
441                }
442            }
443        }
444    }
445    v12 = 0LL;
446    if ( *v11 == 10 )
447    {
448        v12 = 1;
449    }
450    v10 = 3;
451    v11 = v2;
452    v12 = *(_QWORD *)a1 + 8;
453    v13 = *(_QWORD *)a1 + 16;
454    v4 = *(_QWORD *)a1 + 24;
455    v5 = *(_QWORD *)a2;
456    if ( (signed int)respond(v4, (__int64)v5) < 0 )
457    {
458        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
459        v6 = v5;
460        if ( v6 < 0 )
461        {
462            v9 = *(_QWORD *)a2 + 32;
463            v11 = *(_BYTE *)a2 + 24;
464            if ( v9 )
465            {
466                v12 = 0LL;
467                if ( *v11 == 10 )
468                {
469                    v12 = 1;
470                }
471            }
472        }
473    }
474    v12 = 0LL;
475    if ( *v11 == 10 )
476    {
477        v12 = 1;
478    }
479    v10 = 3;
480    v11 = v2;
481    v12 = *(_QWORD *)a1 + 8;
482    v13 = *(_QWORD *)a1 + 16;
483    v4 = *(_QWORD *)a1 + 24;
484    v5 = *(_QWORD *)a2;
485    if ( (signed int)respond(v4, (__int64)v5) < 0 )
486    {
487        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
488        v6 = v5;
489        if ( v6 < 0 )
490        {
491            v9 = *(_QWORD *)a2 + 32;
492            v11 = *(_BYTE *)a2 + 24;
493            if ( v9 )
494            {
495                v12 = 0LL;
496                if ( *v11 == 10 )
497                {
498                    v12 = 1;
499                }
500            }
501        }
502    }
503    v12 = 0LL;
504    if ( *v11 == 10 )
505    {
506        v12 = 1;
507    }
508    v10 = 3;
509    v11 = v2;
510    v12 = *(_QWORD *)a1 + 8;
511    v13 = *(_QWORD *)a1 + 16;
512    v4 = *(_QWORD *)a1 + 24;
513    v5 = *(_QWORD *)a2;
514    if ( (signed int)respond(v4, (__int64)v5) < 0 )
515    {
516        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
517        v6 = v5;
518        if ( v6 < 0 )
519        {
520            v9 = *(_QWORD *)a2 + 32;
521            v11 = *(_BYTE *)a2 + 24;
522            if ( v9 )
523            {
524                v12 = 0LL;
525                if ( *v11 == 10 )
526                {
527                    v12 = 1;
528                }
529            }
530        }
531    }
532    v12 = 0LL;
533    if ( *v11 == 10 )
534    {
535        v12 = 1;
536    }
537    v10 = 3;
538    v11 = v2;
539    v12 = *(_QWORD *)a1 + 8;
540    v13 = *(_QWORD *)a1 + 16;
541    v4 = *(_QWORD *)a1 + 24;
542    v5 = *(_QWORD *)a2;
543    if ( (signed int)respond(v4, (__int64)v5) < 0 )
544    {
545        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
546        v6 = v5;
547        if ( v6 < 0 )
548        {
549            v9 = *(_QWORD *)a2 + 32;
550            v11 = *(_BYTE *)a2 + 24;
551            if ( v9 )
552            {
553                v12 = 0LL;
554                if ( *v11 == 10 )
555                {
556                    v12 = 1;
557                }
558            }
559        }
560    }
561    v12 = 0LL;
562    if ( *v11 == 10 )
563    {
564        v12 = 1;
565    }
566    v10 = 3;
567    v11 = v2;
568    v12 = *(_QWORD *)a1 + 8;
569    v13 = *(_QWORD *)a1 + 16;
570    v4 = *(_QWORD *)a1 + 24;
571    v5 = *(_QWORD *)a2;
572    if ( (signed int)respond(v4, (__int64)v5) < 0 )
573    {
574        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
575        v6 = v5;
576        if ( v6 < 0 )
577        {
578            v9 = *(_QWORD *)a2 + 32;
579            v11 = *(_BYTE *)a2 + 24;
580            if ( v9 )
581            {
582                v12 = 0LL;
583                if ( *v11 == 10 )
584                {
585                    v12 = 1;
586                }
587            }
588        }
589    }
590    v12 = 0LL;
591    if ( *v11 == 10 )
592    {
593        v12 = 1;
594    }
595    v10 = 3;
596    v11 = v2;
597    v12 = *(_QWORD *)a1 + 8;
598    v13 = *(_QWORD *)a1 + 16;
599    v4 = *(_QWORD *)a1 + 24;
600    v5 = *(_QWORD *)a2;
601    if ( (signed int)respond(v4, (__int64)v5) < 0 )
602    {
603        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
604        v6 = v5;
605        if ( v6 < 0 )
606        {
607            v9 = *(_QWORD *)a2 + 32;
608            v11 = *(_BYTE *)a2 + 24;
609            if ( v9 )
610            {
611                v12 = 0LL;
612                if ( *v11 == 10 )
613                {
614                    v12 = 1;
615                }
616            }
617        }
618    }
619    v12 = 0LL;
620    if ( *v11 == 10 )
621    {
622        v12 = 1;
623    }
624    v10 = 3;
625    v11 = v2;
626    v12 = *(_QWORD *)a1 + 8;
627    v13 = *(_QWORD *)a1 + 16;
628    v4 = *(_QWORD *)a1 + 24;
629    v5 = *(_QWORD *)a2;
630    if ( (signed int)respond(v4, (__int64)v5) < 0 )
631    {
632        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
633        v6 = v5;
634        if ( v6 < 0 )
635        {
636            v9 = *(_QWORD *)a2 + 32;
637            v11 = *(_BYTE *)a2 + 24;
638            if ( v9 )
639            {
640                v12 = 0LL;
641                if ( *v11 == 10 )
642                {
643                    v12 = 1;
644                }
645            }
646        }
647    }
648    v12 = 0LL;
649    if ( *v11 == 10 )
650    {
651        v12 = 1;
652    }
653    v10 = 3;
654    v11 = v2;
655    v12 = *(_QWORD *)a1 + 8;
656    v13 = *(_QWORD *)a1 + 16;
657    v4 = *(_QWORD *)a1 + 24;
658    v5 = *(_QWORD *)a2;
659    if ( (signed int)respond(v4, (__int64)v5) < 0 )
660    {
661        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
662        v6 = v5;
663        if ( v6 < 0 )
664        {
665            v9 = *(_QWORD *)a2 + 32;
666            v11 = *(_BYTE *)a2 + 24;
667            if ( v9 )
668            {
669                v12 = 0LL;
670                if ( *v11 == 10 )
671                {
672                    v12 = 1;
673                }
674            }
675        }
676    }
677    v12 = 0LL;
678    if ( *v11 == 10 )
679    {
680        v12 = 1;
681    }
682    v10 = 3;
683    v11 = v2;
684    v12 = *(_QWORD *)a1 + 8;
685    v13 = *(_QWORD *)a1 + 16;
686    v4 = *(_QWORD *)a1 + 24;
687    v5 = *(_QWORD *)a2;
688    if ( (signed int)respond(v4, (__int64)v5) < 0 )
689    {
690        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
691        v6 = v5;
692        if ( v6 < 0 )
693        {
694            v9 = *(_QWORD *)a2 + 32;
695            v11 = *(_BYTE *)a2 + 24;
696            if ( v9 )
697            {
698                v12 = 0LL;
699                if ( *v11 == 10 )
700                {
701                    v12 = 1;
702                }
703            }
704        }
705    }
706    v12 = 0LL;
707    if ( *v11 == 10 )
708    {
709        v12 = 1;
710    }
711    v10 = 3;
712    v11 = v2;
713    v12 = *(_QWORD *)a1 + 8;
714    v13 = *(_QWORD *)a1 + 16;
715    v4 = *(_QWORD *)a1 + 24;
716    v5 = *(_QWORD *)a2;
717    if ( (signed int)respond(v4, (__int64)v5) < 0 )
718    {
719        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
720        v6 = v5;
721        if ( v6 < 0 )
722        {
723            v9 = *(_QWORD *)a2 + 32;
724            v11 = *(_BYTE *)a2 + 24;
725            if ( v9 )
726            {
727                v12 = 0LL;
728                if ( *v11 == 10 )
729                {
730                    v12 = 1;
731                }
732            }
733        }
734    }
735    v12 = 0LL;
736    if ( *v11 == 10 )
737    {
738        v12 = 1;
739    }
740    v10 = 3;
741    v11 = v2;
742    v12 = *(_QWORD *)a1 + 8;
743    v13 = *(_QWORD *)a1 + 16;
744    v4 = *(_QWORD *)a1 + 24;
745    v5 = *(_QWORD *)a2;
746    if ( (signed int)respond(v4, (__int64)v5) < 0 )
747    {
748        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
749        v6 = v5;
750        if ( v6 < 0 )
751        {
752            v9 = *(_QWORD *)a2 + 32;
753            v11 = *(_BYTE *)a2 + 24;
754            if ( v9 )
755            {
756                v12 = 0LL;
757                if ( *v11 == 10 )
758                {
759                    v12 = 1;
760                }
761            }
762        }
763    }
764    v12 = 0LL;
765    if ( *v11 == 10 )
766    {
767        v12 = 1;
768    }
769    v10 = 3;
770    v11 = v2;
771    v12 = *(_QWORD *)a1 + 8;
772    v13 = *(_QWORD *)a1 + 16;
773    v4 = *(_QWORD *)a1 + 24;
774    v5 = *(_QWORD *)a2;
775    if ( (signed int)respond(v4, (__int64)v5) < 0 )
776    {
777        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
778        v6 = v5;
779        if ( v6 < 0 )
780        {
781            v9 = *(_QWORD *)a2 + 32;
782            v11 = *(_BYTE *)a2 + 24;
783            if ( v9 )
784            {
785                v12 = 0LL;
786                if ( *v11 == 10 )
787                {
788                    v12 = 1;
789                }
790            }
791        }
792    }
793    v12 = 0LL;
794    if ( *v11 == 10 )
795    {
796        v12 = 1;
797    }
798    v10 = 3;
799    v11 = v2;
800    v12 = *(_QWORD *)a1 + 8;
801    v13 = *(_QWORD *)a1 + 16;
802    v4 = *(_QWORD *)a1 + 24;
803    v5 = *(_QWORD *)a2;
804    if ( (signed int)respond(v4, (__int64)v5) < 0 )
805    {
806        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
807        v6 = v5;
808        if ( v6 < 0 )
809        {
810            v9 = *(_QWORD *)a2 + 32;
811            v11 = *(_BYTE *)a2 + 24;
812            if ( v9 )
813            {
814                v12 = 0LL;
815                if ( *v11 == 10 )
816                {
817                    v12 = 1;
818                }
819            }
820        }
821    }
822    v12 = 0LL;
823    if ( *v11 == 10 )
824    {
825        v12 = 1;
826    }
827    v10 = 3;
828    v11 = v2;
829    v12 = *(_QWORD *)a1 + 8;
830    v13 = *(_QWORD *)a1 + 16;
831    v4 = *(_QWORD *)a1 + 24;
832    v5 = *(_QWORD *)a2;
833    if ( (signed int)respond(v4, (__int64)v5) < 0 )
834    {
835        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
836        v6 = v5;
837        if ( v6 < 0 )
838        {
839            v9 = *(_QWORD *)a2 + 32;
840            v11 = *(_BYTE *)a2 + 24;
841            if ( v9 )
842            {
843                v12 = 0LL;
844                if ( *v11 == 10 )
845                {
846                    v12 = 1;
847                }
848            }
849        }
850    }
851    v12 = 0LL;
852    if ( *v11 == 10 )
853    {
854        v12 = 1;
855    }
856    v10 = 3;
857    v11 = v2;
858    v12 = *(_QWORD *)a1 + 8;
859    v13 = *(_QWORD *)a1 + 16;
860    v4 = *(_QWORD *)a1 + 24;
861    v5 = *(_QWORD *)a2;
862    if ( (signed int)respond(v4, (__int64)v5) < 0 )
863    {
864        v5 = protocol_request(*(_QWORD *)a2 + 24, *(_QWORD *)a2 + 32, (__int64)v24, 0LL, 0LL);
865        v6 = v5;
866        if ( v6 < 0 )
867        {
868            v9 = *(_QWORD *)a2 + 32;
869            v11 = *(_BYTE *)a2 + 24;
870            if ( v9 )
871            {
872                v12 = 0LL;
873                if ( *v11 == 10 )
874                {
875                    v12 = 1;
876                }
877            }
878        }
879    }
880    v12 = 0LL;
881    if ( *v11 == 10 )
882    {
883        v12 = 1;
884    }
885    v10 = 3;
886    v11 = v2;
887    v12 = *(_QWORD *)a1 + 8;
888    v13 = *(_QWORD *)a1 + 16;

```

<pre> 1 __int64 __fastcall respond(int fd, __int64 a2) 2 { 3 __int64 v2; // rax@1 4 size_t v3; // rbx@2 5 int v4; // er13@3 6 __int64 v5; // rbp@5 7 __int64 v6; // r13@5 8 ssize_t v7; // rax@8 9 fd_set writefds; // [sp+10h] [bp-10B8h]@2 10 char s[4152]; // [sp+90h] [bp-1038h]@1 11 12 v2 = protocol_response(s, 0x1000uLL, a2); 13 if (v2 < 0) 14 return 0xFFFFFFFFLL; 15 v3 = v2; 16 memset(&writefds, 0, sizeof(writefds)); </pre>	<pre> 1 __int64 __fastcall respond(int fd, __int64 a2) 2 { 3 __int64 v2; // rax@1 4 size_t v3; // rbx@2 5 int v4; // er13@3 6 __int64 v5; // rbp@5 7 __int64 v6; // r13@5 8 ssize_t v7; // rax@8 9 fd_set writefds; // [sp+10h] [bp-20B8h]@2 10 char s[8248]; // [sp+90h] [bp-2038h]@1 11 12 v2 = protocol_response(s, 0x2000uLL, a2); 13 if (v2 < 0) 14 return 0xFFFFFFFFLL; 15 v3 = v2; 16 memset(&writefds, 0, sizeof(writefds)); </pre>
--	--

It seems that the only change in this function is, actually, the size of a stack variable and the given size. If we're looking for the new functionality added to the product, it can be irritating going through a big list of small changes. We will re-diff both databases: run again Diaphora by executing diaphora.py and, in the dialog select this time “Relaxed calculations on difference ratios” as shown below:



Press OK and wait for it to finish. When it's finished, go to the “Best matches” tab and find the “respond” or “scan_response” functions:

Line	Address	Name	Address 2	Name 2	Description
00002438	006c362c	__cxa_pure_virtual	006c362c	__cxa_pure_virtual	Equal assembly
00002439	006c3638	__gmon_start__	006c3638	__gmon_start__	Equal assembly
00002440	0040bd70	respond	0040bd70	respond	Bytes hash and names (ratio 1.00)
00002441	0040ce10	scan_response	0040ce90	scan_response	Bytes hash and names (ratio 1.00)
00002442	0040d200	exclude_response.isra.7	0040d440	exclude_response.isra.9	Bytes hash and names (ratio 1.00)
00002443	0040d3c0	license_response.isra.9	0040d600	license_response.isra.11	Bytes hash and names (ratio 1.00)
00002444	0040d6a0	protocol_response	0040d8e0	protocol_response	Bytes hash and names (ratio 1.00)

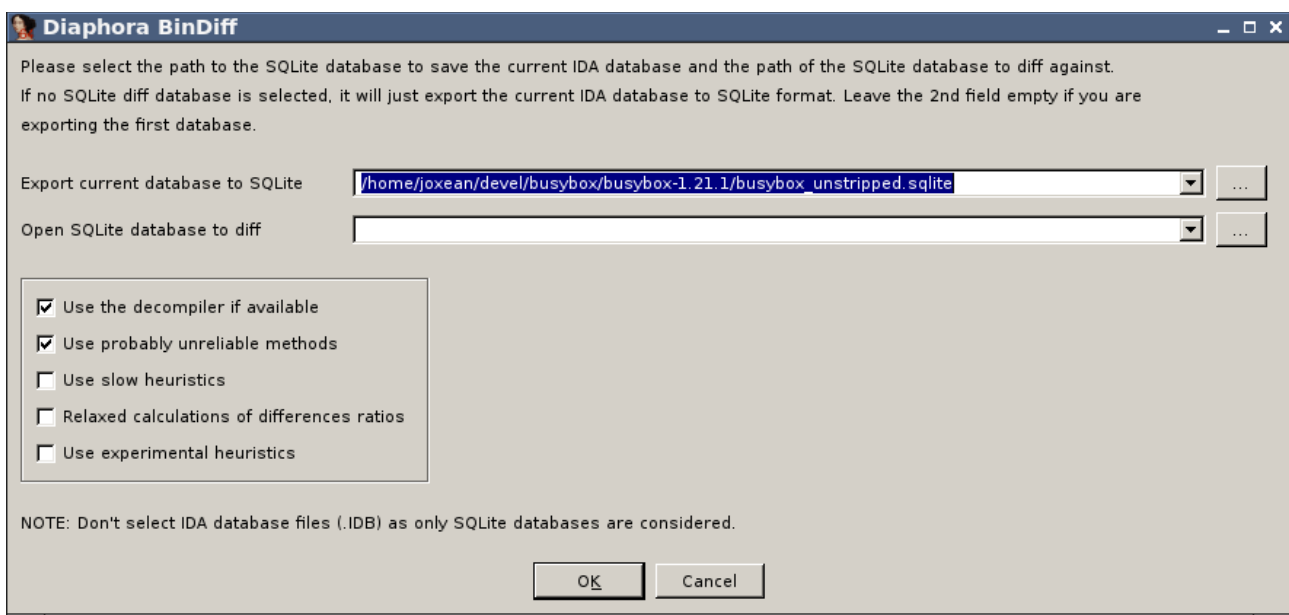
This time, as we can see, both functions appear in the “Best matches”, the list of functions that are considered equal, so you don't need to go through a big list with small changes here and there: the “Partial matches” tab will show only functions with bigger changes, making it easier to discover the

new functionalities added to the program.

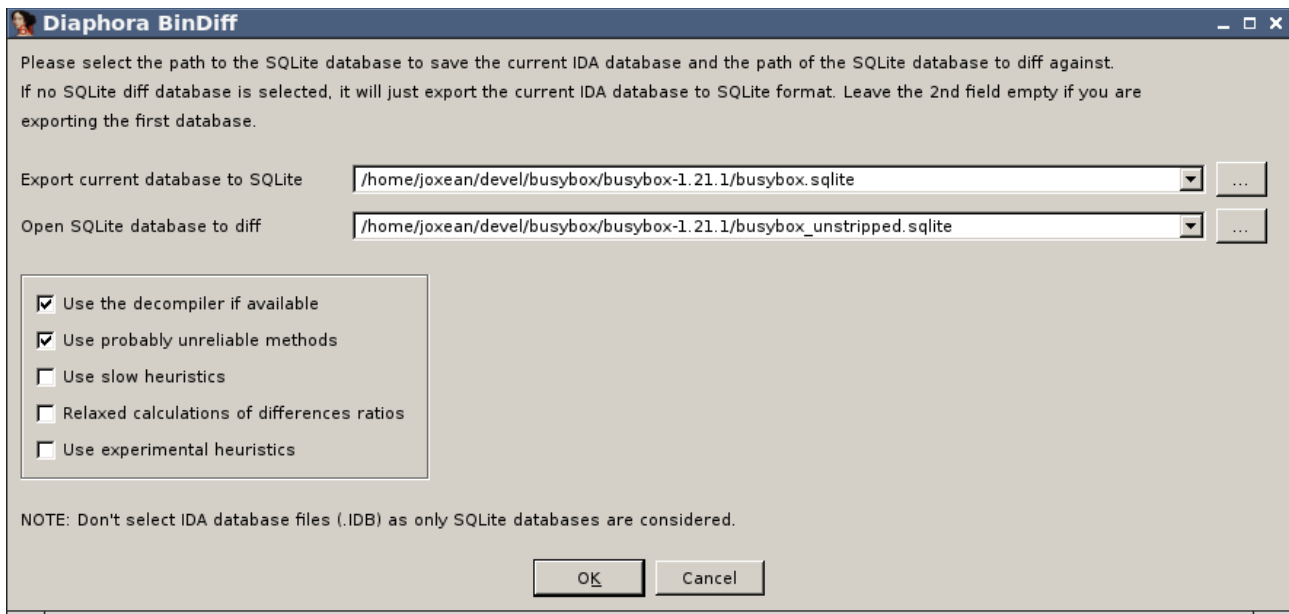
Porting symbols

One of the most common tasks in reverse engineering, at least from my experience, is porting symbols from previous versions of a program, library, etc... to the new version. It can be quite frustrating having to port function names, enumerations, comments, structure definitions, etc... manually to new versions, specially when talking about big reverse engineering projects.

In the following example, we will import the symbols, structures, enumerations, comments, prototypes, etc... from one version full of symbols to another version with symbols stripped. We will use Busybox 1.21-1, compiled in Ubuntu Linux for x86_64. After downloading and compiling it, we will have 2 different binaries: “busybox” and “busybox_unstripped”. The later, is the version with full symbols while the former is the one typically used for distribution, with all the symbols stripped. Launch IDA and open, first, the “busybox_unstripped” binary containing full symbols. Let's IDA finish the initial auto-analysis and, after this, run Diaphora by either running diaphora.py. In the dialog just opened, press OK:



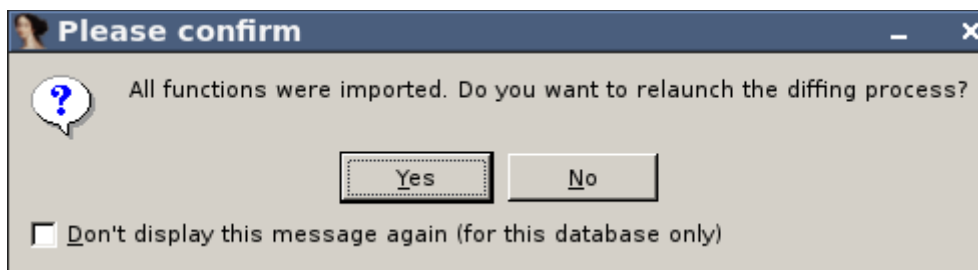
Wait until Diaphora finishes exporting to SQLite the current database. When it finishes, close the current IDA database and open the binary “busybox”, wait until IDA finishes the initial auto-analysis and, then, launch again Diaphora. In the next dialog select as the SQLite database to diff the previous one we just created, the one with all the symbols from the previous binary:



Press OK and wait until it finishes comparing both databases. After a while, it will show various tabs with all the unmatched functions in both databases, as well as the “Best”, “Partial” and “Unreliable” matches tabs.








Line	Address	Name	Address 2	Name 2	Description
00000000	004084dc	sub_4084DC	004084dc	skip_dev_pfx	Bytes hash and names (ratio 0.545455)
00000001	00408cf0	sub_408CF0	00408cf0	xstrndup	Bytes hash and names (ratio 0.657895)
00000002	00408d40	sub_408D40	00408d40	xlopen	Bytes hash and names (ratio 0.750000)
00000003	00408d5d	sub_408D5D	00408d5d	xopen3	Bytes hash and names (ratio 0.857143)
00000004	00408d91	sub_408D91	00408d91	open3_or_warn	Bytes hash and names (ratio 0.750000)
00000005	00408dc4	sub_408DC4	00408dc4	xunlink	Bytes hash and names (ratio 0.692308)
00000006	00408de2	sub_408DE2	00408de2	xrename	Bytes hash and names (ratio 0.631579)
00000007	00408e0a	sub_408E0A	00408e0a	rename_or_warn	Bytes hash and names (ratio 0.750000)

As we can see, Diaphora did a decent work matching 796 functions labeled as “Best Matches” and 2296 labeled as “Partial matches”, a total of 3092 functions out of 3134. Let's go to the “Best matches” tab. All the functions here are these that were matched with a high confidence ratio. Let's say that we want to import all the symbols for the “best matches”: right click on the list and select “Import all functions”. It will ask if we really want to do so: press YES. It will import all function names, comments, function prototypes, structures, enumerations and even IDA's type libraries (TILs). When it's done it will ask us if we want to relaunch again the diffing process:

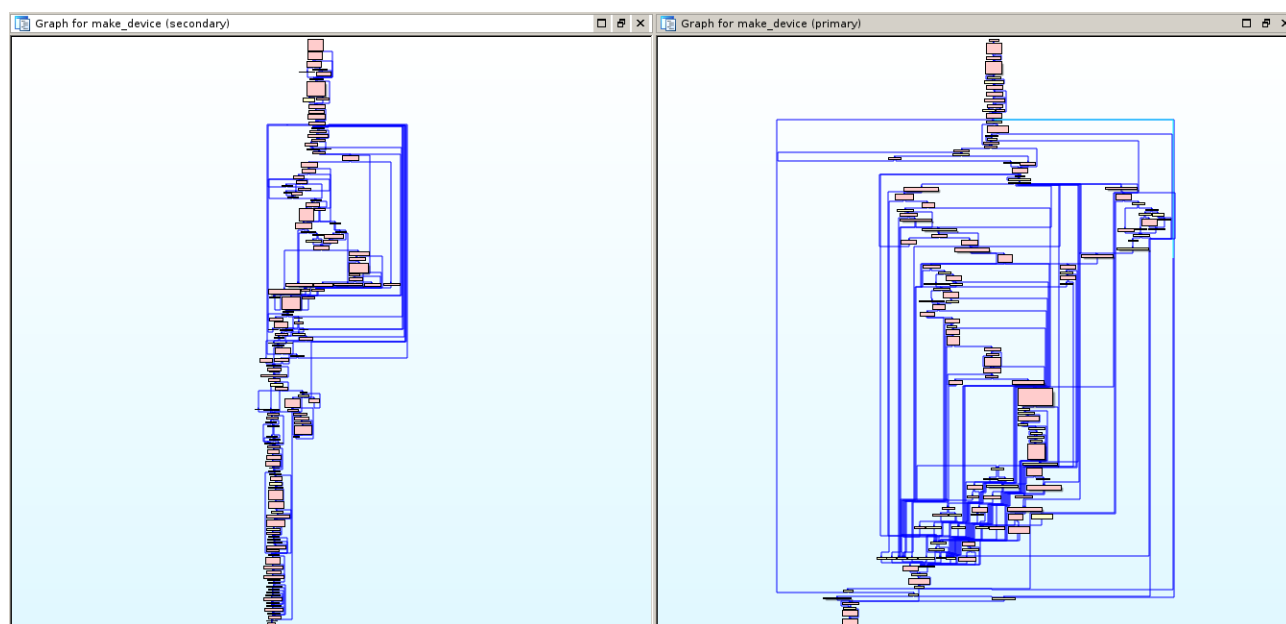


While Diaphora import symbols, at the same time, it updates the database with the exported data from the primary database and, as so, with the new information it may be possible to match new functions not discovered before. In this case we will just say “NO” to this dialog.

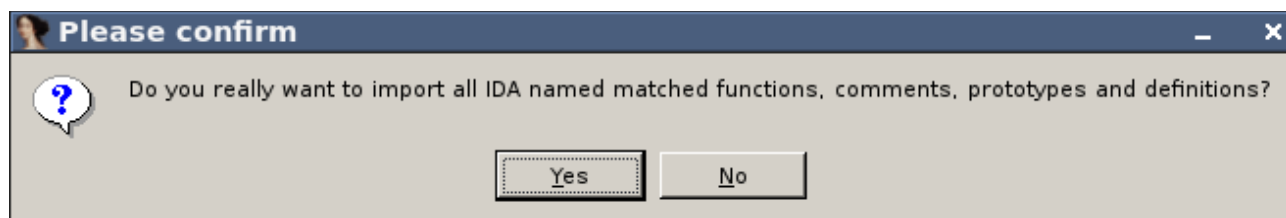
Now, go to the “Partial matches” tab. In this list we have some matches that doesn't look like good ones:

Line	Address	Name	Address 2	Name 2	Description
 00000083	004372ca	sub_4372CA	004372ca	rtntl_rtntype_n2a	Bytes hash and names (ratio 0.962963)
 00000038	00411dae	sub_411DAE	00411dae	bus_state_value	Bytes hash and names (ratio 0.964286)
 00002295	0046c671	make_device	004654e3	make_device	Perfect match, same name (ratio 0.142721)
 00002291	006d042c	sigaction	006d042c	isatty@@GLIBC_2.2.5	Same address, nodes, edges and mnemonics (ratio 0.000000)
 00002292	006d0588	pipe	006d0588	close@@GLIBC_2.2.5	Same address, nodes, edges and mnemonics (ratio 0.000000)
 00002293	006d0854	flock	006d0854	semctl@@GLIBC_2.2.5	Same address, nodes, edges and mnemonics (ratio 0.000000)
 00002294	006d086c	sysinfo	006d086c	shmget@@GLIBC_2.2.5	Same address, nodes, edges and mnemonics (ratio 0.000000)

As we can see, the ratios are pretty low: from 0.00 to 0.14. Let's diff the graphs of the “make_device” function (matched with the “same name” heuristic):



It doesn't look like a good match. And, if it's, it's rather big to verify yet. Let's delete this result: go back to the “Partial matches” tab, select the “make_device” match and, simply, press “DEL”. It will just remove this result. Now, do the same for the next results with 0.00 confidence ratio. OK, we removed the bad results. Now, it's time to import all the partial matches: right click in the list and select “Import all data for sub_* functions”. It will import everything for functions that are IDA's auto-named, these that start with the “sub_” prefix but will not touch any function with a non IDA auto-generated name. It will ask us for confirmation:



Press “Yes”, and wait until it exports everything and updates the primary database. And, that's all! We just imported everything from function names and comments to structures and enumerations into the new database and we can just continue with our work with the new database and with all the data imported from the database we used to worked on before.

Diffing huge databases (or exporting smaller SQLite databases)

Some IDA databases can be huge: for example, IDA databases for firmware images or IDA databases for >100MB binaries. In such cases, exporting and diffing big databases takes a lot of time and space. In order to make it a bit faster and requiring less disk space to store the .sqlite databases that Diaphora uses the following new options were added in the last release candidate:

- Export only non-IDA generated functions. It will only export the functions that are not IDA's auto-generated names, thus, exporting only the functions for which we have symbols or we already assigned a name.
- Do not export instructions and basic blocks. It will export everything about the functions but will not export basic blocks, basic block's relationships or the instructions of all functions. It results in less export time as well as in significantly smaller SQLite databases.

Heuristics

Diaphora uses multiple heuristics to find matches between different functions. The next list shows all the heuristics implemented in the Diaphora Release Candidate 1:

Best matches

- The very first try is to find if everything in both databases, even the primary key values are equals. If so, the databases are considered 100% equals and nothing else is done.
- **Equal pseudo-code.** The pseudo-code generated by the Hex-Rays decompiler are equals. It can match code from x86, x86_64 and ARM interchangeably.
- **Equal assembly.** The assembly of both functions is exactly equal.
- **Bytes hash and names.** The first byte of each assembly instruction is equal and also the referenced true names, not IDA's auto-generated ones, have the same names.
- **Same address, nodes, edges and mnemonics.** The number of basic blocks, their addresses, the names of edges and the mnemonics in both databases are equal

Partial and unreliable matches (according to the confidence's ratio):

- **Same name.** The mangled or demangled name is the same in both functions.
- **Same address, nodes, edges and primes (re-ordered instructions).** The function has the same address, number of basic blocks, edges and a the prime corresponding to the cyclomatic complexity are equal. It typically matches functions with re-ordered instructions.
- **Import names hash.** The functions called from both functions are the same, matched by the

demangled names.

- **Nodes, edges, complexity, mnemonics, names, prototype, in-degree and out-degree.** The number of basic blocks, mnemonics, names, the function's prototype the in-degree (calls to the function) and out-degree (calls performed to other functions) is the same.
- **Nodes, edges, complexity, mnemonics, names and prototype.** The number of basic blocks, edges, the cyclomatic complexity, the mnemonics, the true names used in the function and even the prototype of the function (stripping the function name) are the same.
- **Mnemonics and names.** The functions have the same mnemonics and the same true names used in the function. It's done for functions with the same number of instructions.
- **Small names difference.** At least 50% of the true names used in both functions are the same.
- **Pseudo-code fuzzy hash.** It checks the normal fuzzy hash (calculated with the DeepToad's library kfuzzy.py) for both functions.
- **Pseudo-code fuzzy hashes.** It checks all the 3 fuzzy hashes (calculated with the DeepToad's library kfuzzy.py) for both functions. This is considered a slow heuristic.
- **Similar pseudo-code.** The pseudo-code generated by the Hex-Rays decompiler is similar with a confidence ratio bigger or equal to 0.729. This is considered a slow heuristic.
- **Pseudo-code fuzzy AST hash.** The fuzzy hash calculated via SPP (small-primes-product) from the AST of the Hex-Rays decompiled function is the same for both functions. It typically catches C constructions that are re-ordered, not just re-ordered assembly instructions.
- **Partial pseudo-code fuzzy hash.** At least the first 16 bytes of the fuzzy hash (calculated with the DeepToad's library kfuzzy.py) for both functions matches. This is considered a slow heuristic.
- **Same high complexity, prototype and names.** The cyclomatic complexity is at least 20, and the prototype and the true names used in the function are the same for both databases.
- **Same high complexity and names.** Same as before but ignoring the function's prototype.

Unreliable matches

- **Bytes hash.** The first byte of each instruction is the same for both functions. This is considered a slow heuristic.
- **Nodes, edges, complexity and mnemonics.** The number of basic blocks, relations, the cyclomatic complexity (naturally) and the mnemonics are the same. It can match functions too similar that actually perform opposite operations (like add_XXX and sub_XXX). Besides, this is considered a slow heuristic.
- **Nodes, edges, complexity and prototype.** Same as before but the mnemonics are ignored and only the true names used in both functions are considered. This is considered a slow heuristic.
- **Nodes, edges, complexity, in-degree and out-degree.** The number of basic blocks, edges, cyclomatic complexity (naturally), the number of functions calling it and the number of functions called from both functions are the same. This is considered a slow heuristic.

- **Nodes, edges and complexity.** Same number of nodes, edges and, naturally, cyclomatic complexity values. This is considered a slow heuristic.
- **Similar pseudo-code.** The pseudo-codes are considered similar with a confidence's ratio of 0.58 or less. This is considered a slow heuristic.
- **Same high complexity.** Both functions has the same high cyclomatic complexity, being it at least 50. This is considered a slow heuristic.

Experimental (and likely to be removed or moved or changed in the future):

- **Similar small pseudo-code.** The pseudo-code generated by the Hex-Rays decompiler is less or equal to 5 lines and is the same for both functions. It matches too many things and the calculated confidence's ratio is typically bad.
- **Small pseudo-code fuzzy AST hash.** Same as “Pseudo-code fuzzy AST hash” but applied to functions with less or equal to 5 lines of pseudo-code. Like the previous heuristic, it matches too many things and the calculated confidence's ratio is typically bad..
- **Similar small pseudo-code.** Even worst than “**Similar small pseudo-code**”, as it tries to match similar functions with 5 or less lines of pseudo-code, matching almost anything and getting confidence's ratios of 0.25 being lucky.
- **Equal small pseudo-code.** Even worst than before, as it matches functions with the same pseudo-code being 5 or less lines of code long. Typically, you can get 2 or 3 results, that are, actually, wrong.
- **Same low complexity, prototype and names.** The prototype of the functions, the true names used in the functions and its cyclomatic complexity, being it less than 20, is the same. It worked for me once, I think.
- **Same low complexity and names.** The cyclomatic complexity, being it less than 20, and the true names used in the function are the same. It typically matches functions already matched by other heuristics, so it's usefulness is really limited.