Diaphora – An IDA Python BinDiffing plugin

Index

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:

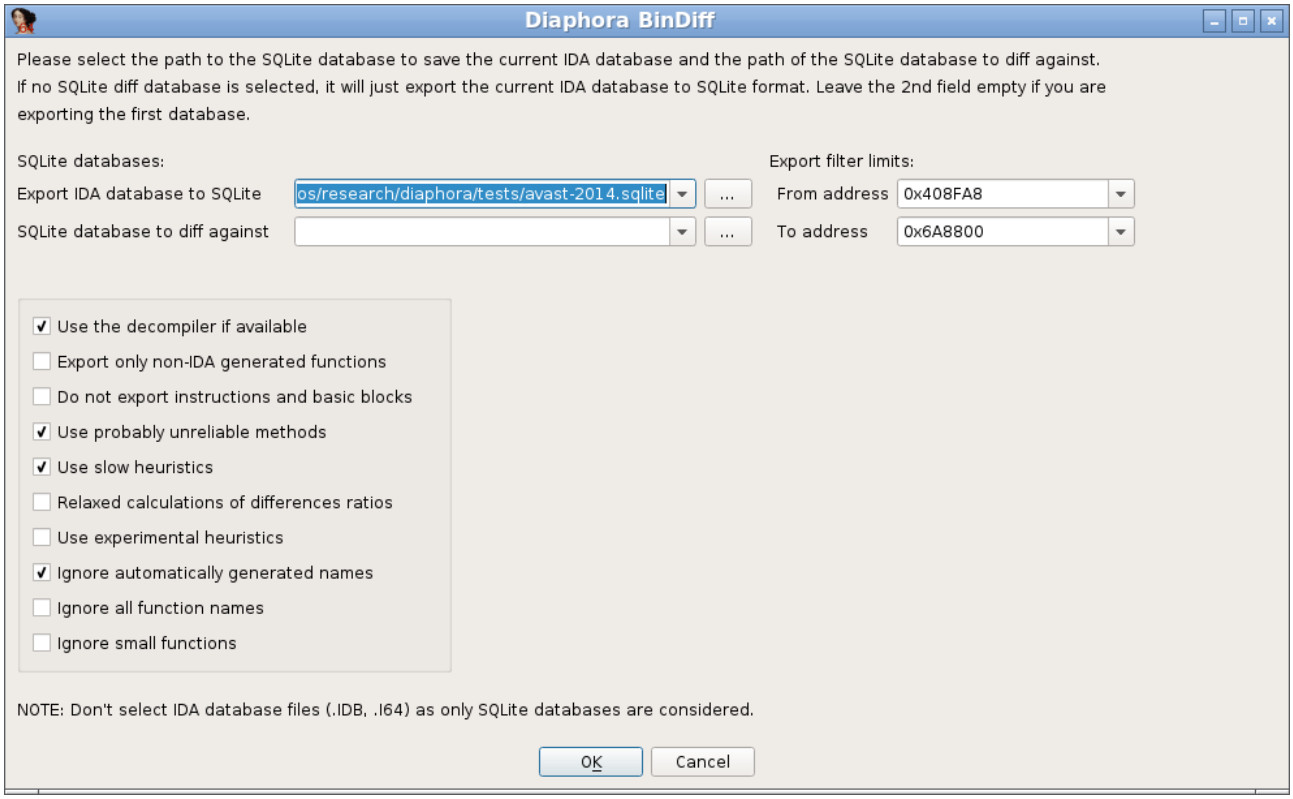
1. diaphora.py: The main IDAPython plugin. It contains all the code of the heuristics, graphs displaying, export interface, etc...
2. jkutils/kfuzzy.py: This is an unmodified version of the kfuzzy.py library, part of the DeepToad project, a tool and a library for performing fuzzy hashing of binary files. 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 directoy 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 it can be a bit confusing at first, is used for both exporting the current IDA database to SQLite format as well as 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. **Export only non-IDA generated functions.** Self-explanatory, only functions with non IDA autogenerated names will be exported.
3. **Do not export instructions and basic blocks.** Export only function summaries. When exporting huge databases, it may help speeding up operations. However, the diffing capabilities will be more limited.
4. **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 my 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”.

5. **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.
6. **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.
7. **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.
8. **Ignore automatically generated names.** When performing the comparison between databases, it tells Diaphora to ignore in the “Same name” heuristic functions with the same IDA's autogenerated name (i.e., when there are two function sub_01020304 in both databases but they aren't actually the same function). **Used only when comparing**.
9. **Ignore all function names.** Just disable the “Same name” heuristic. **Used only when comparing**.
10. **Ignore small functions.** Ignore functions with less than 6 assembly instructions. **Used only when comparing**.

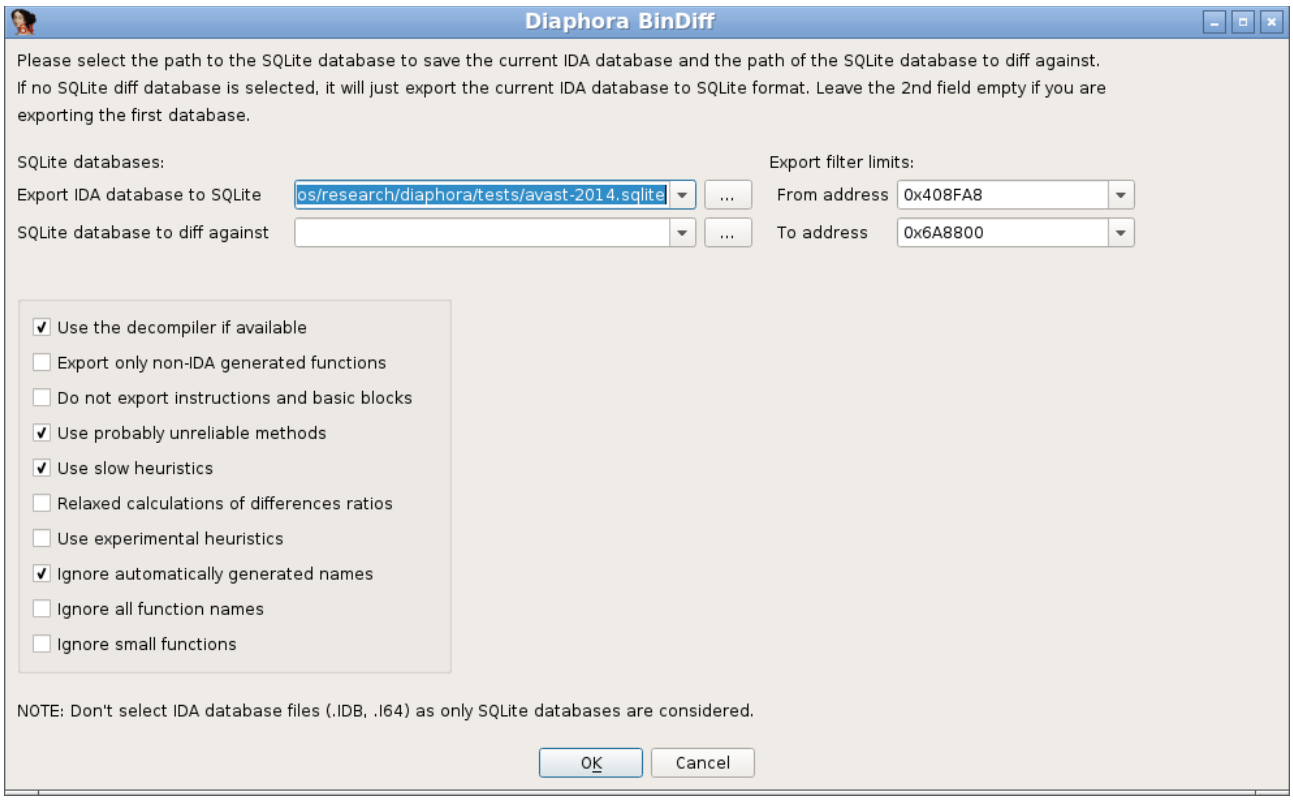
Diaphora quick start

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. c0092cf0cb1286cfd8399681bcab68a4 avast-2014
2. 716ff77e74e47d3ee619df49995536ec avast

The file “avast-2014” is an old version from 2014 and the binary “avast” is the latest version. Launch IDA Pro for 64 bits (idaq64) and open the file “avast-2014”. Once the initial auto-analysis finishes launch Diaphora by either running the script “diaphora.py”. The following dialog will open:

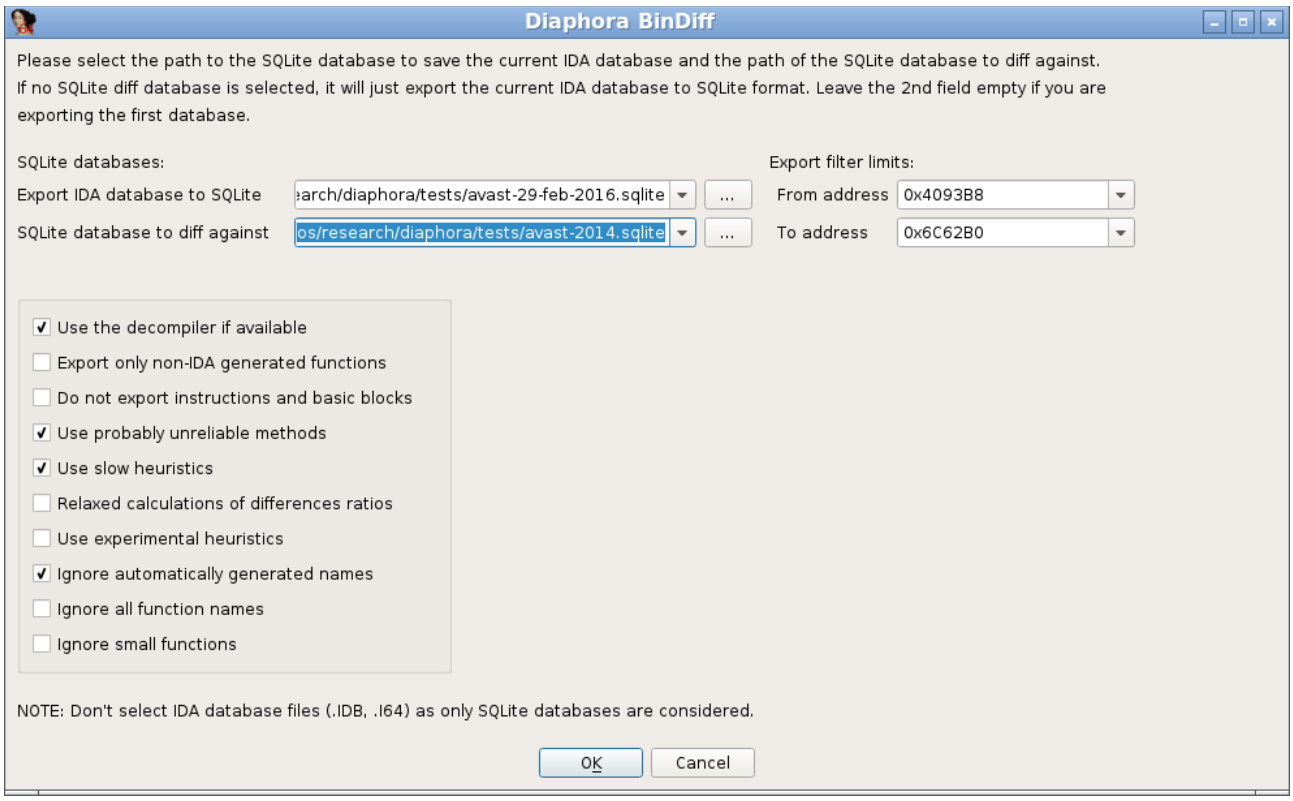


We only need to care about 2 things:

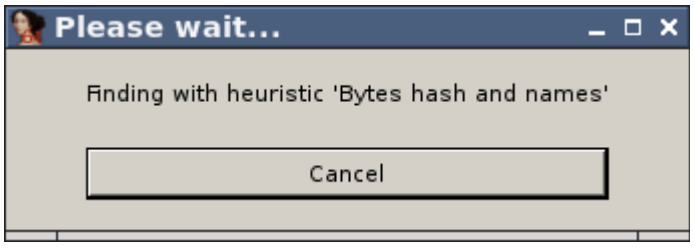
1. Field "**Export current database to SQLite**". This is the path to the SQLite database that will be created with all the information extracted from the IDA database of this avast binary.
2. Field "**Use the decompiler if available**". If the Hex-Rays decompiler is available and we want to use it, 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	ffffffffffffffff	UnixUnlockDefinitionsFolder(void...
00000002	ffffffffffffffff	UnixLockDefinitionsFolder(char c...
00000003	ffffffffffffffff	asw::root::CGenericFile::writesa...
00000004	0043d470	dep_fsUnlockFile
00000005	0043d510	dep_fsLockFile
00000006	ffffffffffffffff	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

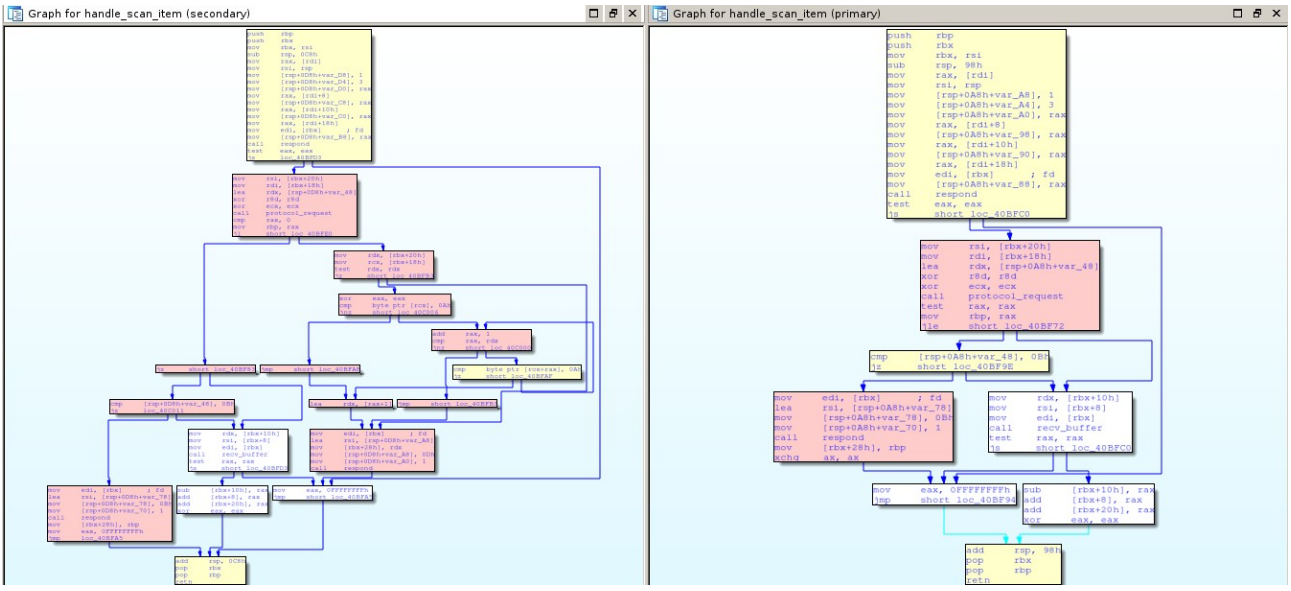
It seems they added various functions to check for SSE, AVX, etc... Intel instructions. Also, they added 2 new functions called handler and context. Let's take a look now to the “Best matches” tab opened:

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

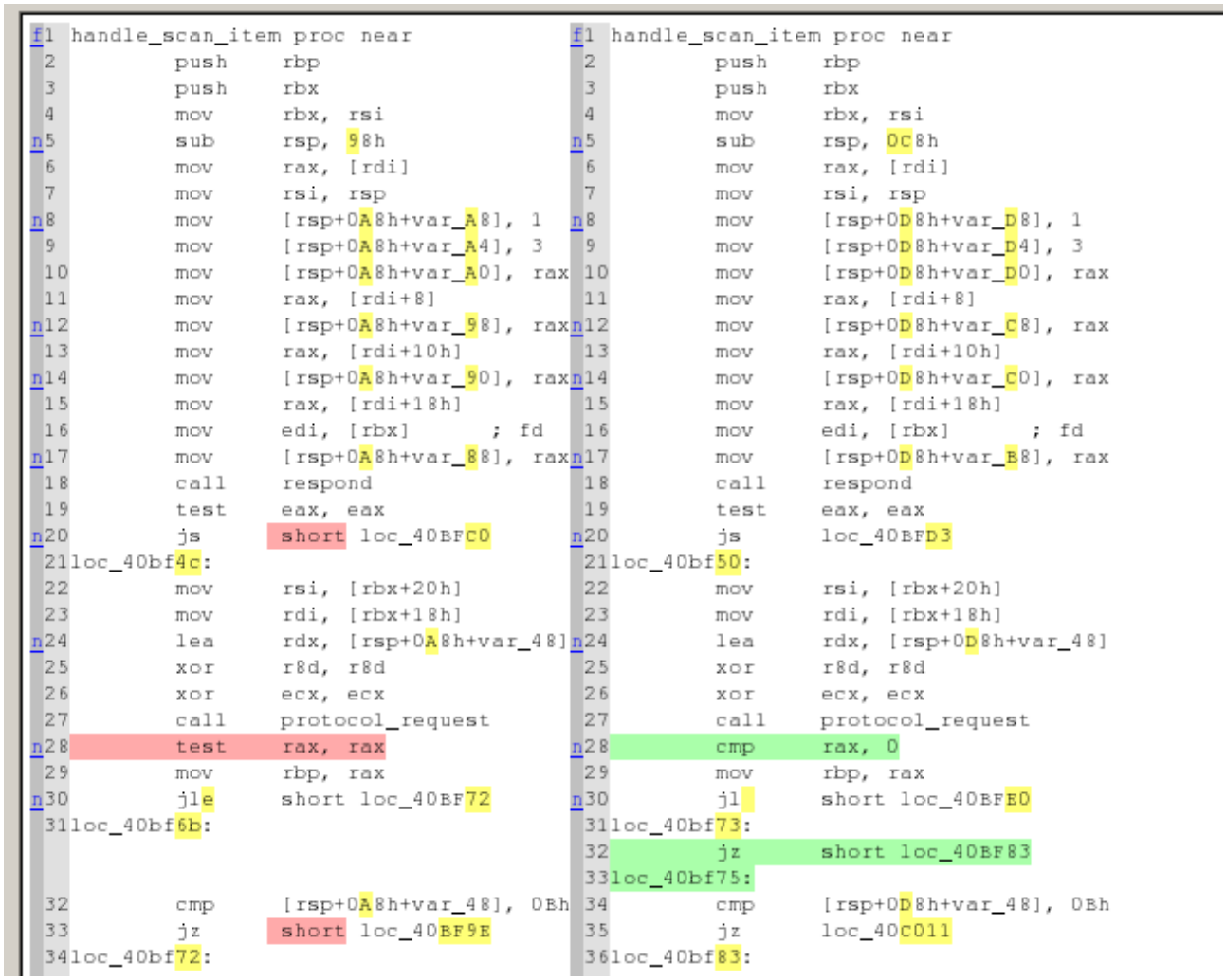
There are many functions in the “Best matches” tab, 2556 functions, and in the primary database there are 2659 functions. The results shown in the “Best matches” tab are these functions matched with some heuristic (like “100% equal”, where all attributes are equal, or “Equal pseudo-code”, where the pseudo-code 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	cmnbInit	0042b130	cmnbInit	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”:

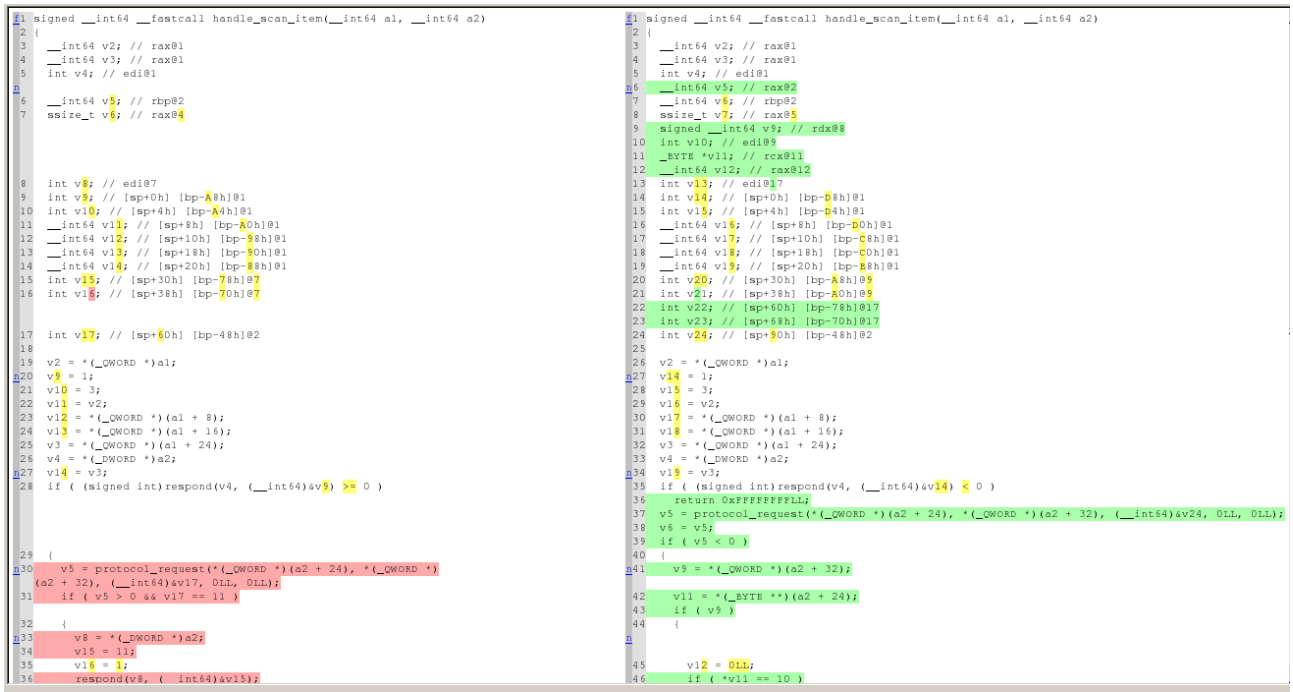


```
handle_scan_item proc near
        push    rbp
        push    rbx
        mov     rbx, rsi
        sub     rsp, 98h
        mov     rax, [rdi]
        mov     rsi, rsp
        mov     [rsp+0A8h+var_A8], 1
        mov     [rsp+0A8h+var_A4], 3
        mov     [rsp+0A8h+var_A0], rax
        mov     rax, [rdi+8]
        mov     [rsp+0A8h+var_98], rax
        mov     rax, [rdi+10h]
        mov     [rsp+0A8h+var_90], rax
        mov     rax, [rdi+18h]
        mov     edi, [rbx]      ; fd
        mov     [rsp+0A8h+var_88], rax
        call    respond
        test    eax, eax
        js      short loc_40BFC0
loc_40bf4c:
        mov     rsi, [rbx+20h]
        mov     rdi, [rbx+18h]
        lea     rdx, [rsp+0A8h+var_48]
        xor     r8d, r8d
        xor     ecx, ecx
        call    protocol_request
        test    rax, rax
        mov     rbp, rax
        jle     short loc_40BF72
loc_40bf6b:


        cmp     [rsp+0A8h+var_48], 0Bh
        jz      short loc_40BF9E
loc_40bf72:
```

```
handle_scan_item proc near
        push    rbp
        push    rbx
        mov     rbx, rsi
        sub     rsp, 0C8h
        mov     rax, [rdi]
        mov     rsi, rsp
        mov     [rsp+0D8h+var_D8], 1
        mov     [rsp+0D8h+var_D4], 3
        mov     [rsp+0D8h+var_D0], rax
        mov     rax, [rdi+8]
        mov     [rsp+0D8h+var_C8], rax
        mov     rax, [rdi+10h]
        mov     [rsp+0D8h+var_C0], rax
        mov     rax, [rdi+18h]
        mov     edi, [rbx]      ; fd
        mov     [rsp+0D8h+var_B8], rax
        call    respond
        test    eax, eax
        js      loc_40BFD3
loc_40bf50:
        mov     rsi, [rbx+20h]
        mov     rdi, [rbx+18h]
        lea     rdx, [rsp+0D8h+var_48]
        xor     r8d, r8d
        xor     ecx, ecx
        call    protocol_request
        cmp     rax, 0
        mov     rbp, rax
        jl      short loc_40BFE0
loc_40bf73:
        jz      short loc_40BF83
loc_40bf75:
        cmp     [rsp+0D8h+var_48], 0Bh
        jz      loc_40C011
loc_40bf83:
```

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”:



```
signed __int64 __fastcall handle_scan_item(__int64 a1, __int64 a2)
{
  __int64 v2; // rax@1
  __int64 v3; // rax@1
  int v4; // edi@1

  __int64 v5; // rbp@2
  ssize_t v6; // rax@4




  int v8; // edi@7
  int v9; // [sp+0h] [bp-A8h]@1
  int v10; // [sp+4h] [bp-A4h]@1
  __int64 v11; // [sp+8h] [bp-A0h]@1
  __int64 v12; // [sp+10h] [bp-98h]@1
  __int64 v13; // [sp+18h] [bp-90h]@1
  __int64 v14; // [sp+20h] [bp-88h]@1
  int v15; // [sp+30h] [bp-78h]@7
  int v16; // [sp+38h] [bp-70h]@7


  int v17; // [sp+60h] [bp-48h]@2

  v2 = *(_QWORD *)a1;
  v9 = 1;
  v10 = 3;
  v11 = v2;
  v12 = *(_QWORD *)(a1 + 8);
  v13 = *(_QWORD *)(a1 + 16);
  v3 = *(_QWORD *)(a1 + 24);
  v4 = *(_DWORD *)a2;
  v14 = v3;
  if ( (signed int)respond(v4, (__int64)&v9) >= 0 )




  {
    v5 = protocol_request(*(_QWORD *)(a2 + 24), *(_QWORD *)
(a2 + 32), (__int64)&v17, 0LL, 0LL);
    if ( v5 > 0 && v17 == 11 )

    {
      v8 = *(_DWORD *)a2;
      v15 = 11;
      v16 = 1;
      respond(v8, (__int64)&v15);
```

```
signed __int64 __fastcall handle_scan_item(__int64 a1, __int64 a2)
{
  __int64 v2; // rax@1
  __int64 v3; // rax@1
  int v4; // edi@1
  __int64 v5; // rax@2
  __int64 v6; // rbp@2
  ssize_t v7; // rax@5
  signed __int64 v9; // rdx@8
  int v10; // edi@9
  _BYTE *v11; // rcx@11
  __int64 v12; // rax@12
  int v13; // edi@17
  int v14; // [sp+0h] [bp-D8h]@1
  int v15; // [sp+4h] [bp-D4h]@1
  __int64 v16; // [sp+8h] [bp-D0h]@1
  __int64 v17; // [sp+10h] [bp-C8h]@1
  __int64 v18; // [sp+18h] [bp-C0h]@1
  __int64 v19; // [sp+20h] [bp-B8h]@1
  int v20; // [sp+30h] [bp-A8h]@9
  int v21; // [sp+38h] [bp-A0h]@9
  int v22; // [sp+60h] [bp-78h]@17
  int v23; // [sp+68h] [bp-70h]@17
  int v24; // [sp+90h] [bp-48h]@2

  v2 = *(_QWORD *)a1;
  v14 = 1;
  v15 = 3;
  v16 = v2;
  v17 = *(_QWORD *)(a1 + 8);
  v18 = *(_QWORD *)(a1 + 16);
  v3 = *(_QWORD *)(a1 + 24);
  v4 = *(_DWORD *)a2;
  v19 = v3;
  if ( (signed int)respond(v4, (__int64)&v14) < 0 )
    return 0xFFFFFFFFLL;
  v5 = protocol_request(*(_QWORD *)(a2 + 24), *(_QWORD *)(a2 + 32), (__int64)&v24, 0LL, 0LL);
  v6 = v5;
  if ( v5 < 0 )
  {
    v9 = *(_QWORD *)(a2 + 32);

    v11 = *(_BYTE **)(a2 + 24);
    if ( v9 )
    {


      v12 = 0LL;
      if ( *v11 == 10 )
```

As we can see, it shows all the differences in the pseudo-code in a 2 sides diff, like with the assembly diff. After you know how the 3 different ways to see differences work, you can choose your favourite or use all of the 3 for specific cases.

Saving and loading diffing results

Now that we have the diffing results we may want to store the results for checking them later on instead of re-launching the diffing process. This can be done by clicking on the “IDA View” tab (required because of IDA's behaviour) and then selecting from the main menu the option Edit → Plugins → Diaphora - Save Results.

In order to load the results, one need to execute the supplied script “diaphora_load.py” and then select the previously saved *.diaphora results file.

By the way: in case one closes a single tab it isn't required to relaunch the whole diffing process or reopen the *.diaphora stored results file, one can simply press the key “F3” or, alternatively, go to the menu Edit → Plugins → Diaphora – Show Results, it will display again any closed tab.

Ignoring small differences (Finding new functionalities)

Sometimes, you don't need to care about small changes when diffing 2 databases. For example, you maybe finding just the new features added to this or that program instead of finding bugs fixed in a product. We will continue with the previous binaries for this example. Go to the tab “Partial matches” and find the functions “respond” and “scan_reponse”:

Address	Name	Address 2	Name 2	Description
0040bd70	respond	0040bd70	respond	Bytes hash and names (ratio 0.934783)
0040ce10	scan_response	0040ce90	scan_response	Bytes hash and names (ratio 0.925000)
0040d200	exclude_response.isra.7	0040d440	exclude_response.isra.9	Bytes hash and names (ratio 0.955556)
0040d3c0	license_response.isra.9	0040d600	license_response.isra.11	Bytes hash and names (ratio 0.898089)
0040d6a0	protocol_response	0040d8e0	protocol_response	Bytes hash and names (ratio 0.936842)
0040e880	load_vps	0040eab0	load_vps	Bytes hash and names (ratio 0.825000)
0040ec80	setup_ini_callback	0040ed70	setup_ini_callback	Bytes hash and names (ratio 0.923077)
0040ecc0	engine_init	0040edb0	engine_init	Bytes hash and names (ratio 0.753846)

According to the ratios shown it seems these functions are almost equal with small changes. Let's see the differences in the function respond: right click on the respond function and select “Diff pseudo-code”:

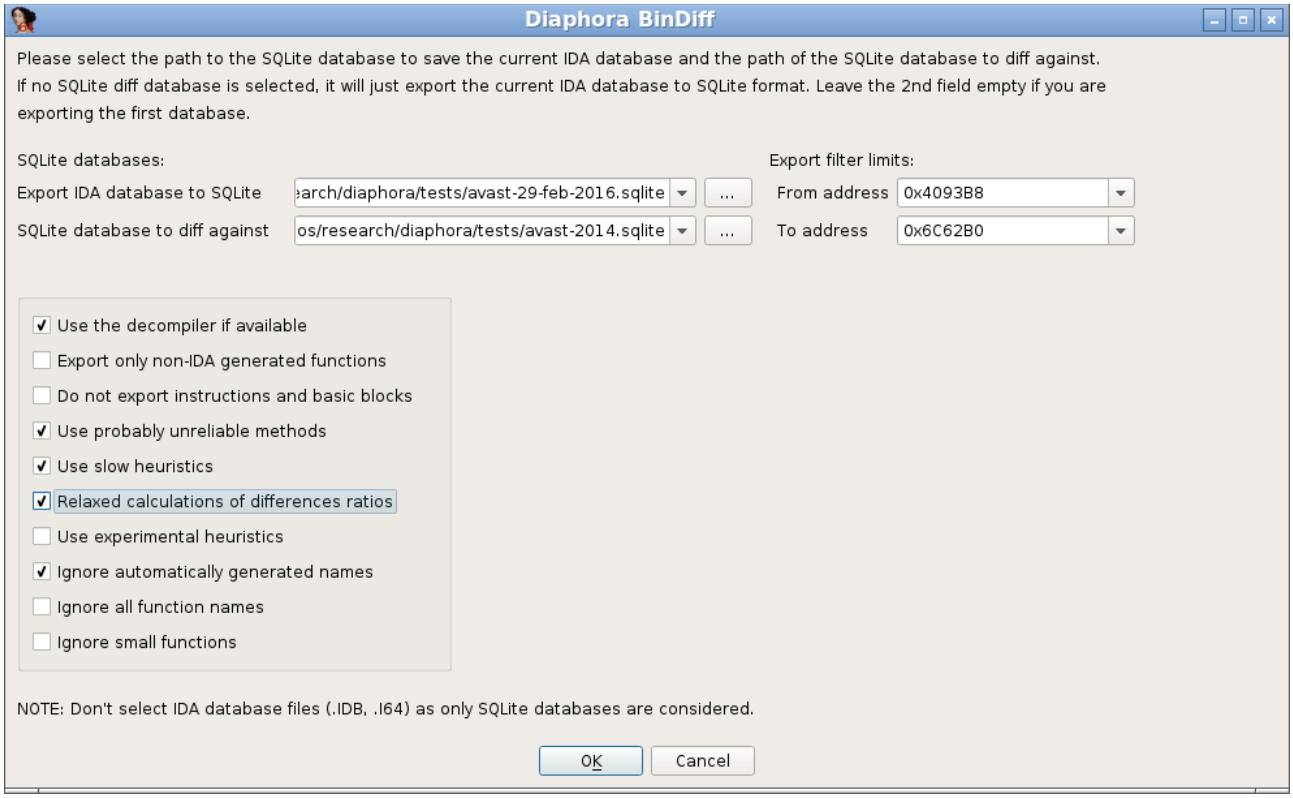
```
__int64 __fastcall respond(int fd, __int64 a2)
{
  __int64 v2; // rax@1
  size_t v3; // rbx@2
  int v4; // er13@3
  __int64 v5; // rbp@5
  __int64 v6; // r13@5
  ssize_t v7; // rax@8
  fd_set writefds; // [sp+10h] [bp-10B8h]@2
  char s[4152]; // [sp+90h] [bp-1038h]@1

  v2 = protocol_response(s, 0x1000uLL, a2);
  if ( v2 < 0 )
    return 0xFFFFFFFFLL;
  v3 = v2;
  memset(&writefds, 0, sizeof(writefds));
```

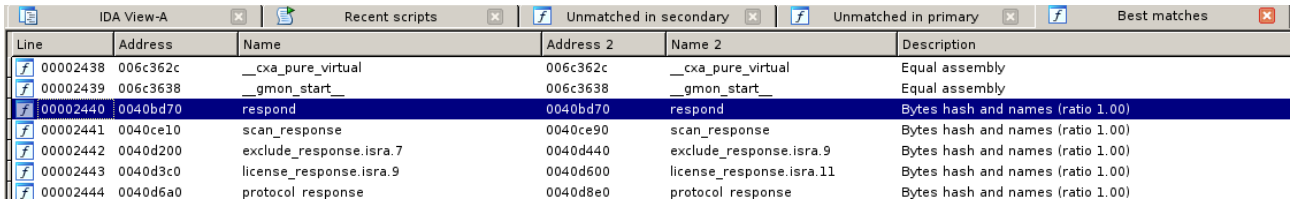
```
__int64 __fastcall respond(int fd, __int64 a2)
{
  __int64 v2; // rax@1
  size_t v3; // rbx@2
  int v4; // er13@3
  __int64 v5; // rbp@5
  __int64 v6; // r13@5
  ssize_t v7; // rax@8
  fd_set writefds; // [sp+10h] [bp-20B8h]@2
  char s[8248]; // [sp+90h] [bp-2038h]@1

  v2 = protocol_response(s, 0x2000uLL, a2);
  if ( v2 < 0 )
    return 0xFFFFFFFFLL;
  v3 = v2;
  memset(&writefds, 0, sizeof(writefds));
```

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 bellow:



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:



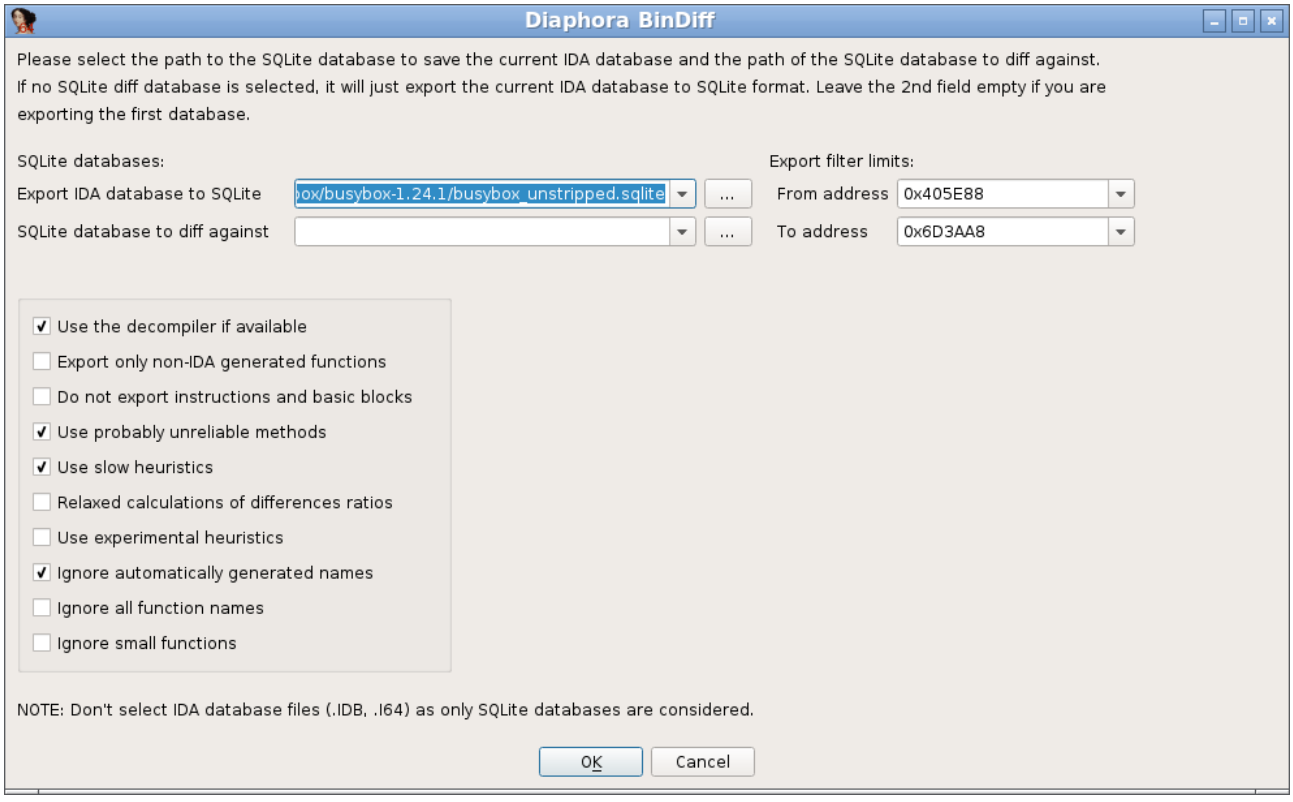
Line	Address	Name	Address 2	Name 2	Description
00002438	006c362c	__cxa_pure_virtual	006c362c	__cxa_pure_virtual	Equal assembly
00002439	006c3638	__gmon_start__	006c3638	__gmon_start__	Equal assembly
00002440	0040bd70	respond	0040bd70	respond	Bytes hash and names (ratio 1.00)
00002441	0040ce10	scan_response	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.

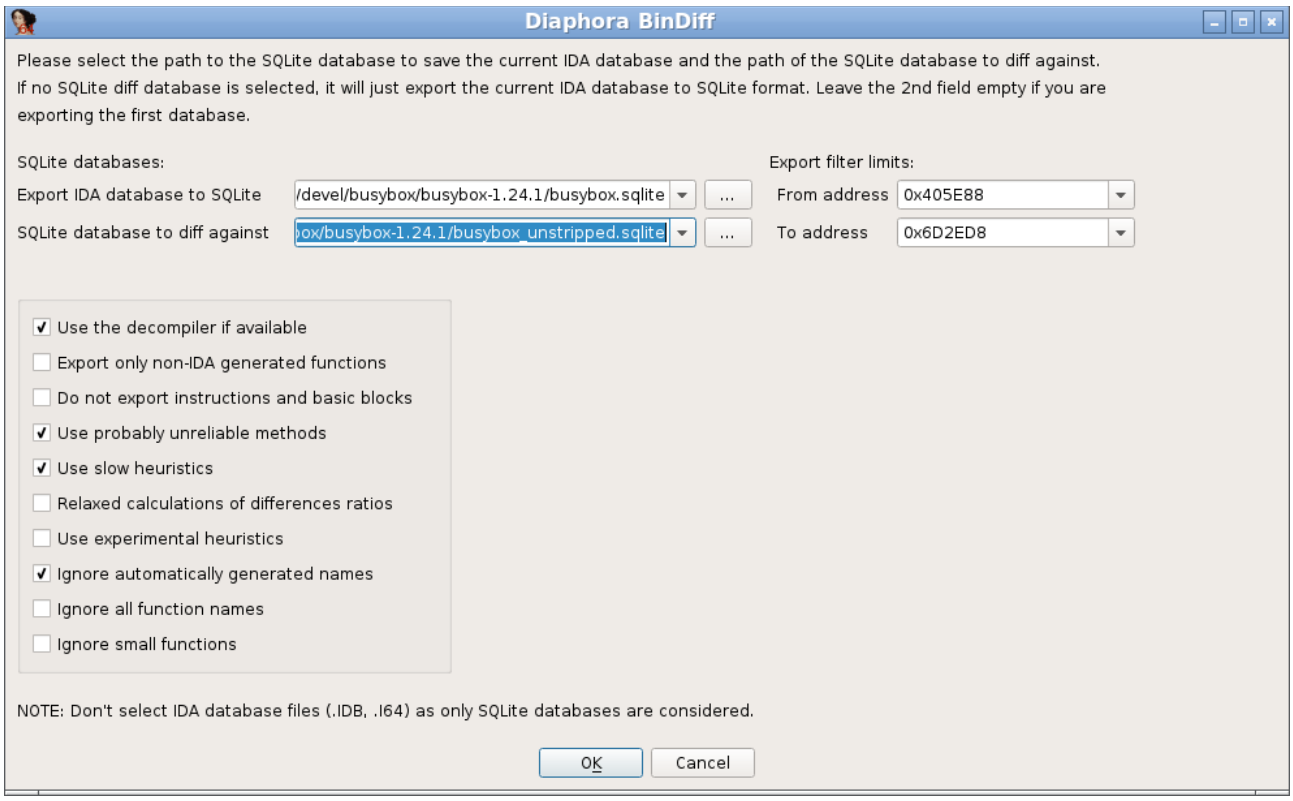
Porting symbols

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.

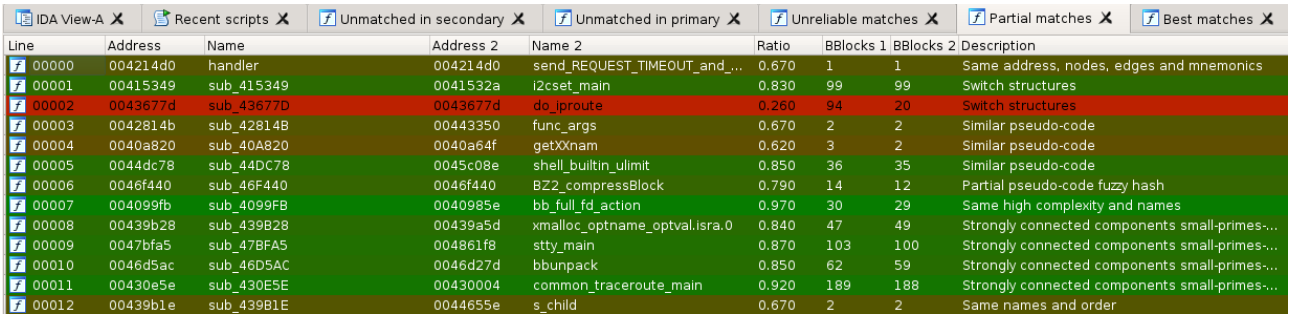
In the following example, we will import the symbols, structures, enumerations, comments, prototypes, etc... from one version full of symbols to another version with symbols stripped. We will use Busybox 1.21-1, compiled in Ubuntu Linux for x86_64. After downloading and compiling it, we will have 2 different 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:

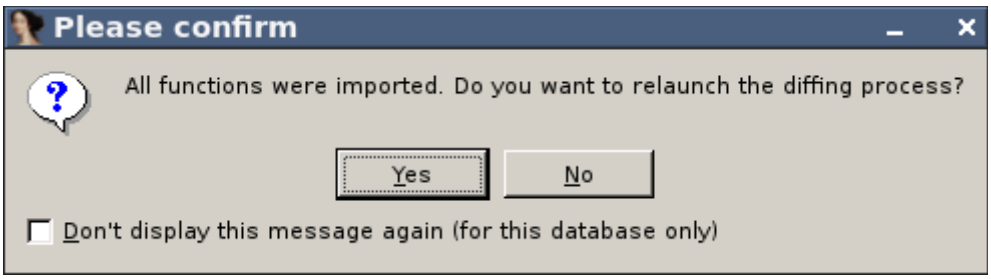


Press OK and wait until it finishes comparing both databases. After a while, it will show various tabs with all the unmatched functions in both databases, as well as the “Best”, “Partial” and “Unreliable” matches tabs.



Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00000	004214d0	handler	004214d0	send_REQUEST_TIMEOUT_and_...	0.670	1	1	Same address, nodes, edges and mnemonics
00001	00415349	sub_415349	0041532a	i2cset_main	0.830	99	99	Switch structures
00002	0043677d	sub_43677D	0043677d	do_iproute	0.260	94	20	Switch structures
00003	0042814b	sub_42814B	00443350	func_args	0.670	2	2	Similar pseudo-code
00004	0040a820	sub_40A820	0040a64f	getXXnam	0.620	3	2	Similar pseudo-code
00005	0044dc78	sub_44DC78	0045c08e	shell_builtin_ulimit	0.850	36	35	Similar pseudo-code
00006	0046f440	sub_46F440	0046f440	BZ2_compressBlock	0.790	14	12	Partial pseudo-code fuzzy hash
00007	004099fb	sub_4099FB	0040985e	bb_full_fd_action	0.970	30	29	Same high complexity and names
00008	00439b28	sub_439B28	00439a5d	xmalloc_optname_optval.isra.0	0.840	47	49	Strongly connected components small-primes-...
00009	0047bfa5	sub_47BFA5	004861f8	stty_main	0.870	103	100	Strongly connected components small-primes-...
00010	0046d5ac	sub_46D5AC	0046d27d	bbunpack	0.850	62	59	Strongly connected components small-primes-...
00011	00430e5e	sub_430E5E	00430004	common_traceroute_main	0.920	189	188	Strongly connected components small-primes-...
00012	00439b1e	sub_439B1E	0044655e	s_child	0.670	2	2	Same names and order

As we can see, Diaphora did a decent work matching 3112 functions labeled as “Best Matches” and 13 labeled as “Partial matches”, a total of 3125 functions out of 3630. 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 at function and instruction level, function prototypes, structures, enumerations, IDA's type libraries (TILs) and even rename global variables and labels with names. When it's done it will ask us if we want to relaunch again the diffing process:

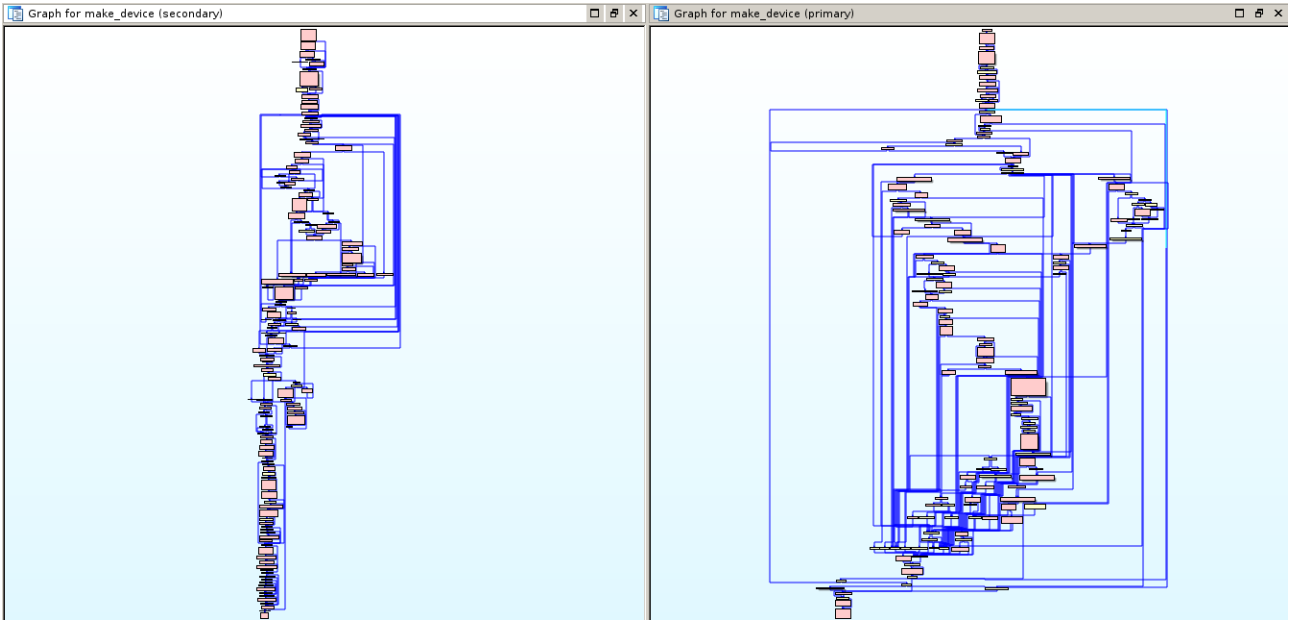


While Diaphora imports symbols, at the same time, it updates the database with the exported data from the primary database and, as so, with the new information it may be possible to match new functions not discovered before. In this case we will just say “NO” to this dialog.

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

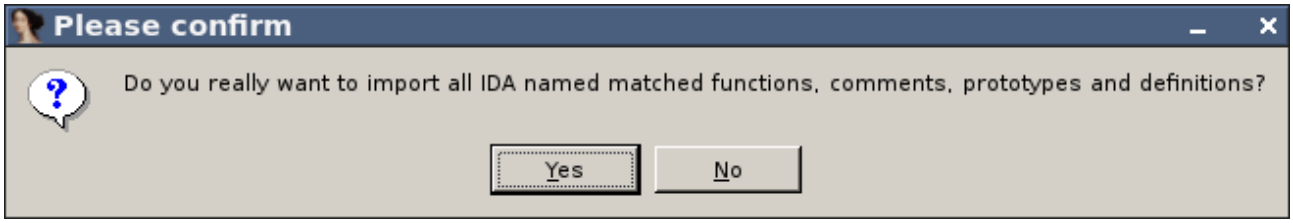
Line	Address	Name	Address 2	Name 2	Description
00000083	004372ca	sub_4372CA	004372ca	rtnl_rtntype_n2a	Bytes hash and names (ratio 0.962963)
00000038	00411dae	sub_411DAE	00411dae	bus_state_value	Bytes hash and names (ratio 0.964286)
00002295	0046c671	make_device	004654e3	make_device	Perfect match, same 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 a too low confidence ratio (i.e., 0.00). 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
- **Same RVA and hash**. The RVA (Relative Virtual Address) and the bytes hash is the same for both databases.
- **Same order and hash**. Both functions have the same bytes hash and were discovered by IDA at the very same position in the database (i.e., both functions are the 100th function in the database).
- **Function hash**. The calculated function hash is equal for both functions. The hash is calculated as the MD5 of the concatenation of all the instruction bytes in a function.
- **Bytes hash**. The calculated bytes hash is equal for both functions. The hash is calculated as the MD5 of the concatenation of all the instruction bytes in a function **but**, in opposite to **function hash,** it does so by stripping any byte that can be variable depending on the address of the instruction, like displacements or relative calls and jumps.
- **Bytes sum**. Both the size of the function in bytes and the summatory of all the bytes in the function are the same for both functions.

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

- **All or most attributes.** All the attributes of a function (basic blocks, primes values, hashes, etc...), or most of them are equal in both functions.
- **Switch structures**. The cases and values of all the switch statements in both functions are equal.
- **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.
- **Mnemonics small-primes-product**. The SPPs, calculated by assigning primes to mnemonics, in both functions are the same. It's sensible to changes in IDA: if the IDA's API GetInstructionList(), at some point, reorders the instructions, all exported Diaphora databases would not be comparable to new databases.

- **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 0.6. This is considered a slow heuristic.
- **Similar pseudo-code and names**. Same as before but also the true names used in both functions are equal.
- **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.
- **Topological sort hash**. Both the strongly connected components as well as the topological sort hash of the graph of both functions are the same.
- **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.
- **Strongly connected components**. The sets of strongly connected components of the graph are the same in both databases. This is considered a slow heuristic.
- **Strongly connected components small-primes-product**. The SPP calculated by assigning prime numbers to each strongly connected component for each set of strongly connected components in the graph is the same for both functions.
- **Loop count**. The number of loops (more than 1) in the function is the same for both databases and the confidence ratio is at least 0.49. This is considered a slow heuristic.
- **Same nodes, edges and strongly connected components**. The number of basic blocks, relationships between them and the sets of strongly connected components in the function graph are the same for both functions.

Unreliable matches

- **Strongly connected components.** The sets of strongly connected components are the same and, at least, there are 2 components. This is considered a slow heuristic.
- **Loop count**. The number of loops is the same for both functions. The comparison is made without checking the number of basic blocks. 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):

- **Call address sequence.** Check for similar or equal functions by sequentially looking over all the list of matches (both “Best” and “Partial”). The current implementation is far from being “good” but still works. However, as it isn't working properly all the time, it's considered an experimental heuristic.
- **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.
- **Same graph**. By looking to most attributes of the functions, the graph seems to be the same in both databases. It's a really slow heuristic that causes some false positives and for huge databases might cause the comparison to even crash because SQLite requires more than 3GB of memory (if the IDA process is a 32 bit, the default as of 2016).