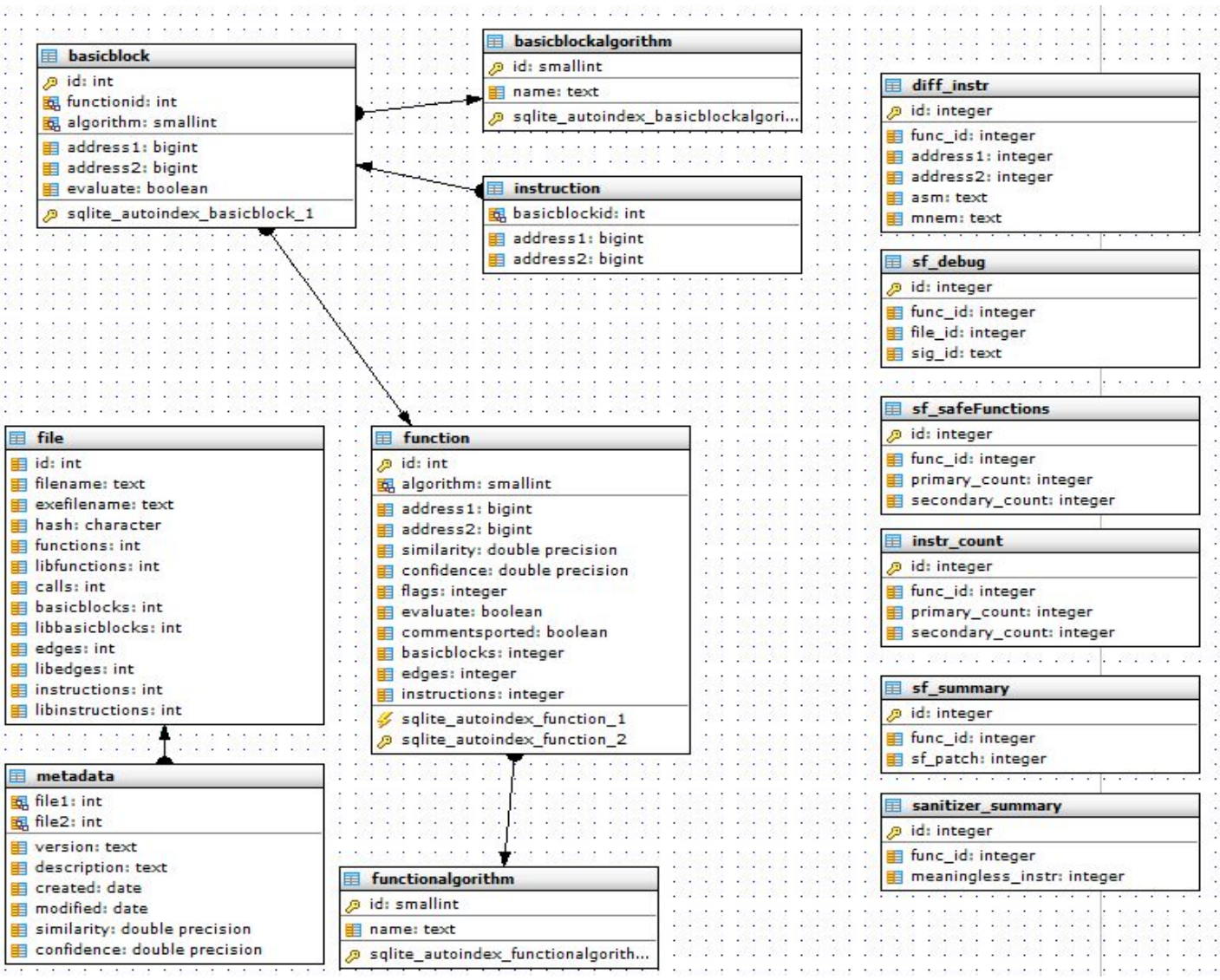
AutoDiff

1 Architecture

1.1 Database

BinDiff sqlite DB diagram with added tables by AutoDiff.



2 How does it work ?

2.1 Data Collection

Before any action is taken by AutoDiff each module needs to collect necessary information's for its further actions.

2.1.1 Sanitizer

Because BinDiff database does not provide information's about:

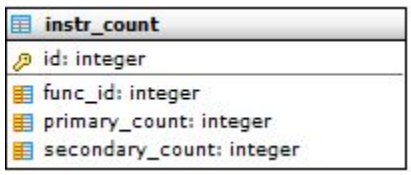
- instructions count
- address of different instruction(s) for particular function

Sanitizer collects this information's for own purposes.

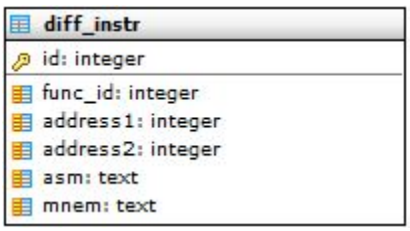
PROBLEM

Simple instructions counting for particular function can be problematic, but details related with this problem I will discuss in **Futures** section.

Related table:



Using available information's about matched instructions in BinDiff database is easy to find addresses of instructions added/removed to the function, doing simple subtraction on instructions' addresses set's. These information's are stored into the following table:

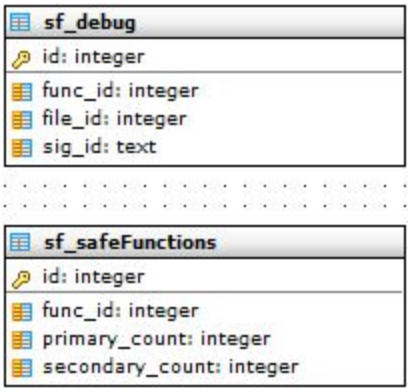


NOTE: Instructions count and addresses of different instructions are include in BinDiff results but in .BinExport files. Those are binary files with undocumented format, but at first glance is quiet easy to see some patterns, so reversing this format should be a problem. However I decided to collect this informations "manually".

2.1.2 SignificantFunctions

This module collects information's about usage of functions from IntSafe library.

Collected informations are aggregated in the following tables:



PROBLEM

Because of some functions specific layout simple walking over instruction set can be not enough to find all references to IntSafe functions calls. Problem is quiet simple to solve and solution for it gonna be discuss in **Futures** section.

2.2 Filtering algorithms

Keeping things in KISS philosophy PoC ver of AutoDiff and its module have implementation of simple filtering and rating. "Filtering algorithms" is a big word here, BUT still, simple data collection/rate/filter made by implemented modules brings interesting results and conclusions.

2.2.1 Sanitizer

Base on collected informations about differing set of instructions, Sanitizer can rate difference in functions as meaningless base on the following conditions (having primary function as reference point instructions can be added or removed from it):

- added instructions mnemonics set are subset of meaningless instruction set
- removed instructions mnemonics set are subset of meaningless instruction set
- sets of added and removed instructions are equal. *

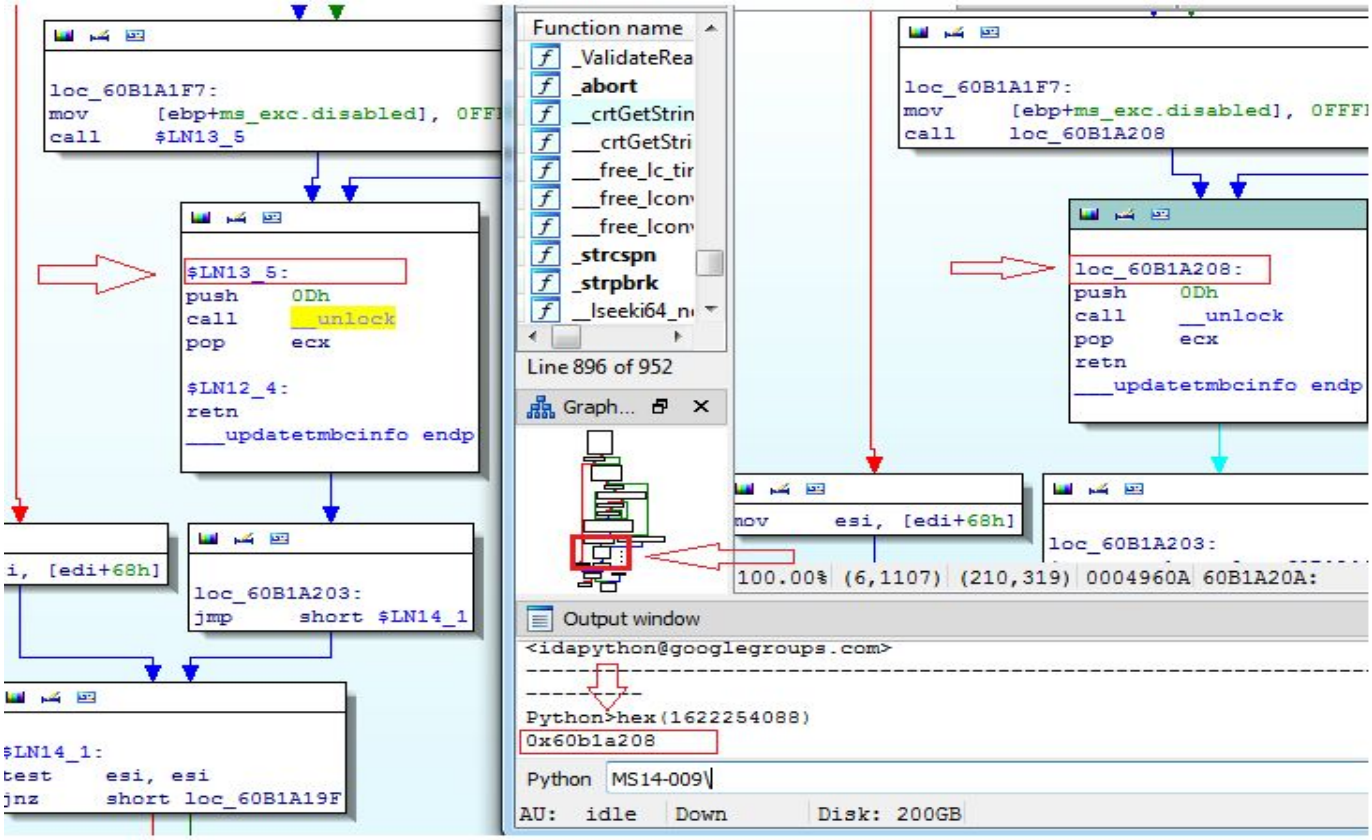
* That situation can appear in current implementation because sometimes BinDiff points on places which :

*All examples based on comparison of **MS14-009\NDP40-KB2898855-v2-x86\sos_dll_x86 vs MS14-009\NDP40-KB2835393-x86\sos_dll_x86***

- are not really functions, just place in the middle of bigger function

Example:

Function id : 857

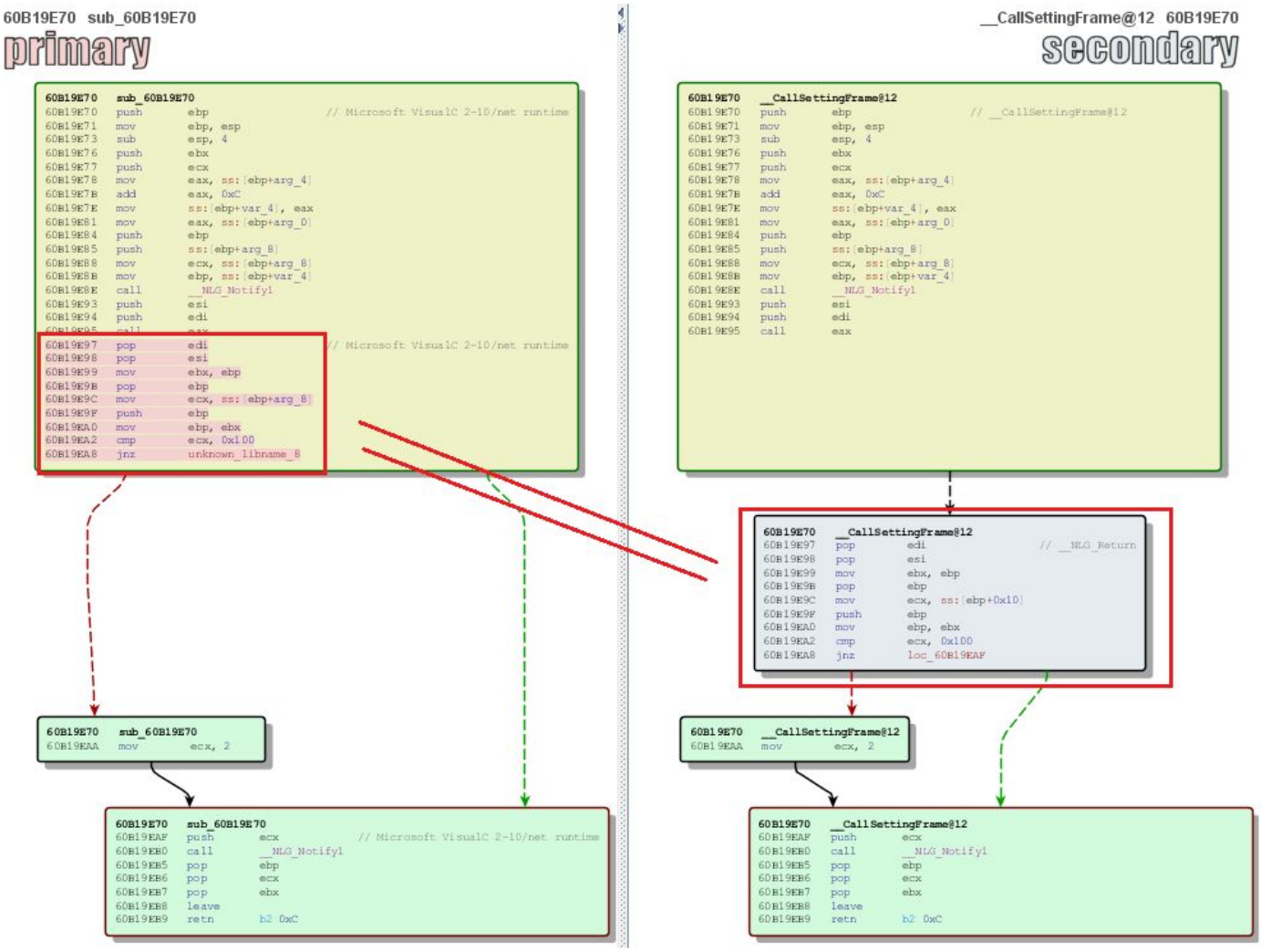


- because of IDA fault,BinDiff heuristic fault, lame implementation of instruction traversal in AutoDiff there was no possibility to reach instructions pointed by BinDiff as different and at the end, result sets are empty

- some BB was created/moved so added/removed instructions set is the same but entire function is seen by BinDiff as bit different

Example

Function id : 848

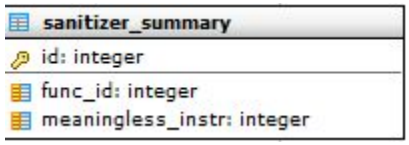


Present meaningless instructions set looks like this:

```
meaningLessInstr = set(['int','nop','ret','retn'])
```

As we can notice, eventually parameters for these instructions don't have meaning in context of our comparison, also so limited (for this moment) set does not bring FP but still is nice indicator for meaningless patches.
If any of this three conditions mentioned above is met, patch for particular functions pair is marked as meaningless.

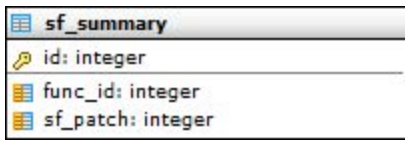
Summary is stored in this table:



2.2.2 SignificantFunctions

Using collected information's about amount of calls to functions from IntSafe in particular function, SF module can assesses whether secure patch had place. It's indicated by bigger number of calls to IntSafe functions in function from primary file than function in secondary file.

Results are stored to sf_summary table:



2.2.3 Summary of implemented heuristics (filtering algorith)

2.2.3.1 Meaning less instruction detection

Simple detection whether set of instructions which has been added/removed is a subset of "meaningless" instructions for us in this context. Defined set of meaningless instructions looks like this:

```
set(['int','nop','ret','retn'])
```

2.2.3.2 Comparing differing sets of mnemonics

Two sets of instructions mnemonics from added(patched file) and removed(unpatched file) are compared whether their sets are the same. This algorithm is basic version of algorithm which compares sets of instructions taking in count that registers could change where order and amount of instructions stay the same. Because of simplicity of this algo it can generate in some cases with big probability false positives.

2.2.3.3 Finding significant functions

Algorithm searches for functions defined in signature file (mostly functions from IntSafe lib) and count numbers of their calls in particular function. If number of class of "safe functions" in primary file in particular function is bigger than number of calls in secondary file, it means that there is a patch made with usage of IntSafe function. Notice that this algo is one of "positive" algorithms, it points us interesting function which definitely contains security related patch instead of pointing on function which should be take in count as interesting.

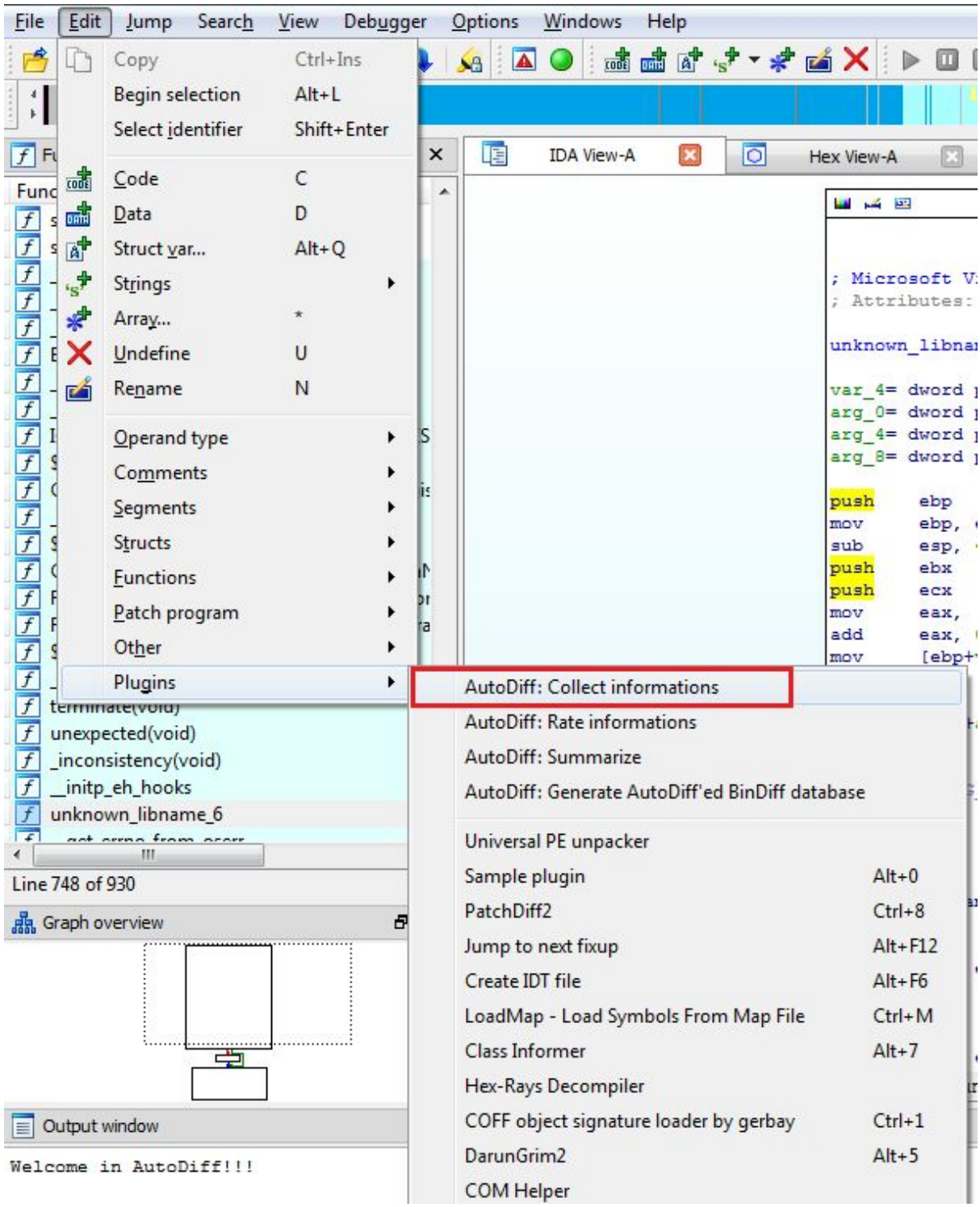
2.3 Example of usage

Let's we see AutoDiff with implemented so simple mechanism in it in action.
Before we load and execute AutoDiff in IDA we should have already ready BinDiff DB (.BinDiff file)

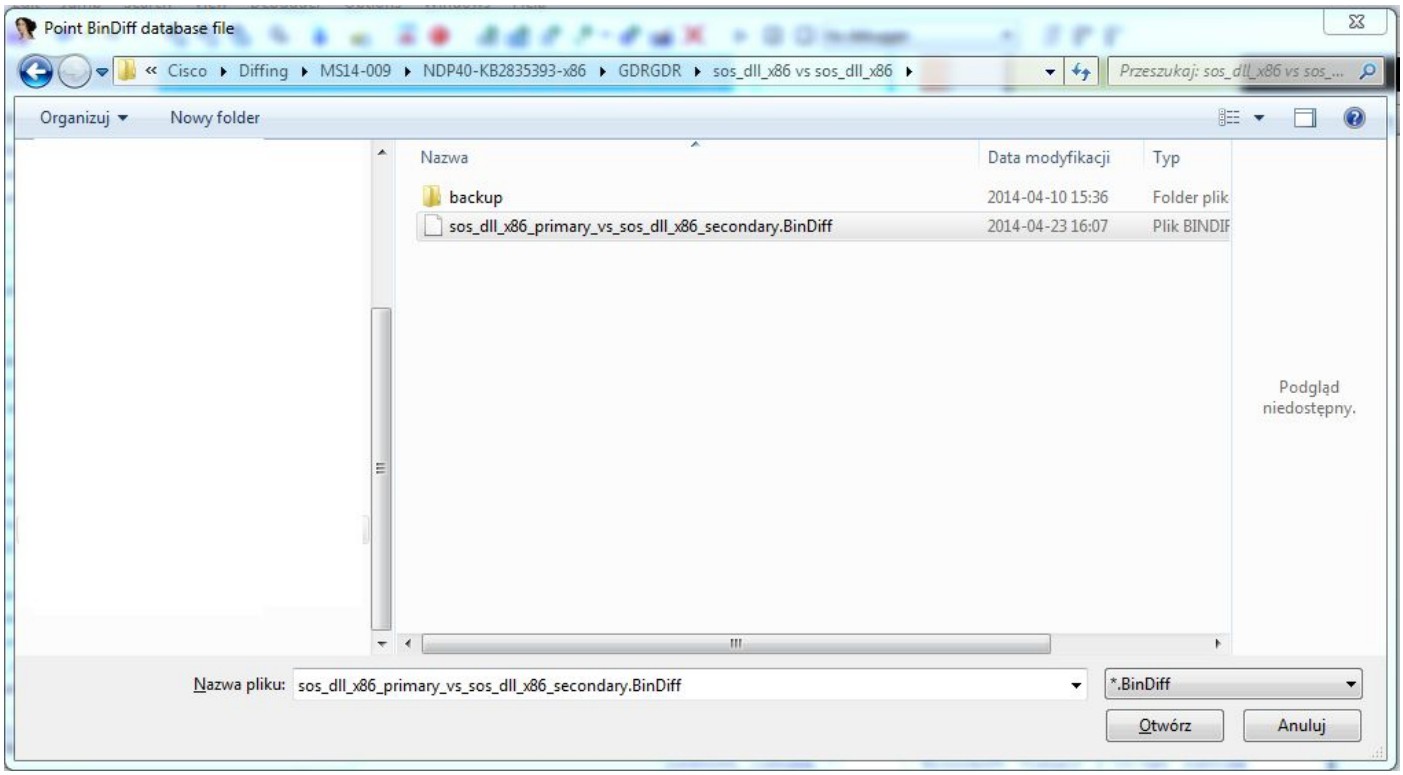
Assuming that we have BinDiff results, we can load AutoDiff into IDA.
Usage example will be present with mentioned sos.dll's used as sample data.

2.3.1 Data collection

First of all each module needs to collect proper informations from each IDB.
Let's collect informations from primary IDB.

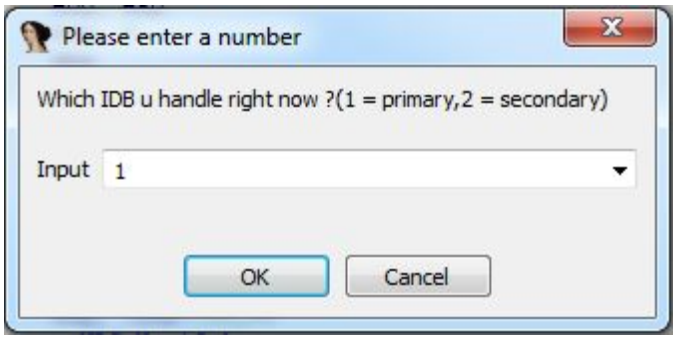


Point BinDiff db to analysis:



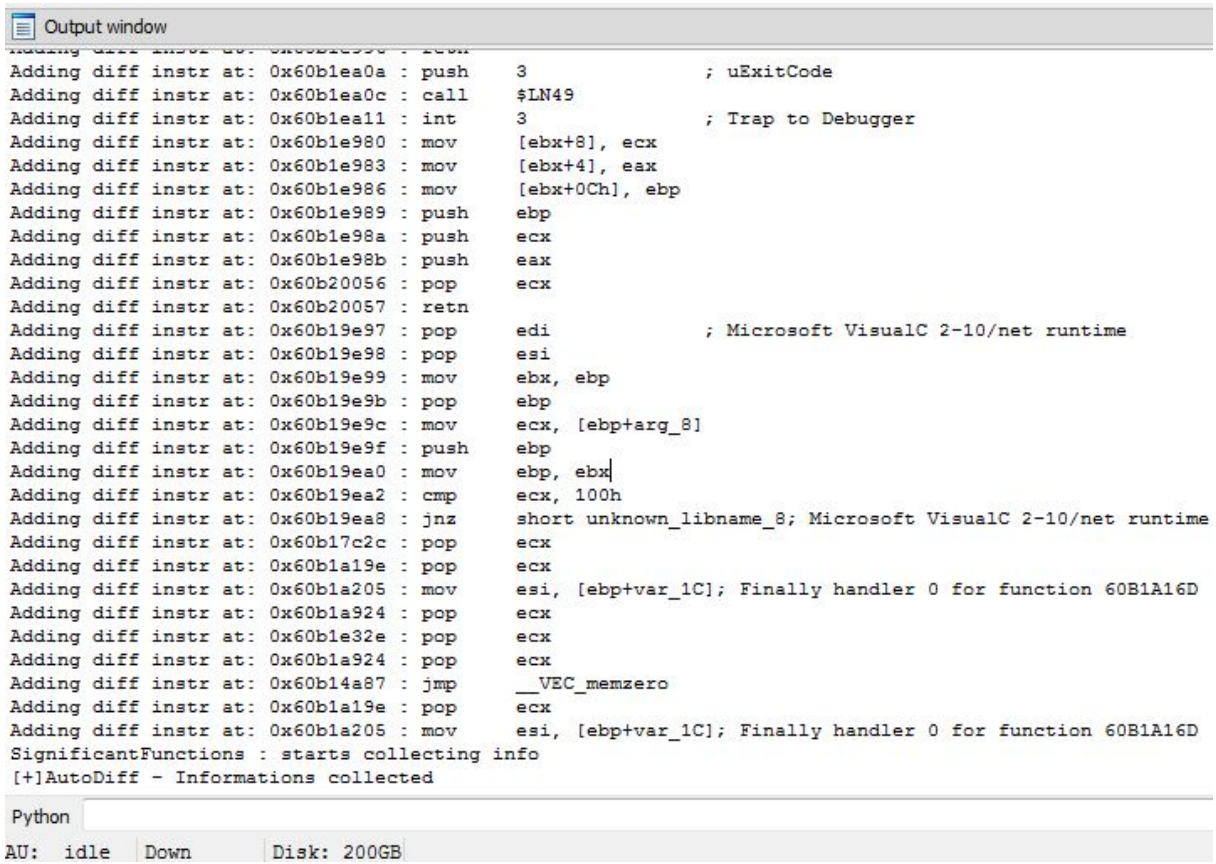
2.3.2

Next choose proper IDB file.



2.3.3

After it, modules should start collecting informations. Just wait for finish



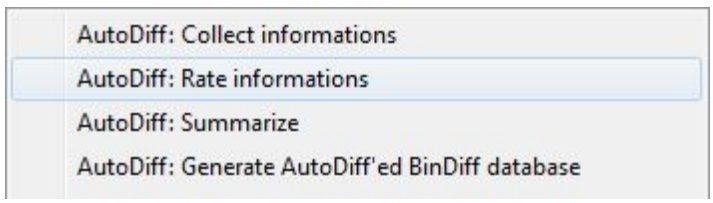
[UPDATE]

Launching separate IDA instance for second IDB it's not necessary. AutoDiff automatically spawn IDA instance for second IDB and collect proper informations.

Repeat the same steps for secondary IDB . Unfortunately for this moment u need to do this manually == execute another IDA instance load secondary IDB...

When You collected informations for both IDB's, AutoDiff is ready for Rating.

2.3.4 Rating



During rating u can notice informations about rated entities by modules in output window:

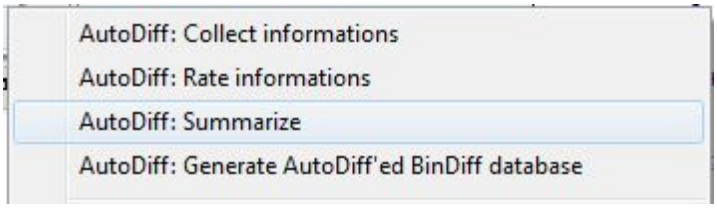
Output window

```
Meaningless instruction change detected: function id 867
Instruction added:
pop     ecx
pop     ecx
Instruction removed:
pop     ecx
pop     ecx
c
Meaningless instruction change detected: function id 653
Instruction added:
jmp     __VEC_memzero
Instruction removed:
jmp     __VEC_memzero
c
Meaningless instruction change detected: function id 856
Instruction added:
pop     ecx
mov     esi, [ebp+var_1C]; Finally handler 0 for function 60B1A16D
pop     ecx
mov     esi, [ebp+var_1C]; Finally handler 0 for function 60B1A16D
Instruction removed:
pop     ecx
mov     esi, [ebp+var_1C]; Finally handler 0 for function 60B1A16D
pop     ecx
mov     esi, [ebp+var_1C]; Finally handler 0 for function 60B1A16D
[+]AutoDiff - Functions have been rated
```

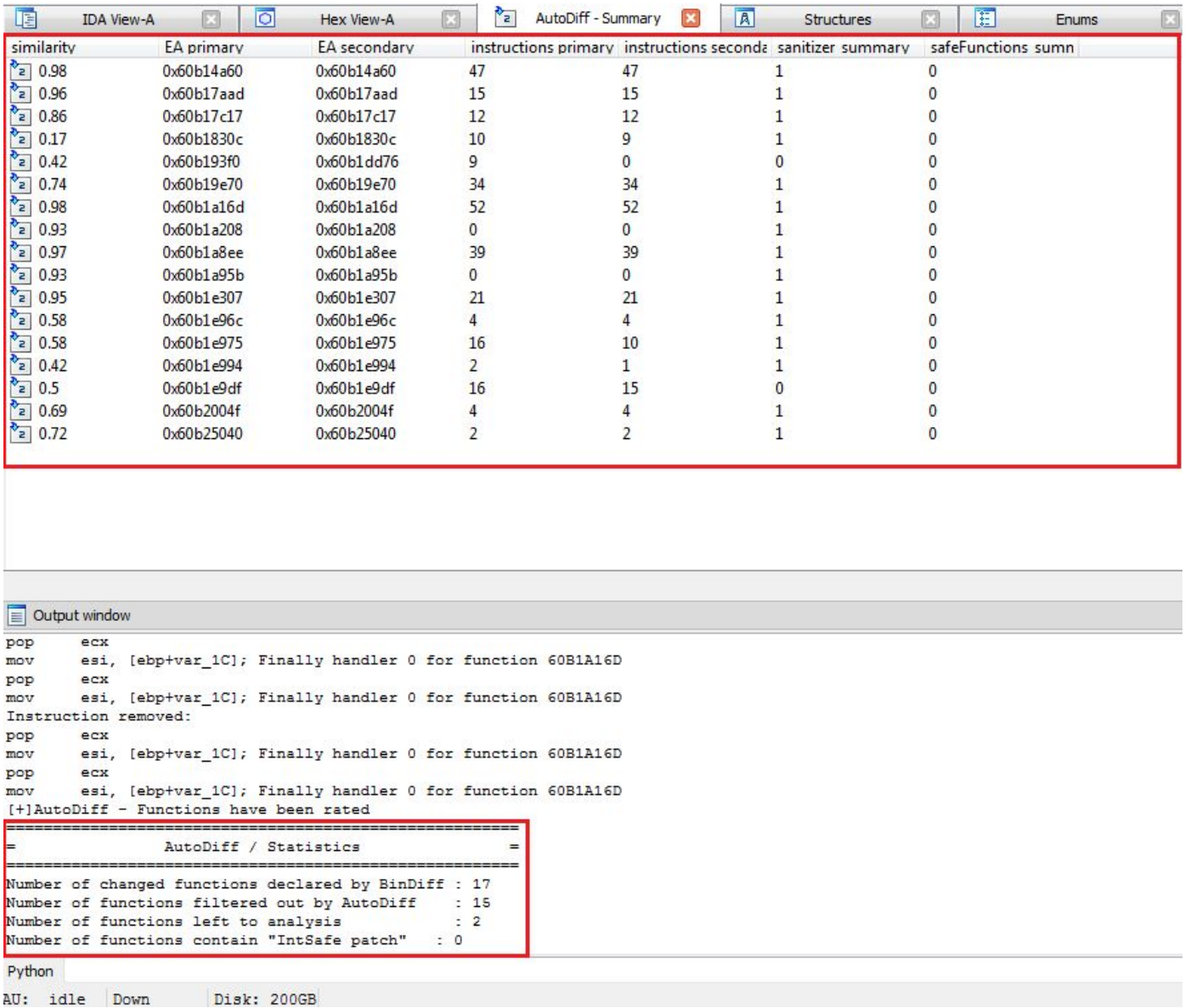
Python

Going further...

2.3.5 Summarize

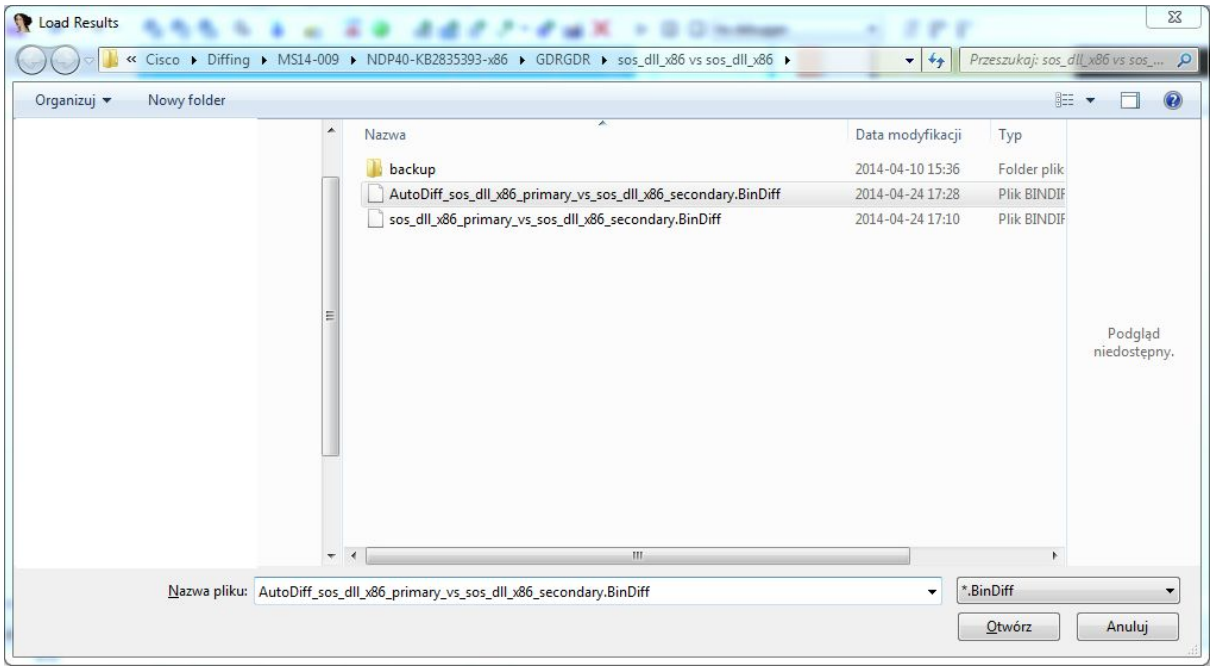


If everything went ok, we should see AutoDiff summarize window:



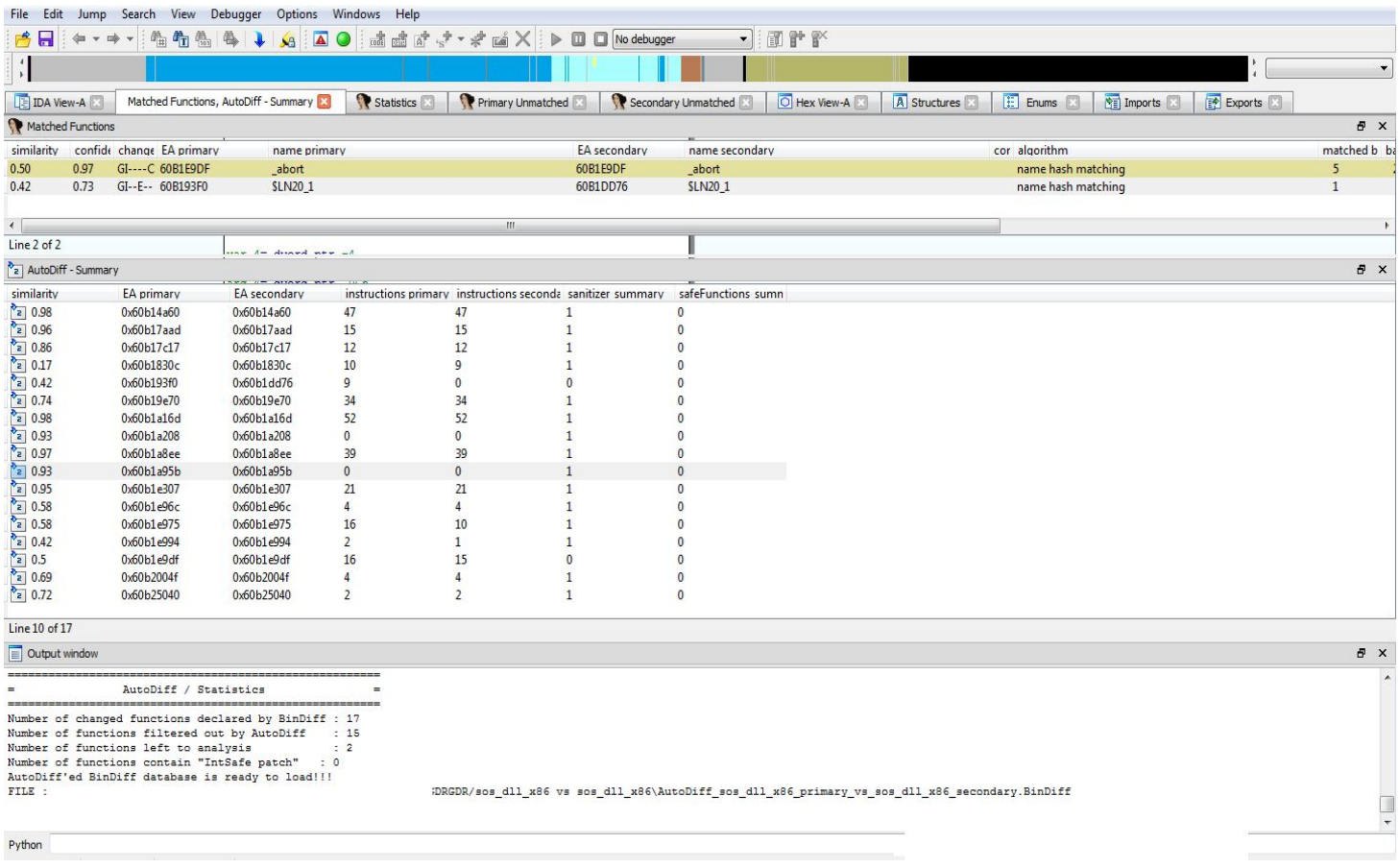
2.3.6 Generate AutoDiff'ed BinDiff database

Because BinDiff window does not give as possibility to simple filtering like "hide all functions which similarity is > =1.0" , AutoDiff using summary informations and constant condition that we are interested only in function marked as changed (similarity < 1.0) can generate modified BinDiff database for us. Just click "Generate AutoDiff'ed BinDiff database" option. Next, new BinDiff database is ready to load, it's file is in the same folder like oryginal BinDiff DB but file name starts with "**AutoDiff_**"



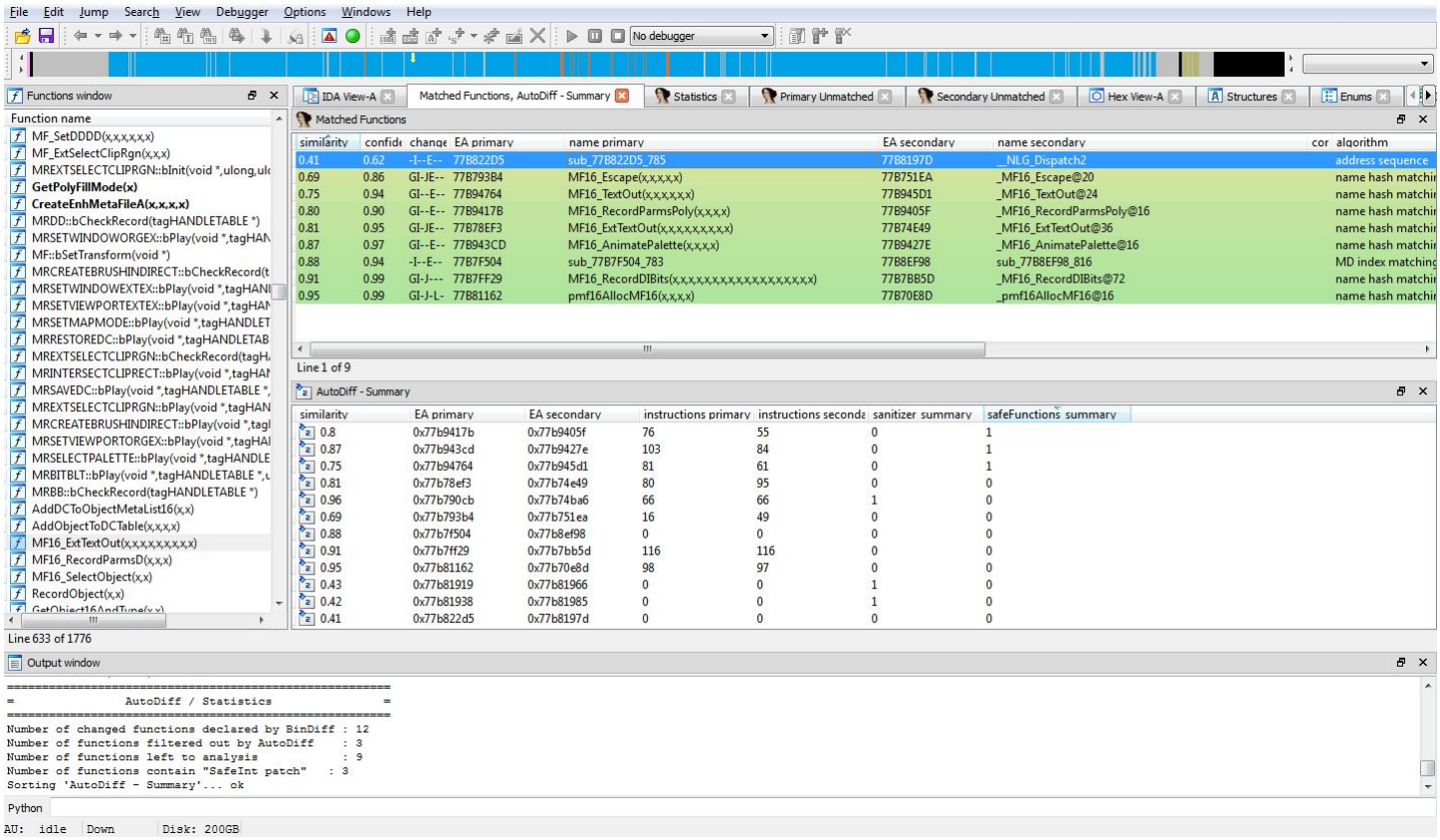
2.3.7

Setting together AutoDiff summary window and modified BinDiff results we see the following picture:



As we can see, this time BinDiff database contains only 2 matched functions to analysis.

Another results for GDI.dlls from BinDiff Challenge



Summarize 2.0

Using BinDiffFilter module u can achieve new way of summation.

Instead of creating new window with results, AutoDiff will modify on the fly (original bindiff database is not touched) BinDiff “Matched Functions” window and put there all necessary informations from AutoDiff modules.

More info in BinDiffFilter documentation:

https://docs.google.com/a/sourcefire.com/document/d/1jZxK26-t487_wTgIHgKxr6YTnK9H1sgZzcX8L-i63-8/edit?usp=sharing

Batch mode

You can execute AutoDiff in batch mode in the following way:

```
idaq.exe -A -S"AutoDiff.py -f 1 -b \"gdi32_primary_vs_gdi32_secondary.BinDiff\" -d \"X:\GDI\old\gdi32.idb\" -a" X:\GDI\patched\gdi32.idb
```

where parameters are:

```
self._optMethods = {
                "-f"      : self._setFileID,
                "-b"      : self._setBinDiffDB,
                "-d"      : self._setSecondIDB,
                "-a"      : self._batchMode,
                "-c"      : self._collectInformations,
                "-r"      : self._rate,
                "-s"      : self._getSummary
```

-a option is equal to - c -r -s

Order of parameters really matter!!!

If everything went ok, u can find log.txt file in GDI\old and GDI\patched directory plus summary.txt in GDI\patched with summary informations generated by AutoDiff.

Logging options

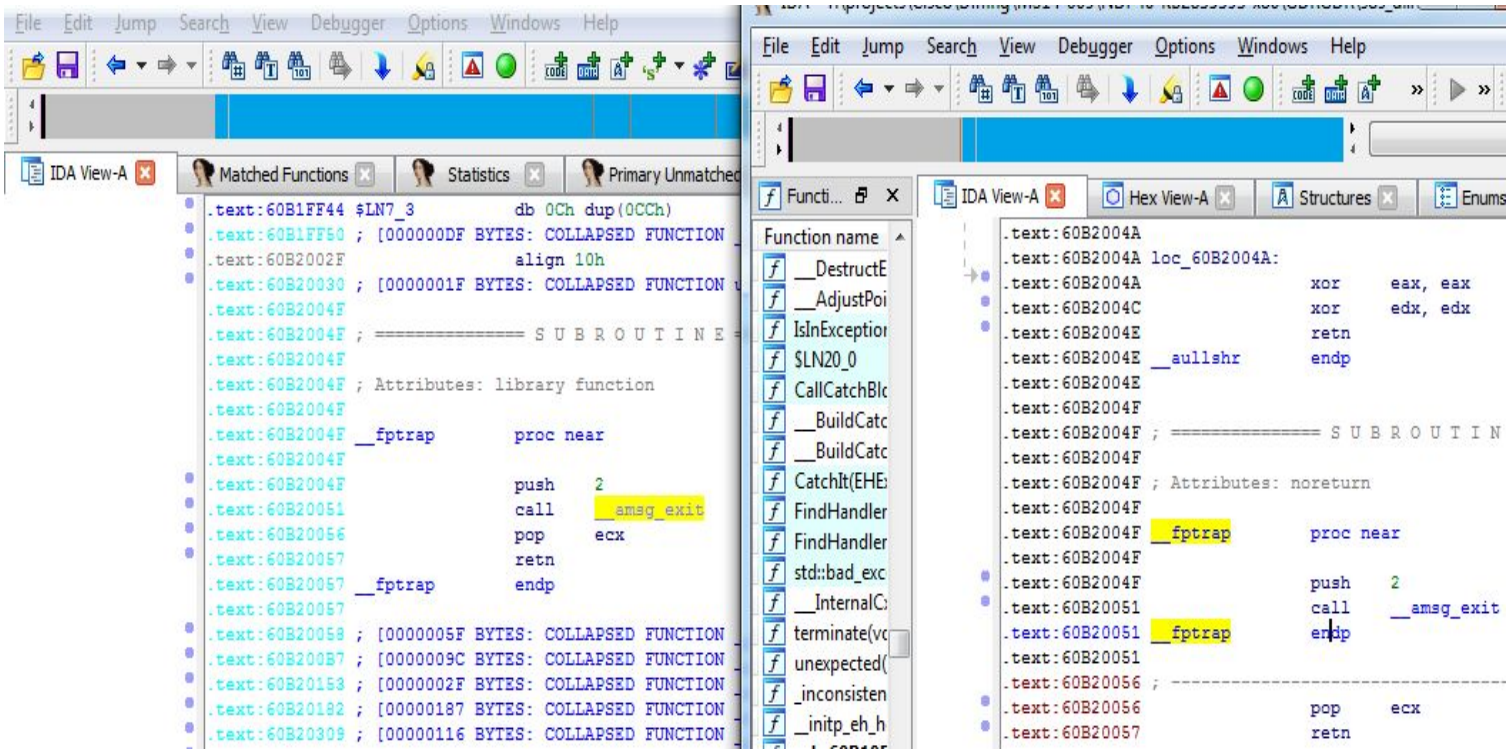
Logging to windows cmd console is possible via setting Logger.init(Logger.CMD).

Useful when u want to observe current progress in AutoDiffing because in normal scenario when AutoDiff logs info to IDA Python output window message are cached and present nearly at the end when script stop working. Logger with CMD option gives u possibility to observe all message from AutoDiff in runtime.

2.4 Problems

2.4.1 Function boundary

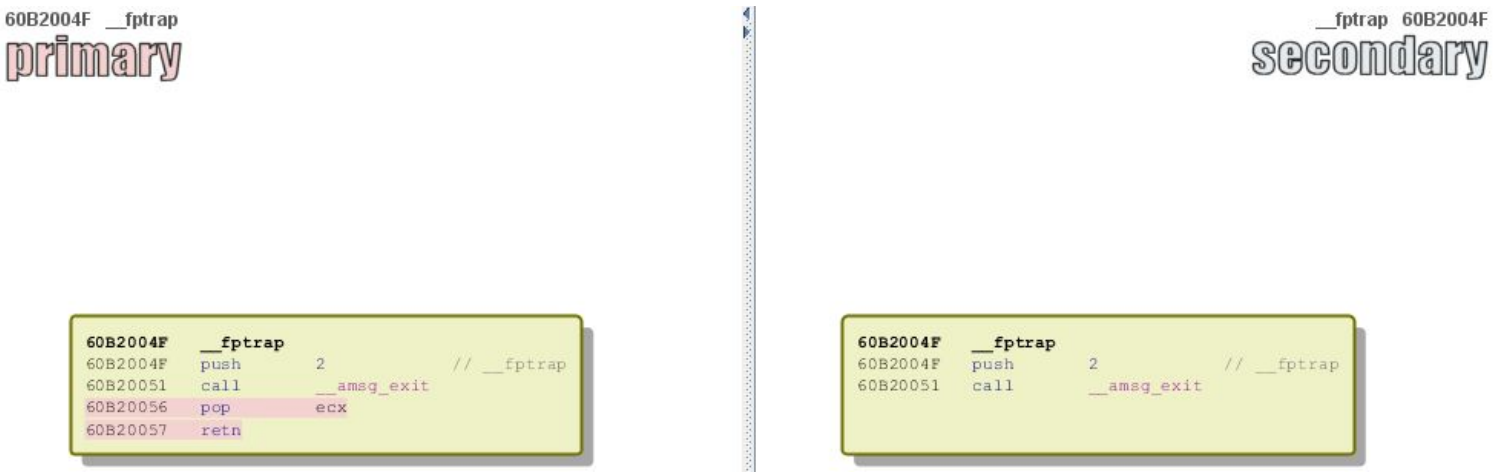
There is a lot of examples when IDA badly set function boundary which have latter a lot of bad implications for BinDiff results and also AutoDiff.
Let's we analyze one of that example:
Still we are using sos.dll



As we can see in primary file (on the left) , that __fptrap functions consist of 4 instructions but unfortunatelly IDA in secondary IDB shorten function only to 2 instructions and:
pop ecx
retn
instructions are out of boundary.

Implications for BinDiff:

His heuristic here fails because last instruction in __fptrap function from secondary IDB contains "exit" in name. My experience allow me to claim that BinDiff in some uses that kind of "signatures" to eventually move forward and look for real function end or just end in that place.
As summary of this mistakes we see that BinDiff shows us that this functions are different:



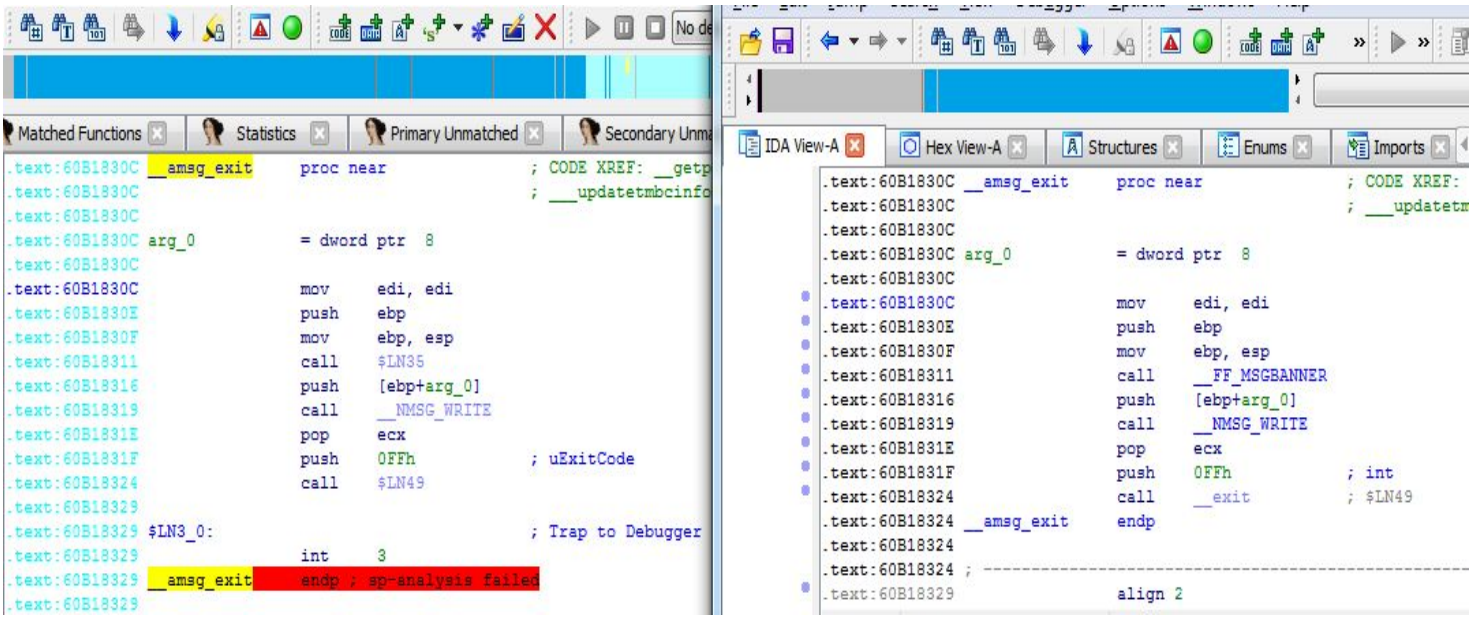
Implications for AutoDiff:

As we can see above unmatched instructions, are pop ecx and retn.
Sanitizer won't mark them as meaningless cos they are not in defined set.
for this example added_instr = set([pop,retn]) is NOT subset of meaninglessInstr

Ohh well we can add them to the set...with that approach we can add elements to the death and for a moment end with so big set which will return a lot of false positives.
Ofc a little bit better approach would be to create even pairs/ of instructions or defined subsets which should be treated as meaningless, but let's go back to the roots.

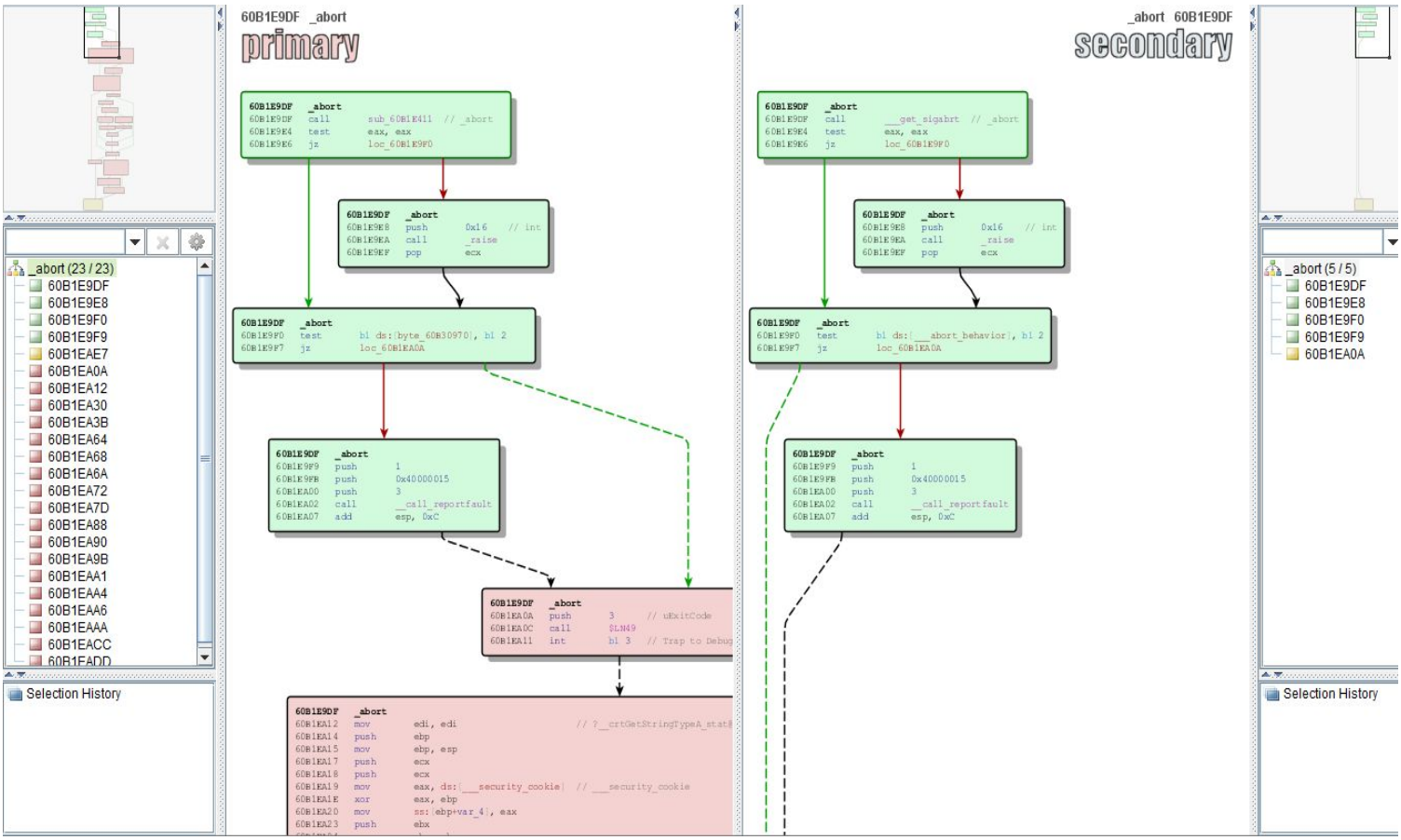
Solution
Solution here would be heuristically finding proper function ends before even starting AutoDiff collecting infor procedure or even after that intervention BinDiff should be run again on updated IDB's.

How to find function end ?
Scenario #1
Above case show example when IDA did not add proper instructions/BB to the function.
Finding functions which end on CALL instruction and checking whether another instruction is valuable (not e.g INT3, NOP,etc) would be a good approach. Generally algo should search here for in somehow defined function prologe (ret/retn instructions?).
Scenario #2
In this example IDA was too much greedy which later had bad implications on BinDiff.
Let's see what exactly happened:



As we can see in primary function IDA badly set end of function add unnecessary INT 3 instruction to functions body.
Implications for BinDiff:
Mentioned in #1 scenario BinDiff heuristic act here wrongly because the second to the last instruction is CALL to end but in primary IDB symbols changed and now call does not contain exit in name + last instruction UNFORTUNATELLY added by IDA is INT 3, so BinDiff similar to IDA acts

greedy and starts looking for proper functions end adding in the same way a lot of unrelated to this



function BB.

IMO solution here would be to check instructions at the end of functions and if they are different than standard one in func. epilog (ret/retn/call _exit), function should be shorten to the nearest meaning instr. In this example it gonna be call $LN49

2.4.2 Data treated as a Code

There are places where data (string, structure,...) is treated as a code which can lead to a lot of FP's during diffing. One of example we can observe in clr.dll
primary - MS14-009\NDP40-KB2898855-v2-x86\GDRGDR\
secondary - MS14-009\NDP40-KB2835393-x86\GDRGDR\

Code example:
String treated by IDA as code leading to unnecessary comparison

```
.text:79307E0C _g_RGCH_subscription:
.text:79307E0C              jnb     short near ptr byte_79307E83
.text:79307E0E              bound   esi, [ebx+63h]
.text:79307E11              jb      short loc_79307E7C
.text:79307E13              jo      short loc_79307E89
.text:79307E15              imul    ebp, [edi+6Eh], 90909000h
```

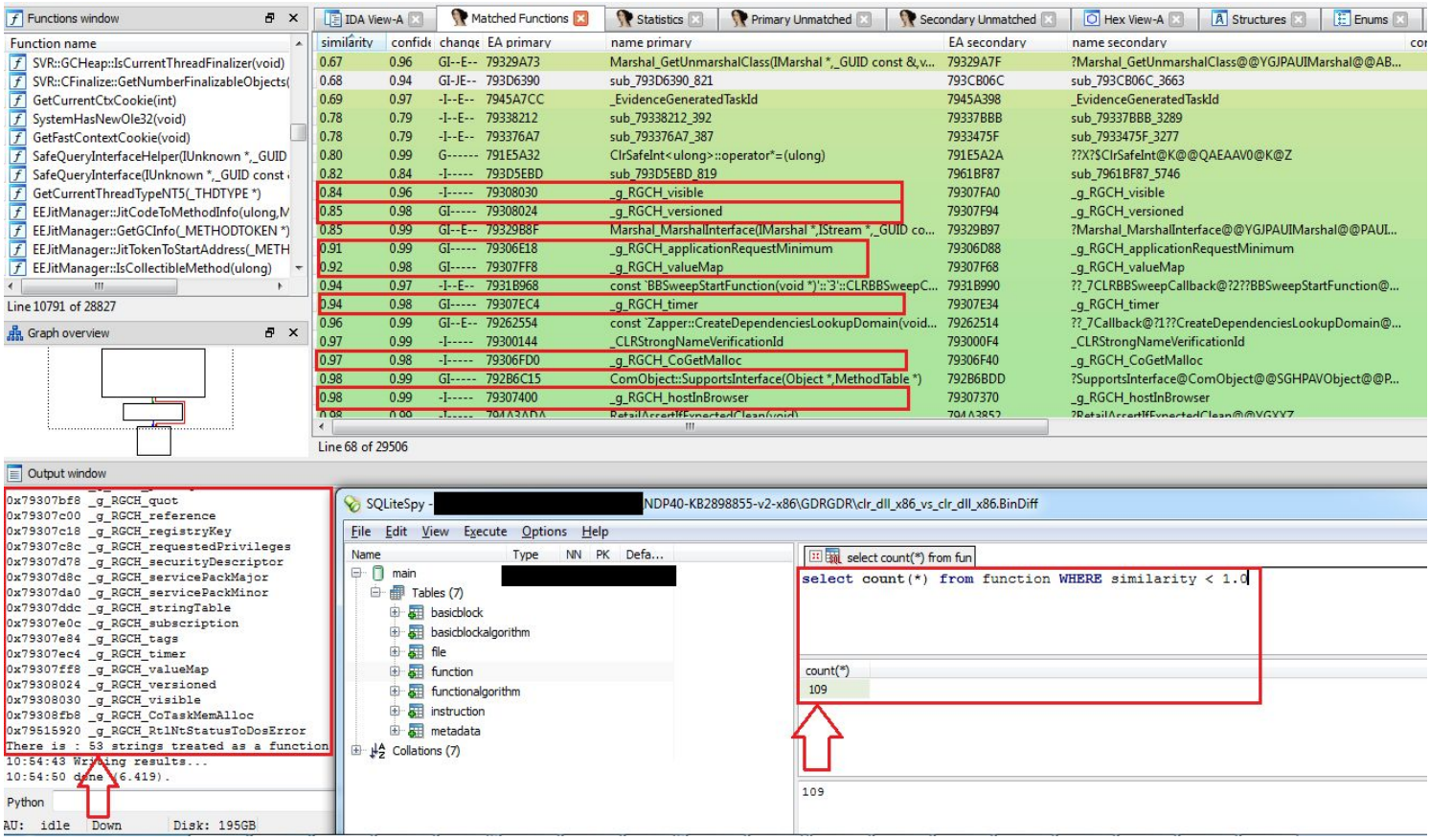
Experienced reverse quickly recognize that above code does not have too much sense and it's probably result of badly interpreted data, dissembler mistake, etc.

Converting above code into string data we achieve the following results:

String Data

```
.text:79307E0C _g_RGCH_subscription db 'subscription',0
.text:79307E19                 db  90h ; É
.text:79307E1A                 db  90h ; É           ; CODE XREF:
_g_RGCH_servicePackMinor:loc_79307DA2j
.text:79307E1B                 db  90h ; É
```

It turns out that in this diffing there is 53 that strings treated as a function where all matched functions with similarity < 1.0 109 !!! Which gives 49% of FP!!!



Potential solutions:

Entropy measurement - differently than regular code, strings should have bigger entropy or just quiet big rate if we collate :

set(instructions/opcodes)

---

number of instructions

Manual filtering - just filter & persistence removal

This solution will be discuss in **Futures** section.

3 Future plans

3.1 AutoDiff mass analysis

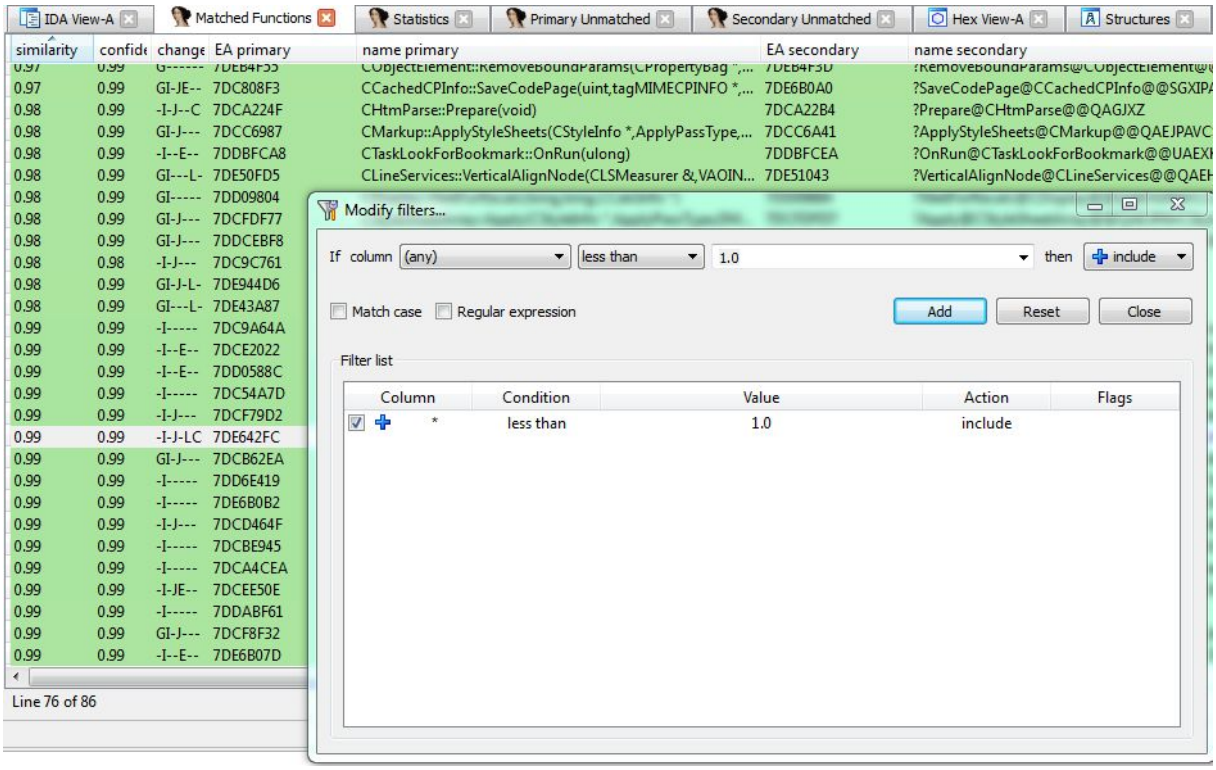
[+]Already implemented in AutoDiff

Currently April.2014 AutoDiff PoC ver need few manual move to achieve step when we see summary window. In future operator should be only obligated to execute AutoDiff "start" from IDA instance with primary IDB and wait for results.
Analysis of secondary IDB would be made automatically via spawning IDA instance/Loading AutoDiff script with proper parameters/ AutoDiff signaling end of analysis to primary instance/ Primary instance starts work on rating.
Dream scenario would be having full automation for dll sets.
AutoDiff --dir-primary C:\patched --dir-sceondary C:\unpatched
Question is whether there will be possibility to automate BinDiff???

3.2 Interface for m*anual filtering - just filter & persistence removal*

[+] BinDiffFilter - implements this functionality

Standard BinDiff summary window allows only on sorting by chosen column.
With big set of functions potentially to analysis there is very often need to filter out some positions and at the beginning focus only on those which can really matter.
Going back to example where strings where treated as a code, reverse after notice patter in name "_g_" of this elements, having filter window would be able easily hide these results.
Let I give u couple examples of potential filters:
- hide functions with small amount of instructions (2-5)
- hide functions where changed instructions set does not contain e.g: cmp or test
  (can be added in patch for boundary checking)
- show only functions containing calls to unsafe functions
(...)
and so on. Somebody can say that some of these filter can be done by just sorting via column or modifying BinDiff db, yep probably yes, but when set of matched functions is big and we wanna create complicated filter it won't be so easy. IMO definitely having that interface would simplify work in some cases.
[UPDATE]
It turns out that since IDA 6.2 this control which is used by BinDiff to results presentation has by default filtering functionality.



3.3 Correlation between modules' assessment

Having currently two modules there is no need for correlation. Especially in example of Sanitizer which indicates:

- "negative changes" - functions which should be dismissed/filtered out
- NOT interested for analyst functions

and

Significant Functions which indicates :

-function having sec patches

- INTERESTING places for analyst to look into

If both modules work correctly there should not be collision between them and also correlation is no needed because like you can see they summary should be treated in different way "negative and positive". But when there would be more modules which will indicates that some function is meaningless in context of analysis or function should receive additional attention because of some valuable artifacts ? How to accumulate those results ? Add some rank ? Do some results from different modules should exclude each other ? Maybe there would be need to create modules dependencies? e.g

Module A rate some func. as meaningless but only when results from module B is negative.

3.4 Custom grammar

Currently Sanitizer point functions to filter out based on 3 simple IF's. As PoC is fairly enough but continue that approach lead to nowhere. IMO there should be created custom grammar which will allow AutoDiff users to create database with signatures giving possibility for different actions.

Let we see potential example of signature in xml form:

```
<signature name="NOPs">
      <action>remove</action>
      <conditions>
            <operation type="sql_query">select mnem from diff_instr WHERE
func_id = {$CURRENT}<operation>
            <operator>eq<operator>
            <predicated_value>nop</predicated_value>
      <conditions>
</signature>
```

3.5 BAP/IDAOCaml -> SMT formulas

Combining power of BAP and IDAOcaml maybe there would be possibility to check functions on logical level. There is a lot of FP examples reported by BinDiff where simple filtering won't be effective because number of x86 instr mutations which lead to the same result is nearly infinite. Good example for function change which provide nothing new to its logic is:

Primary	**Secondary**
push 0	xor ebx,ebx
push 0	push ebx
call someFunc	push ebx
	call someFunc

3.6 AutoDiff modules improvments

3.6.1 Sanitizer

3.6.1.1 (Un)conditional jmp detection which destination is next instruction

Sometimes there is a lot of that kind of noise in functions added or removed which exactly brings nothing for function only FB in BinDiff summary. Detecting that kind of instr should be problematic.

3.6.1.2 Mismatched function detection

[Ryan's description]

BinDiff will match most functions properly, but there are often a handful of functions that get mismatched between two versions of a binary. This is probably because they are found to be good fits when matching algorithms with much higher precedence than matching by instruction count are applied.

matched basicb	basicblocks prir	basicblocks secor	matched instructior	instructions primary	instructions seconda	matched edges	edges primary	edges seco
1	1	1	7	7	14	0	0	0
1	1	1	7	8	14	0	0	0
1	1	1	11	14	21	0	0	0
13	28	14	64	288	109	8	45	22
13	14	28	64	109	288	8	22	45
8	9	12	37	126	138	6	12	19
2	4	8	7	36	68	0	4	11
3	3	16	16	19	184	1	3	22
8	16	9	58	184	126	6	22	12
3	3	35	4	14	243	1	3	49
4	8	7	9	68	69	2	11	8
1	1	3	4	14	19	0	0	3
1	1	3	1	6	14	0	0	3
15	25	52	35	169	413	6	39	80
1	4	1	1	18	6	0	4	0
1	1	4	5	34	36	0	0	4
15	52	25	35	413	169	6	80	39
1	8	1	6	39	7	0	12	0
1	1	9	6	28	54	0	0	12
1	9	1	9	88	28	0	12	0
1	1	9	7	14	88	0	0	12
1	9	1	5	54	34	0	12	0
4	35	4	4	243	18	3	49	4
1	12	1	6	138	8	0	19	0
1	18	1	9	278	13	0	25	0
6	6	57	16	34	373	4	6	83
6	57	6	16	373	34	4	83	6
2	4	3	13	81	19	0	4	3
3	3	18	10	19	278	1	3	25
4	7	4	13	69	81	3	8	4

Example:

Once these functions have been audited by hand and manually matched, they are recognized as identical by the differ. These mismatches create additional noise in the pool of changed functions. For a small number of changed functions, where there is a limited dispersion of unique instruction counts, these are easy to sort out, but for larger dlls, it may be too time consuming and I've found these mismatches contribute to the majority of the changed functions observed in the Microsoft dlls I've looked at.

solution

I've contacted the Zynamics guys and it looks like the precedence is predetermined, so the configuration file will only allow you to enable/disable specific algorithms. Unfortunately, it looks like the mismatches are an edge case and the algorithms responsible produce suitable matches in most other situations. The solution I'd propose is doing a secondary pass on all the changed functions once BinDiff completes and grouping them into sets with identical numbers of instructions. We can then do a direct binary comparison between only the functions in each set to determine which are identical.

[Ryan's description]

3.6.2 SignificantFunction

3.6.2.1 Searching for dangerous functions calls and their luck/replacement in patched version

Currently only usage of IntSafe function is searched but searching also for dangerous functions and manually security checks made related with them can be another portion of information to filter valuable function from garbage's.

3.6.2.2 Manual security patch detection

If BinDiff detects changes in function which contains e.g strcpy but patch does not provide usage of its safe equivalent, still there can appear manual patch like:

```
if ( strlen(buf2) < MAX_PATH )
        strcpy(buf1,buf2);
```

similar situation for integer overflow

```
  if ( a12 + v18 < -1 - 2 * (unsigned int)(unsigned __int16)v19 )
  {
    v20 = LocalAlloc(0, v18 + 2 * (v19 + ((a12 + 1) >> 1)));
    v21 = v20;
```

3.6.2.3 Way of marking functions considered as valuable

Currently functions pointed by Sanitizer can be removed from analysis but functions indicated by SignificantFunction module are not in any way represented in BinDiff window results. I did not find easy way to modify BinDiff window to add e.g another color for functions containing IntSafe security patches or even column with value indicating that this function is mark by SF module.

3.6.3 Sanitizer & SignificantFunction

3.6.3.1 Walking over instruction set belongs to function

Problem mentioned during instruction counting and finding all references to safe function usage. Simple walking over instructions via

```
for head in Heads(f.start,f.end):
```

For some functions where BB address belonging to function does not lie in range (f.start,f.end) won't be touch for simple enumeration presented above. It's simple case to travel over graph but mentioned here cos it's not implemented in current PoC ver. of AutoDiff.
This simple change will bring again better results for both modules and it's luck leaded in couple examples to FP assessment.