Diaphora, reviving binary diffing

Joxean Koret

R2Con 2017

What is Diaphora?

- Diaphora is an open source (GPL) tool for performing program diffing.
  - Patch diffing; finding how a vulnerability was fixed in a closed source product.
  - Porting symbols between IDA databases.
  - Finding new functionality in a closed source product.
  - Plagiarism detection.
  - Malware indexation using Radare2, IDA, ...
  - etc...

Why?

- There were many reasons to decide to write one more program diffing tool:
  - The main tool I used to use (Zynamics BinDiff) was not updated for long times and lacks many features I wanted to have.
  - The other Open Source approaches were not as good as I want or I considered them too hard to modify to my needs.
  - I always prefer to write my own tools when I don't like existing ones.

Why? A bit of history

- I typically work doing reverse engineering. Many times, I need to port my work from an old version to a new version of my target.
- For this purpose I used to work with BinDiff.
- However, I'm a heavy user of:
  - Structures.
  - Enumerations.
  - The decompiler after I have the previous elements.
- Structures and enumerations cannot be ported with BinDiff. Ouch.
- ...

Why? A bit of history

- When porting symbols from a database to another, I found many-many cases where functions very similar in the Hex-Rays pseudo-code (decompiled code) and different in the assembly were missed by Zynamics BinDiff.
  - Specially, when comparing code for multiple architectures (ARM/x86/x86_64).
- I used to write many project specific ugly IDA Python scripts (hacks) to import and apply structures, enumerations and to match functions according to the pseudo-code.
- And, one day, I got tired and decided to write my own code.
  - Writing project specific ugly codes is **wrong**.
  - Writing generic good code for such tasks is better.
- The first version of the program diffing tool was written in a weekend. It worked better, for my testing projects, than the other existing solutions.

Internals

Internals

- Internally, Diaphora does the following:
  - Exports the whole Radare or IDA databases to my own SQLite format.
  - Compares all the artefacts from the 2 exported databases by simply issuing SQL queries.
  - Shows the matches (only in IDA, for now).
    - There are plans to try to integrate with Iaito for Radare2, when a GUI API becomes available.
- Basically, this is all it does.
- However, it's more complex than that.

Internals: exporting

- From each database the following elements are exported:
  - Functions and all their attributes (flags, prototype, relative addresses, etc...).
  - Basic blocks and their attributes.
  - Instructions and their attributes.
  - The pseudo-code of each function, if the decompiler is available (both IDA and Radare2).
  - The Abstract Syntax Tree (AST) of the pseudo-code (IDA specific).
- ...

Internals: exporting

- For each function or basic block, also, some information is calculated:
  - A hash based on the bytes (the non-changing bytes of each instruction).
  - The cyclomatic complexity (CC).
  - The strongly connected components of the flow graph, the topological sort of the flow graph, the MD Index of the flow graph, etc...
  - The small-primes-product (SPP) for each strongly connected component.
  - A set of 3 fuzzy hashes of the raw pseudo-code text.
  - A fuzzy hash based on the Abstract Syntax Tree (AST) of the pseudo-code (SPP of each AST's instructions).
      - IDA only.
  - The number of loops.
  - The switch structures.
      - IDA only.

Internals: exporting

- For each database, the following information is calculated:
  - A fuzzy call graph hash based on the SPP of each function's CC.
  - All primes calculated for the call graph.
  - The SPP hash of the call graph will be used to determine how related 2 binaries are, at diffing time.

Internals: comparing

- When the 2 databases are exported from IDA or Radare2 to the SQLite format understood by Diaphora, SQL queries are launched to find function matches.
- The SQL queries launched try to match as much arguments as possible using different heuristics.
  - First, the most robust heuristics are launched.
  - Then, the less robust ones.
- For each match, a similarity ratio is calculated and assigned.
- The results are then labelled as either “Best”, “Partial”, or “Unreliable”.
- In IDA, one can see the results.

Internals: comparing

- Diaphora has 47 heuristics implemented for finding matches. Examples:
  - Equal/similar assembly or pseudo-code (both raw and cleaned-up).
  - Bytes hash and names.
  - Same (function) name.
  - All or most attributes from the function.
  - Same address, nodes, edges and primes (re-ordered instructions).
  - Import names hash.
  - Pseudo-code fuzzy hashes and pseudo-code fuzzy AST hash.
  - Topological sort hash.
  - Same rare MD-Index.
  - Strongly connected components small-primes-product.
  - Same graph.
  - ...and many more.

Internals: comparing

- For each match that the SQL queries give, a similarity ratio is calculated using 4 methods:
  - Python's SequenceMatcher().quick_ratio for the cleaned-up assembly.
    - Basically, it gives out a similarity ratio of 2 strings.
  - Same as before, but with the cleaned-up pseudo-code.
  - A difference ratio based on the SPP of the AST.
    - Primes in both sets are ignored and a ratio is calculated based on the primes that are only in one of the 2 sets.
  - ...

Internals: comparing

- Another method used to calculate a similarity ratio is by using the flow graph's MD Index:
  - If the MD-Index is the same, we consider the flow graph to be the same.
    - i.e., we ignore added or re-ordered instructions not affecting the graph.
  - If they are different, we calculate how different they look like, calculate a ratio, and use this ratio with the previous other calculated ratio.
  - The final ratio is useful to compare “apples” to “oranges”.
    - i.e.: x86 with ARM, MIPS with SH4, etc...

Internals: comparing

- According to the ratio, the matches are grouped in a set of either Best, Partial or Unreliable results.
- Best results are these with a similarity ratio of 1.0.
- Partial results are these with a similarity ratio bigger or equal to 0.5.
  - In general, some heuristics are considered reliable even when the ratio is lower than 0.5.
  - For example: the “Same name” heuristic.
- Unreliable results are these with less than 0.5 similarity ratio or the heuristic that found the match or matches is known to cause too many false positives.
  - Or is, yet, considered experimental.

Heuristics

Heuristics

- There are, as previously said, many heuristics used by Diaphora in order to find matches.
- Some of them are very simple, others are more complex.
- Also, some of them are reliable while others are rather unreliable.
- Let's see some of them...

Heuristics

- Heuristic “Bytes hash and names”.
- It simply compares the non-changing bytes of each instruction and the referenced true names.
  - IDA only for now.
- A true name is a name like a function name or string reference with a name that is not generated by IDA.
- One of the simplest and best heuristics.
- Let's see the whole code of this heuristic in the next slide...

Bytes hash and names

```
sql = """ select distinct f.address ea, f.name name1, df.address ea2, df.name name2,
                 'Bytes hash and names' description,
                 f.pseudocode, df.pseudocode,
                 f.assembly, df.assembly,
                 f.pseudocode_primes, df.pseudocode_primes
            from functions f,
                 diff.functions df
           where f.names = df.names
             and f.bytes_hash = df.bytes_hash
             and f.names != '[]'""" + postfix
log_refresh("Finding with heuristic 'Bytes hash and names'")
```

Heuristics

- The previous heuristic is one of the most common one in program diffing tools.
- Let's see some heuristics that are unique to Diaphora (I think):
  - “Same cleaned up assembly or pseudo-code.”
  - “Pseudo-code fuzzy AST hash.”
  - “Strongly connected components SPP.”
  - “Switch structures.”
  - “Same rare MD-Index.”

Same cleaned up assembly or pseudo-code

- Partially based on the pseudo-code generated by the available decompiler, naturally.
- It generates a “clean” textual representation of the assembly, pseudo-code or both, that can be used for comparing.
- Example:
  - mov eax, sub_AF0908 → mov eax, xxx
  - Int v1 = v2 + 0x1024 → int xxx = xxx + 0x1024
- This really simple heuristics works pretty well over all and causes a low ratio of false positives.

Same cleaned up assembly

- This is how this heuristic works:

```
engine_error proc near
        push    r12
        cmp     edi, 0A40Fh
        push    rbp
        mov     rbp, rsi
        push    rbx
        mov     rbx, rdx
        jle     short XXXX
XXXX:
        lea     r9d, [rdi-0A410h]
        xor     eax, eax
        jmp     short XXXX
XXXX:
        add     rax, 1
        cmp     rax, 4Ah
        jz      short XXXX
XXXX:
        mov     r8, rax
        shl     r8, 4
        cmp     ds:XXXX[r8], r9d
        jnz     short XXXX
```

```
engine_error proc near
        push    r12
        cmp     edi, 0A40Fh
        push    rbp
        mov     rbp, rsi
        push    rbx
        mov     rbx, rdx
        jle     short XXXX
XXXX:
        lea     r9d, [rdi-0A410h]
        xor     eax, eax
        jmp     short XXXX
XXXX:
        add     rax, 1
        cmp     rax, 4Ah
        jz      short XXXX
XXXX:
        mov     r8, rax
        shl     r8, 4
        cmp     ds:engine_errors[r8], r9d
        jnz     short XXXX
```

- IDA or Radare auto-generated names like labels, object names, etc... are ignored (see the XXXX).
- The final comparison Diaphora makes internally would lead to only this difference.

Same cleaned up assembly or pseudo-code

- One can argue that, in the previous slide, it should also ignore non IDA/Radare2 generated names.
- Doing so, with the previous example, it would give a similarity ratio of 1.0 instead of 0.990 as it does.
- But, what if the non IDA/R2 generated name, true name, changes from a call to add_element to remove_element?
  - This is the reason why I'm not doing so.
  - Writing heuristics for program diffing tools is harder than what it may looks like at first.
  - Unless you don't care about having a gazillon false positives.

Pseudo-code fuzzy AST hash

- It's based on the Hex-Rays decompiler.
  - It seems getting the AST from SnowMan decompiler is tricky.
- It takes each expression (cinsn_t) in the AST and assigns a prime number to it.
  - Example: cif_t → 2, cdo_t → 5, etc…
- All primes are multiplied together and a fuzzy AST hash is generated.
- For now, only perfect matches (equal hash) are used. In the future, partial matches will be considered too (i.e., 90% of primes are matched).
- For non-trivial functions, it finds good matches and generates a low ratio of false positives.

Strongly connected components SPP

- Using small-primes-product, as with the previous heuristic, it assigns a prime for each strongly connected component based on the number of strongly connected components.
- The final product of the calculated primes is the hash.
- As before, only perfect matches (equal hashes) are considered.
- In the future, partial hashes (a big percent of primes match) will be considered as well.

Switch Structures

- The total number of cases as well as the actual values of the switch structures are used to match functions.
- There are some switch values that can cause collisions (i.e., value 1, 2, 3, 4, 5...) but others don't cause at all (i.e., value 0xFEFD0001...)
- Before implementing it, I thought that heuristic would cause too much false positives and would be unreliable.
- The reality proved otherwise: it works better than other heuristics that I thought would cause little or no false positives at all.
- IDA only, for now.

DEMO time

- Small examples
- Porting symbols
- Finding new functionality

Future

Current status

- Currently, in my opinion, is the best open source or otherwise program diffing tool out there.
- Event better if you have decompiler(s).
- In order to make Diaphora better, I have a number of changes and improvements on mind.

More heuristics

- More and more heuristics will be added. Some examples:
  - Use an intermediate language, apply optimizations, and use the intermediate representation for finding matches.
    - It will make possible to directly compare, for example, MIPS and Solaris Sparc or even Z80.
  - Symbolic execution of basic blocks.
    - Finding matches based on the symbolic execution result of a set of basic blocks.
  - Both will use the VEX intermediate language.
  - ...

GUI Changes

- A new, independent, GUI tool will be written.
- It will allow doing both program diffing and even analysing an already exported R2/IDA database.
  - What about a read-only viewer for exported IDA databases that can be used in an iPad or shared with non-IDA users?
  - Note: Some users actually asked me for this.
- The program would be able to consume data from other tools (not only IDA):
  - SnowMan and Radare2 is what people is asking for.
  - Once, one person asked for support for Hopper and Binary Ninja too.

In the not so near future...

- Adapt Diaphora to perform source code to binary matching.
  - What about finding matches between C/C++ source code and a binary? Comparing ASTs, as I know no other possible methods yet.
  - It would make way easier to import symbols from open source libraries that are linked statically in a binary.
  - Also, it opens the door to find matches and import symbols when having partial source code (i.e., non-compilable source).
    - Naturally, a fuzzy C/C++ parser is required and this is a non-trivial task that can be considered a whole long-time project by itself.
    - But I never said it would be easy.

Conclusion

- And this is all for now!
- Diaphora is open source (GPL), use it, modify it, adapt it to your needs, and send me back patches ;)
  - AND IT EVEN SUPPORTS RADARE2!!!1!!
- You can download it from the following URL:
  - https://github.com/joxeankoret/diaphora

Questions?