Call Graph Agnostic Malware Indexing

Joxean Koret

EuskalHack 2017

Introduction

- The Idea
- Previous Approaches
- The Tool
- IT'S A CONPIRACY!!!1!
- Future of the Tool
- Conclusions

The Idea

- Finding similarities between binary executable malware samples by finding small and unique (or almost unique) commonalities.
- Agnostic of the call graph: whole program comparisons are avoided.
- Example: A set of very rare functions shared between samples of malware campaign X and Y of, supposedly, different actors.
- Could be useful, perhaps, for attribution.

Previous Approaches

- Antivirus: Create signatures of very specific artifacts in malware samples.
  - i.e.: Byte streams, specific strings, cryptographic hashes, etc...
  - Very-very false positive prone.
- Yes, I know. Antivirus products use a lot of other mechanisms. But they aren't any good for my purpose:
  - Find unique commonalities.

Previous Approaches

- Cosa Nostra/VxClass:
  - Whole program matching calculating differences between malware samples.
  - Output phylogenetic trees of, supposedly, malware families.
  - Uses the differences as indicators of distance from the “initial” sample(s).
- Really-really slow.
- Also, whole program matching suffers from many other problems (will be explained later).

Usual Problems

- AV approach: too error prone. A single string in a binary is enough for an AV signature.
  - Not useful to determine relationships.
- CosaNostra/VxClass: problems comparing programs with small real code but big statically compiled libraries.
  - i.e.: Two DLLs with one or more big static libraries, like SQLite and OpenSSL, built in, where only the code at DllMain changes.
  - Obvious but non viable option to remove such false positives: Build a lot of popular libraries with many compilers and optimization levels/flags & create signatures to discard them.
    - Does not scale “well”, at all, IMHO.
  - Also, very slow.

“Other“ Approaches?

- Basically, what I want to do is program diffing, but in a not so usual form.
- Let's explain what are, in my opinion, the most popular program diffing usages...
  - (With permission from a blonde guy over here...)

Program Diffing

- Program diffing is a method used to find commonalities between 2 or more programs using many means (like graph theory based algorithms or matching constants, in example).
- Program/binary diffing, can be used for a wide variety of different scenarios.
- I'll just discuss some of the most common ones.

Program Diffing

- Whole program matching: How much code between 2 programs is different or common.
  - New features detection between 2 versions of a program, plagiarism detection, etc...
- Patch Diffing: Detect portions of code modified between 2 different versions of the same program.
  - Find fixed vulnerabilities to write exploits or detection methods.
  - Or, for example, to create signatures to do vulnerability extrapolation.

Program Diffing Tools

- For the 2 previous methods there are various tools like, for example:
  - BinDiff/Diaphora: General usage program diffing tools.
  - Cosa Nostra/VxClass: Create phylogenetic trees of malware families using as indicator of distance between leafs the differences between the call and flow graphs.
- Such tools are useful to detect similarities and differences between versions, fixed bugs, added functionalities, etc...
- But they aren't specifically created to find small unique similarities between programs that could help in, say, attribution.
  - Attribution: which actor is behind some malware campaign?

Time to show the tool...

Enter... MalTindex!

MAL tindex

Mal Tindex

- “Mal Tindex” is an Open Source Malware Indexing set of tools.
- It aims to help malware researchers in attributing a malware campaign by finding small rare similarities between malware samples.
  - E.g.: A function set that only exists in 2 binaries out of a significantly **big** set.
    - “Significantly big” is very “significant” here.
- The name for the project was proposed by a friend as it tries to match “couples” of things.

How it works?

- It first exports a set of signatures of each function in a program using Diaphora.
- Then, the most significant and less false positive prone signature types are used as indicators and indexed.
- After a big enough number of program samples is indexed, the signatures can be used to find very specific functions that are only shared between rather small sets of programs.
- ...

Signatures

- As explained before, MalTindex uses Diaphora to export binaries to a database and generate signatures.
- Diaphora exports almost everything from each function, including pseudo-code, graph related data, etc...
- The signature types (table fields generated by Diaphora) that I found caused the less number of false positives were these that I chose.
- They are explained in the next slides.

Signature Types

- Bytes hash: Just a MD5 over the whole function's bytes, for “big enough” functions.
  - Pretty robust, almost 0 false positives found so far.
- Function hash: Similar as before, but removing the bytes that are variable (i.e., non position independent parts).
  - Same as before, with some false positives.
- MD Index: A hash function for CFGs based on the topological order, in-degrees and out-degrees of a function. Invented by Thomas et al (MP-IST-091-26).
  - More “fuzzy” than the others, thus, more false positive prone during my testing. However, unique MD-Index values are pretty useful. And, actually, this is what we're looking for here: unique signatures.
- Pseudo-code primes: A small-primes-product (SPP) based on the AST of the pseudo-code of a function (if a decompiler is available).

Matching

- MalTindex doesn't do anything “magical” or very special to match:
  - Just compares equal but rare enough signatures.
  - That's, basically, all it does.
- Every time a malware sample is exported with Diaphora, a set of tables is updated with the set of rare signatures.
- These tables with the rare signatures are the ones used to find rare enough but common similarities.
  - The actual malware indexes.

Using Mal Tindex

- How to use it:

  $ export DIAPHORA_DB_CONFIG=/path/to/cfg

  $ diaphora_index_batch.py /ida/dir/ samples_dir

- When all the samples are indexed:

  $ maltindex.py <database path>

  MalTindex> match MD5

  or

  MalTindex> match MD5_1 MD5_2

  or

  MalTindex> unique MD5

- And it will print all the matches or “apparently unique” matches between the whole dataset or for a specific sample.

DEMO

Time to discuss about more problems of this approach.

And how it can lead to... bad attribution.

Attribution

- Attribution based on ”shared” chunks of code should be taken with big care.
- Otherwise, it leads to, basically, choosing your own conspiracy.
- Specially for not big enough datasets.
- Let's see some examples...

Conspiranoia case #1:
The Sony Attack (2014)

Attribution

- During my tests with ~9,000 files, I found a curious match:
  - D1C27EE7CE18675974EDF42D4EEA25C6
  - E4AD4DF4E41240587B4FE8BBCB32DB15
- The first sample was used in the Sony attack (wypall).
- The second sample is a “DLL” from the NSA leak.
  - Looks cool, right?
- It is a false positive: a specific version of zlib1.dll appeared first, in my dataset, with one of the NSA dumps.

The Match

```
unsigned int __cdecl sub_403B69(int a1, void *Dst, int a3)
{
  unsigned int v3; // edi@1
  int v5; // eax@5
  int v6; // eax@6

  v3 = *(_DWORD *)(a1 + 4);
  if ( v3 > a3 )
    v3 = a3;
  if ( !v3 )
    return 0;
  *(_DWORD *)(a1 + 4) -= v3;
  v5 = *(_DWORD *)(*(_DWORD *)(a1 + 28) + 24);
  if ( v5 == 1 )
  {
    v6 = sub_4058CC(*(_DWORD *)(a1 + 48), *(_DWORD *)a1, v3);
LABEL_9:
    *(_DWORD *)(a1 + 48) = v6;
    goto LABEL_10;
  }
  if ( v5 == 2 )
  {
    v6 = sub_4070B8(*(_DWORD *)(a1 + 48), *(_DWORD *)a1, v3);
    goto LABEL_9;
  }
LABEL_10:
  memcpy(Dst, *(const void **)a1, v3);
  *(_DWORD *)a1 += v3;
  *(_DWORD *)(a1 + 8) += v3;
  return v3;
}
```

```
unsigned int __usercall sub_5A4C1BD2@<eax>(_DW
{
  unsigned int v3; // edi@1
  int v5; // eax@5
  int v6; // eax@6

  v3 = a1[1];
  if ( v3 > a3 )
    v3 = a3;
  if ( !v3 )
    return 0;
  a1[1] -= v3;
  v5 = *(_DWORD *)(a1[7] + 24);
  if ( v5 == 1 )
  {
    v6 = adler32(a1[12], *a1, v3);
LABEL_9:
    a1[12] = v6;
    goto LABEL_10;
  }
  if ( v5 == 2 )
  {
    v6 = crc32(a1[12], *a1, v3);
    goto LABEL_9;
  }
LABEL_10:
  memcpy(Dst, (const void *)*a1, v3);
  *a1 += v3;
  a1[2] += v3;
  return v3;
}
```

The Sony Attack

- After I realized that it was a code chunk in zlib1.dll (1.2.5) I asked my friends to send me more zlib1.dll files.
- After +25 files indexed the match remained unique.
- After ~30 files indexed the unique match disappeared:
  - A zlib1.dll in the popular Windows Xampp application matched with the previous 2 samples.
- End of case. A false positive. My dataset was not big enough.
  - Solution: index more programs.

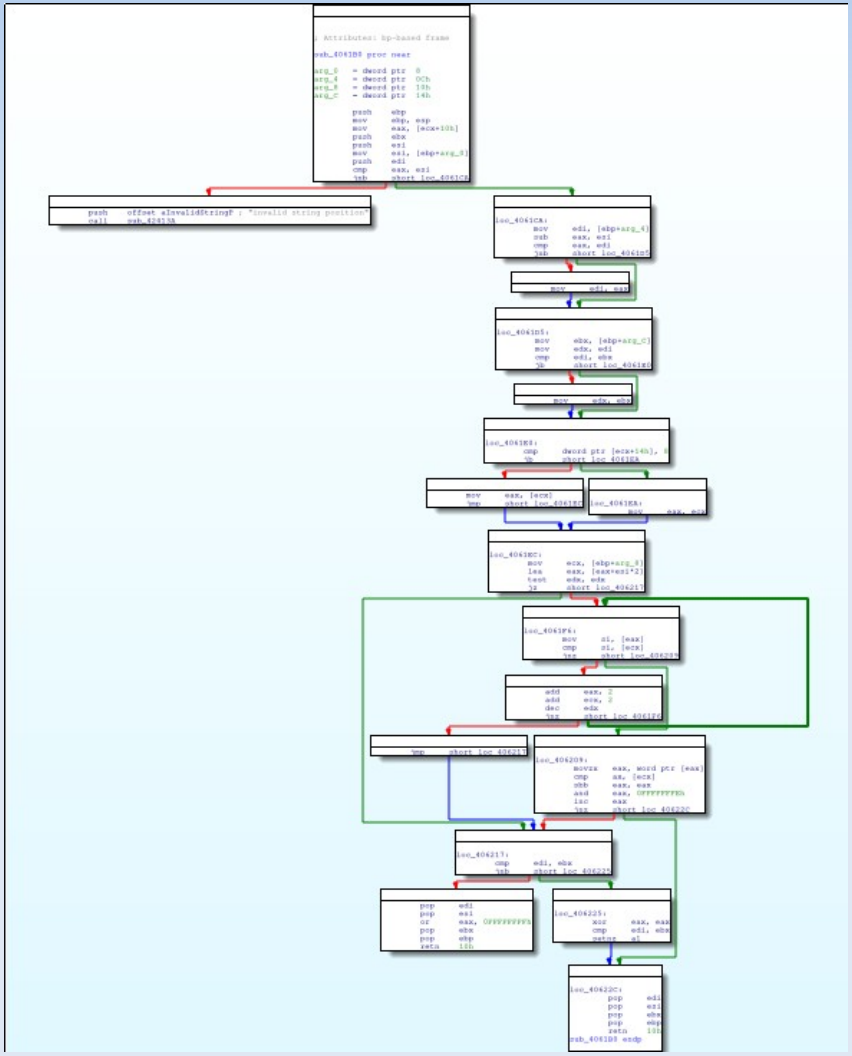
Conspiranoia #2:
The Lazarus Group & Amonetize case

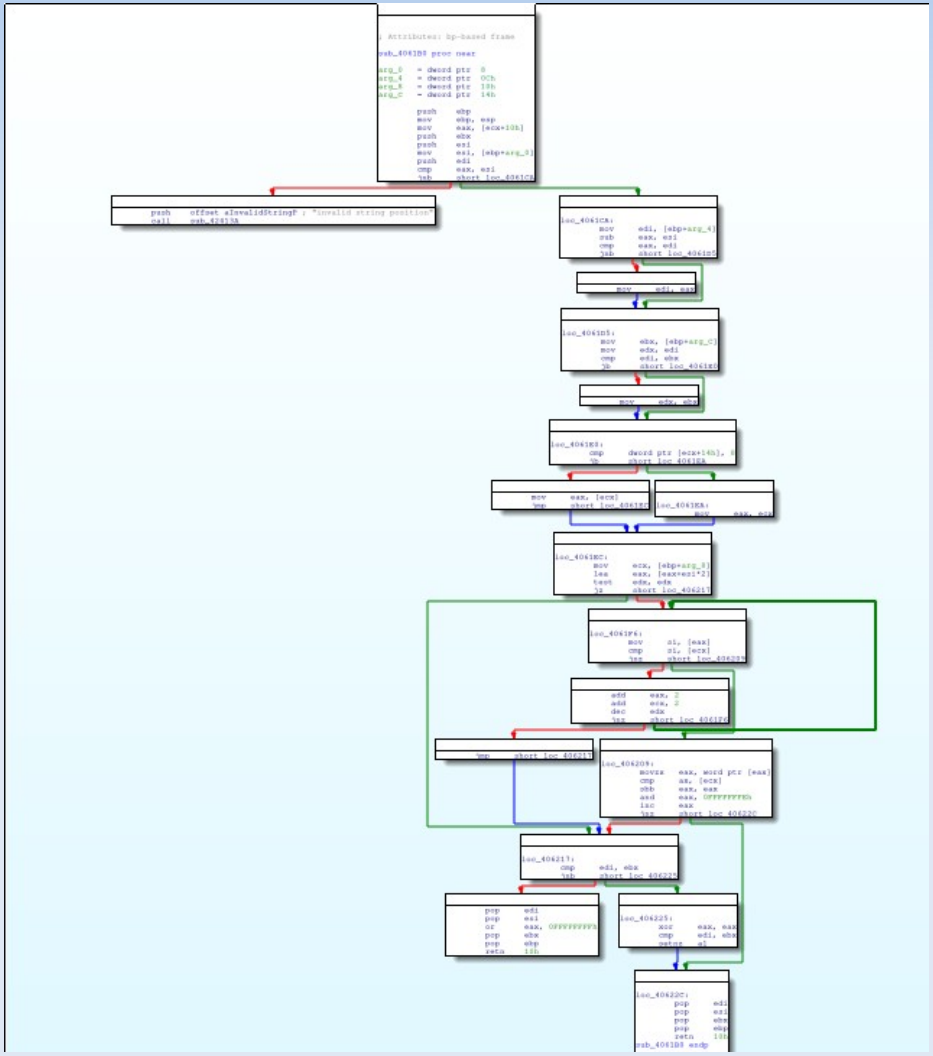
The relationship is clear!!!1!

Lazarus & Amonetize

- Again, during my tests, I found yet another case of a match that looked interesting (or at least weird enough as to research it):
  - 18A451D70F96A1335623B385F0993BCC
  - CBDB7E158155F7FB73602D51D2E68438
- The first sample is Ratankba, used by Lazarus group to attack a Polish bank.
- The 2nd sample is just an adware: Amonetize.
- What do they have in common?
  - Let's see the shared code...

The Match





The Match (Fragments)

```
signed int __thiscall sub_4025E0(_DWORD *this,
{
  unsigned int v5; // eax@1
  unsigned int v6; // edi@3
  unsigned int v7; // eax@3
  unsigned int v8; // edx@5
  _DWORD *v9; // eax@8
  _WORD *v10; // ecx@10
  _WORD *v11; // eax@10
  signed int result; // eax@14

  v5 = this[4];
  if ( v5 < a2 )
    sub_4188FC("invalid string position");
  v6 = a3;
  v7 = v5 - a2;
  if ( v7 < a3 )
    v6 = v7;
  v8 = v6;
  if ( v6 >= a5 )
    v8 = a5;
  if ( this[5] < 8u )
    v9 = this;
  else
    v9 = (_DWORD *)*this;
  v10 = a4;
  v11 = (_WORD *)((char *)v9 + 2 * a2);
  if ( !v8 )
    goto LABEL_15;
  while ( *v11 == *v10 )
  {
    ++v11;
    ++v10;
    if ( !--v8 )
      goto LABEL_15;
  }
```

```
signed int __thiscall sub_4061B0(_DWORD *this, u
{
  unsigned int v5; // eax@1
  unsigned int v6; // edi@3
  unsigned int v7; // eax@3
  unsigned int v8; // edx@5
  _DWORD *v9; // eax@8
  _WORD *v10; // ecx@10
  _WORD *v11; // eax@10
  signed int result; // eax@14

  v5 = this[4];
  if ( v5 < a2 )
    sub_42413A("invalid string position");
  v6 = a3;
  v7 = v5 - a2;
  if ( v7 < a3 )
    v6 = v7;
  v8 = v6;
  if ( v6 >= a5 )
    v8 = a5;
  if ( this[5] < 8u )
    v9 = this;
  else
    v9 = (_DWORD *)*this;
  v10 = a4;
  v11 = (_WORD *)((char *)v9 + 2 * a2);
  if ( !v8 )
    goto LABEL_15;
  while ( *v11 == *v10 )
  {
    ++v11;
    ++v10;
    if ( !--v8 )
      goto LABEL_15;
  }
```

The Match

- Is the match good enough? Is the function big enough?
  - I would say “yes”.
- Is the function unique across the whole dataset?
  - It was. Until I added more samples to my dataset...
  - 0925FB0F4C06F8B2DF86508745DBACB1 (Dalbot)
  - 9144BE00F67F555B6B39F053912FDA37 (QQDownload, not malware)
- Is this a false positive?
  - Absolutely. This function seems to be from a specific MFC/ATL version.
- End of case.
  - Solution: Index more and more programs.

Conspiranoia #3:
Stuxnet and Duqu

Stuxnet & Duqu

- I was trying to find matches between the dumped NSA tools and other things. I found nothing too exciting.
- Then, I decided to try to match Duqus with other things. And Stuxnet matched.
  - 546C4BBEBF02A1604EB2CAAAD4974DE0
  - A Driver.
- This is, probably, the first non false positive result I have had. I think.
- Let's see...

Match

- Stuxnet:

```
void __stdcall DriverReinitializationRoutine(struct _DRIVER_OBJECT *DriverObject, PVOID Context, ULONG Count)
{
  void *v3; // ecx@1

  if ( !KeGetCurrentIrql() && Count <= 0x65 && !sub_10580(v3) )
  {
    if ( sub_10300() )
      sub_108D2();
    else
      IoRegisterDriverReinitialization(DriverObject, DriverReinitializationRoutine, 0);
  }
}
```

- Duqu:

```
void __stdcall DriverReinitializationRoutine(struct _DRIVER_OBJECT *DriverObject, PVOID Context, ULONG Count)
{
  if ( !KeGetCurrentIrql() && Count <= 0x65 && !sub_10580() )
  {
    if ( (unsigned __int8)sub_10300() )
      sub_108D2();
    else
      IoRegisterDriverReinitialization(DriverObject, DriverReinitializationRoutine, 0);
  }
}
```

The first match

- The DriverReinitializationRoutine matches 1 to 1. However, it isn't big enough.
- But there are other matches.
- Let's see them...

Stuxnet

```
int __stdcall sub_104C8(_BYTE *a1)
{
  char v1; // al@5
  _DWORD *v2; // edi@5
  int result; // eax@7
  unsigned __int8 (*v4)(void); // edi@11
  int v5; // [esp+8h] [ebp-120h]@6
  __int16 v6; // [esp+11Ch] [ebp-Ch]@9
  _DWORD *v7; // [esp+124h] [ebp-4h]@5

  *a1 = 1;
  if ( !KeGetCurrentIrql() )
  {
    sub_105A8();
    if ( (unsigned __int8)sub_1089A(0) || (unsigned __int8)sub_1089A(1) )
      return 0;
    v1 = sub_1089A(2);
    v2 = v7;
    if ( v1 )
    {
      memset(&v5, 255, 0x11Cu);
      v5 = 284;
      if ( !*v7 )
        return -1073741823;
      result = ((int (__stdcall *)(int *))*v7)(&v5);
      if ( result )
        return result;
      if ( !v6 )
        return v2[1] != 0 ? 0xC0000001 : 0;
    }
    v4 = (unsigned __int8 (*)(void))v2[1];
    if ( v4 )
    {
      if ( v4() )
```


```
int __stdcall sub_104C8(_BYTE *a1)
{
  char v1; // al@5
  _DWORD *v2; // edi@5
  int result; // eax@7
  unsigned __int8 (*v4)(void); // edi@11
  int v5; // [esp+8h] [ebp-120h]@6
  __int16 v6; // [esp+11Ch] [ebp-Ch]@9
  _DWORD *v7; // [esp+124h] [ebp-4h]@5

  *a1 = 1;
  if ( !KeGetCurrentIrql() )
  {
    sub_105A8();
    if ( (unsigned __int8)sub_1089A(0) || (unsigned __int8)sub_1089A(1) )
      return 0;
    v1 = sub_1089A(2);
    v2 = v7;
    if ( v1 )
    {
      memset(&v5, 255, 0x11Cu);
      v5 = 284;
      if ( !*v7 )
        return -1073741823;
      result = ((int (__stdcall *)(int *))*v7)(&v5);
      if ( result )
        return result;
      if ( !v6 )
        return v2[1] != 0 ? 0xC0000001 : 0;
    }
    v4 = (unsigned __int8 (*)(void))v2[1];
    if ( v4 )
    {
      if ( v4() )
```

Stuxnet/Duqu

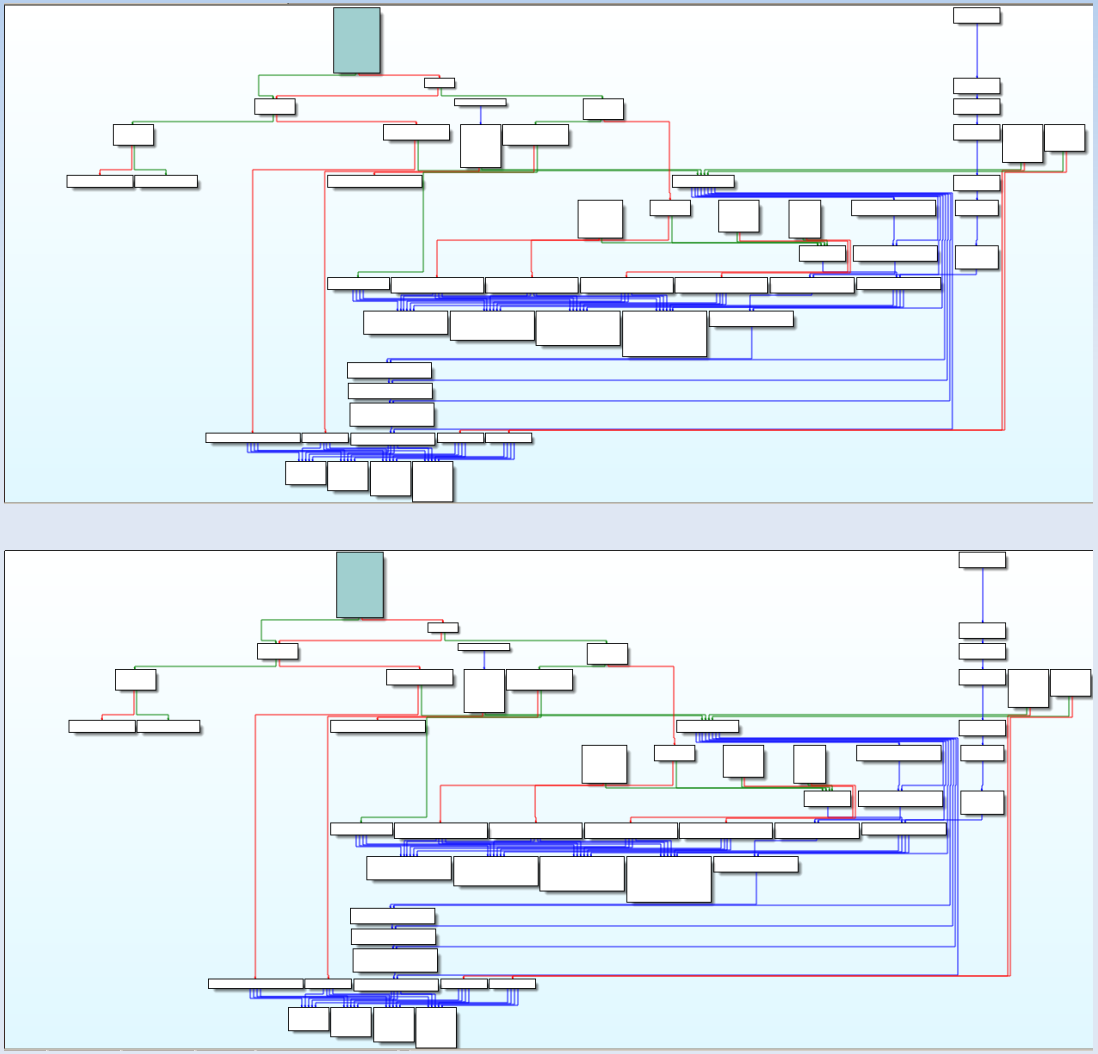
- The match is perfect. Indeed, even most addresses in the driver actually match.
- This is not an isolated false positive and the proof is conclusive.
  - There is no single match, there are multiple matches.
  - F8153747BAE8B4AE48837EE17172151E
  - C9A31EA148232B201FE7CB7DB5C75F5E
  - And many others.
- End of history. Both Symantec and F-Secure are right: Duqu is Stuxnet or shared code.

Conspiranoia #3:
Dark Seoul and Bifrose

Dark Seoul & Bifrose???

- When my dataset was not “big enough”, I found a curious and apparently unique match:
  - 5FCD6E1DACE6B0599429D913850F0364
  - 0F23C9E6C8EC38F62616D39DE5B00FFB
- The 1st sample is a Dark Seoul (DPRK).
- The 2nd sample is a ghetto Bifrose.
- Do they have something in common???
  - Actually, no. But the dataset was not big enough.
- Let's see the match...

The Match



Dark Seoul & Bifrose?

- The function is big enough and complex enough. So, the match is good.
- But can we consider this proof conclusive?
  - Not at all.
- Indeed, when I started feeding more samples to my dataset, it wasn't any more a unique match.
  - 46661C78C6AB6904396A4282BCD420AE (Nenim)
  - 67A1DB64567111D5B02437FF2B98C0DE (Infected with a version of Sality).
- There is no other match so... end of case. Is a false positive.
- I still don't know which function it is. But I know the solution:
  - Index more and more programs.

Conspiranoia #4:
WannaCry & Lazarus Group

WannaCry & DPRK!!!!??

- On 15th May, Neel Mehta, a Google engineer published the following tweet:

- This is a match (independently reproduced by me based on their MD Index) between WannaCry and a malware from the Lazarus Group (DPRK).
- Let's see the match...

WannaCry & Lazarus Group

- MalTindex finds the same match between samples:

```
MalIndex> match 9C7C7149387A1C79679A87DD1BA755BC AC21C8AD899727137C4B94458D7AA8D8
Searching for matches between 9C7C7149387A1C79679A87DD1BA755BC and AC21C8AD899727137C4B94458D7AA8D8...
Type | Name 1 | MD5 | Name 2 | Hash | Filename
-----------------------------------------------
M sub_402560 AC21C8AD899727137C4B94458D7AA8D8 sub_10004BA0 4.592895023151108863613470134 /home/joxean/
db
M sub_10004BA0 9C7C7149387A1C79679A87DD1BA755BC sub_402560 4.592895023151108863613470134 /home/joxean/
db
Total of 2 row(s)
```

- Searching for the specific MD Index, it only finds the same 2 matches:

```
MalIndex> mdindex 4.592895023151108863613470134
md5 | name | file_name
---------------------
AC21C8AD899727137C4B94458D7AA8D8 sub_10004BA0 /h
9C7C7149387A1C79679A87DD1BA755BC sub_402560 /hom
Total of 2 row(s)
```

Lazarus Group (Fragment)

```
  v1 = a1;
  v2 = *a1;
  LOBYTE(v2) = 1;
  v3 = (char *)a1 + 5;
  *a1 = v2;
  *(v3 - 1) = 3;
  *v3 = 1;
  sub_100016B0((int)a1 + 6, 32);
  v4 = time(0);
  *(int *)((char *)a1 + 6) = sub_10004CC0(v4, v4 >> 31, 4);
  *((_BYTE *)a1 + 38) = 0;
  v5 = (_WORD *)((char *)a1 + 39);
  v6 = 0;
  v7 = 6 * (rand() % 5 + 2);
  if ( v7 > 0 )
  {
    v15 = (_WORD *)((char *)a1 + 41);
    while ( 1 )
    {
      v8 = rand() % 0x4Bu;
      v9 = 0;
      v14 = v8;
      if ( v6 > 0 )
        break;
LABEL_8:
      if ( v8 == -1 )
        goto LABEL_10;
      LOWORD(v8) = word_10012AA4[v8];
      *v15 = dword_1001336C(v8);
LABEL_11:
      ++v6;
      ++v15;
      if ( v6 >= v7 )
        goto LABEL_12;
```

```
  v1 = (int *)a1;
  v2 = *(_DWORD *)a1;
  LOBYTE(v2) = 1;
  v3 = (_BYTE *)(a1 + 5);
  *(_DWORD *)a1 = v2;
  *(v3 - 1) = 3;
  *v3 = 1;
  sub_408130(a1 + 6, 32);
  v4 = time(0);
  *(_DWORD *)(a1 + 6) = htonl(v4);
  *(_BYTE *)(a1 + 38) = 0;
  v5 = (u_short *)(a1 + 39);
  v6 = 0;
  v7 = 6 * (rand() % 5 + 2);
  if ( v7 > 0 )
  {
    v15 = (u_short *)(a1 + 41);
    while ( 1 )
    {
      v8 = rand() % 0x4Bu;
      v9 = 0;
      v14 = v8;
      if ( v6 > 0 )
        break;
LABEL_8:
      if ( v8 == -1 )
        goto LABEL_10;
      *v15 = htons(hostshort[v8]);
LABEL_11:
      ++v6;
      ++v15;
      if ( v6 >= v7 )
        goto LABEL_12;
    }
```

WannaCry & Lazarus Group

- The function is rare enough. In my dataset, there is only this match.
- Apparently, Google only has this same match.
- However, for me, it's totally non conclusive.
- Can we associate an actor to a malware family just because one function matches between them?
  - Not enough evidence, in my opinion, and it can be done on purpose. Actually, in the future, I will try to automate that process.
- Also, the logic says that a group stealing millions of USD is not an actor asking for a $300 ransom per box.
- End of case?

Conspiranoia #5:

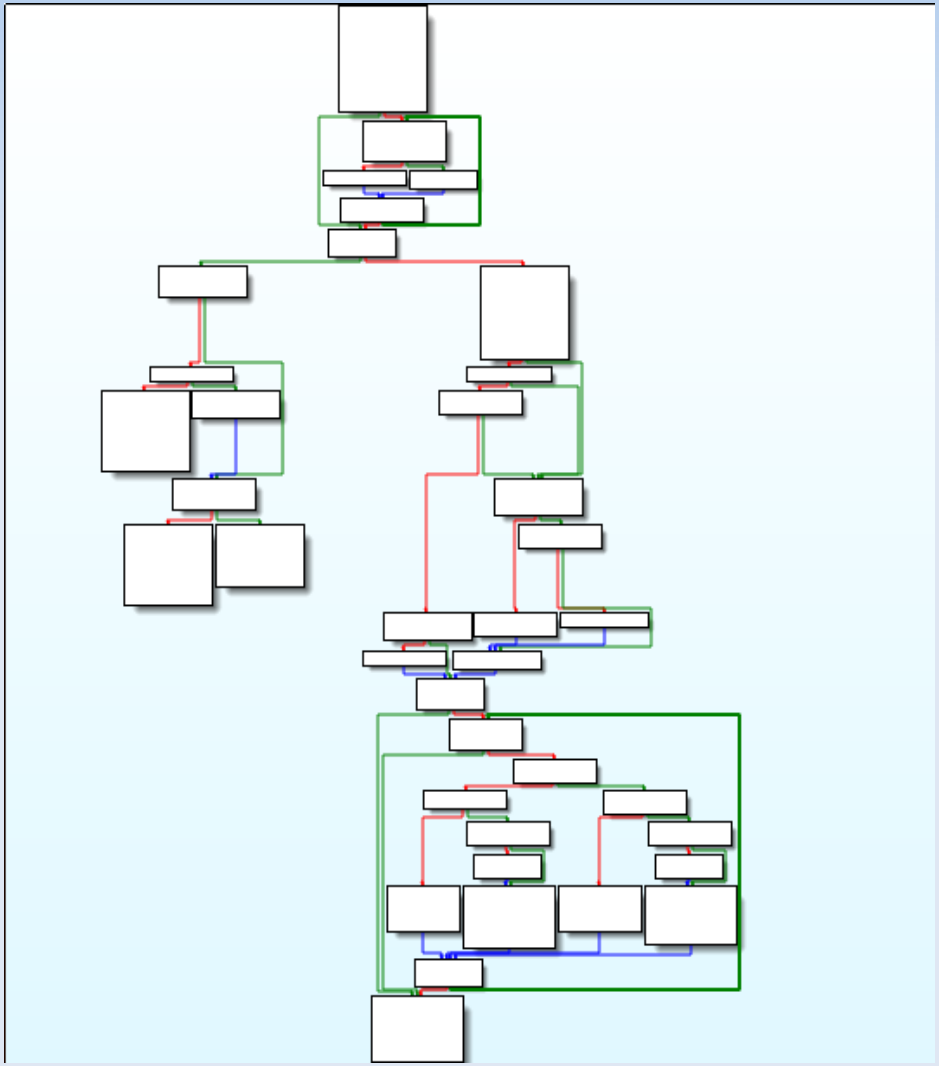
Bundestrojaner, NSA, WannaCry and 2 shitty malwares

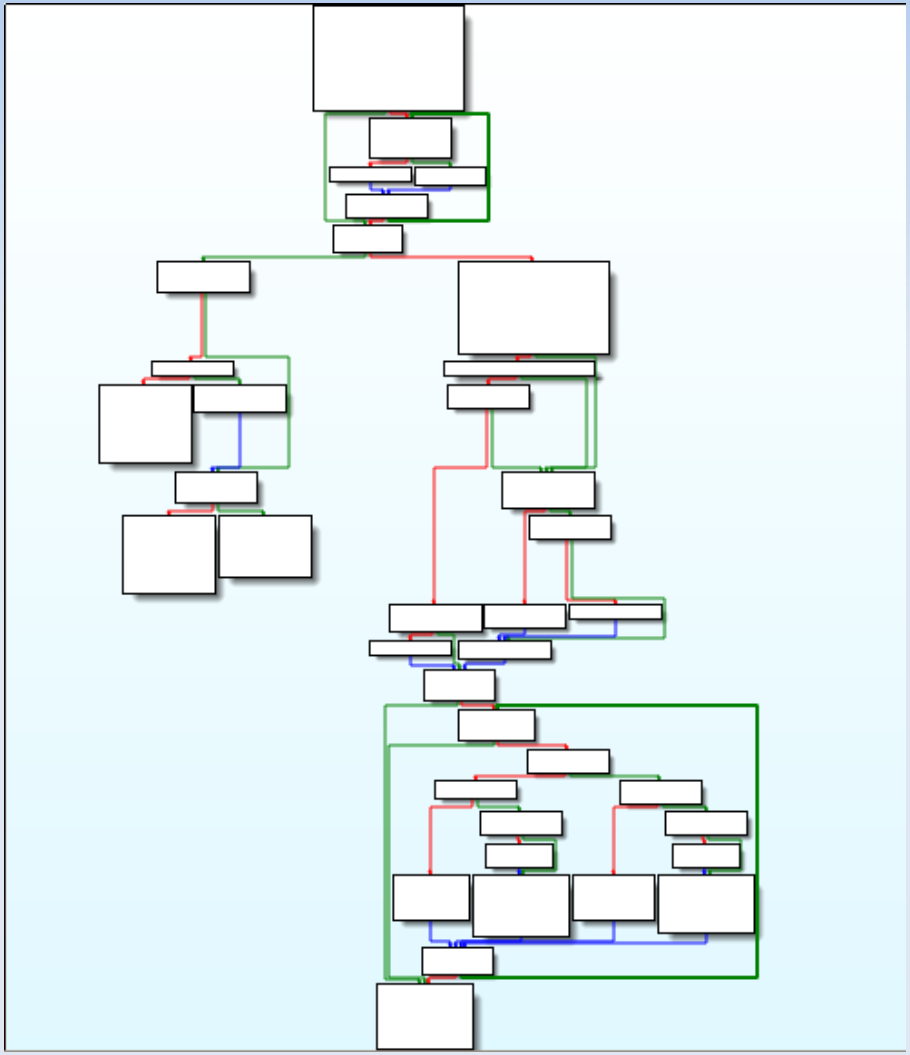
LOL, WUT?

LOL, WUT?

- One of my favourite false positives ever. Searching for a specific MD-Index (11.272122399876039724401052 68) just 5 files appear:
  - DB5EC5684A9FD63FCD2E62E570639D51: NSA's GROK GkDecoder.
  - 930712416770A8D5E6951F3E38548691: Bundestrojaner!
  - 7257D3ADECEF587636146 4088EF3E26B: Some Krap?
  - 0EB2E1E1FAFEBF8839FB5E3E2AC2F7A8: Microsoft calls it Nenim.
  - DB349B97C37D22F5EA1D1841E3C89EB4: WannaCry!
- Naturally, it must be a false positive. Right?

Bundestrojan vs WannaCry





Bundestrojan vs WannaCry

```
.text:1000F8D0 sub_1000F8D0 proc near                  ; CODE XREF: sub_1000B800+33↑p
.text:1000F8D0
.text:1000F8D0 var_4   = dword ptr -4
.text:1000F8D0 arg_0   = dword ptr  4
.text:1000F8D0 arg_4   = dword ptr  8
.text:1000F8D0
.text:1000F8D0         push    ecx
.text:1000F8D1         push    ebx
.text:1000F8D2         push    ebp
.text:1000F8D3         push    esi
.text:1000F8D4         mov     edx, dword_1005231C
.text:1000F8DA         push    edi
.text:1000F8DB         mov     edi, ecx
.text:1000F8DD         mov     ebx, [esp+14h+arg_4]
.text:1000F8E1         mov     al, 1
.text:1000F8E3         mov     ecx, [edi+4]
.text:1000F8E6         mov     esi, ecx
.text:1000F8E8         mov     ebp, [ecx+4]
.text:1000F8EB         cmp     ebp, edx
.text:1000F8ED         jz      short loc_1000F909
.text:1000F8EF
.text:1000F8EF loc_1000F8EF:                           ; CODE XREF: sub_1000F8D0+37↓j
.text:1000F8EF         mov     eax, [ebx]
.text:1000F8F1         mov     esi, ebp
.text:1000F8F3         cmp     eax, [ebp+0Ch]
.text:1000F8F6         setl    al
.text:1000F8F9         test    al, al
.text:1000F8FB         jz      short loc_1000F902
.text:1000F8FD         mov     ebp, [ebp+0]
.text:1000F900         jmp     short loc_1000F905
.text:1000F902 ; ---------------------------------------------------------------------------
.text:1000F902
.text:1000F902 loc_1000F902:                           ; CODE XREF: sub_1000F8D0+2B↑j
.text:1000F902         mov     ebp, [ebp+8]
```

```
.text:00408390 sub_408390 proc near                    ; CODE XREF: sub_401370+130↑p
.text:00408390                                         ; sub_401370+161↑p ...
.text:00408390
.text:00408390 var_4   = dword ptr -4
.text:00408390 arg_0   = dword ptr  4
.text:00408390 arg_4   = dword ptr  8
.text:00408390
.text:00408390         push    ecx
.text:00408391         push    ebx
.text:00408392         push    ebp
.text:00408393         push    esi
.text:00408394         mov     edx, dword ptr current_executable_name+118h
.text:0040839A         push    edi
.text:0040839B         mov     edi, ecx
.text:0040839D         mov     ebx, [esp+14h+arg_4]
.text:004083A1         mov     al, 1
.text:004083A3         mov     ecx, [edi+4]
.text:004083A6         mov     esi, ecx
.text:004083A8         mov     ebp, [ecx+4]
.text:004083AB         cmp     ebp, edx
.text:004083AD         jz      short loc_4083C9
.text:004083AF
.text:004083AF loc_4083AF:                             ; CODE XREF: sub_408390+37↓j
.text:004083AF         mov     eax, [ebx]
.text:004083B1         mov     esi, ebp
.text:004083B3         cmp     eax, [ebp+0Ch]
.text:004083B6         setl    al
.text:004083B9         test    al, al
.text:004083BB         jz      short loc_4083C2
.text:004083BD         mov     ebp, [ebp+0]
.text:004083C0         jmp     short loc_4083C5
.text:004083C2 ; ---------------------------------------------------------------------------
.text:004083C2
.text:004083C2 loc_4083C2:                             ; CODE XREF: sub_408390+2B↑j
```

Yet another false positive

- My guess, considering which malwares appear to share this function is that it's a false positive.
  - Ones seems to be an installer doing RAR stuff.
  - Perhaps this function “decrypts” something?
  - No idea, I haven't found what it actually does.
- This function is the only “evidence” that can be used to relate such malwares.
  - ...and groups

End of case or... it's a conspiracy!

- One shared function is not enough evidence.
  - Unless you want to believe!
  - Now you've “proof” to relate WannaCry with the Bundestag, the NSA, and various crappy malware groups.
  - If you love conspiranoias, it's better than chem trails.
- End of case. Totally. For real.

Future

Future Plans

- I have a few ideas on mind for the not so far future like:
  - Make the dataset (the indexes) public. Probably creating a web service to fetch/query/upload data.
  - Create an IDA plugin to get symbol names from indexed sources.
  - Support both Radare2 and IDA as backends.
  - Perhaps, support also DynInst.
  - Implement a different and better way to determine rareness. Probably based on Halvar's Bayes thing.

Web Service

- The idea is to, basically, allow others to:
  - Index sample and upload its indexed data to a public server.
  - Allow others to query the samples I already indexed or the ones people upload.
  - Allow others to research/find match between malware families, different actors/groups, etc...

IDA Plugin

- Like FLIRT signatures or Zynamics BinCrowd.
- The plugin would search for symbols for the local database in a remote server, finding a match with a big database with the most common and non common stuff indexed.
  - Open source libraries, well known malwares, etc...
- Every single component would be open sourced:
  - You can deploy your own symbols server in your organization.
  - Or you can use the public one I will publish some day.

Exporters

- So far, Mal Tindex is based on Diaphora and it just supports IDA.
- In The Not So Far Future (TM), I plan to add support for exporting into Diaphora for the following tools:
  - Radare2.
  - Maybe, DynInst.

Conclusions

Conclusions

- We can find seemingly unique similarities in many cases when just looking to functions between different groups/actors.
  - Indeed, we can actually “build” them.
- In my opinion, a single function matching in just 2 binaries is not enough evidence.
  - During my testing, I was only able to prove a match was not a false positive when there were various unique functions matched, not just one.
  - Regardless of its size.

Conclusions

- We should be skeptical when a company says that they have a unique match between 2 groups based on just a single function.
- Why?
  - We don't have their dataset, thus, we cannot independently reproduce.
  - We've to trust companies that might have some interest on the specific case.
- On the other hand, if a company makes such a claim and others are able to find another unique match, their credibility would be deteriorated.
  - E.g.: It's unlikely they would use such an argument unless they believe on it.

Conclusions

- The size of the dataset is highly important when doing malware indexing.
  - The bigger it's, the better.
- However, considering how expensive it's in both terms of storage and indexing time, it might be prohibitive for most organizations.
  - On average, a binary program takes 5 minutes to index (IDA auto analysis + indexing tool execution).
  - When my testing (SQLite) database was around 9,000 files, it was ~10 GB. And it was too small.
- Do the math to calculate its requirements for a real, production, use case.

Conclusions

- Having a really big dataset is a must to attribute/link groups based on malware indexing. But, when is it big enough?
  - I don't have an answer to it. Perhaps Google's dataset is big enough. Or Microsoft's one. Or none.
  - According to a friend, probably, a minimum of 1 million files.
- Can datasets be biased?
  - Almost always: you will see only what you feed to your system. You will only see your **bubble**.
  - Are you filtering samples? Your dataset is **biased**.
- Can we reproduce what $company says?
  - Hardly, unless they share their whole dataset.
- Can $company use attribution politically?
  - Lol.


Thank you!
Questions?