# Automated Detection of Obfuscated Code

10 Aug 2021 - Tim Blazytko

In a previous blog post, we already discussed that it is valuable to know which code areas are obfuscated; those areas often guard sensitive code and are worth a closer look. Furthermore, we designed a heuristic to automatically detect control-flow flattening and state machines in binaries by identifying specific loop characteristics in the control-flow graph. However, other code obfuscation techniques such as opaque predicates, complex arithmetic encodings or virtualization are not necessarily covered by this heuristic, especially if the control-flow graph is loop-free. For these cases, we have to develop new heuristics to identify obfuscation.

In this blog post, we will have a look at some generic heuristics that allow us to quickly identify interesting code parts. For this, we first discuss the general idea to detect code obfuscation. Afterward, we have a closer look at different heuristics and what they identify. In the end, we investigate how the heuristics work on (partially) obfuscated binaries.

We implemented all heuristics in a Binary Ninja plugin called Obfuscation Detection. The repository also contains (most of the) sample binaries we use for evaluation. If you would like to play around with it, check it out.

## Detection Heuristics

Our goal is to develop heuristics that pinpoint code which shares similar characteristics to obfuscated code. Early in the reverse engineering process when we want to get a better overview over the binary, we can use these heuristics to spot *interesting* code locations that we can inspect manually. Even if these code locations are not obfuscated, they are still relevant for reverse engineers since they often implement complex dispatching routines, cryptographic algorithms or other important program logic.

One way to look at code obfuscation is that it tries to impede reverse engineering by artificially increasing the code's complexity. Therefore, we can identify obfuscated code by looking for *complex* code, such as functions with large basic blocks or control-flow graphs. Another way to look at code obfuscation is that it tries to confuse reverse engineers by playing with their assumptions and analysis tools. This way, we can look for *anomalies* in our analysis tooling, such as overlapping instructions or meaningless disassembly.

If we want to apply such heuristics to large binaries (e.g., several hundreds of megabytes in size), they have to be efficient and provide a minimal analysis overhead. As a consequence, we rely on data points that are easy to obtain and heuristics that are cheap to compute.

In practice, it is useful to apply several heuristics independently: While different heuristics may find the same code locations, they may also find other locations since they look for different characteristics. In the following, we will get to know three such heuristics that accomplish the aforementioned requirements.

### Complex Functions

Intuitively, large functions implement a complex program logic such as file parsing, network protocols, dispatching routines or cryptographic algorithms. If functions are large due to cod obfuscation, they often contain dead code, (nested) opaque predicates or control-flow flattening.

To determine a function's complexity, we could, for example, count its number of instructions, basic blocks or bytes. However, in all of these cases, we ignore the function's branch characteristics: We ignore if the function consists contains loops or (nested) branches. A more generic way to measure a function's complexity is to measure the complexity of its control-flow graph; for this, we can calculate its cyclomatic complexity. In short, the cyclomatic complexity measures the number of independent paths in a function and is calculated by `#edges - #blocks + 2`. If the control-flow graph has only a single basic block and no edges, we get `0 - 1 + 2 = 1`; if it contains five basic blocks and eight edges, we get `8 - 5 + 2 = 5`. The second control-flow graph is more complex. We can implement this heuristic within a few lines in Binary Ninja:

```
def calc_cyclomatic_complexity(function):
    # number of basic blocks
    num_blocks = len(function.basic_blocks)
    # number of edges in the graph
    num_edges = sum([len(b.outgoing_edges) for b in function.basic_blocks])

    return num_edges - num_blocks + 2
```

To build a meaningful heuristic that is based on the cyclomatic complexity, we can *sort* all functions by their complexity and print the *upper* 10%:

```
def find_complex_functions(bv):
    # sort functions by cyclomatic complexity
    sorted_functions = sorted(bv.functions, key=lambda x: calc_cyclomatic_complexity(x))

    # bound to print only the top 10%
    bound = math.ceil(((len(bv.functions) * 10) / 100))

    # print top 10% (iterate in descending order)
    for f in list(reversed(sorted_functions))[:bound]:
        print(f"{hex(f.start)}: {calc_cyclomatic_complexity(f)}")
```

Overall, the function's cyclomatic complexity provides a good heuristic to fingerprint complex graphs. However, it ignores the size of basic blocks and the number of instructions in the function; data points which are interesting to pinpoint other code constructs.

### Large Basic Blocks

Large basic blocks guarantee that a sequence of code is executed in a row. Often, they implement complex calculations, (unrolled) cryptographic algorithms or initialization routines. For obfuscated code, large basic blocks often contain dead code, initialize virtual machines or hide simple arithmetic calculations in complex arithmetic encodings.

To pinpoint large basic blocks, we could sort all basic blocks in the binary by their number of instructions. However, this might create a lot of noise, especially if there are a lot of large blocks within the same function. Alternatively, we can consider large basic blocks in a *function context* by calculating the *average number of instructions* per basic block, relative to the number of basic blocks in the function:

```
# instructions in the function
------------------------------
# basic blocks in the function
```

The average number of instructions is high if a function contains only a single large basic block or if a function contains several larger blocks. This way, we can again pinpoint complex code on the function level.

In Binary Ninja, we implement this as follows:

```
def calc_average_instructions_per_block(function):
    # number of basic blocks
    num_blocks = len(function.basic_blocks)
    # number of instructions
    num_instructions = sum(
        [b.instruction_count for b in function.basic_blocks])

    return num_instructions / num_blocks
```

Similar to the previous heuristic, we can sort all functions based on their average basic block size and *print* the *upper* 10%.

```
def find_large_basic_blocks(bv):
```

```python
    # sort functions by average basic block size
    sorted_functions = sorted(bv.functions, key=lambda x: calc_average_instructions_per_block(x))

    # bound to print only the top 10%
    bound = math.ceil(((len(bv.functions) * 10) / 100))
    # print top 10% (iterate in descending order)
    for f in list(reversed(sorted_functions))[:bound]:
        print(f"{hex(f.start)}: {math.ceil(calc_average_instructions_per_block(f))}.")
```

### Instruction Overlapping

Up until now, the developed heuristics aimed at detecting complex code. However, sometimes code obfuscation tries to confuse disassemblers by introducing opaque control transfers to addresses that are *in the middle* of valid instructions. This way, the disassembler does not know how to proceed and build the control-flow graph, since two instructions *overlap*. In non-obfuscated code, this can also happen in cases where the disassembler mistakenly interprets data as code, therefore creates *meaningless* disassembly.

To detect instruction overlapping, we can simply walk over all instructions and store the addresses of *all* bytes that belong to the corresponding instruction in a set. If we check before inserting a byte, whether it is already part of the set, we know that instructions do not overlap. Otherwise—if an address is already in the set before insertion—we found two instructions overlap. In this case, we can print the function the instruction belongs to. If we want to go a step further, we can also highlight the corresponding instructions in the disassembler.

Compared to the other heuristics in which we walked over all functions and interacted with the graph API, we now have to walk over all function bytes in the program, making the implementation and runtime overhead more expensive (we omit the code listing for readability). However, in practice, the overhead is still acceptable for large binaries (up to several minutes).

## Evaluation

To get a feeling on how good these heuristics work, we will evaluate them on three binaries: two malware samples and the Windows kernel. Intuitively, we expect to find obfuscated code as well as other interesting program logic. So, let's dig in and have a look how the heuristics perform on the individual samples.

### Emotet

The Emotet sample sample uses a custom implementation of control-flow flattening. While most of it functions are obfuscated, the sample contains also non-obfuscated code.

```
Cyclomatic Complexity
Function 0x4063f0 (sub_4063f0) has a cyclomatic complexity of 76.
Function 0x4012a0 (sub_4012a0) has a cyclomatic complexity of 36.
Function 0x405800 (sub_405800) has a cyclomatic complexity of 35.
Function 0x402b60 (sub_402b60) has a cyclomatic complexity of 35.
Function 0x409e20 (sub_409e20) has a cyclomatic complexity of 31.
Function 0x404f50 (sub_404f50) has a cyclomatic complexity of 29.
Function 0x40a4b0 (sub_40a4b0) has a cyclomatic complexity of 27.
Function 0x402210 (sub_402210) has a cyclomatic complexity of 26.
Function 0x4025a0 (sub_4025a0) has a cyclomatic complexity of 24.
Function 0x40a9d0 (sub_40a9d0) has a cyclomatic complexity of 22.
Function 0x409530 (sub_409530) has a cyclomatic complexity of 22.
[snip]
```

We see that one function is significantly more complex than all others; the others, however, are in a comparable dimension. Most of the functions (including the most complex one) are obfuscated via control-flow flattening; the functions that are not obfuscated implement some sequential dispatching logic.

```
Large Basic Blocks
Basic blocks in function 0x405e40 (sub_405e40) contain on average 19 instructions.
Basic blocks in function 0x405d20 (sub_405d20) contain on average 19 instructions.
Basic blocks in function 0x405e00 (sub_405e00) contain on average 16 instructions.
Basic blocks in function 0x405c90 (sub_405c90) contain on average 13 instructions.
Basic blocks in function 0x404ee0 (sub_404ee0) contain on average 13 instructions.
Basic blocks in function 0x405ea0 (sub_405ea0) contain on average 12 instructions.
Basic blocks in function 0x405db0 (sub_405db0) contain on average 12 instructions.
Basic blocks in function 0x405d70 (sub_405d70) contain on average 12 instructions.
[snip]
```

The large basic block heuristic does not produce any anomaly; all the values are in a comparable range. If we dig into the individual functions, we see that most of them share a similar structure: They are single-basic block functions that set initialize memory values. If we analyze the functions' usages, we learn that they are called from obfuscated functions to update the control-flow flattening state:

```c
int32_t sub_405c90() {
    int32_t ecx
    int32_t var_4_4 = ecx
    int32_t var_4 = 0x2224
    int32_t eax
    int32_t edx
    edx:eax = mulu.dp.d(0xcccccccd, 0x7acbf5eb)
    return ((edx u>> 6) - 0xc617) ^ 0x1882c5b
}
```

The instruction overlapping heuristic does not find any overlapping instructions. So far, both complexity heuristics pinpoint different code locations, effectively clustering the obfuscated code in complex state machines and helper functions.

### Adylkuzz

The Adylkuzz sample is protected by the VMProtect obfuscator. VMProtect is a virtualization-based obfuscator that heavily relies on opaque predicates thwarting disassemblers with instruction overlapping/disalinged control flow. Furthermore, it uses dead code to bloat its code size.

```
Cyclomatic Complexity
Function 0x70c597 (sub_70c597) has a cyclomatic complexity of 79.
Function 0x5c0821 (sub_5c0821) has a cyclomatic complexity of 76.
Function 0x6ff664 (sub_6ff664) has a cyclomatic complexity of 75.
Function 0x70b66e (sub_70b66e) has a cyclomatic complexity of 74.
Function 0x6ff79b (sub_6ff79b) has a cyclomatic complexity of 74.
Function 0x70feea (sub_70feea) has a cyclomatic complexity of 73.
Function 0x709927 (sub_709927) has a cyclomatic complexity of 73.
Function 0x5c36db (sub_5c36db) has a cyclomatic complexity of 60.
Function 0x6fefe5 (sub_6fefe5) has a cyclomatic complexity of 41.
Function 0x5c0bfc (sub_5c0bfc) has a cyclomatic complexity of 31.
Function 0x7086ab (sub_7086ab) has a cyclomatic complexity of 28.
Function 0x703be1 (sub_703be1) has a cyclomatic complexity of 28.
Function 0x70dd19 (sub_70dd19) has a cyclomatic complexity of 27.
Function 0x6f9fd3 (sub_6f9fd3) has a cyclomatic complexity of 27.
Function 0x6e53bf (sub_6e53bf) has a cyclomatic complexity of 27.
Function 0x6fbbed (sub_6fbbed) has a cyclomatic complexity of 26.
Function 0x70ac91 (sub_70ac91) has a cyclomatic complexity of 24.
[snip]
```

Most of the identified functions with a high cyclomatic complexity are garbage, since the disassembler produces an invalid disassembly due to overlapping instructions. However, the few valid functions initialize the VM and import the hidden API calls (via `LoadLibraryA`).

```
Large Basic Blocks
Basic blocks in function 0x70d941 (sub_70d941) contain on average 112 instructions.
Basic blocks in function 0x6baa2e (sub_6baa2e) contain on average 104 instructions.
Basic blocks in function 0x5becad (sub_5becad) contain on average 67 instructions.
Basic blocks in function 0x6ba981 (sub_6ba981) contain on average 59 instructions.
Basic blocks in function 0x5b98a3 (sub_5b98a3) contain on average 49 instructions.
Basic blocks in function 0x5b37a2 (sub_5b37a2) contain on average 49 instructions.
Basic blocks in function 0x6f836d (sub_6f836d) contain on average 48 instructions.
Basic blocks in function 0x5b7ac6 (sub_5b7ac6) contain on average 48 instructions.
```

```
Basic blocks in function 0x5ba8e9 (sub_5ba8e9) contain on average 46 instructions.
Basic blocks in function 0x5b1236 (sub_5b1236) contain on average 45 instructions.
Basic blocks in function 0x6fe79c (sub_6fe79c) contain on average 44 instructions.
[snip]
```

For the large basic block heuristic, the results are a bit different: While the functions with the highest scores are also garbage, many other identified functions implement the instruction semantics handler of the virtualization-based obfuscation (VM handler) within a single basic block.

```
Overlapping instructions in function 0x5bedbd (sub_5bedbd).
Overlapping instructions in function 0x5bf05a (sub_5bf05a).
Overlapping instructions in function 0x5bf4d6 (sub_5bf4d6).
Overlapping instructions in function 0x5bf7de (sub_5bf7de).
Overlapping instructions in function 0x5c0125 (sub_5c0125).
Overlapping instructions in function 0x5c01b5 (sub_5c01b5).
Overlapping instructions in function 0x5c0363 (sub_5c0363).
Overlapping instructions in function 0x5c03bf (sub_5c03bf).
Overlapping instructions in function 0x5c0821 (sub_5c0821).
Overlapping instructions in function 0x5c0bfc (sub_5c0bfc).
Overlapping instructions in function 0x5c1003 (sub_5c1003).
Overlapping instructions in function 0x5c1447 (sub_5c1447).
Overlapping instructions in function 0x5c1563 (sub_5c1563).
[snip]
```

As indicated by the results of the other heuristics, the instruction overlapping heuristic identifies a magnitude of functions that contain overlapping instructions and produce incorrect disassembly. While many functions can be immediately ignored since they are only garbage, some functions *might* contain valid instructions; in those functions, only parts of the disassembly are broken. However, by purely static analysis, it is hard to tell if the instructions are valid or not.

In summary, we can say that all heuristics identified garbage code. If we remove all functions that are identified by the instruction overlapping heuristic from the results of the other heuristics, we can again group the identified functions into two categories: The complex functions perform VM-related initialization routine or decrypt API calls, while the large basic block heuristic pinpoints the VM handlers.

### Windows Kernel

After evaluating the heuristics on obfuscated malware samples, let us have a look on how they work on a commercial real-world application: the latest version of the Windows kernel, `ntoskrnl.exe` (11 MiB, MD5: `c9d2f9ada42052c2a34cb3e0743caf48`). While most parts of the Windows kernel are not obfuscated, it contains an anti-tamper protection called PatchGuard which employs a lightweight obfuscation by Microsoft's in-house obfuscation framework Warbird.

```
Function 0x140a1bee4 (sub_140a1bee4) has a cyclomatic complexity of 2964.
Function 0x1409f7010 (FsRtlMdlReadCompleteDevEx) has a cyclomatic complexity of 2371.
Function 0x1403da6d0 (sub_1403da6d0) has a cyclomatic complexity of 1506.
Function 0x1405d3a40 (PropertyEval) has a cyclomatic complexity of 718.
Function 0x14069fcf0 (NtSetInformationProcess) has a cyclomatic complexity of 642.
Function 0x14068ecb0 (ExpQuerySystemInformation) has a cyclomatic complexity of 435.
Function 0x14066bc78 (SPCall2ServerInternal) has a cyclomatic complexity of 414.
Function 0x14022fba0 (MmCheckCachedPageStates) has a cyclomatic complexity of 318.
Function 0x140a0b0fc (sub_140a0b0fc) has a cyclomatic complexity of 281.
Function 0x1406a9da0 (NtSetSystemInformation) has a cyclomatic complexity of 274.
Function 0x140675a50 (IopParseDevice) has a cyclomatic complexity of 271.
```

We notice that the first three functions have a very high complexity; afterward, the values drop quickly. Public research about PatchGuard's internals are not very well documented, so it's hard to tell what these functions do. However, the first three functions are definitely related to PatchGuard: The first one is related to PatchGuard's initialization routine, while the second (`FsRtlMdlReadCompleteDevEx`) is known to perform some PatchGuard related checks. The third function is called by `KiFilterFiberContext`, which is also a known PatchGuard function.

```
Large Basic Blocks
Basic blocks in function 0x140a62e04 (SepInitSystemDacls) contain on average 491 instructions.
Basic blocks in function 0x1403e8f74 (SymCryptSha256AppendBlocks_ul1) contain on average 236 instruc
Basic blocks in function 0x1404bdc60 (HalpRestoreHvEnlightenment) contain on average 147 instruction
Basic blocks in function 0x140a55cb8 (MiInitializeDummyPages) contain on average 133 instructions.
Basic blocks in function 0x1409a7744 (HalpBlkInitializeProcessorState) contain on average 103 instru
[snip]
```

The functions with the largest average basic block size do not seem to be related to PatchGuard. Instead, based on their function names, they initialize different data structures (`SepInitSystemDacls`, `HalpRestoreHvEnlightenment`, `MiInitializeDummyPages` and `HalpBlkInitializeProcessorState`) or implement cryptographic algorithms (`SymCryptSha256AppendBlocks_ul1`).

While the instruction overlapping heuristic also pinpoints some functions, they can be ignored: The results are all false positives, since the disassembler wrongly interprets data as code.

Overall, we can summarize the experiments and say that the heuristics pinpoint all kinds of interesting code parts, no matter if the code is obfuscated, implements a complex state machine, initialization routines or cryptographic algorithms. As we have seen for Emotet and the Windows kernel, it can be beneficial to pay special intention to peaks in the values. Furthermore, we observed that heuristics often produce different results, but can also identify the same code locations.

## Setting the Scene

In a previous blog post, we introduced a heuristic to detect control-flow flattening and state machines in binaries. This time, we developed more generic heuristics to pinpoint code obfuscation and complex code. While the heuristics are different in their nature, they all are easy to implement, cheap to calculate and exploit characteristics that are shared by obfuscated as well as interesting non-obfuscated code.

As part of our day-to-day reverse engineering, we can use these heuristics to get an initial overview over the binary; we can spot which code areas might be worth a closer look. If the code is obfuscated, we then can try to understand the context in which the obfuscation is embedded. Afterward—if we want to better understand the obfuscated code—we may look for patterns and come up with a strategy to automatically remove the obfuscation (as we'll do in my code deobfuscation training classes).

## Contact

For questions, feel free to reach out via Twitter @mr_phrazer, mail tim@blazytko.to or various other channels.