Situational Awareness

Reverse Engineering Algorithms

Jasenko Hosic
Matt Martin

March 5, 2013

Contents

Introduction

This document details the algorithms developed to identify various code blocks within disassembled code. The algorithms are general, but are implemented specifically for use with the Oxide disassembler. In some cases, references are made to the Python dictionary created by the Oxide disassembler module.

Each section will include sample code in order to illustrate the conceptual side of the algorithms. Extra code samples were created to test the validity of the algorithms. These testing samples have been placed in the appendix at the end of this document. Every section will outline the results of testing the heuristics against the code samples. Lastly, the algorithms were tested on open source projects found on the internet. The results were manually verified with the source code and the results were noted in the appropriate sections.

Conditional Statements

The different types of conditional statments are identified in similar ways. All loops in C an C++ seem to have the same structure on an assembly level, whether they be *for*, *while*, or *do... while*. Looping can be accomplished in an obscure manner through recursion, and while this is not directly recognized as a loop, it is noted as a recursion. The heuristics used to identify recursive code blocks are discussed in a later section.

If Statements

If statements are quite simple to identify. The general structure of an if statement in C/C++ is:

Algorithm 1 Generic If Statement

```
if condition then
    Action 1
else if condition2 then
    Action 2
    ...
else
    Do this if no other condition is true
end if
```

In assembly, if statements follow a similar convention. In each case, there must be a conditional statement to check if the actions will execute. If the condition is deemed *true*, the next statement is a jump to the code containing the action of the if statement (an address *greater* than that of the conditional statement). If it is false, the code drops through to code that is after the if statement in question. Alternatively, the assembly code may convert the condition to the *opposite* of what the programmer originally wrote in C/C++. In such a case, the *true* evaluation of the condition results in a jump statement that skips the actions of the if statement. This address is still *greater* than the address of the conditional statement. In either case, an if statement is represented as a conditional statement, followed by some sort of jump to an address that is greater. The following code samples illustrate these cases.

Algorithm 2 Example If Statement

```
if $i > 29$ then
    $g \leftarrow i + 5$
else
    $g \leftarrow 0$
end if
```

The above example is reproduced with assembly code below. Case 1 shows the instance where the conditional check is the same as that in the C/C++ code. Case 2 has the opposite conditional check. These variants arise from compiler differences as well as variations in optimization levels.

Algorithm 3 Case 1: If Statement Checking For Same Condition

```
4199112: cmp $ebp+4, 1Dh        //compare the variable to 29
4199116: jg 4199148             //jump to 4199148 if greater
4199118: mov $eax, $ebp+4       //The ELSE block
4199121: add $eax, 5
4199124: mov $eax, $ebp+8
...
4199148: mov $ebp+8, 0          //The IF block
...
```

Algorithm 4 Case 2: If Statement Checking For Opposite Condition

```
4199112: cmp $ebp+4, 1Dh        //compare the variable to 29
4199116: jle 4199148            //jump to 4199148 if less than or equal to
4199118: mov $ebp+8, 0          //The IF block
...
4199148: mov $eax, $ebp+4       //The ELSE block
4199151: add $eax, 5
4199154: mov $eax, $ebp+8
...
```

Testing If Statements

In order to test the validity of the assertions in the previous section, several test programs were created and manually verified. The relevant portions of the code for these tests and the results are reproduced in the appendix.
All of the if statements are appropriately identified in each of the code samples in appendix A. Test 5 did not have an If statement, and a false positive was not given by the algorithm. In test 1 (and similar tests), if the print statements are removed and static values are set instead within the action of the if statements, higher optimization levels remove the if statements entirely. This is acceptable as the intent is to identify actual behavior, not original code. It should be noted that more than just the user-defined if statements are identified as the compiler adds extra code

Loops

The process used to identify loops is quite similar to the one formulated in the if statements section. This time, instead of checking for jumps to addresses ahead of the conditional statement, the heuristic checks for jumps to previous addresses. This jump can either be the immediate jump after the conditional check (the superceding jump), or a jump that is between the superceding jump and its target address. As with if statements, the several cases exist, but their similarity is simple and requires no explanation. A sample loop is shown below in C code as well as assembly.

Algorithm 5 Basic Loop In C

```
int i = 0;
do
{
    printf("testing... %d\n", i);
    i++;
}while(i < 100);
```

Algorithm 6 Basic Loop In Assembly

```
4199105: mov eax, ebp+4
4199108: mov esp+18h+14, eax
4199112: mov esp+18h+18, "testing... %d\n"
4199119: call printf
4199124: lea eax, ebp+4
4199127: inc eax
4199129: cmp ebp+4, 63h  //the conditional
4199133: jg 4199137      //the jump right after the conditional
4199135: jmp 4199105     //the jump back we are looking for
4199137: retn      //end of loop
```

Testing Loops

All of the tests were used to check the correctness of the loop heuristic. The results were unanimously stellar. All of the loops were recognized correctly and no false positives were given. Nesting did not present a problem as long as jumps and addresses were checked in an outside-in fashion (as with parsing curly braces).

API Calls

An API, or Application Programming Interface, is a specification for software to communicate with other software. This can take the form of accessible pre-existing libraries. When a program makes a call to a function that exists in a library (such as the *C++ Standard Template Library*), this instruction can be identified. Calls to functions such as *printf*, *fprintf*, and *cout* are important to identify as their functionality is well documented. Knowing where these calls are being made can greatly elucidate the flow of the whole program.

Calls to functions (both user-defined and otherwise) in the Oxide disassembler are shown as:

```
call ADDRESS_LOCATION_HERE
```

Within the Oxide API, there exists a function that can be used to match all the call addresses to the names of their functions. The *map_calls* function produces a dictionary of addresses and function names. User-defined functions are named as generic internal functions while actual API calls are named properly. The addresses in the *map_calls* function are the addresses of the lines where the call occurred. There is, however, an offset caused by the differences in real and virtual addresses. This offset can be calculated with the following Oxide-specifc Python function:

```
def convert_rva_offset(oid, rva):
    header = api.get_field("object_header", oid, "header")
    try:
        rva = int(rva)
    except:
        raise ShellSyntaxError("Unrecognized address %s" % rva)
    return header.get_offset(rva)
```

In summation, when a call is encountered, the address of the line (with the converted real/virtual offset calculated) maps to the name of function in the dictionary produced by the *map_calls* module.

As this is primarily the work of built-in Oxide modules, extensive testing was not necessary. Each time this method was used, the calls were marked correctly. To date, we have found no errors with this algorithm.

There are many types of API calls that reveal essential program functionality. Looking for these is a matter of hard coding a search for the function names. Lists of functions for various purposes (such as registry manipulation, printing, network access, etc) are provided in Appendix B.

Exception Handling

Exceptions in code are special circumstances which require the normal flow of a program to be subverted in order to process something else. For instance, consider the following function that divides two numbers:

Algorithm 7 Division Function In C

```
double divide(int x, int y)
{
    return x / y;
}
```

In instances where y is 0, the result cannot be computed. An exception needs to be made in this special circumstance. The result should not be computed and an error should be given to the user. Test 7 of Appendix A shows a corrected version of this function with proper exception handling.

Finding an occurrence of exception handling in code is a matter of matching the correct API calls. The way that exceptions are handled is different from compiler to compiler. In fact, it is one of the many things that make up a *compiler signature* (discussed in a later section).

The following API calls were found on code samples such as Test 7 of Appendix A using Visual Studio, G++, and Dev-C++(Cygwin):

Algorithm 8 Exception Handling Functions

```
raiseexception
_cxxthrowexception
__cxa_allocate_exception
__cxa_end_catch
__cxa_get_exception_ptr
__cxa_throw
__cxa_begin_catch
setlasterror
getlasterror
```

User-defined Functions

User-defined functions work very much like the API calls discussed previously. The primary difference is that the name of the function can not be found using the same techniques. When using the *map_calls* module, the returned dictionary contains a system_calls section and an internal_functions section. The system_calls entries are the API calls while the internal_functions are the user-defined procedures. Since the names given to the functions by the programmer cannot be retrieved, the procedures are given arbitrary names in the form of "internal_function_####".

Identifying Recursion

Finding recursive calls is perhaps the easiest task of all. A recursive function, by definition, calls itself. The cycle is then broken once a base case is reached. Technically, however, infinite recursion is completely valid. As such, the only thing necessary to identify recursion is to find a function calling itself. Within the module that contains these algorithms, functions are tagged as recursive with a "yes" string in the semantics dictionary.

Encryption Schemes

For all of the cryptographic algorithms, only a select set of the instructions were looked at to find a positive detection. These instructions were as follows:

```
AND
OR
XOR
NOT
*Any jump instruction*
ROR
ROL
SHR
SHL
CALL
INC
DEC
```

However from these functions, operations used to clear registers were removed from those that were examined. Examples of these operations were those such as xoring a register with itself, or anding a register with 0. These were taken out to reduce the number of false positives.

All of these algorithms return a probability of there being a correct implementation of the given cryptographic algorithm within the binary file. This is calculated by taking the quotient of the parts of the algorithm found and the number of total parts that are being looked for. At the present, all parts of the algorithm are weighted the same. At a later time, these weights may change based on testing.

SHA1

The first algorithm that the crypto plugin looks for is SHA-1. This algorithm is composed of a pre-processing stage followed by 4 distinct rounds. The algorithm looks for these 5 parts as well as looking for constants that are used by this algorithm and initialization values. An explanation of how each of the parts are identified is shown below.

Pre-Processing:

To positively identify the preprocessing stage of the SHA-1 algorithm it must contain both of the following features. The first of which is three xor operations in a row. Note that these must show up in the new list of instructions that only contains the above instructions. The second feature is a rotate left by 1, or rotate right by 31. These are equivilent instructions. Another option is usings both a shift left and shift right and anding the two results together which has been seen in some versions.*** The second feature must come after the first.

**** Note that everywhere in this document where a rotate right or left is looked for that it could also be replaced by these equivilent values.*

Rounds:

Each of the 4 rounds have the same structure. They all first start with a rotate left by 5. They all end with a rotate right by 2. Each have a set of instructions between these two rotates that will be explained below.

Round 0:

The 0th round has one of the following series of instructions which are different only by order that the operations are executed or logical equivilence. The same is true for the following rounds as well.

```
and not and or
not and and or
xor and xor
```

Round 1:

```
xor xor
```

Round 2:

```
and and and or or
and and or and or
or and and or
```

Round 3:

```
xor xor
```

Constants:

The constants that this algorithm were looked for within move instructions as well as stored elsewhere in the program. The constants that were looked for are as follows:

```
5a827999
6ed9eba1
8f1bbcdc
ca62c1d6
```

Initializations:

The initializations that this algorithm were looked for within move instructions as well as stored elsewhere in the program. The constants that were looked for are as follows:

```
67452301
efcdab89
98badcfe
10325476
c3d2e1f0
```

SHA256

For identifying this algorithm 5 things are looked for. Initializations, Pre-Processing, rounds, big sigmas, and small sigmas. Explanations of each of these parts are explained below.

Initializations:

These set of initializations were looked for in the instructions as well as elsewhere in the binary.

```
6a09e667
bb67ae85
3c6ef372
a54ff53a
510e527f
9b05688c
1f83d9ab
5be0cd191
```

Pre-Processing:

This set of rotates and shifts must be found

```
shift right 3
shift right 10

rotate right 17
rotate right 19
rotate right 7
rotate right 18
```

Rounds:

These two sets of instructions must be found.Note there are multiple forms for each set.

1

```
and not and xor
not and and xor
```

2

```
        and and and xor xor
        not and xor and xor
```

Big Sigmas:

This string of rotates must be found.

```
  rotate right 2
  rotate right 13
  rotate right 22
  rotate right 6
  rotate right 11
  rotate right 25
```

Small Sigmas:

This string of rotates and shifts must be found

```
      rotate right 7
      rotate right 18
      shift right 3
      rotate right 17
      rotate right 19
      shift right 10
```

AES

Compiler Signatures

Algorithm Identification

While it is helpful to find semantic descriptions of programs through miscellaneous identifiable pieces in disassembled code, it is far more beneficial to be able to find the occurrence of entire algorithms. In order to find algorithms of all sorts, the methods of identifying them had to be fairly generic. Since algorithms range in complexity, it would be impractical to employ the same process in all cases. The key components of an encryption algorithm such as SHA1, for instance, and the essential instructions of the max function are quite different.

As a result, a three-pronged (or layered) scheme was devised to allow for versatility based on varying needs. Each layer of the identification process increases in complexity and offers more options. In order to specify which algorithm to identify, a standardized template file is utilized (located in *plugins/templates*). In the template file, the various layers and their details are written. The algorithm reads the template file and sifts through the assembly code, searching for the specified instruction sets. A sample template file for an encryption algorithm is reproduced in Appendix D, and its general structure is indicative of the proper formatting for all template files. The following is an explanation of the three-layered process that occurs once a template file is provided to the *AlgIdent* plugin.

Layer 1

Layer 1 is primarily a string matching algorithm. It identifies the commands that are part of the algorithm and passes them to the subsequent layer. It is broken down into *Include* and *Exclude*, which are both further broken down into *Simple* and *Complex*. The *Include* portion of the layer denotes the instructions that are part of the algorithm being identified, while the *Exclude* instructions are those that should be ignored. The items placed into the *Simple* portions can only be comma-delimited lists of single instructions that are independent of each other and stripped of their operands. Any set of instructions after the *Complex* label, however, can contain specific operands and terminal variables.

Terminal variables act much like wildcard characters. They are strictly capital letters that represent an operand whose value is unimportant to the user. If the same terminal variable is used multiple times in a line, the only commands that will match the line are those that contain the same value in all instances of that terminal variable.

Algorithm 9 Terminal Variable Example

```
xor A A
```

In the above example of a terminal variable, only instructions that attempt to xor a value (whether it be any number or any variable) with itself will be a match.

Layer 1 also contains code to pair API calls to their respective addresses so that the template file can accept the full name of a function rather than its memory address.

Layer 2

Layer 2 is far more complex than its predecessor. Its structure is broken down into a user-defined quantity of grammar rules. The grammar rules are applied to the extracted commands passed in from the completed execution of the previous layer in order to identify combinations of instructions. Each rule is given its own name by the user for easy recognition during the output. Each rule contains two sections, the terminals required and the structure of the rule itself. If the grammar needs to represent the rule with the use of three terminals, for instance, the values A, B, and C, can be placed under the *Terminals:* label. Under the *Rule:* heading, the order of the instructions determines what the algorithm attempts to identify. Just like in the previous layer, any reuse of a terminal within the same line or across multiple lines means that in order for the line(s) to be matched in the disassembled code, the various values must be identical in each of those locations.

For clarity, consider the following rule:

Algorithm 10 Rule Example

```
cool_rule:
Terminals:
          A, B, C
Rule:
          xor A B
          xor 1 C
          xor C C
```

In this scenario, terminals A, B, and C are used as wildcards. The algorithm will search the set of included lines (from the first layer) until an xor is encountered. The operands will be temporarily assigned to terminal variables A and B, respectively. The algorithm then continues forth from that line and seeks for an xor whose first operand is 1 and second operand is any value. If that is located, the process is repeated, this time looking for an xor statement whose operands are both identical to the second operand of the previously matched line. If the operands are different, the line is ignored.

From the example, three important details become evident. First, this process is heavily nested and therefore requires a lot of computational resources. This is difficult to avoid due to the large number of variations that are possible. The preprocessing done in the first layer and its exclusion of certain instructions reduces the search space and runtime considerably. Second, the more specificity included in a template file when representing an algorithm, the faster the algorithm will execute. Lastly, since the process must utilize a sequential search and there is no limit to the number of possible unwanted instructions located between two desired commands, two lines that are implausibly far apart may be incorrectly identified as belonging to the same algorithm. While not without its flaws, a solution to this issue is contained within the third layer.

Layer 2 can still make use of the API call component of the first layer. For example, finding a chain of calls that open, delete, and close a registry key becomes trivial. The template file that achieves this result is reproduced below.

Algorithm 11 API Call Template Example

```
Layer1:
    Include:
        Simple:
        Complex:
                call RegOpenKeyA
                call RegDeleteKeyA
                call RegCloseKey

    Exclude:
        Simple:
        Complex:
                xor A A  ;unnecessary, but irrelevant

Layer2:
    Registry_Ops:
        Terminals:
        Rule:
                call RegOpenKeyA
                call RegDeleteKeyA
                call RegCloseKey
```

Layer 3

Layer 3 is, at its core, an error correction scheme. The layer receives all possible rule matches found in the previous layer, attempts to identify those that have unlikely gaps between instructions, and removes them. This is done by using a maximum span constraint that isolates only the identified algorithms that start and end within a certain number of lines or addresses. The constraint is defaulted to 50 lines, but can be specified by the user when executing the *AlgIdent* plugin. The application of this error correction technique greatly reduces the number of false-positives when attempting to locate the existence of specific algorithms.

Equivalency Database

There are many ways to write the same algorithm. Likewise, even if the code remains identical, there are multiple ways to compile it. As such, it is difficult to determine exactly which representation of an algorithm to needs to be defined within a template file in order to accurately prodce a match. In order to combat this issue, an equivalency database was established that would compliment the *AlgIdent* plugin.

When the template file is read in by the plugin, the data is passed to the equivalency database. Each line in a rule is examined and expanded into all other reasonable equivalent statements. If the algorithm identification process then encounters a line that is equivalent but different from the one specified by the template file, the match can still made.

Rather than attempting to maintain a gigantic list of all possible equivalencies, the algorithm just contains functions that expand certain instructions. For instance, the equivalency database contains a *ror* function which expands any rotate-right bitshift into all possible identical statements while preserving its operands. Although the database of functions is not fully fleshed out, it is easy to make additions. As more equivalencies are discovered and required, updates will be made to the plugin.

Appendix A - Code Samples

Note: The code in several of these tests include specific integer values or odd strings. These values aid in finding the code snippets within the assembly code

Algorithm 12 Test 1 - If Statements

```
void main()
{
    int a= 1, b = 2, c = 3;
    if(a == b)
    {
        printf("if 1");
    }
    else if(a >= b)
    {
        printf("if 2");
    }
    else if(a < b && a > c)
    {
        printf("if 3");
    }
    else
    {
        printf("if 4");
    }
    printf("testing testing");
    if(c > a)
        printf("\n");
    printf("second test");
    if(b < c)
        printf("\n2");
    printf("blah");

}
```

Algorithm 13 Test 2 - File IO

```
void main()
{
    FILE *in, *out;
    char name[10];
    int value;
    in = fopen("input.txt", "r");
    if (in == NULL)
    {
        fprintf(stderr, "Can't open input file.\n");
        return;
    }
    out = fopen("output.txt", "w");
    if (out == NULL)
    {
        fprintf(stderr, "Can't open output file.\n");
        return;
    }
    while(fscanf(in, "%s %i", name, &value) != EOF)
    {
        fprintf(out, "%i,%s\n", value, name);
    }
    fclose(in);
    fclose(out);
}
```

Algorithm 14 Test 3 - System Calls

```
int main ()
{
    int i;
    printf ("Checking if processor is available...");
    if (system(NULL)) puts ("Ok");
    else exit (1);
    printf ("Executing command ls...\n");
    i=system ("ls");
    printf ("The value returned was: %d.\n",i);
    return 0;
}
```

Algorithm 15 Test 4 - Nested Conditionals

```
void main()
{
    printf("START!\n"); doIfs(); doLoops(); doBoth();
}
void doIfs()
{
    char name[20];
    int age;
    printf("Enter your name: ");
    scanf("%s", &name);
    printf("Enter your age: ");
    scanf("%i", &age);
    if(age >= 21)
    {
        printf("you can drink!\n");
        if(name[0] == 'j' || name[0] == 'J')
        {
            printf("your name starts with an awesome letter!\n");
            if(age == 22)
            {
                printf("Cool age, bro\n");
            }
        }
    }
}
void doLoops()
{
    int i = 13, j = 0;
    for(i; i > 0; i--)
    {
        for(j=0; j< i; j++)
        {
            printf("* ");
        }
        printf("\n");
    }
}
void doBoth()
{
    int num, i;     printf("Enter number: ");
    scanf("%i", &num);
    for(i = 0; i < num; i++)
    {
        if(i == 7) { printf("NO!\n"); }
        else { printf("YES!\n"); }
    }
}
```

Algorithm 16 Test 5 - Write Bit Sequence

```
void main()
{
    int counter = 0;
    FILE * write = fopen("test.jpg", "wb");
    for(counter; counter < 1000; counter++)
    {
        fprintf(write, "10001011011101");
    }
    fclose(write);
}
```

Algorithm 17 Test 6 - Passing Parameters

```
void main()
{
    display();
    mystery(19, 18, "Hello, World!");
    int k = 23;
    int p = 89;
    char text[] = "TEXT!";
    mystery(p, k, text);
}
void display()              //Testing no parameters
{
    printf("AHHH!\n");
}
double mystery(int x, int y, char temp[])
{
    if(x*y > 26)
    {
        printf("%s", temp);
    }
    blah(x, y);
}
void blah(int x, int y)
{
    if(x+y > 21)
        display();
}
```

Algorithm 18 Test 6 - Passing Parameters

```
#include <iostream>
using namespace std;
double division(double, double);
int main()
{
    double x, y;
    cin >> x >> y;
    try
    {
        cout << "Answer: " << division(x, y) << endl;
    }
    catch(int z)
    {
        if(z==1669)
        {
            cout << "ERROR 1669: Can't divide by zero!" << endl;
        }
    }
    return 0;
}

double division(double x, double y)
{
    if(y==0)
        throw 1669;
    return x/y;
}
```

Appendix B - API Calls

Algorithm 19 Network Access Functions

```
wsacleanup
wsastartup
inet_addr
htons
socket
wsagetlasterror
bind
listen
accept
inet_ntoa
recv
recvfrom
closesocket
send
sendto
tcpseqmypspec
send_ip_packet_eth
winsock
traceroute
winpcap
tcpip
```

Algorithm 20 File I/O Functions

```
createfilea
deltefilea
readfile
writefile
flushfilebuffers
```

Algorithm 21 Registry Manipulation Functions

```
regopenkeya
regqueryvalueexa
regclosekey
regcreatekeya
regsetvalueexa
regdeletekeya
regenumkeya
regcreatekeyexa
regdeletevaluea
```

Algorithm 22 Printer Access Functions

```
startdocprintera
startpageprinter
enddocprinter
closeprinter
writeprinter
endpageprinter
enumprintersa
openprintera
```

Algorithm 23 Exception Handling Functions

```
raiseexception
_cxxthrowexception
__cxa_allocate_exception
__cxa_end_catch
__cxa_get_exception_ptr
__cxa_throw
__cxa_begin_catch
setlasterror
getlasterror
```

Appendix C - Other Testing Repositories

The following are links to open source projects that were used to test the validity of the algorithms contained in this document. Various portions of the code in these files were hand-matched with the output of these algorithms. Exceptions, conditionals, API calls, etc. were all checked throughout with great success.

```
Pacman_QT: http://sourceforge.net/projects/pacmanlike/
Chat: http://sourceforge.net/projects/chat
MD5Deep/HashDeep: http://sourceforge.net/projects/md5deep/
MTPaint: http://sourceforge.net/projects/mtpaint/
```

Appendix D - Sample Algorithm Identification Template File

```
Layer1:
    Include:
        Simple:
                or, xor, not, and, jmp, call, dec, inc, shr, shl, ror, rol, je,
                jg, jge, jl, jle, jnc, jnz ;previous line continued
        Complex:

    Exclude:
        Simple:
        Complex:
                 xor A A

Layer2:
    SHA1_Pre:
        Terminals:
                   A, B, C, D, E
        Rule:
              xor A B
              xor A C
              xor A D
              shr E 31
    SHA1_Round0:
        Terminals:
                   A, B, C, D, E, F, G,H
        Rule:
              not A
              and A D
              and B C
              or A B
```

```
;Continued
                    shl E 5
                    shr F 27
                    or F E
                    shr G 2
                    shl H 30
                    or H G
      SHA1_Round1:
            Terminals:
                           A, B, C, D, E, F, G
            Rule:
                    shl E 5
                    shr F 27
                    or F E
                    xor A B
                    xor A C
                    shr D 2
                    shl G 30
                    or G D
      SHA1_Round2:
            Terminals:
                           A, B, C, D, E, F, G,H,I
            Rule:
                    or A B
                    and A C
                    and D B
                    or A E
                    shl F 5
                    shr G 27
                    or G F
                    shr H 2
                    shl I 30
                    or I H

      SHA1_Round3:
            Terminals:
                           A, B, C, D, E, F, G
            Rule:
                    shr E 27
                    shl F 5
                    or E F
                    xor A B
                    xor A C
                    shr D 2
                    shl G 30
                    or G D
```