EVALUATION OF THE EFFICACY OF CONTROL FLOW OBFUSCATION AGAINST PROFILING AND INTELLIGENT STATIC ATTACKS

By

SRINIVASAN CHANDRASEKHARAN

A Thesis submitted to the Honors College
In Partial Fulfillment of the Bachelors degree
With Honors in
Computer Science
UNIVERSITY OF ARIZONA
DECEMBER 2003

Approved by

Dr. Saumya Debray

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for a degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Signed: ____

INTRODUCTION

In today's world of code tampering, code theft and software piracy, protecting one's intellectual property is of utmost importance for every programmer. Protecting one's intellectual property has come into focus once again with the introduction of architecture independent code format (Java bytecode [1]), and because of the emergence of reverse engineering tools such as decompilers [2, 3]. Programmers have played with the idea of encryption for many years and have succeeded in implementing the safest algorithms to protect their data. This is very safe when the attacker does not physically have the source code but can only execute the code. This means that the software runs on a remote server and consumers pay to use the software remotely. This is not economically viable for the consumer and for the software developer for large applications due to network constraints. The immediate solution for the software developer is to encrypt the source code and make it available to the consumer. This shall work only if the entire decryption, encryption and execution process happens in the hardware. But again in the execution process, if the intermediate code runs on a virtual machine interpreter then the code and can be intercepted and decompiled [4] which is the case for most architecture independent code. Therefore one is left with no other choice but to use architecture specific code. But that too can be reverse engineered with the help of architecture specific decompilers [2].

Code Obfuscation was introduced in the security and cryptographic communities to tackle the problem of attackers reverse engineering intermediate code to source code. Code Obfuscation is an approach whereby one transforms sensitive code to another form. This transformed code is behaviorally same as the original, but in the process of obfuscation the transformations results in intermediate code that is difficult for the attacker to understand even after decompiling it to source level. This is because obfuscation inserts extra computations into the original code so as to divert and disillusion the mind of the attacker by giving him a lot of options as to where the code might go next. Though this process results in software that is slower and larger than the original, it does to an extent protect the intellectual property of the software developer.

Formal Definition of Obfuscation

Given a set of obfuscating transformations T = {T_1 … T_n} and a program P consisting of source code objects {classes, methods, statements, etc.} {S_1 … S_k}, find a new program P^1 = {…, $S^1_j = T_i (S_j)$ …} such that:

- P^1 has the same observable behavior as P, which means that the semantics are preserved.
- The obscurity of P^1 is maximized, which means that reverse engineering and understanding P^1 will be more time consuming than reverse engineering and understanding P.
- The resilience of P^1 is maximized, where by the transformations cannot be undone through automatic tools.
- The stealth of each transformation is maximized while the cost (execution time/ space) incurred is minimized. [7]

Obfuscation can be mainly classified into two kinds

- Control Transformation Obfuscation
- Data Transformation Obfuscation

Control Transformation Obfuscation

We define Control Transformation Obfuscation as those obfuscation schemes that change the control flow of the program. These are mainly done through the use of Opaque Constructs. Opaque Predicates are those constructs that always have one value. We can consider them as Boolean constants. They are always either true or false.

Data Transformation Obfuscation

We define Data Transformation Obfuscation as those obfuscation schemes that alter the underlying data structure or the variables used in the program.

Our focus is to figure out the efficacy of obfuscated code through Control Transformations hence we shall limit our research to Buggy Code and Bogus Predicates. Both of the above two mentioned are control flow obfuscation schemes developed by the Sandmark team at the University of Arizona [6]. We shall test how the two obfuscation schemes fare against profiling and some intelligent static attacks namely, *Detection of Exception Handling Blocks* and *Dead Method Elimination*.

BACKGROUND

Compatibility of code across different platforms has become one of the most important aspects of software development today. Therefore we see the emergence of architecture independent code formats such as Java bytecode, C# etc. which give rise to the distribution of code in intermediate code format.

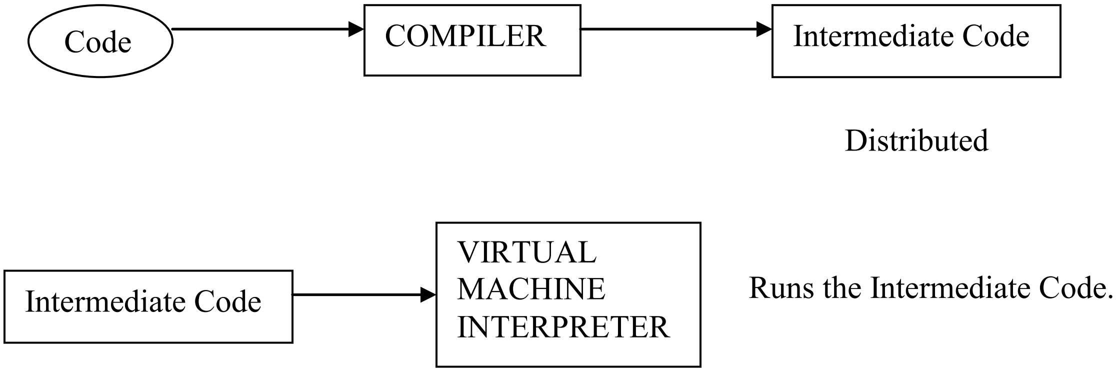


Figure 1

Code is first compiled into intermediate code that is architecture independent. It is then the distributed amongst the consumers who just need the architecture specific virtual machine interpreter to interpret the intermediate code that they have bought. Figure 1 illustrates as to how architecture independent code is compiled, converted into Intermediate Code, which is then distributed, and then executed with the help of a virtual machine that is architecture specific. This shows us the flexibility and advantage of using architecture independent code because the software developer doesn't have to worry about customizing his code to fit the needs of different architectures. We use Java Byte Code, intermediate code generated by the Java Compiler (javac) for our research.

Java Byte Code:

Java Byte Code is the intermediate, architecture independent code that is generated when a .java (source file) is compiled. This intermediate code is stored in a class file, which inherits the same name as the .java (source file) with a .class extension. This intermediate code can be distributed as is and can be run on a computer that has the Java Virtual Machine (JVM) installed on it with the help of the java class file executor (java).

Java File:

The java file is the source file for the application / applet. It contains the source code written in the higher-level language, with the Java language syntax specifications.

Class File:

The class file is the intermediate code that is generated when a Java file is compiled. It contains the byte code instructions that are needed by the JVM to run the program.

Jar File:

The Java Archive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file will contain the class files and auxiliary resources associated with applets and applications. For our purposes we make our JAR files (test cases) to be Executable Jar files. (java –jar <ExecutableJarFileName>).

Obfuscation Tool and Libraries:

The obfuscation tool that we have used is Sandmark, developed at the University of Arizona, Computer Science Department [6]. We make use of another library called the BCEL (Byte-Code Engineering Library) developed by Marcus Dahm [5]. It is a tool that allows manipulation of byte-code at runtime after which one can save the transformed class/jar files. Sandmark also uses BCEL to manipulate byte-code instructions for its obfuscation schemes.

OBFUSCATION SCHEMES

Buggy Code

This algorithm runs through an application (jar file) looking at all its class files. It then runs through all the methods of each class file looking for a significant section of code that doesn't have a conditional or an unconditional branch Instruction. When it has found such a section it makes a copy of that code and inserts some "junk" instructions that manipulate bogus variables that it has declared at the beginning of the function. The algorithm then uses an opaque predicate in a branch Instruction with the obfuscated code as a fall-through to the branch Instruction and the correct (original) code as a jump to the branch Instruction. The opaque predicate used is always true.

Before Obfuscation

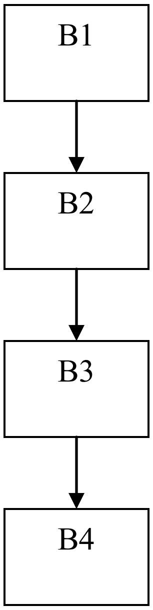


Figure 2.1

After Obfuscation

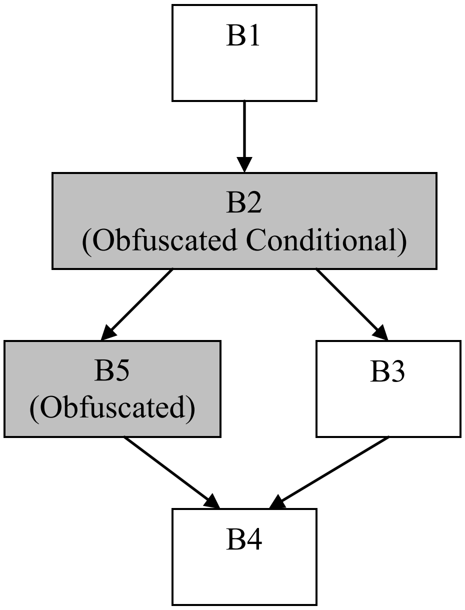


Figure 2.2

Figure 2.1 shows the control flow graph containing the basic blocks of a function before obfuscation. Figure 2.2 on the other hand shows the same function with a colored section, which indicates to the obfuscated code, and the non-colored part, which indicates to the original path of code execution. Since the obfuscated part of the code is branched over by the Opaque Predicate therefore the colored section (obfuscated code) is never executed. Hence if we are able to somehow keep track of how many times a particular basic block is executed then we shall successfully identify the colored section (obfuscated code).

Bogus Predicates

This algorithm also makes use of opaque predicates like the buggy code obfuscation scheme with a slight difference that instead of looking for sections of code containing no branch instructions it looks for sections of code that do contain conditional branch instructions. Once it has found one it appends the branch instruction conditions with a randomly selected opaque predicate.

There is a list of opaque constructs maintained by the obfuscation scheme. At run time whenever a conditional expression is encountered one of the opaque predicates is selected from the list randomly and is appended to the current conditional expression thereby creating another path in the control flow of the program. As a result of the above there is a single induced edge for every embedded opaque construct in the conditional. This induced edge is never traversed for any input because it is a fall through edge of a conditional that is always true (embedded opaque predicate). Hence if we are able to find out how many times a particular edge is traversed then we can identify this induced edge. Once this induced edge is identified we can hypothesize that both the parent and the child of the edge is obfuscated.

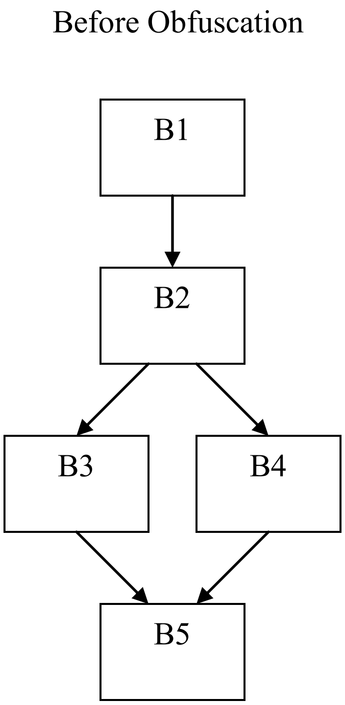


Figure 3.1

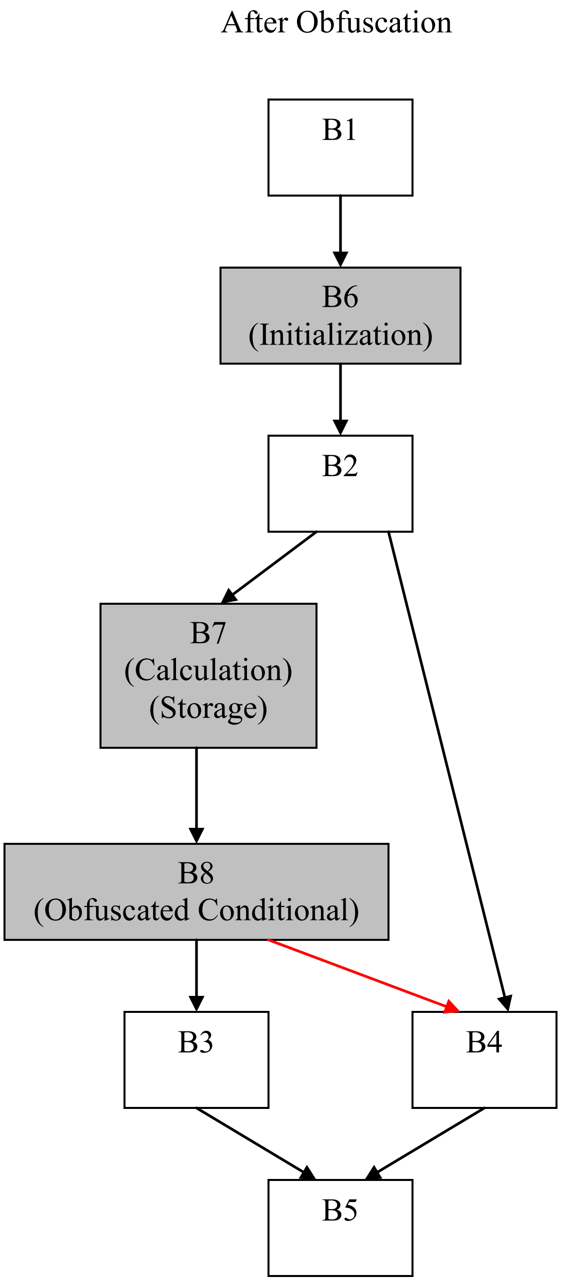


Figure 3.2

Figure 3.2 illustrates how an opaque predicate is initialized (B6), calculated and then stored (B7). If we compare Figure 3.1 and 3.2 we clearly see the induced edge (in red) between basic block # 8 and basic block # 4 which shall never be executed for any input.

THEORY BEHIND THE IMPLEMENTATION

After looking at the algorithms of both buggy code and bogus predicates one comes to the conclusion that buggy code obfuscation scheme shall crack when subjected to basic block profiling. Basic Block Profiling [8] is a technique whereby one can find out certain properties of a certain basic block in a control flow of a program. The property that we shall implement is to find out the count of how many times a particular block is executed.

Bogus predicates obfuscation scheme on the other hand is vulnerable to an attack through edge profiling. Edge Profiling [8] is a technique whereby one can find out certain properties of edges in a control flow of a program. Edges are the arbitrary arrows (defining flow of control) between two basic blocks. The property that we shall implement is the count of how many times a certain edge is traversed when the program is executed.

Both these implementations give results after the program has done executing hence they are dynamic attacks.

We subject the two obfuscation schemes to a set of static attacks namely identification of exception-handling routines and Dead Method Elimination. This was implemented because most of the code is not executed at runtime hence counts for many basic blocks would come to be zero. Also for a "good" run for a program, which means that there are no exceptions generated while execution, exception-handling routines shall never be executed. The above two mentioned techniques were introduced into the research so as to reduce the huge number of false positives that we were getting after just profiling the obfuscated code.

IMPLEMENTATION

All the tools mentioned above were implemented in Java using Sandmark [6] and BCEL libraries [5].

Basic Block Profiling

In Basic Block Profiling we used a global static array that kept counts of all the basic blocks in the application to be profiled. This array was stored in the main class of the application. Each basic block is associated with a unique number. The mapping is the number to be incremented in the global array. When the program quits a special function is called that prints out the global array with all the counts of each basic block. This function is also inserted in the application while the application is being prepared for profiling.

Since we know the mapping between the identifier of the basic block and the index to the array we insert lines of intermediate code (Java bytecode instructions) right before every basic block which increments the count of that particular basic block by one. So every time a certain basic block is traversed the profiling code is executed and count incremented. Figure 4.1 and 4.2 illustrate non-instrumented and instrumented code respectively. In Figure 4.2 the colored basic blocks are the instrumentation code that increment the count of the successor basic block in the global array of counts of basic blocks. INC(basic block) is the function illustrated in the Figure 4.2 which does the incrementation as explained above.

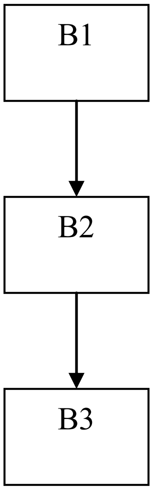


Figure 4.1

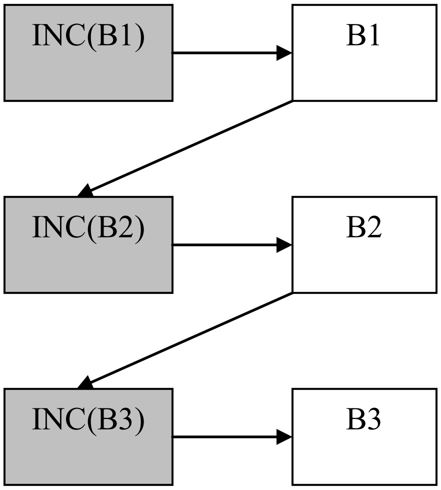


Figure 4.2

Edge Profiling

In this tool we use the same notion of storing counts similar to Basic Block Profiling but here we store the counts for edges. The edges in the control flow graph can be uniquely identified by their source and destination, both of which are basic blocks in the control flow graph of the method. Therefore for each edge we create an edge identifier as the pair of its source and destination. This mapping is used to instrument the code. A function is used for incrementing the global array of counts and just like in Basic Block Profiling the counts are printed out at the end when the application quits. So whenever an edge is traversed count for that edge is incremented in the global array. Figure 5.1 and 5.2 illustrate non-instrumented and instrumented code respectively. In Figure 5.2 the colored basic blocks are the instrumentation code that increment the count of the edge that is traversed by using the edge identifier as described above.

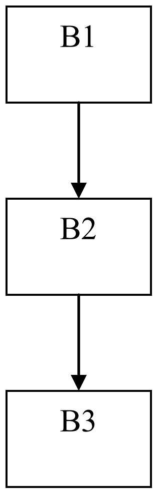


Figure 5.1

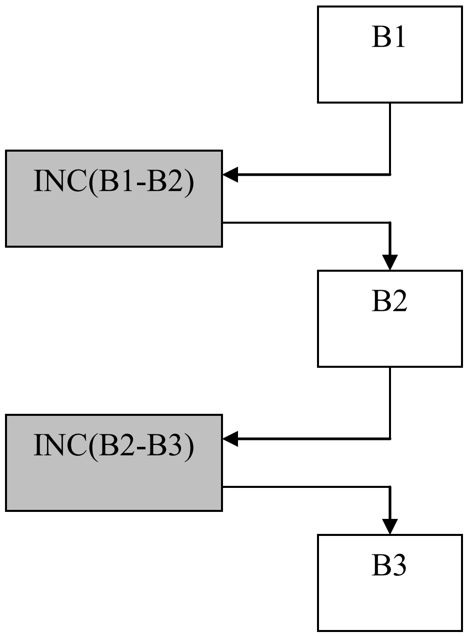


Figure 5.2

Exception Handler Removal

This tool was implemented because for a "good" run of an application exception-handling code is never executed. Exception handling code is present in most Java programs because they use libraries and library methods that throw exceptions. As our hypothesis suspects zero count blocks to be obfuscated we would get huge false positives for a "good" run of the program. So to reduce the number of false positives from the profile data we identify the blocks that correspond to exception-handling routines in the application. This is done through the BCEL library, which extracts exception-handling information from the byte code file. The library therefore allows one to identify the exception-handling code in a given method. We can then find out the basic blocks that are in an exception-handling routine. These blocks are removed from the set of false positives that we contain because we make sure that during the execution of the obfuscated code we provide the application "good" set of inputs that do not cause any of exception-handling routines to be executed. Figure 6 illustrates the exception handling code range (B1 to B5). If there is an exception in the exception handling code range then the control shifts to the exception handling code (EXC1 to EXC2).

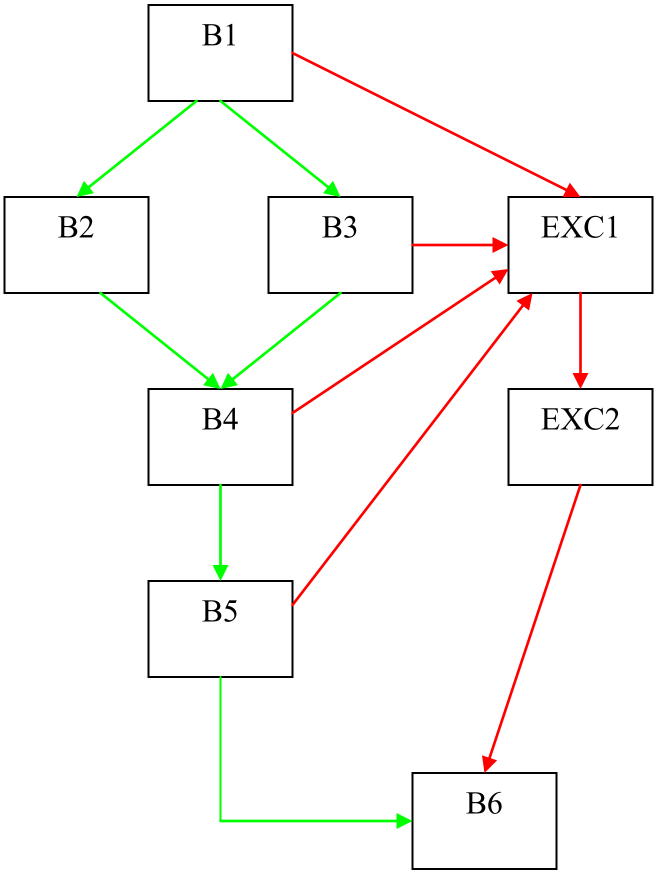


Figure 6.

The green arrows illustrate an execution with no exceptions and the red arrows show the transfer of control to the exception handling routine. This shows that EXC1 and EXC2 will not get executed if the green path is taken.

Dead Method Elimination

The other source of huge false positives could be methods that are never executed. They might be present in the program as mere debugging tools that the programmer forgot to remove. The do no harm to the program as they are never executed hence these shall also pop up in the profile data with zero counts. This is also backed by the 80-20 rule, which says that most of the time is spent in twenty percent of the code. To tackle this problem we implement a simple reachability algorithm that takes into account the name of the method, the class in which it is defined and the parameters that it takes. We construct a graph with vertices as methods signatures (the string comprising of the class name the method is defined in concatenated with the method name concatenated with the parameters) and edges as calls to other functions. We shall name such a graph as a C-Graph. The reachability algorithm figures out if all the methods in the graph were reachable by the "main" function of the Main-Class or not. All those methods not reachable from the "main" function are termed unreachable and their basic blocks removed from the profile data. The method name coupled with the class name of the method and the signature of the method was used as the identifier for the vertex. This takes care of methods having similar names with different parameters (functional overloading).

After we find out the methods that are unreachable we double check with the profile data if they really have zero counts or not. We do this test because Java Reflection could have been used where one can call a method at runtime, whose name might be determined at runtime. Therefore statically it would be impossible to figure out what the method was called. If the blocks of the method termed "unreachable" do have counts greater than zero then we discard those methods as they might have been called through Java Reflection or implicitly (static functions).

Algorithm A

```
List N contains all methods.
Tree T contains the parent as the caller and the child as the callee.
Method boolean ifReachable(String src, String dest)
    reached = false

    If src == null
        return false

    If src == dest
        return true

    findNode of src in the List N

    If src is visited
        return false
    else
        set src to visited

    Iterate through all the children of the src getting srcNode from T
        If child == src
            reached = true and break

    If reached == false
        Iterate through all the children of src getting srcNode from T
            If IfReachable(child, dest) == true
                reached = true and break

    return reached
```

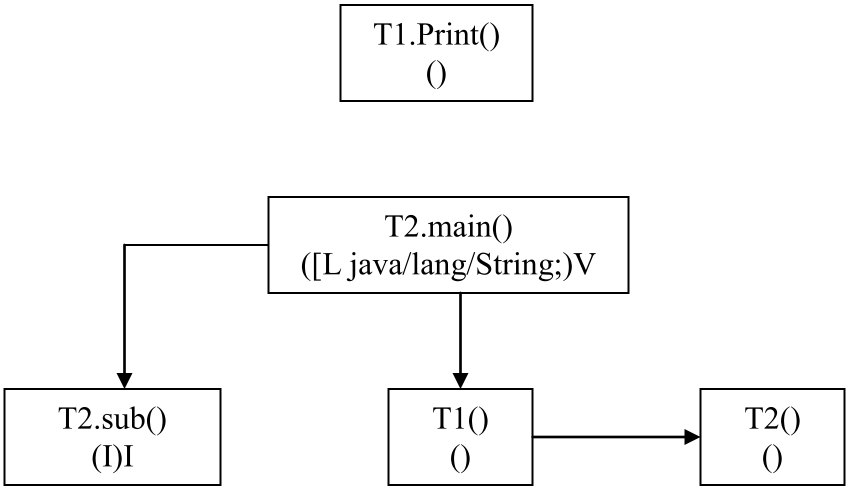
Figure 8 illustrates the above algorithm on a small test case with two class files. The figure is the graph constructed on which the above reachability algorithm is executed. The figure clearly shows that the graph will not contain the unreachable methods and hence the algorithm will point out that those methods are unreachable. Code 1 is the pseudo-code of the test case.

```
Class T1
        Int a

        T1()
                a = 0

        Int sub(Int a)
                Return this.a – a

        Print()
                print a

End of Class
```

```
Class T2
        Int b
        T1 k

        T2()
                b = 0

        main()
                T2 t
                t.k.sub(6)

End of Class
```

Code 1.

The above code generates the following C-Graph.



Therefore when Algorithm A is executed on such a C-Graph we see that reachability of T1.Print() from T2.main() fails. Therefore we can remove all the blocks in T1.Print() from the set of suspected obfuscated blocks.

TEST CASES

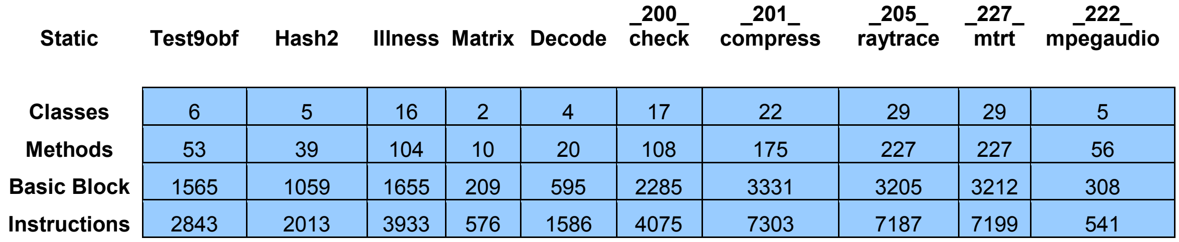
The test cases used in our research for obfuscation and then subsequently used in profiling and static attack techniques were chosen with the following criteria.

We wanted to test our profiling and static attack techniques on real programs, programs that contained redundancy of code and extra methods hence we chose 2 student projects (Test9obf) and (Hash2) that implemented hash tables turned in as a class project for CSC 345 in Spring 2003. The 2nd set of test cases (Illness, Matrix and Decode) was taken from the Sandmark test suite. The 3rd set of test cases was the SpecJvm 98 suite of which 5 test cases were chosen (_200_checkobf, _201_compress, _205_raytrace, _227_mtrt and _222_mpegaudio). After the profiling data was collected we classify the basic blocks based on the count and whether they existed in the list of obfuscated blocks. So we get a 4-bucket arrangement as explained in Figure 8.

	0 COUNT	NON-ZERO COUNT
JUNK	Obfuscated Blocks with 0 Count	Obfuscated Blocks with non 0 Count
NONJUNK	Good Code with 0 Count	Rest of the code

Figure 8.

Following are the characteristics of the test cases used.

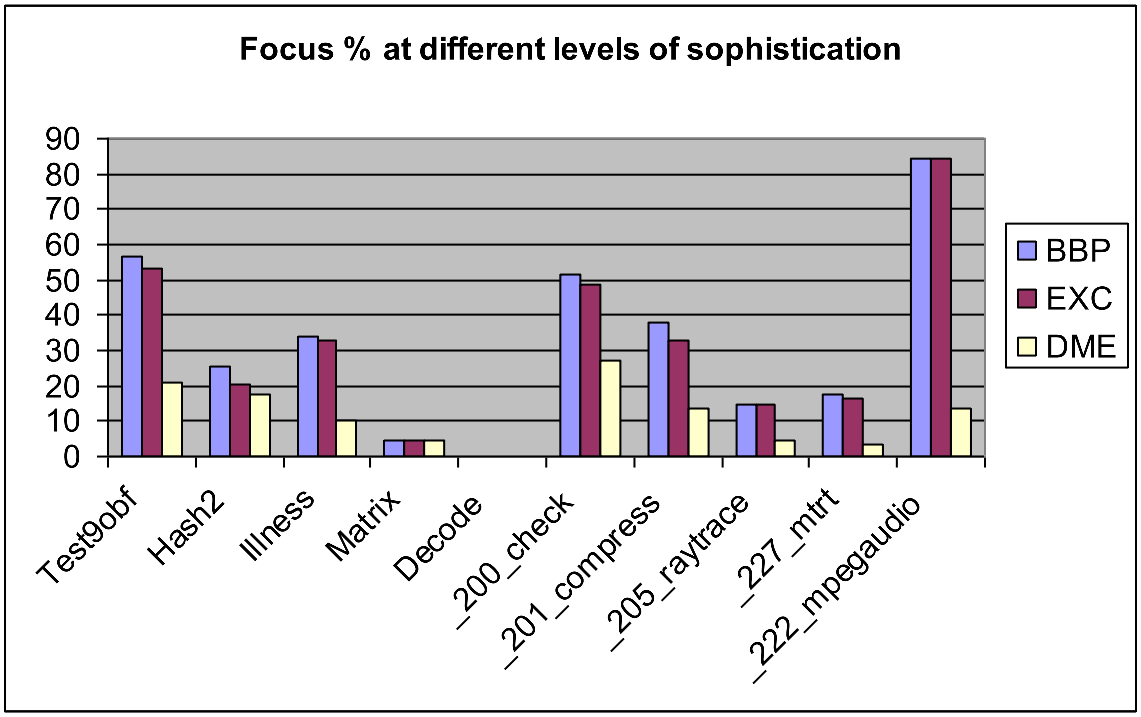


Static	Test9obf	Hash2	Illness	Matrix	Decode	_200_ check	_201_ compress	_205_ raytrace	_227_ mtrt	_222_ mpegaudio
Classes	6	5	16	2	4	17	22	29	29	5
Methods	53	39	104	10	20	108	175	227	227	56
Basic Block	1565	1059	1655	209	595	2285	3331	3205	3212	308
Instructions	2843	2013	3933	576	1586	4075	7303	7187	7199	541

Table 1.

RESULTS AND OBSERVATIONS

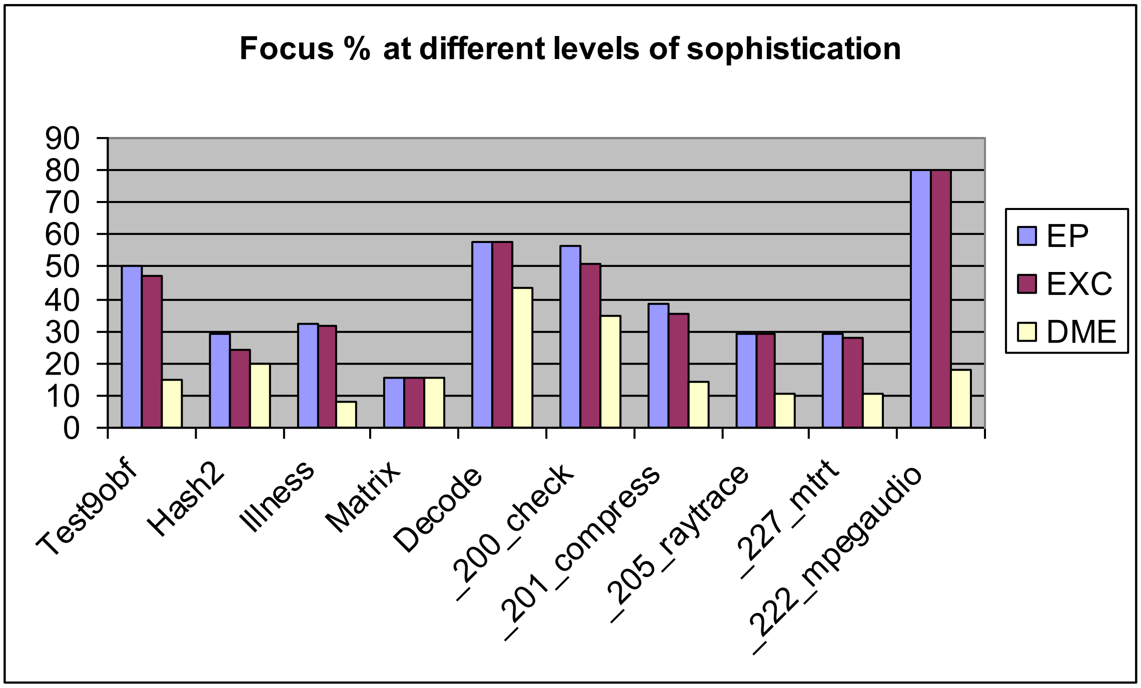
From the data collected we see that we are able to trace down a part of the obfuscated code because we had the list containing the obfuscated basic blocks. But since the attacker doesn't have this list we introduce the term focus percentage, which is defined as the percentage of code the attacker should see with respect to the original code length to find obfuscation. The following chart shows to us that the focus of the attacker is decreased considerably for all the test cases.



BBP = Basic Block Profile
EXC = Isolation of Exception Handling Routines
DME = Dead Method Elimination

Chart 1

From the chart we see that the focus of the attacker is reduced considerably when the code that is obfuscated by buggy code is subjected to Basic Block Profiling and then subsequently to the static attacks shown above.



BBP = Basic Block Profile
EXC = Isolation of Exception Handling Routines
DME = Dead Method Elimination

Chart 2.

Once again we see that by using the above three mentioned techniques we are able to focus the attackers attention to a certain section of the code that does contain the obfuscation. Therefore we conclude that Edge Profiling and static attacks can be used to crack programs obfuscated through Bogus Predicates.

		Test9obf		Hash2		Illness		Matrix		Decode	
		0	Non 0	0	non 0	0	non 0	0	non 0	0	non 0
Profiling	JUNK	1	1	2	0	3	0	3	1	0	0
	NON	885		267		562		9		(Noobf)	
	Blocks	52		52		14		0			
Exception	JUNK	1	1	2	0	3	0	3	1		
	NON	833		215		548		9			
	Blocks	505		30		376		0			
Dead Method	JUNK	1	1	2	0	3	0	3	1		
	NON	328		185		172		9			

		_200_ check		_201_ compress		_205_ raytrace		_227_ mtrt		_222_ mpegaudio	
		0	non 0	0	non 0	0	non 0	0	non 0	0	non 0
Profiling	JUNK	2	1	6	1	4	0	3	1	4	0
	NON	1176		1265		478		571		260	
	Blocks	136, 76=0 60 Non 0		169, 166=0 3 Non 0		35, 1=0 34 Non 0		35		1	
Exception	JUNK	2	1	6	1	4	0	3	1	4	0
	NON	1110		1099		477		536		259	
	Blocks	484		653		419		419		217	
Dead Method	JUNK	2	1	6	1	3	1	3	1	4	0
	NON	626		446		132		117		42	

Table 2.

Table 2 is the data collected from the Buggy Code obfuscation scheme. It shows the 4-bucket analysis of the code after each level of sophistication. We see that as expected the number of false positives decreased as we applied some intelligent static attacks on the code.

		Test9obf		Hash2		Illness		Matrix		Decode	
		0	non 0	0	non 0	0	non 0	0	non 0	0	non 0
Profiling	JUNK	301	70	156	95	44	11	41	55	54	39
	NON	952		366		545		45		394	
	Blocks	57		57		14		0		0	
Exception	JUNK	301		156	95	44	11	41	55	54	39
	NON	895		309		531		45		394	
	Blocks	612		60		397		0		104	
Dead Method	JUNK	301	70	156	95	44	11	41	55	54	39
	NON	283		249		134		45		297	

		_200_ check		_201_ compress		_205_ raytrace		_227_ mtrt		_222_ mpegaudio	
		0	non 0	0	non 0	0	non 0	0	non 0	0	non 0
Profiling	**JUNK**	563	94	174	91	137	327	143	321	24	13
	NON	1607		1331		1029		1015		263	
	Blocks	185		124		35		35		1	
Exception	**JUNK**	563	94	174	91	137	327	143	321	24	13
	NON	1445		1215		1021		980		262	
	Blocks	448		719		650		617		203	
Dead Method	**JUNK**	563	94	174	91	137	327	143	321	24	13
	NON	997		496		371		363		59	

Table 3.

Table 3 is the data collected from the Bogus Predicates obfuscation scheme. Here too we use the same notions as used in Table 2. The 4-bucket analysis again shows to us that the focus of the attacker is brought down considerably.

CONCLUSION AND FURTHER RESEARCH

From the analysis we conclude that we have aided the attacker into finding out where control flow obfuscation is present. This shows to us that control flow obfuscation is weak and can be broken by using simple profiling techniques. Therefore control flow obfuscation is not very effective against dynamic profiling techniques.

Another way that one could further bring down the focus percentage is to analyze branch instructions. Control Flow Obfuscation rely on branch Instructions for their obfuscation and so if one can just profile branch Instructions and deduce some opaque properties of its conditionals then we can further reduce the focus percentage making any control flow obfuscation even more susceptible to dynamic and intelligent static attacks.

References

[1] James, Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996. ISBN 0-201-63451-1

[2] Christina Cifuentes and K. John Gough. "Decompilation of binary programs." *Software – Practice & Experience,* 25(7):811-829, July 1995

[3] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS),* June 1997

[4] Christian Collberg, Clark Thomborson and Douglas Low: *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.* Proc. Symposium on Principle of Programming Languages (POPL '98), Jan 1998.

[5] Marcus Dahm: BCEL (Byte Code Engineering Library). http://bcel.sourceforge.net, 2002.

[6] Christian Collberg, Sandmark: A Tool for the Study of Software Protection Algorithms. http://www.cs.arizona.edu/sandmark/

[7] Christian Collberg, Clark Thomborson: *Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection.* IEEE Transactions on Software Engineering, Volume 28, Number 8, Aug 2002.

[8] Thomas Ball, James R. Larus, "'Optimally Profiling and Tracing Programs," Technical Report #1031, University of Wisconsin, Madison (July 1991)