Code Obfuscation Literature Survey

Arini Balakrishnan, Chloe Schulze
CS701 Construction of Compilers, Instructor: Charles Fischer
Computer Sciences Department
University of Wisconsin, Madison

December 19th, 2005

Abstract

In this paper we survey the current literature on code obfuscation and review current practices as well as applications. We analyze the different obfuscation techniques in relation to protection of intellectual property and the hiding of malicious code. Surprisingly, the same techniques used to thwart reverse engineers are used to hide malicious code from virus scanners. Additionally, obfuscation can be used to protect against malicious code injection and attacks. Though obfuscation transformations can protect code, they have limitations in the form of larger code footprints and reduced performance.

1 Introduction

In the last decade, distribution of code in an architecturally-neutral format has increased the ability to reverse engineer source code from executables. This practice has greatly concerned software companies who desire to protect the intellectual property of their products. Though copyright laws forbid direct piracy of software, developers are worried by possible theft of proprietary data structure and algorithmic design. Though there are several methods for protecting software, such as encryption, server-side execution and native code, obfuscation has been found to be the cheapest and easiest solution to this problem[6].

Code obfuscation is the practice of making code unintelligible, or at the very least, hard to understand. The process of code obfuscation is the application of transformations to the code, which changes the physical appearance of the code, while preserving the black-box specifications of the program. In this project we have surveyed the different obfuscation techniques that are currently being researched, particularly those that can be implemented via a compiler.

Not only can code obfuscation be used to protect intellectual property, it is also used extensively by authors of malicious code to avoid detection. Many viruses utilize obfuscation techniques to subvert virus scanners by continually changing their code signature with obfuscating transformations. We review the types of viruses that commonly use obfuscation and the techniques they employ.

The paper is laid out as follows: in Section 2 we review the general methods used in common obfuscation techniques, then in Section 3 we discuss techniques used to thwart static analysis. Section 4 talks about obfuscation in the disassembly phase, and Section 5 addresses obfuscation techniques used by viruses. Finally, we conclude in section 6.

2 General Methods for Obfuscation

General code obfuscation techniques aim to confuse the understanding of the way in which a

program functions. These can range from simple layout transformations to complicated changes in control and data flow. The techniques described below summarize the work by Collberg et al. [4].

2.1 Obfuscation Quality

The quality of an obfuscating transformation is evaluated according to four criteria:

1. Potency: How much obscurity it adds to the program
2. Resilience: How difficult is it to break for an automatic deobfuscator
3. Stealth: How well the obfuscated code blends with the rest of the program
4. Cost: How much computation overhead it adds to the obfuscated application

2.2 Obfuscating control transformations

The control flow transformations used for obfuscation can be described as affecting the aggregation, ordering or computations of the flow of control. Aggregation transformation breaks up computations that are logically related and merges computations that are not. Control ordering transformations randomize the order in which the computations are carried out. Computation transformations insert new code or make algorithmic changes to source application.

The key to the success of control transformations is the resilience of opaque predicates and variables[5]. Opaque predicates and variables are constructs whose values are known to the obfuscator, but are difficult for the deobfuscator to deduce. An opaque predicate is *trivial if a deobfuscator can deduce it by static local analysis, and* weak if a deobfuscator can deduce it by static global analysis.

2.3 Computation Transformation

There is a strong relation between the complexity of the code and the number of predicates it contains. As the number of predicates increase in a body of code, insertion of dead or irrelevant code into the program becomes easier. Therefore, the presence of opaque predicates provides opportunities to further obfuscate a program. For instance, a basic block, B can be split into two basic blocks by inserting an opaque predicate P^T (which always evaluates to true) in the center of B. Different obfuscations can then be applied to each half of B. We can also obfuscate a loop by making the termination condition complex. This can be done by introducing an opaque predicate P^T (always evaluates to true) or P^F (always evaluates to false) that does not affect the number of times a loop will execute.

2.4 Obfuscating data abstractions

In the following sections we describe transformations that obscure data abstractions used in source code. There are two types of such obfuscation: namely, modifying inheritance relations and restructuring data arrays. This information is gleaned from a further study done by Collberg et al. [7].

2.4.1 Modifying inheritance relations

According to the Chidamber metric[1], the complexity of a program increases with greater depth of the inheritance tree. Along these lines, we can artificially increase the complexity of a program either by splitting up a class or by inserting a new bogus class. Suppose C is the class we want to factor into C1 and C2, we should make sure

[1]The Chidamber metric measures the complexity of a class C according to the number of methods in C, the depth of C in the inheritance tree, number of direct subclasses of C, number of other classes to which C is coupled, number of methods that can be executed in response to a message sent to an object of C, and the degree to which C's methods do not reference the same set of reference variables.

that this factoring respects the scope of the variables V in C. In other words, if we split a class in two, one half must not end up with all the variables, while the other half contains all the methods.

Inheritance relations can also be modified by false refactoring. Refactoring takes place in two steps. Firstly, identify two independent classes that implement the same behavior. Secondly, move the features common to both the classes to a parent class. False refactoring is similar to refactoring except that false refactoring is performed on two independent classes that do not have a common behavior.

2.4.2 Restructure Arrays

There are many different types of transformations that can be devised to obscure the operations performed on an array. These transformations include: splitting an array, merging two or more arrays, flattening an array (i.e. decrease the dimensions of the array), and folding the array (i.e. increase the dimensions of the array).

2.5 Obfuscating Procedural Abstractions

In the following section we discuss different obfuscating transformations that obscure the procedural abstractions in a program. These break up user-defined abstractions or introduce new abstractions, thereby destroying the original structure of the source code.

2.5.1 Table Interpretation

This is one of the most efficient, but expensive, transformations used for obfuscation of code. The fundamental idea is to convert a portion of the code to a different machine code. This new code is then executed by a new virtual machine interpreter included in the obfuscated code. There is usually one to two orders of magnitude slowdown for each level of interpretation, and so this transformation should only be used for those sections of code that consumes a small part of the total runtime.

While this transformation is highly potent, it is not very resilient. The deobfuscator could inline code for each bytecode instruction prior to decompilation. One way to thwart this is to replace the bytecode string by a program that produces it or obfuscate the program itself by introducing bogus predicates and dead code.

2.5.2 Inline and Outline Methods

Inlining, a code optimization technique also finds a use in code obfuscation. Once code is inlined and the procedure itself has been removed there is no trace of that abstraction left in the code. Conversely, outlining is the process of selecting a group of statements in a procedure and using them to create a sub-procedure. One example of applying inlining and outlining could be to inline two procedures A and B which are called one after the other, and then outline a portion of their combined code into a new procedure.

2.5.3 Clone Methods

While examining the purpose of a subroutine, a reverse engineer would examine its body and signature. Also the environments where these functions are called play an important role in the understanding of the code. We can make this process difficult by obfuscating a method's call site to make it appear like different routines are called.

2.6 Obfuscating built-in data types

In the following sections we discuss different obfuscating transformations that obscure basic data types used in the source application. Designing such transformations is difficult because these data types form an integral part of programming languages. These transformations are very high in cost and low in stealth. Cost is high because we are obscuring the data type, but it is not very stealthy because the transformation is easy to identify.

2.6.1 Split variables

Variables of restricted range can be split up into two or more variables. In order to split a variable V of type T into two variables p and q of type U, we need to provide three pieces of information:

1. A function f(p,q) that maps the values of p and q into the corresponding value of V.

2. a function g(V) that maps the value of V into the corresponding values of p and q.

3. new operations cast in terms of operations on p and q.

The potency, resilience and cost of this transformation all grow with the number of variables into which the original variable is split.

2.6.2 Convert static to procedural data

Static strings contain much useful information to a reverse engineer. A simple way of obfuscating them would be to convert the string to a program that computes the string. Aggregating the computation of all static string data into one function, however, may be un-stealthy. We can achieve much higher resilience and potency if the program that computes the string is broken into smaller components and embedded in the normal control flow of a program.

2.6.3 Merge scalar variables

This method of obfuscation involves merging two or more scalar variables into a single variable. The variables $v_1, v_2 \dots v_k$ can be merged into one variable V_m provided the the combined ranges of $v_1, v_2 \dots v_k$ fit within the precision of V_m. Arithmetic on individual variables would be transformed into arithmetic on V_m. The resilience of merging variables, however, is too low to make the transformation profitable.

3 Code obfuscation by obstructing static analysis of programs

Effective alias detection is shown to be NP Hard and this method of code obfuscation aims to introduce non trivial aliases into the program such that data flow analysis is rendered slow or less precise. The transformations discussed below introduce aliases into the program which further hinders the systematic analysis of the program by breaking down control flow. We focus here on techniques used for obstructing static analysis of programs. Static analysis can be either control-flow sensitive or control-flow insensitive. Flow-insensitive analysis does not provide as much information with which to tamper, as does flow-sensitive analysis. Control-flow sensitive analysis first constructs a control flow graph of a program and conducts data flow analysis. The techniques to thwart static analysis increase the data dependency of the control flow graph. This results in increased complexity and decreased precision of the analysis. In the following sections we summarize the work of Wang et al. [13].

3.1 Control-flow Transformations

Control flow transformations are accomplished in two steps. The first step involves decomposing high level control transfers into a series of if-then-goto statements. The second step is to then modify the goto statements such that the target addresses are determined dynamically. This can be achieved by branching on a data variable instead of direct jumps. These transformations make direct branches dependent on data and results in the flattening of the control flow graph. Now, given that there is no direct information about branch target addresses, the problem of building a flow-graph for static analysis of the program depends on identifying the latest definition of the variable at each branching point.

3.2 Data-flow Transformations

Once control-flow transformations have been applied, the problem of constructing a flow-graph now depends on data-flow analysis. Data-flow problems have been proved to be NP complete or harder. The fundamental difficulty in data flow analysis arises from the presence of aliases in the program. The second set of transformations aims to introduce aliases into the program that determine the computation and hence the analysis, of branch targets. The transformation is done in the following steps:

1. In each function, introduce an arbitrary number of pointer variables.
2. Insert artificial blocks, or code in existing blocks, that assign pointers to data variables.
3. Replace references to variables and array elements with indirection through these pointers. Previously meaningful computation on data quantities can be replaced with semantically equivalent computation through the pointers.
4. A definition of a pointer is placed in a different block than its use.

The effect of these transformations is that the static analyzer does not know which blocks to execute and since definitions to pointers and their uses are placed in different blocks the analyzer will not be able to deduce which definition applies to each use of the pointer. This approach of thwarting static disassembly is capable of expanding analysis times and reducing the precision of analysis to useless levels.

4 Code Obfuscation in Disassembly Phase

Most of the obfuscation techniques discussed so far were applied during the decompilation phase. We can also obfuscate code at the disassembly phase. In this section we discuss the two widely used static disassembly algorithms and techniques to thwart each of them.

There are two methods of disassembly: static disassembly, where the file being disassembled by the disassembler is not executed during the course of disassembly; and dynamic disassembly, where the file is executed on some input while the disassembly happens. We focus only on code obfuscation techniques for thwarting static disassembly proposed in the literature. There are two kinds of static disassembly: linear sweep and recursive traversal. A linear sweep disassembler sweeps through the program disassembling instructions as they are encountered. A recursive traversal disassembler would disassemble the instructions according to the flow of control. That is, when a branch instruction is encountered, it determines the possible control flow successors of that instruction and proceeds with disassembly at those instructions. In the following sections we summarize the work of Linn et al. [9].

4.1 Thwarting disassembly

A machine code file consists of different sections that contain various details about the program. One such instruction stored in the machine code of a program is the program entry point, i.e the location in the machine code where the machine instructions begin. To thwart a disassembler, we have to confuse as much as possible its notion of where the instruction boundaries in a program lie. On some instruction sets the disassembly process is self-repairing. That is even when a disassembly error occurs, the disassembler eventually ends up resynchronizing with the actual instruction. So the efforts to confuse disassembly should take this factor into account. Some techniques used to thwart disassembly are discussed in the following sections.

4.2 Junk Insertion

We can thwart disassembly by inserting junk bytes at specific places in the instruction stream. To confuse the disassembler, the junk instruc-

tions should be partial instructions and in order to preserve program semantics the junk instructions inserted should be unreachable at run time. To do so we choose candidate blocks, which are blocks that will allow such junk bytes to be inserted before it. To ensure that these junk instructions are unreachable during program execution, these candidate blocks must not allow the execution to fall through to them. Once this is done, we have to determine which junk instructions to insert so as to confuse the disassembler as much as possible and delay resynchronization for as long as possible.

4.3 Thwarting Linear Sweep

A linear sweep disassembly cannot distinguish data in the text section. This can be exploited in thwarting disassembly by inserting junk bytes at selected locations in the instruction stream. The junk bytes can only be inserted before candidate blocks that do not have execution fall through to it. In programs obtained from a typical optimizing compiler candidate blocks are around 30 instructions apart. To enable introduction of more junk bytes, we have to find more candidate blocks and this can be done with the help of branch flipping. The idea is to invert the sense of conditional jumps, by converting code of the form

$$b_{cc}\ Addr$$

where cc represents a condition to

$b_{\overline{cc}}$ L'
jmp Addr
L':

where $\overline{cc}$ is the complementary condition to cc.

4.4 Thwarting Recursive Traversal

Recursive traversal deals intelligently with control flow and this can be exploited to confuse the disassembly process. Once a branch is encountered, recursive traversal continues disassembly at the possible targets of this branch. Recursive traversal assumes that the normal branches and call functions work reasonably. That is, a conditional branch has two possible targets and a call to a function returns to the point from where it was called. Another aspect of recursive traversal is the difficulty in identifying possible targets of indirect control transfer functions.

4.4.1 Branch functions

The assumption that the call function returns to the instruction immediately following the call instruction can be exploited using branch functions. Given a finite map ϕ over locations in a program

$$\phi = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2 \ldots a_n \rightarrow b_n\}$$

a branch function f_ϕ is a function that, whenever it is called from one of the locations a_i, causes control to be transferred to the corresponding location b_i. Given such a branch function we can replace n conditional statements by the following lines:

A1:jmp b_1
A2:jmp b_2
An:jmp b_n

By calls to branch function

A1:call f_ϕ
A2:call f_ϕ
An:call f_ϕ

Such branch functions have two purposes. One which obscures the computation of the target address within branch function, and the second creates opportunities for misguiding the disassembler. Such branch functions can be implemented in a number of ways. One implementation would be to use the return addresses to look up the table. Another, better, implementation is to have

the callee pass, as an argument the offset from the instructions immediately after it to the target b_i.

4.4.2 Call conversion

A variation of the branch function scheme can be used to insert junk bytes of instructions immediately following call instructions as well. This is done by rerouting call instructions through a specialized branch function, which branches to its appropriate target function, but returns to some predetermined offset from the original call instruction. This can obscure control flow information and also makes it difficult to decipher function entry points while increasing the potential to mislead the disassembler.

4.4.3 Opaque predicates

The assumption that a conditional branch has two targets can be exploited by converting an unconditional branch to a conditional branch, that always goes in one direction. This can be achieved by the use of opaque predicates. Once the unconditional branch is replaced by a conditional branch that uses an opaque predicate, we have a candidate block before which we can insert junk bytes to mislead disassembly.

4.4.4 Jump Table Spoofing

In addition to inserting junk bytes to mislead disassembly we can also insert artificial jump tables to subvert recursive traversal disassembly. Recursive traversal disassembly attempts to determine the size of the jump table to identify possible targets of an indirect jump. This behavior can be exploited by introducing a jump table that is unreachable at runtime.

5 Code Obfuscation as it Relates to Viruses

Not only is code obfuscation used to protect commercial software against reverse engineering; malicious code writers can also use obfuscation to hide their creations from virus scanners. In the end, both uses have the same basic intent: to obscure understanding of the program. Of course each task involves different limiting properties. Viruses need to deal with the fact that their obfuscations must continually change, or else scanners will have a known signature to match against. This brings up time and space restrictions. Conversely, commercial software has performance requirements, which limit the types and extent of obfuscation.

In this section we discuss the differences between obfuscation of commercial software and malicious code. We first introduce the particular types of viruses that use obfuscation techniques; we then discuss the transformations they typically use in subverting detection. Finally, we explore the possibility of learning from these techniques and perhaps transferring them to the task of protecting commercial software. We restrict our survey to viruses in the interest of time and space.

5.1 Virus Types

The two main types of malicious code that use obfuscation techniques to hide themselves from virus scanners are polymorphic and metamorphic viruses. Simple viruses do not change from generation to generation, so virus scanners can rely on simplistic string matching detection to determine if a file has been infected or not. Polymorphic and metamorphic viruses were developed so as to subvert general scanner methods. They rely on techniques that change their code signature each infection generation, making it impossible for string matching algorithms to detect their presence.

5.1.1 Polymorphic

Polymorphic viruses are an extension to encrypted viruses [1, 14]. An encrypted virus simply encrypts its body and then attaches a decryption key to itself, which changes from gen-

eration to generation. A polymorphic virus also encrypts its body, however, it randomly changes its decryption algorithm each time the virus infects a host. Because the decryption scheme is never encrypted, and it is generally tacked on to the end of the virus code, it might be source of detection by virus scanners. Therefore, when a polymorphic virus changes its decryption routine, it changes the part of the signature that scanners could detect.

Virus scanners can still get around this obfuscation by basically waiting for a polymorphic virus to deobfuscate itself. Virus scanners have the ability to run a virus in a "sandbox" or emulator, which allows the virus to run without causing harm. As the virus runs, it must decrypt itself to attempt to infect a file and therefore the emulator can scan for the virus signature when the virus decides to execute. Obviously this technique has limitations, one of which is how to know how long to run.

5.1.2 Metamorphic

Metamorphic viruses, like polymorphic viruses encrypt themselves to hide their signatures from virus scanners. Metamorphic viruses, however, can change their source code and then recompile themselves if compilers are available on the victim machine. This allows the virus to add and remove junk code, which evolves their signature over time. Metamorphic viruses are, of course, an advance in technology beyond the techniques of polymorphic viruses. Whereas, polymorphic viruses will decrypt themselves in memory to propagate and infect hosts, metamorphic viruses do not. In fact, a metamorphic virus never reveals its entire virus body at once, making it almost impossible for a simple signature-matching scanner to detect it.

As a response, virus scanners have evolved to use heuristics to detect metamorphic viruses. Common techniques still involve emulation, or running on a Virtual Machine, which simulates some of the functionality of a operating system. Tracking system calls is a technique that has seen much research in the last few years [12, 8]. M. Christodorescu et al. have recently developed a static analysis technique to detect metamorphic viruses [3]. Techniques to detect a metamorphic virus can be very much like those that are used to reverse engineer commercial software. Static analysis, decompilation and the like can recognize obfuscation efforts such as dead code insertion.

5.2 Obfuscation Techniques

Obfuscations used by virus writers are generally utilized for the purpose of changing the signature of the code, i.e. the sequence of instructions so that a virus scanner will be unable to use search strings to detect the presence of the virus. Common obfuscation techniques, which change the layout of code, are: dead code insertion (also called junk or trash insertion, code transposition, register reassignment and instruction substitution. These obfuscations are explained briefly below, though they generally expand on the earlier descriptions of obfuscation transformations.

5.2.1 Dead Code Insertion

This transformation acts just like it sounds, it involves placing ineffectual instructions in amongst existing instructions. These new instructions can be no-ops, change the state of a program, only to change it right back, or can involve jump instruction placement which will skip the new instructions all together. Though changes such as these most likely will not fool a reverse engineer, it has been shown by M. Christodorescu et al. [3] that these simple transformations can subvert many virus scan engines.

5.2.2 Code Transposition

Code transposition is simply the cosmetic movement of code within a file. For instance the location of procedures could be changed around, or instructions within a procedure, which are independent of one another, can be reordered.

5.2.3 Register Reassignment

Register reassignment refers to the change of registers used by live variables. If a particular register R1 is not used during the live range of a variable, then the register R2 used currently to store the live variable can be replaced by R1. Again, this transformation would not stand up to reverse engineering techniques, but it does have the effect of changing the signature of the program.

5.2.4 Instruction Substitution

When compared to the above transformations, this obfuscation technique is the most resilient, even when it comes to reverse engineering. Instruction substitution is implemented using a library of equivalent instructions. It replaces an instruction in the body of code with one that is equivalent. This can greatly change the signature of the code, and is difficult to deobfuscate, especially if the library of equivalent instructions is unavailable.

5.3 Comparisons

Virus writers obfuscate code to change the signature of the code, so that virus scanners cannot recognize it. Which makes sense because the scanners are searching for an infection that may not be there. This is obviously different than obfuscation from a reverse engineering standpoint. In the latter case the obfuscation is meant to hide important or sensitive data, which the reverse engineer knows is present. In general, obfuscation techniques used by malicious code writers are not as potent as those used for commercial software.

There are practices, however, which malicious code authors use, that may have applications to obfuscation of commercial software. Such an example is a technique used by the Zmist virus to hide its polymorphic decryptor[12]. The decryptor is placed within the host program in chunks separated by pieces of the host code and connected together by jumps. It might be possible to apply this type of transformation on commercial software. An important or sensitive algorithm could be broken into chunks and spread throughout the code. For the most part, however, transformations applied to viruses will not stand up against reverse engineering attacks and so cannot directly be applied to commercial software.

5.4 Another Angle

There is of course another angle to obfuscation of commercial software, which is the use of obfuscation to protect against malicious attacks. Such attacks are generally in the form of code injection, where a malicious entity attempts to insert code into an existing program with the intent that the code will be run and transfer control of the system to the attacker [10]. One can apprehend that to insert code into a program and have it successfully run without notice requires an in-depth understanding of the victim program. To thwart such an attack, clearly it would help if it were difficult to understand the inner workings of the victimized program. And here code obfuscation can take on yet another role.

There is current research in the area of address obfuscation, which aims to vary the location of data and procedures through dynamic change of the program's address space during execution [11, 2]. Not only does this technique subvert understanding of a program, but it also has the ability to protect against repeated attacks because of its ability to change dynamically. This is an especially interesting technique because it seems to influenced by the way in which viruses themselves can mutate to avoid detection.

6 Conclusion

Though code obfuscation can thwart many attacks, given enough time and effort any of the techniques discussed above can be overcome by reverse engineers. No obfuscation has yet been found that can completely resist reverse engi-

neering. In addition to this drawback, code obfuscation increases the code footprint, decreases performance, and can hinder certain compiler optimizations. In spite of these limitations, obfuscation techniques, when used sparingly, and combined appropriately, can add a layer of protection against theft and insertion of malicious code. The literature surveyed produces an inconclusive result as to whether obfuscation techniques used by viruses can be applied to commercial software. Clearly, obfuscation does help to protect against virus infections, but further study is needed to ascertain which techniques produce the best results.

References

[1] Understanding and managing polymorphic viruses, 1996.

[2] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efcient approach to combat a broad range of memory error exploits.

[3] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, pages 169–186, August 2003.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*, July 1997.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[6] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.

[7] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.

[8] A. Lakhotia and E. U. Kumar. Abstract stack graph to detect obfuscated calls in binaries. In *IEEE International Workshop on Source Code Analysis and Manipulation*, September 2004.

[9] C. Linn and S. Debray. *Obfuscation of Executable Code to Improve Resistance to Static Disassembly*, 2003.

[10] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, J. H. Hartman, and P. Moseley. A multi-faceted defence mechanism against code injection attacks.

[11] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation.

[12] Peter Szor and Peter Ferrie. *Hunting for Metamorphic*, September 2001.

[13] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, 12 2000.

[14] T. Yetiser. *Polymorphic Viruses: Implementation, Detection, and Protection*. VDS Advanced Research Group, January 1993.