Obfuscation of Executable Code to Improve Resistance to Static Disassembly *

Cullen Linn Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721.
{linnc, debray}@cs.arizona.edu

Abstract

A great deal of software is distributed in the form of executable code. The ability to reverse engineer such executables can create opportunities for theft of intellectual property via software piracy, as well as security breaches by allowing attackers to discover vulnerabilities in an application. The process of reverse engineering an executable program typically begins with disassembly, which translates machine code to assembly code. This is then followed by various decompilation steps that aim to recover higher-level abstractions from the assembly code. Most of the work to date on code obfuscation has focused on disrupting or confusing the decompilation phase. This paper, by contrast, focuses on the initial disassembly phase. Our goal is to disrupt the static disassembly process so as to make programs harder to disassemble correctly. We describe two widely used static disassembly algorithms, and discuss techniques to thwart each of them. Experimental results indicate that significant portions of executables that have been obfuscated using our techniques are disassembled incorrectly, thereby showing the efficacy of our methods.

1 Introduction

Advances in program analysis and software engineering technology in recent years have led to significant improvements in tools for program analysis and software development. Unfortunately, this same technology can, in many cases, be subverted to reverse engineer software systems with the goal of discovering vulnerabilities, making unauthorized modifications, or stealing intellectual property. These all require an ability to take an executable program and reconstruct its high-level structure to some extent. For example, to identify vulnerabilities in a software system, a hacker has to be able to figure out how it works and where it may be attacked. Similarly, to steal a piece of software with an embedded copyright notice or software watermark, a pirate must reconstruct enough of its internal structure to be able to identify and delete the copyright notice or watermark without affecting the functionality of the program.

The problem can be addressed by maintaining the software in encrypted form and decrypting it as needed during execution [1], or using specialized hardware (e.g., see [16]). While effective, such approaches have the disadvantages of high performance overhead or loss of flexibility because the software can no longer be run on stock hardware. An alternative approach, which we focus on, is to use code obfuscation techniques to enhance software security [9, 10, 11, 12, 28]. The goal is to deter attackers by making the cost of the reconstructing the high-level structure of the program prohibitively high.

The processes of compilation and reverse engineering are illustrated in Figure 1. Compilation refers to the translation of a source-language program to machine code; it consists of a series of steps, each producing successively lower-level program representa-

*This work was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.

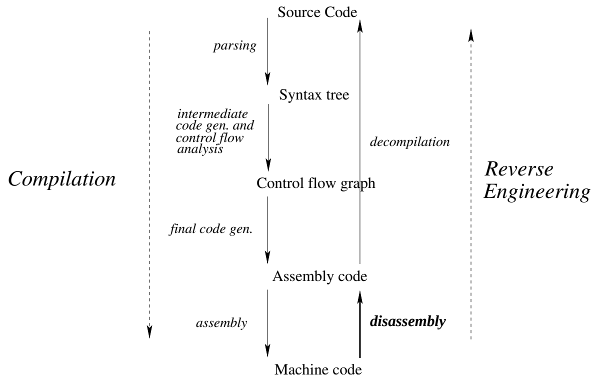


Figure 1: The processes of compilation and reverse engineering

tions. Reverse engineering is the dual process of recovering higher-level structure and semantics from a machine code program. Broadly speaking, we can divide the reverse engineering process into two parts: *disassembly*, which produces assembly code from machine code; and *decompilation*, which reconstructs the higher-level semantic structure of the program from the assembly code. Most of the prior work on code obfuscation and tamper-proofing focus on various aspects of decompilation. For example, a number of researchers suggest relying on the use of difficult static analysis problems, e.g., involving complex Boolean expressions, pointers, or indirect control flow, to make it harder to construct a precise control flow graph for a program [3, 12, 19, 26, 27].

The work described in this paper, by contrast, focuses on the disassembly process. Our goal is to increase the difficulty of statically disassembling a program. As such, our approach is independent of, and complementary to, current approaches to code obfuscation. It is independent of them because our techniques can be applied regardless of whether or not any of these other obfuscating transformations are being used. It is complementary to them because, by making a program harder to disassemble accurately, we add yet another barrier to recovering high-level semantic information about a program.

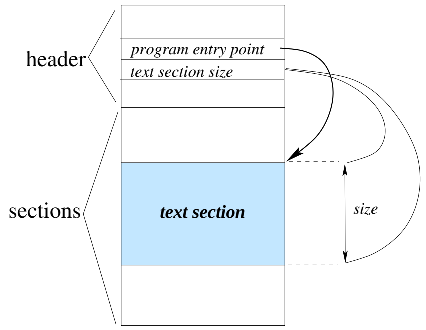


Figure 2: The structure of an executable file

2 Background: Disassembly

A machine code file typically consists of a number of different sections, e.g., *text*, *read-only data*, etc., that contain various sorts of information about the program, together with a header describing these sections. Among other things, the header contains information about the program entry point, i.e., the location in the file where the machine instructions begin (and where program execution begins), and the total size or extent of these instructions[1] (see Figure 2) [15]. Disassembly refers to the process of recovering a sequence of assembly code instructions from such a file, e.g., in a textual format readable by a human being.

Broadly speaking, there are two approaches to disassembly: *static disassembly*, where the file being disassembled is examined by the disassembler but is not itself executed during the course of disassembly; and *dynamic disassembly*, where the file is executed on some input and this execution is monitored by an external tool (e.g., a debugger) to identify the instructions that are being executed. Static disassembly has the advantage of being able to process the entire file all at once, while dynamic disassembly only disassembles a "slice" of the program, i.e., those instructions that were executed for the particular input that was used. Another advantage of static disassembly is that it takes time proportional to the size of the program, while the time taken by dynamic disassembly is typically proportional to the number of instructions

[1] This applies to most file formats commonly encountered in practice, including Unix *a.out*, ELF, COFF, and DOS EXE files. The information about the entry point and code section size is implicit in DOS COM files.

executed by the program at runtime—the former tends to be considerably less than the latter (often by several orders of magnitude), making static disassembly considerably more efficient than dynamic disassembly.

This paper focuses on static disassembly. There are two generally used techniques for this: *linear sweep* and *recursive traversal* [22]. The remainder of this section sketches each of them.

2.1 Linear Sweep

The linear sweep algorithm begins disassembly at the input program's entry point, and simply sweeps through the entire text section disassembling each instruction as it is encountered:

```
global startAddr, endAddr;
proc DisasmLinear(addr)
begin
   while (startAddr ≤ addr ≤ endAddr) do
      I := decode instruction at address addr;
      addr += length(I);
   od
end

proc main()
begin
   ep := program entry point;
   size := text section size;
   startAddr := ep; endAddr := ep + size;
   DisasmLinear(ep);
end
```

This method is used by programs such as the GNU utility *objdump* [18] as well as a number of link-time optimization tools [8, 17, 24].

The main weakness of this algorithm is that it is prone to disassembly errors resulting from the misinterpretation of data that is embedded in the instruction stream. Only under special circumstances, e.g., when an invalid opcode is encountered, can the disassembler become aware of such disassembly errors.

2.2 Recursive Traversal

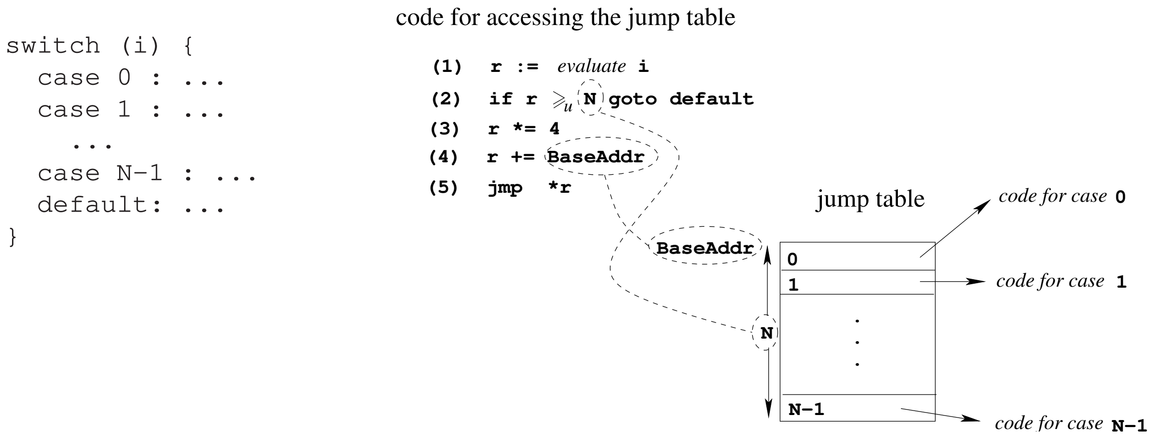
The problem with the linear sweep algorithm is that, because it does not take into account the control flow behavior of the program, it cannot "go around" data (e.g., alignment bytes, jump tables, etc.) embedded in the instruction stream, and mistakenly interprets them as executable code. An obvious fix would be to take into account the control flow behavior of the program being disassembled in order to determine what to disassemble. Intuitively, whenever we encounter a branch instruction during disassembly, we determine the possible control flow successors of that instruction, i.e., addresses where execution could continue, and proceed with disassembly at those addresses (e.g., for a conditional branch instruction we would consider the branch target and the fall-through address):

```
global startAddr, endAddr;
proc DisasmRec(addr)
begin
   while (startAddr ≤ addr ≤ endAddr) do
      if (addr has been visited already) return;
      I := decode instruction at address addr;
      mark addr as visited;
      if (I is a branch or function call)
         for each possible target t of I do
            DisasmRec(t);
         od
         return;
      else addr += length(I);
   od
end

proc main()
begin
   ep := program entry point;
   size := text section size;
   startAddr := ep; endAddr := ep + size;
   DisasmRec(ep, startAddr, endAddr);
end
```

Variations on this basic approach to disassembly, which we term *recursive traversal*, are used by a number of binary translation and optimization systems [4, 23, 25].

A virtue of this algorithm is that, by following the control flow behavior of the program being processed, it is able to "go around" and thus avoid disassembly of data embedded in the text section. Its main weakness is that its key assumption, that we can precisely identify the set of control flow successors of each control transfer operation in the program, may not always hold in the case of indirect jumps. Imprecision in determining the set of possible targets of such a jump will result either in a failure to disassemble some reachable code (if the set of targets is underestimated) or erroneous disassembly of data (if the set of targets is overestimated).



(a) Source code

(b) Implementation using a jump table

Figure 3: A example of a C **`switch`** statement and its implementation using a jump table

Some researchers have proposed *ad hoc* extensions to the basic algorithm outlined above to handle common cases of indirect jumps. As an example, one of the commonest uses of indirect jumps involves *jump tables*, a construct used by compilers to implement C-style **`switch`** statements [2]. This is illustrated in Figure 3. The jump table itself is a contiguous array of *N* code addresses, corresponding to the *N* cases in the **`switch`** statement. The code to access the jump table evaluates the index expression *i*; checks to see whether this expression falls within the bounds of the table, i.e., whether $0 \leq i < N$; adds the scaled value of the index expression to the base address of the table to obtain the address of the i^{th} entry in the table; then jumps indirectly through this location. The check of whether the index expression falls within the table bounds can be accomplished using a single unsigned comparison (denoted by $\geq_u$ in instruction (2) in Figure 3(b)) [2]. To determine the possible target addresses of an indirect jump through a jump table, a disassembler needs to know the base address of the table and its extent, i.e., the values of `BaseAddr` and `N` in Figure 3(b). This can be done by scanning back from the indirect jump instruction to find the instruction that adds the scaled index to the base address (instruction (4) in Figure 3(b)), whence the base address can be extracted; and the unsigned compare of the index (instruction (2) in Figure 3(b)), whence the table size can be determined. Once this has been done, disassembly can continue at each target identified from the *N* table entries starting at location `BaseAddr` [6].

Code that is reachable only through indirect control transfers may not be found using the basic algorithm above. To handle this problem, some systems, e.g., the UQBT binary translation system [5], resort to "speculative disassembly." The idea is to process undisassembled portions of the text segment that appear to be code, in the expectation that they might be the targets of indirect function calls; a "speculative bit" is set when this is done, and speculative disassembly of a particular region of memory is abandoned if an invalid instruction is encountered.

3 Thwarting Disassembly

It is easy to see that in order to thwart a disassembler, we have to somehow confuse, as much as possible, its notion of where the instruction boundaries in a program lie. This section discusses some ways in which this can be achieved. We first discuss a phenomenon that we had not expected: that of disassembly errors that "repair" themselves within a relatively short distance. This is followed by a discussion of a general technique we use to inject "junk bytes" into the instruction stream to introduce disassembly errors. After this we discuss specific details of the way in which this is done to confuse the two disassembly algorithms discussed in the previous section.

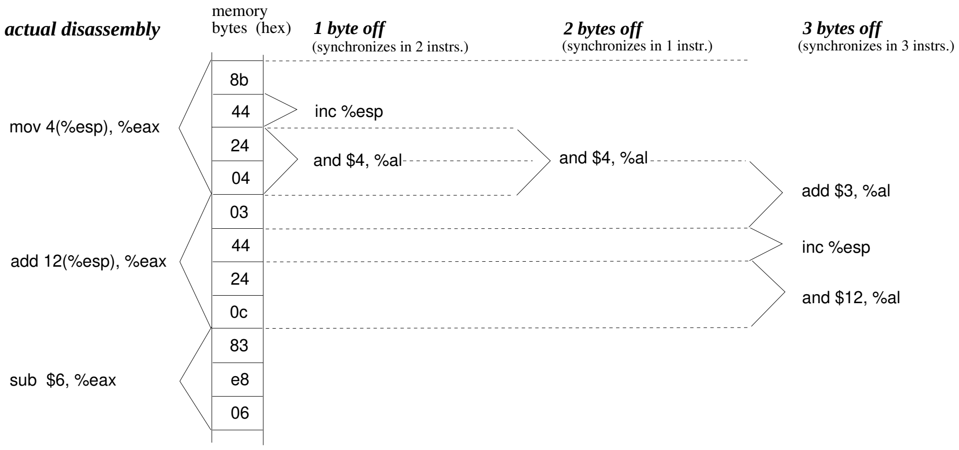


Figure 4: An example of self-repairing disassembly

3.1 Self-Repairing Disassembly

One way to understand the process of static disassembly, and what happens when a disassembly error occurs, is to compare the set of instruction start addresses (i.e., the addresses where each statically disassembled instruction begins) identified by a static disassembler with the "actual" instruction addresses that would be encountered when the program is actually executed. If the static disassembly is completely correct, these two sets will be identical. The effect of a disassembly error is to cause these sets to become different. The goal of the techniques discussed in this paper is to maximize this difference.

It turns out that on some instruction sets—most notably, that of the Intel IA-32 architecture (i.e., the x86 and Pentium)—the instruction structure is such that, very often, the disassembly process is self-repairing: even when a disassembly error occurs (e.g., due to the disassembly of data), the disassembler eventually ends up re-synchronizing with the actual instruction stream. In other words, with such instruction sets the disassembly error results in a nonzero difference between the instruction start addresses identified by the disassembler and the "actual" instruction addresses, but this difference typically goes to zero as the disassembly continues, and after some time the instruction start addresses identified by the disassembler begin to coincide with the actual instruction addresses.

This is illustrated by the example in Figure 4, which shows a typical byte sequence in memory, together with the actual disassembly (on the left), and the disassemblies we obtain if the disassembler is off by 1, 2, or 3 bytes, on the right. When the disassembly is initially off by a single byte, the disassembler produces two erroneous instructions but is back in synchrony with the original disassembly by the second instruction in the actual disassembly sequence. A similar phenomenon occurs when the disassembler is initially off by two bytes: it resynchronizes with the second instruction in the actual disassembly after producing a single incorrectly disassembled instruction. If the disassembler is initially off by three bytes, it generates three incorrectly disassembled instructions but resynchronizes by the third instruction in the actual disassembly.

Obviously, the actual resynchronization behavior on a particular program will depend on its particular distribution of instructions. In practice, however, we have found that disassembly errors usually resynchronize quite quickly—often within just one or two instructions beyond the point at which the disassembly error occurred. Efforts to confuse disassembly have to take this self-repairing aspect of disassembly into account.

3.2 Junk Insertion

We can introduce disassembly errors by inserting "junk" bytes at selected locations in the instruction stream where the disassembler is likely to expect code. (An alternative approach involves partially or fully overlapping instructions, e.g., see [7]: this is discussed in Section 5.) It is not difficult to see that any such junk bytes must satisfy two properties. First, in order to actually confuse the disassembler, the junk bytes must be partial instructions, not complete instructions. Second, in order to preserve program semantics, such partial instructions must be inserted in such a way that they are unreachable at runtime. To this end, define a basic block as a *candidate block* if it can have such junk bytes inserted before it. In order to ensure that any junk so inserted is unreachable during execution, a candidate basic block cannot have execution fall through into it. In other words, the basic block immediately before a candidate block must end in an unconditional control transfer, e.g., an unconditional jump or a return from a function. Candidate blocks can be identified in a straightforward way by scanning the basic blocks of the program after their final memory layout has been determined.

As mentioned in Section 3.1, the static disassembly process very often manages to "re-synchronize" itself after a disassembly error. Once a candidate block B has been identified, we have to determine what junk bytes to insert before it so as to confuse the disassembler as much as possible, i.e., delay this re-synchronization for as long as possible. To do this, we take a particular n-byte instruction I (our current implementation considers a 6-byte bitwise-OR instruction, but it is easy to extend this to other instructions), and determine how far away this re-synchronization would occur if the first k bytes of I were to be inserted immediately before the candidate block B, for each k, $0 < k < n$. To determine the re-synchronization point, for each such k we simulate disassembly for the candidate block, assuming that the disassembler encounters the first k bytes of I at the beginning of B, then continuing with the byte sequence comprising the machine-level encodings of the instructions actually in B. Using this approach we determine the value k_{max} of k for which the re-synchronization distance is maximized, and insert the first k_{max} bytes of I immediately before block B.

3.3 Thwarting Linear Sweep

As observed in Section 2.1, linear sweep disassembly is generally unable to distinguish data embedded in the text section. We can exploit this weakness by inserting "junk" bytes at selected locations in the instruction stream, as discussed in Section 3.2. One point to note here is that since the simulation of disassembly scans forward from each candidate to determine the number of "junk" bytes to be inserted there, it is important to ensure that such decisions made for one candidate are not subsequently invalidated by the insertion of junk into subsequent candidates. To avoid such effects, we consider candidate blocks in a backwards order when inserting junk.

With the approach described thus far, we find that we are typically able to attain a "confusion factor" of about 26%–30% on average—i.e., 26%–30% of the instructions in a program are incorrectly disassembled (confusion factors are discussed in more detail in Section 4). The reason that it is not higher is that candidates for the insertion of junk bytes cannot have execution fall through into them: the preceding block has to end in an unconditional control transfer. We have found that, in programs obtained from a typical optimizing compiler, candidate blocks tend to be around 30 instructions apart on average.[2] This distance, combined with the self-repairing nature of disassembly, means that when disassembly goes wrong after the insertion of junk before a candidate, it typically manages to re-synchronize before the next candidate is encountered. We increase the number of candidates by a transformation called *branch flipping*. The idea is to invert the sense of conditional jumps, by converting code of the form

$$\texttt{b}\mathit{cc}\ \mathit{Addr}$$

where cc represents a condition, e.g., '`eq`' or '`ne`', to

$$\begin{array}{ll} & \texttt{b}\overline{cc}\ L' \\ & \texttt{jmp}\ \mathit{Addr} \\ L': & \end{array}$$

where $\overline{cc}$ is the complementary condition to cc, e.g., a '`beq ...`' is converted to a '`bne ...`'. The basic block at L' now becomes a candidate. With this transformation, the distance between candidate blocks drops to about 12 instructions on average, and the confusion factor rises to about 70%.

[2]These data reflect the SPECint-95 benchmark suite compiled using *gcc* at optimization level `-O3`. The averages given here were computed as geometric means.

3.4 Thwarting Recursive Traversal

The main strength of the recursive disassembly algorithm—its ability to deal intelligently with control flow and thereby disassemble around data embedded in the text segment—also turns out to be a weakness that we can take advantage of to confuse the disassembly process. There are two (related) aspects of recursive traversal that we can exploit. The first is that when it encounters a control transfer, disassembly continues at those locations that are deemed to be the possible control transfer targets. In this context, disassemblers typically assume that commonly encountered control transfers, such as conditional branches and function calls, behave "reasonably." For example, a conditional branch is assumed to have two possible targets: the branch target and the fall through to the next instruction. Similarly, a function call is assumed to return to the instruction immediately following the call instruction.

The second aspect of recursive traversal is that identifying the set of possible targets of indirect control transfers is difficult. Recursive traversal disassemblers therefore generally resort to *ad hoc* techniques, such as examining bounds checks associated with jump tables, or disassembling speculatively, to handle commonly encountered situations involving indirect jumps.

Below we discuss different ways in which these characteristics can be exploited to confuse recursive traversal disassembly.

3.4.1 Branch Functions

The assumption that a function returns to the instruction following the call instruction can be exploited using what we term *branch functions*. The idea is illustrated in Figure 5. Given a finite map φ over locations in a program

$$\varphi = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$$

a branch function f_φ is a function such that, whenever it is called from one of the locations a_i, causes control to be transferred to the corresponding location b_i, $1 \leq i \leq n$. Given such a branch function f_φ, we can replace n unconditional branches in a program,

a_1 : `jmp` b_1
...
a_2 : `jmp` b_2
...
a_n : `jmp` b_n

by calls to the branch function:

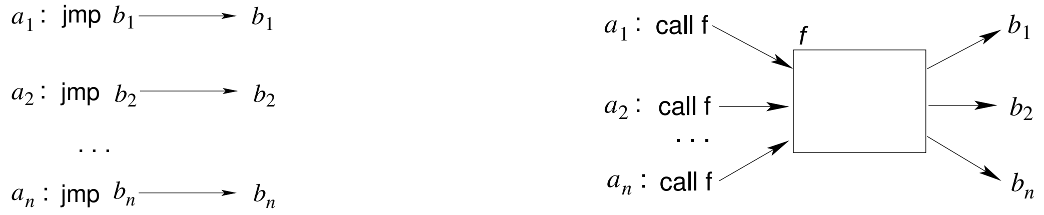
a_1 : `call` f_φ
...
a_2 : `call` f_φ
...
a_n : `call` f_φ

The code for the branch function is responsible for determining the target location b_i based on the location a_i it was called from, then branching to the appropriate b_i. Moreover, it has to do this in such a way that the program state is that which would have been encountered at the location b_i in the original code with unconditional branches. Note that a branch function does not behave like "normal" functions, in that it typically does not return to the instruction following the call instruction, but instead branches to some other location in the program that depends, in general, on where it was called from.[3]

Branch functions serve two distinct purposes. The first is to obscure the flow of control in the program: by sufficiently obscuring the computation of the target address b_i within the branch function, we can make it difficult for an attacker to reconstruct the original map φ it realizes. The second is to create opportunities for misleading a disassembler: since a disassembler will typically continue disassembly at the instruction following the call instruction, we can introduce errors in the disassembly by inserting junk bytes at the point immediately after each '`call` f_φ' instruction in a manner similar to that discussed in Section 3.3.

Branch functions can be implemented in a number of ways. Trivial implementations involve looking up a table (via a simple linear or binary search) using the return address passed to the branch function to determine the target address. Such implementations have the disadvantage of being relatively straightforward to reverse engineer. A more sophisticated approach might use perfect hashing, e.g., as discussed by Fredman *et al.* [14], or other schemes that are difficult to reverse engineer. Indeed, the complexity of

[3] This can, however, have adverse performance implications on some architectures, e.g., the Intel Pentium, by interfering with the branch prediction and/or return stack buffer mechanisms.



(a) Original code

(b) Code using a branch function

Figure 5: Branch functions

a branch function's implementation, and the way in which it is accessed, offer an interesting tradeoff between execution speed, on the one hand, and difficulty of reverse engineering, on the other. For example, we can choose different branch function implementations for jump instructions depending on their execution frequencies: frequently executed jump instructions might be directed to a lightweight branch function, less frequently executed ones to a more complex branch function, and so on. Moreover, a particular jump instruction in a program can be made part of the branch mapping for several different branch functions, with one of them being chosen in some arbitrary (and dynamically variable) manner at runtime.

Our current implementation uses a scheme where the callee passes, as an argument to the branch function, the offset from the instruction immediately after it (whose address is passed to the branch function as the return address) to the target b_i. The branch function simply adds the value of its argument to the return address, so that the return address becomes the address of the original target b_i. The code we use for this, on the Intel IA-32 architecture, is as follows:[4]

```
xchg %eax, 0(%esp)       # I1
add  8(%esp), %eax       # I2
pop  %eax                # I3
ret                      # I4
```

Instruction I1 exchanges the contents of register `%eax` with the word at the top of the stack, effectively saving the contents of `%eax` and at the same time loading the displacement to the target (passed to the branch function as an argument on the stack) into `%eax`. Instruction I2 then has the effect of adding this displacement to the return address. I3 restores the previously saved value of `%eax`, and I4 then has the effect of branching to the address computed by the function. Since the resulting code is nevertheless quite a bit more expensive, in execution cost, than the single branch instruction in the original program, we use execution profile information to apply the transformation only to code that is not "hot," i.e., that is not frequently executed; the details are discussed in Section 4.

3.4.2 Opaque Predicates

The assumption that a conditional branch has two possible targets can be exploited by disguising an unconditional branch as a conditional branch that happens, at runtime, to always go in one direction—i.e., either it is always taken, and never falls through; or it is never taken, and always falls through. This technique relies on using predicates that always evaluate to either the constant *true* or the constant *false*, regardless of the values of their inputs: such predicates are known as *opaque predicates* [12]. Other researchers have discussed techniques for synthesizing opaque predicates; their ideas translate in a straightforward way to our context, so we do not discuss this issue further.

Once an unconditional branch has been replaced by a conditional branch that uses an opaque predicate, we have a location—either the branch target or the fall through, depending on whether the opaque predicate is always false or always true—that appears to be a legitimate continuation for execution from the condi-

[4] If any of the condition code flags is live at the call point, they have to be saved by the caller just before the call, and restored at the target.

tional branch but, in fact, is not. We can then insert junk bytes at this point, as discussed earlier, to mislead the disassembly.

3.4.3 Jump Table Spoofing

We can generalize the notion of opaque predicates to that of opaque expressions—i.e., expressions that always evaluate to the same constant value, but where it is difficult to discern this from an examination of the code to evaluate the expression. Let $mem[a]$ denote the contents of a memory location with address a. Then, given an opaque predicate e that always evaluates to 0 (i.e., *false*), and a set S of arbitrary but legal memory addresses, we can construct an opaque expression that always evaluates to 0 as

$$(\sum_{\ell \in S} mem[\ell]) \ \& \ e$$

where & denotes a bitwise-AND operation. The actual value of the sum expression $\sum_{\ell \in S} mem[\ell]$ is unimportant here, since the only purpose of the memory loads is to confuse an attacker or a disassembler about the value that is being computed; we care only that the addresses in the set S be legal readable addresses, so that the memory references do not generate any exceptions. Since the opaque predicate e always evaluates to 0, the result of the bitwise-AND operation is also 0. Similarly, given an opaque predicate e' that always evaluates to 1 (i.e., *true*), the expression

$$(\sum_{\ell \in S} mem[\ell]) \ \& \ (1 - e')$$

will always evaluate to 0. To obtain an opaque expression that evaluates to some other value c, we can then simply add the number c to the result of an opaque expression that evaluates to 0.

We can use the values of opaque expressions with artificial jump table constructs to mislead a disassembler. We refer to this technique as *jump table spoofing*. Recall that, as discussed in Section 2.2, recursive traversal disassemblers attempt to use the bounds check for a jump table to identify its size, and thereby determine the set of possible targets of an indirect jump through a jump table. The basic idea is that we take an unconditional jump to an address ℓ and convert it to an indirect jump through a jump table where the address ℓ appears as the k^{th} table entry. The table is indexed by the value of an opaque expression that always evaluates to k. However, the bounds check for the table uses a table size $m > k$, leading the disassembler to believe that the jump table contains m entries. Only one of these m entries contains a real code address: we can put "junk addresses"—text segment addresses that do not correspond to actual instructions—into each of the other entries, thereby confusing the disassembler.

3.5 Implementation Status

We have implemented our ideas using PLTO, a binary rewriting system developed for Intel IA-32 executables running Linux [21]. The system reads in statically linked executables,[5] disassembles the input binary, and constructs a control flow graph (eliminating, in the process, unreachable code such as unused library routines that have been linked in). This control flow graph is then processed in one of two ways. If the user specifies that profiling is to be carried out, instrumentation code is inserted to generate an edge profile when the resulting binary is executed. If, on the other hand, the user requests obfuscating transformations to be carried out, the system reads in edge profile information if available, generates a memory layout for the code using the profile data [20], carries out branch flipping to increase the number of candidate blocks (Section 3.3), applies various obfuscating transformations, and writes out the resulting executable.

The transformations currently implemented in the system are junk insertion (Section 3.2) and transformation of unconditional jumps to branch function calls (Section 3.4.1). We expect to have additional transformations, such as jump table spoofing (Section 3.4.3), implemented in the near future.

4 Experimental Evaluation

We evaluated the efficacy of our techniques using the SPECint-95 benchmark suite. Our experiments were run on an otherwise unloaded 2 GHz Pentium IV system with 1 GB of main memory running RedHat

[5] The requirement for statically linked executables is a result of the fact that PLTO relies on the presence of relocation information to distinguish addresses from data. The Unix linker `ld` refuses to retain relocation information for executables that are not statically linked.

Linux 7.3. The programs were compiled with *gcc* version *egcs-2.91.66* at optimization level `-O3`. The programs were profiled using the SPEC training inputs and these profiles were used to identify any hot spots during our transformations. The final performance of the transformed programs were then evaluated using the SPEC reference inputs. Each execution time reported was computed as the average of three runs.

We experimented with three different "attack disassemblers" to evaluate our techniques. Two of these are disassemblers we wrote ourselves: a straightforward linear sweep disassembler, and a recursive disassembler that uses bounds checks to handle indirect jumps through jump tables and which incorporates speculative disassembly (see Section 2). In order to avoid unduly optimistic results, we provide the recursive disassembler with additional information about the address and size of each jump table in the program as well as the start and end address of each function. The results obtained from this disassembler therefore serve as a lower bound estimate of the extent of obfuscation achieved. Our third disassembler is IDA Pro [13], a commercially available disassembly tool that is generally regarded to be among the best disassemblers available.

For each of these, the efficacy of obfuscation was measured by computing "confusion factors" for the instructions, basic blocks, and functions. Intuitively, the confusion factor measures the fraction of program units (instructions, basic blocks, or functions) in the obfuscated code that were incorrectly identified by a disassembler. For instructions, we compare the actual instruction sequence produced by our tool with that obtained by applying a conventional disassembler to the output of our tool, and use a minimal edit distance computation (in essence, a *diff*) to count the number of instructions that were incorrectly disassembled. The computations for functions and basic blocks are similar, the only conceptual difference being that a basic block or function is counted as being "incorrectly disassembled" if any of the instructions in it is incorrectly disassembled. The reason for computing confusion factors for basic blocks and functions as well as for instructions is to determine whether the errors in disassembling instructions are clustered in a small region of the code, or whether they are distributed over significant portions of the program.

PROGRAM	EXECUTION TIME (SECS)		
	Original (T_0)	*Obfuscated* (T_1)	*Slowdown* (T_1/T_0)
compress	44.45	75.70	1.70
gcc	26.08	37.61	1.44
go	57.89	75.52	1.30
ijpeg	44.15	52.27	1.18
li	29.13	48.18	1.65
m88ksim	31.66	56.90	1.80
perl	30.46	68.46	2.24
vortex	55.40	63.47	1.14
Geo. mean			1.52

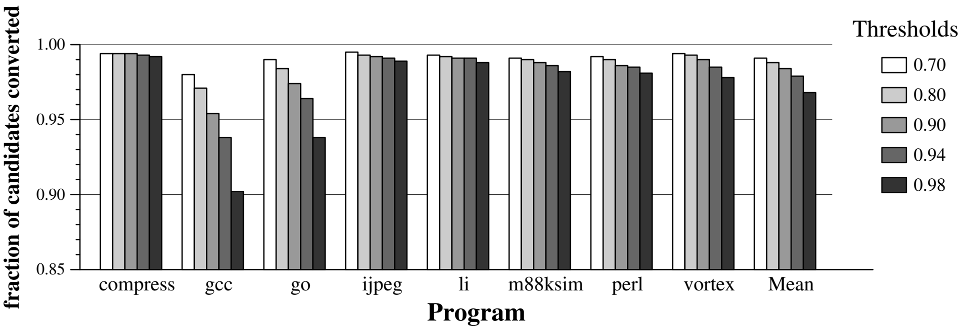
Figure 8: Effect of obfuscation on execution speed ($\theta = 0.98$)

As mentioned in Section 3.4.1, we transform jumps to branch function calls only if the jump does not occur in a "hot" basic block. The first questions we have to address, therefore, are: how are hot basic blocks identified, and what is the effect of different choices of what consititutes a "hot" block on the extent of obfuscation achieved and the performance of the resulting code? To identify the "hot," or "frequently executed," basic blocks, we start with a (user-defined) fraction θ ($0.0 < \theta \leq 1.0$) that specifies what fraction of the total number of instructions executed at runtime should be accounted for by "hot" basic blocks. For example, $\theta = 0.8$ means that hot blocks should account for at least 80% of all the instructions executed by the program. More formally, let the *weight* of a basic block be the number of instructions in the block multiplied by its execution frequency, i.e., the block's contribution to the total number of instructions executed at runtime. Let *tot_instr_ct* be the total number of instructions executed by the program, as given by its execution profile. Given a value of θ, we consider the basic blocks b in the program in decreasing order of execution frequency, and determine the largest execution frequency N such that

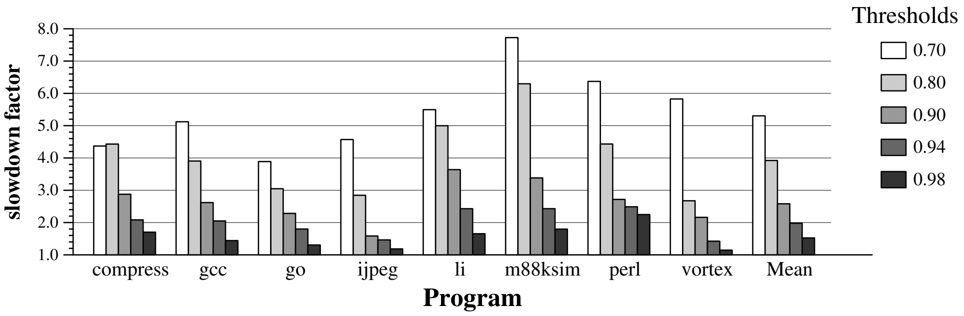
$$\sum_{b:freq(b) \geq N} weight(b) \quad \geq \quad \theta \cdot tot\_instr\_ct.$$

Any basic block whose execution frequency is at least N is considered to be hot.

The effect of varying the hot code threshold θ on performance (both obfuscation and speed) is shown in



(a) No. of jumps converted to branch function calls



(b) Slowdown in execution speed

Figure 6: Effect of “hot code threshold” on branch function conversion and execution speed

Figure 6. Figure 6(a) shows the fraction of candidates that are converted to branch function calls at different thresholds; this closely tracks the overall confusion factors achieved. Figure 6(b) shows the concomitant degradation in performance. It can be seen, from Figure 6(a), that most programs have a small and well-defined hot spot, and as a result varying the threshold from a modest 0.70 to a value as high as 0.98 does not dramatically affect the number of candidates converted. The benchmark that is affected the most is *gcc*, and even here over 90% of the candidates are converted at $\theta = 0.98$. On average, about 97% of the candidates are converted at $\theta = 0.98$. However, as illustrated in Figure 6(b), varying the hot code threshold has a significant effect on execution speed. For example, at $\theta = 0.70$ the programs slow down by a factor of 5.3× on average, with the *m88ksim* benchmark experiencing the largest slowdown, by a factor of 7.7×. However, as θ is increased the slowdown factor drops off quickly, to 2.6× at $\theta = 0.9$ and 1.5× at $\theta = 0.98$. In summary, choosing a threshold θ that is close to 1.0 still results in most of the candidate blocks in the program being converted to branch function calls without excessive performance penalty. For the purposes of this paper, therefore, we give measurements for $\theta = 0.98$.

Figure 7 shows the efficacy of our obfuscation transformations for both of the disassembly methods discussed in Section 2. The confusion factors achieved for linear sweep disassembly are quite high: on average, 75% of the instructions, 56% of the basic blocks, and 88% of the functions are incorrectly disassembled. For recursive traversal, the confusion factors attained are somewhat lower because in this case the disassembler can understand and deal with control flow somewhat better than with linear sweep. Nevertheless, we find that, on average, over 40% of the instructions in the program incur disassembly errors. As a result, over 21% of the basic blocks and close to 48% of the functions, on average, are incorrectly disassembled using

PROGRAM	Confusion factor (%)					
	LINEAR SWEEP			RECURSIVE TRAVERSAL		
	Instructions	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*
compress	82.0	59.6	88.0	47.6	26.7	51.1
gcc	70.5	52.8	86.7	34.1	15.1	50.7
go	75.5	55.7	92.7	38.3	19.2	50.1
ijpeg	75.4	58.2	88.5	39.8	22.4	47.7
li	77.5	57.3	77.1	44.4	24.6	39.8
m88ksim	80.0	59.3	90.0	44.2	24.2	51.1
perl	75.2	56.5	89.0	44.0	24.3	49.9
vortex	65.4	50.6	92.9	34.5	17.8	43.4
Geo. mean	75.0	56.2	88.0	40.6	21.4	47.8

Figure 7: Efficacy of obfuscation: confusion factors ($\theta = 0.98$)

this disassembly method. This is achieved at the cost of a 52% penalty in execution speed (see Figure 8); we are currently working on techniques to reduce this performance penalty by significant amounts.

The reason the confusion factors for basic blocks are lower than those for instructions and functions, for both disassembly methods, is that basic blocks can vary quite widely in size, ranging from a single instruction in many cases to several instructions in others. When a disassembly error occurs, there is a high likelihood that several nearby instructions will be incorrectly disassembled. However, if a group of such incorrectly disassembled instructions falls within a basic block, that will count as just a single incorrectly disassembled block. On the other hand, when a basic block consists of just a single instruction, and this instruction is correctly disassembled, it counts as a correctly disassembled block. In other words, when we view the distribution of disassembly errors at the granularity of basic blocks, the clustering effects of incorrect disassembly result in a smaller confusion factor. By contrast, when viewed at the level of functions, the clustering effects are less significant (because functions rarely get as small in size as basic blocks), and the resulting confusion factors are higher.

The data reported in Figure 7 are actually quite conservative: in an effort to avoid unduly optimistic results, we supply our disassemblers with additional information, e.g., the size and extent of all the jump tables in the program, that helps with disassembly. To evaluate the efficacy of our techniques in a more realistic situation, we used a commercial disassembly tool, IDA Pro version 4.3x [13], which is widely considered to be the most advanced disassembler available. The results of this experiment are reported in Figure 9. It can be seen that this tool fails on most of the program: close to 94% of the instructions, and about 83% of the functions in the program, are disassembled incorrectly. Part of the reason for this high degree of failure is that IDA Pro only disassembles addresses that (it believes) can be guaranteed to be instruction addresses. This has two effects: first, large portions of the code that are reached by branch function addresses are simply not disassembled, being presented instead to the user as a jumble of hex data; and second, the location immediately following a branch function call is treated as an address to which control returns, and this causes some junk bytes to be erroneously disassembled. Overall, this shows that our techniques are effective even against state-of-the-art disassembly tools.

Finally, Figure 10 shows the impact of obfuscation on code size, both in terms of the number of instructions (which increases, for example, due to branch flipping), as well as the number of bytes occupied by the text section. The latter includes the effects of the new instructions inserted as well as all junk bytes added to the program. Overall, it can be seen that there is a 42% increase in the total number of instructions, and a 34% increase in the size of the text section of the resulting executables.

The techniques described here apply to a wide variety of architectures. The insertion of partial instruc-

PROGRAM	NO. OF INSTRUCTIONS			TEXT SECTION SIZE (BYTES)		
	Original (I_0)	Obfuscated (I_1)	Change (I_1/I_0)	Original (S_0)	Obfuscated (S_1)	Change (S_1/S_0)
compress	74785	108794	1.455	265975	360793	1.356
gcc	327131	463982	1.418	1128263	1514985	1.343
go	124422	176296	1.417	468527	616494	1.316
ijpeg	105764	148328	1.402	363159	483349	1.331
li	89307	127816	1.431	310291	418347	1.348
m88ksim	104209	150078	1.440	368788	498547	1.352
perl	137945	198231	1.437	484184	653461	1.350
vortex	174958	241583	1.381	592066	783316	1.323
Geo. mean			1.422			1.340

Figure 10: Effect of obfuscation on code size ($\theta = 0.98$)

PROGRAM	*Confusion factor (%)*		
	Instructions	*Basic blocks*	*Functions*
compress	98.5	73.2	83.8
gcc	91.0	66.2	82.2
go	93.1	68.7	89.6
ijpeg	94.1	72.4	83.9
li	93.9	70.2	70.8
m88ksim	96.8	71.8	85.1
perl	95.8	71.8	84.6
vortex	87.3	68.1	88.6
Geo. mean	93.8	70.3	83.4

Figure 9: Efficacy of obfuscation: IDA Pro disassembler ($\theta = 0.98$)

tions to confuse disassembly, as discussed in Section 3.2, is applicable to variable-length instruction sets, such as those on the widely used Intel Pentium and Motorola 680x0, as well as the StrongArm (together with the Thumb 16-bit instruction encoding) and other mixed-mode architectures such as the MIPS32/MIPS16. Branch functions and jump table spoofing can be used on any architecture.

5 Related Work

The only work we are aware of that addresses the problem of making executable programs harder to disassemble is by Cohen, who proposes overlapping adjacent instructions to fool a disassembler [7]. We are not aware of any actual implementations of this proposal. We implemented this idea as well as a number of variations on the basic scheme, but found the results disappointing: the resulting confusion factors were typically less than 1%. The reason for this is that in order to overlap two adjacent instructions I and J, we have to satisfy several conditions, among them:

(i) execution cannot fall through from I to J; and

(ii) the trailing k bytes of I must be identical with the leading k bytes of J for some $k > 0$.

There tend to be relatively very few candidates satisfying these criteria (e.g., the largest number of overlaps we achieved was for the *gcc* benchmark, where we found only 27 overlaps out of 360,152 instructions; by contrast, our approach can use 11,205 candidates before branch flipping on this program, and 38,927 candidates after branch flipping). Variations on this theme, e.g., by judicious insertion, immediately before the instruction J, of no-ops or dead code that satisfy the second condition above, do not seem to help matters significantly either. This scarcity of candidates for overlapping, together with the self-repairing property of disassembly errors discussed in Section 3.1, results in poor confusion factor numbers using this approach.

There is a considerable body of work on code obfuscation that focuses on making it harder for an attacker to decompile a program and extract high level semantic information from it [3, 12, 19, 26, 27, 28]. Typically, these authors rely on the use of computationally difficult static analysis problems, e.g., involving complex Boolean expressions, pointers, or indirect control flow, to make it harder to construct a precise control flow graph for a program. Of the references cited, only Wroblewski focuses specifically on obfuscation of executable programs [28]. Our work is orthogonal to these proposals, and complementary to them. We aim to make a program harder to disassemble correctly, and to thereby sow uncertainty in an attacker's mind about which portions of a disassembled program have been correctly disassembled and which parts may contain disassembly errors. If the program has already been obfuscated using any of these higher-level obfuscation techniques, our techniques add an additional layer of protection that makes it even harder to decipher the actual structure of the program.

Even greater security may be obtained by maintaining the software in encrypted form and decrypting it as needed during execution, as suggested by Aucsmith [1]; or using specialized hardware, as discussed by Lie *et al.* [16]. While extremely effective, such approaches have the disadvantages of high performance overhead (in the case of runtime decryption in the absence of specialized hardware support) or a loss of flexibility because the software can no longer be run on stock hardware.

6 Conclusions

A great deal of software is distributed in the form of executable code. Such code is potentially vulnerable to reverse engineering, in the form of disassembly followed by decompilation. This can allow an attacker to discover vulnerabilities in the software, modify it in unauthorized ways, or steal intellectual property via software piracy. This paper describes and evaluates techniques to make executable programs harder to disassemble. Our techniques are seen to be quite effective: applied to the widely used SPECint-95 benchmark suite, they result in disassembly errors over much of the program; the best commercially available disassembly tool fails to correctly disassemble over 93% of the instructions, and 83% of the functions, in the obfuscated binaries.

References

[1] D. Aucsmith. Tamper-resistant software: An implementation. In *Information Hiding: First International Workshop: Proceedings*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.

[2] R. L. Bernstein. Producing good code for the case statement. *Software—Practice and Experience*, 15(10):1021–1024, October 1985.

[3] W. Cho, I. Lee, and S. Park. Againt intelligent tampering: Software tamper resistance by extended control flow obfuscation. In *Proc. World Multiconference on Systems, Cybernetics, and Informatics*. International Institute of Informatics and Systematics, 2001.

[4] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.

[5] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.

[6] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2–3):171–188, July 2001.

[7] F. B. Cohen. Operating system protection through program evolution, 1992. `http://all.net/books/IP/evolve.html`.

[8] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.

[9] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Proc. 26th. ACM Symposium on Principles of Programming Languages (POPL 1999)*, pages 311–324, January 1999.

[10] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protecti on. Technical Report TR00-03, The Department of Computer Science, University of Arizona, February 2000.

[11] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 IEEE International Conference on Computer Languages*, pages 28–38.

[12] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. 25th. ACM Symposium on Principles of Programming Languages (POPL 1998)*, pages 184–196, January 1998.

[13] DataRescue sa/nv, Liége, Belgium. IDA Pro. `http://www.datarescue.com/idabase/`.

[14] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[15] J. R. Levine. *Linkers and Loaders*. Morgan Kaufman Publishers, San Francisco, CA, 2000.

[16] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. 9th. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[17] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere. `alto`: A link-time optimizer for the Compaq Alpha. *Software—Practice and Experience*, 31:67–101, January 2001.

[18] Objdump. *GNU Manuals Online*. GNU Project—Free Software Foundation. `http://www.gnu.org/manual/binutils-2.10.1/html_chapter/binutils_4.html`.

[19] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEEE Trans. Fundamentals*, E86-A(1), January 2003.

[20] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[21] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[22] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pages 45–54, October 2002.

[23] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[24] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[25] H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. 7th Conference on Real-Time Computing Systems and Applications*, December 2000.

[26] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, July 2001.

[27] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, 12 2000.

[28] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.