Binary Obfuscation Using Signals[*]

Igor V. Popov, Saumya K. Debray, Gregory R. Andrews
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{ipopov, debray, greg}@cs.arizona.edu

ABSTRACT

Reverse engineering of software is the process of recovering higher-level structure and meaning from a lower-level program representation. It can be used for legitimate purposes—e.g., to recover source code that has been lost—but it is often used for nefarious purposes—such as to search for security vulnerabilities in binaries or to steal intellectual property. The first step in reverse engineering a binary program is to disassemble the machine code into assembly code. This paper addresses the topic of making reverse engineering of binaries hard by making it difficult to statically disassemble machine code. The starting point is an executable binary program. The executable is then obfuscated by changing many control transfers into signals (traps) and inserting dummy control transfers and "junk" instructions after the signals. The resulting code is still a correct program, but current disassemblers are unable to disassemble from 30 to 80 percent of the instructions in the program. Furthermore, the disassemblers have a mistaken understanding of over half of the control flow edges. However, the obfuscated program necessarily executes more slowly than the original. Experimental results quantify the tradeoff between the degree of obfuscation and the increase in execution time.

1. INTRODUCTION

Software is often distributed in binary form, without source code. Many groups have developed technology that enables one to reverse engineer binary programs and thereby reconstruct the actions and structure of the program. This is accomplished by disassembling machine code into assembly code and then possibly decompiling the assembly code into higher level representations [2, 4, 5, 15, 16, 23]. While reverse-engineering technology has many legitimate uses, it can also be used to discover vulnerabilities, make unauthorized modifications, or steal intellectual property. For example, a hacker might probe for security vulnerabilities by figuring out how a software system works and where it might be attacked. Similarly, a software pirate might steal a piece of software with an embedded copyright notice or software watermark by reconstructing enough of its internal structure to identify and delete the copyright notice or watermark without affecting the functionality of the program.

One way to prevent reverse engineering is to ship and store binaries in encrypted form. This can provide theoretically perfect protection, but it requires decrypting binaries during execution [1] or having execute-only memory and decrypting binaries only when loading them into that memory [20]. Consequently, this approach has high performance overhead and requires special hardware. An alternative approach is to leave binaries in executable form, but to use code obfuscation techniques to make reverse engineering difficult [9, 10, 11, 12]. The goal here is to deter attackers by making the cost of reconstructing the high-level structure of a program prohibitively high.

Most of the prior work on code obfuscation and tamper-proofing has focused 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 hard to construct a precise control flow graph for a program [3, 12, 25, 30, 31]. By contrast, the work described in this paper focuses on making static disassembly hard. Thus, our work 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 other obfuscating transformations are being used. It is complementary to them because making a program harder to disassemble adds another barrier to recovering high-level information about a program.

This paper describes two techniques for obfuscating binaries that together confound current disassemblers. The primary technique is to replace control transfer instructions—jumps, calls, and returns—by instructions that raise traps at runtime; these traps are then fielded by signal handling code that carries out the appropriate control transfer. The effect is to replace control transfer instructions with either apparently innocuous arithmetic or memory operations, or with what appear to be illegal instructions that suggest an erroneous disassembly. The secondary technique is to insert (unreachable) code after traps that contains fake control transfers and that makes it hard to find the beginning of the true next instructions. We also disguise the traps and the actions of the signal handler in order to further conceal what is actually going on. These techniques significantly extend ones that were introduced in an earlier paper from our research group [21], and they provide a much higher degree of obfuscation. In particular, they cause the best disassemblers to miss from 30 to 80 percent of the instructions in test programs and to make mistakes on over half of the control flow edges.

The remainder of the paper is organized as follows. Section 2 provides background information on the disassembly process. Section 3 describes the new techniques for thwarting disassembly and explains how they are implemented. Section 4 describes how we evaluate the efficacy of our approach. Section 5 gives experimental results for programs in the SPECint-2000 benchmark suite. Section 6 describes related work, and Section 7 contains concluding remarks.

2. DISASSEMBLY ALGORITHMS

The techniques described in this paper thwart the disassembly process by replacing unconditional control transfers by traps to signal handlers and by inserting junk "instructions" into the binary at

[*]This work was supported in part by NSF Grants EIA-0080123, CCR-0113633, and CNS-0410918

selected places. To provide the context needed to understand these techniques and how they work, this section summarizes the operation of disassemblers.

A machine code file typically consists of a number of different sections—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 location in the file where the machine instructions begin (and where program execution begins), and the total size or extent of these instructions[1] [19]. 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 processes the entire file all at once, while dynamic disassembly only disassembles those instructions that were executed for the particular input that was used. Moreover, with static disassembly it is easier to apply various offline program analyses to reason about semantic aspects of the program under consideration. For these reasons, static disassembly is a popular choice for low level reverse engineering.

This paper focuses on static disassembly. There are two generally used techniques for this: *linear sweep* and *recursive traversal* [26]. The linear sweep algorithm begins disassembly at the input program's first executable byte, and simply sweeps through the entire text section disassembling each instruction as it is encountered. This method is used by programs such as the GNU utility `objdump` [24] as well as a number of link-time optimization tools [8, 22, 28]. The main weakness of linear sweep is that it is prone to disassembly errors resulting from the misinterpretation of data that is embedded in the instruction stream (e.g., jump tables). Only under special circumstances, e.g., when an invalid opcode is encountered, can the disassembler become aware of such disassembly errors.

The recursive traversal algorithm uses the control flow behavior of the program being disassembled in order to determine what to disassemble. It starts with the program's entry point, and disassembles the first basic block. When the algorithm encounters a control flow instruction, it determines the possible successors of that instruction—i.e., addresses where execution could continue—and proceeds with disassembly at those addresses. Variations on this basic approach to disassembly are used by a number of binary translation and optimization systems [5, 27, 29]. The main virtue of recursive traversal is that by following the control flow of a program, it is able to "go around" and thus avoid disassembly of data embedded in the text section. Its main weakness is that it depends on being able to determine the possible successors of each such instruction, which is difficult for indirect jumps and calls.[2] The algorithm obviously also depends on being able to find all the instructions that affect control flow. Errors in determining control flow will result either in 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).

[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.

[2]For common cases of indirect jumps, this can be handled by *ad hoc* extensions to the basic algorithm. For example, switch statements are often compiled into code that uses jump tables, and the disassembler can often find the base and size of such a table by looking for and then analyzing a code pattern that indicates a jump table is being used. A more general technique is to use "speculative disassembly" [6]. The idea here 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.

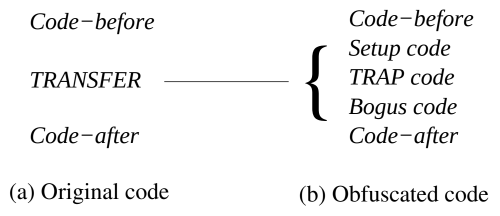


Figure 1: Summary of Source Code Transformations

3. SIGNAL-BASED OBFUSCATION

In order to confuse a disassembler, we have to disrupt its notion of where the instructions are, what they are doing, and what the control flow is. Our goal is to cause a lot of confusion without actually changing the observable behavior of the program. The choices we have for altering the program are (1) changing instructions to others that produce the same result, and (2) adding instructions that do not have visible effects. Simple, local changes will obviously not confuse a disassembler or a human. More global and drastic changes are required.

Our main obfuscation technique is to change many unconditional control transfers—jumps, calls, and returns—into code sequences that cause traps and raise signals, and to add a signal handler to the program that simulates the effects of the altered control transfers. This muddies the output of a linear sweep disassembler, because the control transfer instructions are replaced by seemingly innocuous "normal" instructions, and it confuses a recursive traversal disassembler, because it obscures transfer targets and hence often prevents the disassembler from finding and disassembling reachable code.

Our secondary obfuscation technique is to add bogus instructions after the trap points (which are unreachable code) to further confuse a disassembler's notion of control flow and instruction boundaries. In particular, we add a conditional jump, which will be disassembled but will not ever be taken, and we add junk bytes that will cause the disassembler to incorrectly disassemble one or more of the real (i.e., original) instructions that actually follow the trap point.

Below we give an overview of how these techniques are implemented. Then we describe in detail how control transfers are changed to signal-raising and bogus code, how the signal handler works, and how we deal with interactions between our signals and other signals or signal handlers that might be present in the original program.

3.1 Overview

Figure 1 summarizes how we obfuscate a program. Part (a) contains a fragment of machine code in the original program. TRANSFER is a jump, call, or return instruction. For ease of implementation, we consider only direct jumps and calls—i.e., those whose target address is specified in the code.[3] All return instructions are considered, because the target of the return is stored in a known location on the stack. The TRANSFER instruction is preceded and followed by some other code in the original program; these are indicated by *Code-before* and *Code-after*.

[3]Indirect jumps and calls are much less frequent in practice, so this restriction has essentially no effect on our obfuscation results.

Figure 1(b) contains the corresponding code fragment in the obfuscated program. The TRANSFER instruction is replaced by three code sequences, which are described in detail in the next section:

- Setup code, which prepares for raising a signal.
- TRAP code, which causes a trap that raises a signal.
- Bogus code, which will be disassembled but will not actually be executed.

The code that was before and after the TRANSFER instruction is unchanged and remains before and after the new code that is inserted in the binary in place of the TRANSFER instruction.

When an obfuscated TRANSFER instruction is executed, it causes a trap that in turn raises a signal. The role of the signal handler is to transfer control to the original target of the TRANSFER instruction. To effect this, we build a table that contains mappings from the addresses of TRAP instructions to the target(s) of the corresponding TRANSFER instruction. For a jump, the target is the address in the original instruction. For a call, there are two targets: the address of the called function, and the address to which the call would normally return. For a return, the target address will be on the stack (and hence does not need to be stored in the mapping table).

When our signal handler gets control, the source address S of the trap that raised the signal is in a known location. (The address is placed on the stack by the processor when a trap occurs.) Our signal handler saves this value in a different stack location, then "tricks" the kernel into returning control to a special block of code, the Restore block. The Restore block uses the saved value of S to index the mapping table and retrieve the target address(es). The Restore block then restores the machine state to what it was just before the TRANSFER in the original program and finally transfers control to the original target of the TRANSFER instruction.

We could conceivably obfuscate every jump, call, or return in the source code. However, this would cause the program to execute *much* more slowly because of signal-processing overhead. We allow the user to specify a hot-code threshold, and we only obfuscate control transfers that are not in hot parts of the original program. (See Section 4 for how the hot-code threshold is defined and computed.) Thus, before obfuscating a program, we first instrument the program to gather edge profiles, and then we run the instrumented version on a training input.

The obfuscation process itself has three steps. First, using the profile data and user-specified threshold, determine which control transfers should be obfuscated and modify each such instruction as shown in Figure 1. Second, recompute memory layout and construct the table of mappings from TRAP instructions to target addresses. Finally, assemble a new, obfuscated binary that includes our signal handler and restore code.

3.2 Program Obfuscations

We now describe in detail the implementation of the obfuscations shown in Figure 1. Within our obfuscator, the original program is represented as an interprocedural control-flow graph (ICFG). The nodes are basic blocks of machine instructions; the edges represent the control flow in the program.

We obfuscate all direct jumps, direct calls, and returns that are in cold code blocks—as specified by a hot-code threshold that is an input parameter to our obfuscator. In addition, we use a code transformation called *branch flipping* to increase the number of jumps in cold code blocks [21]. For example, suppose we have a conditional branch instruction that branches if the result of a prior comparison of two values found that they were equal:

```
        je    Addr
```

If this branch is in a cold code block, we replace it by

```
        jne   L
        jmp   Addr
    L:
```

The first instruction flips the sense of the branch, so if the comparison was equal, now the conditional branch falls through and the code then jumps to Addr. Other kinds of conditional branches can be flipped similarly. We only do branch flipping in cold code blocks because it slows the program down, although not nearly as much as raising a signal.

Obfuscating Control Transfers

After some initialization actions, our obfuscator makes one pass through the original program to do branch flipping on conditional branches in cold code. It then makes a second pass through the program to find and modify all control transfer instructions that are to be obfuscated.

For a jump instruction, the control flow graph will contain two basic blocks that contain the following:

Code-before
`jmp` *Addr*

Code-after

Again, *Code-before* and *Code-after* are instructions from the original program. There is a control flow edge from the first basic block to the block corresponding to location *Addr*. There is no control-flow edge from the first block above to the second, but we link all basic blocks together in the order in which they were laid out in memory in the original program. This link is important when we insert bogus code as described below.

The first code block is obfuscated by changing it into the following:

Code-before
Setup code
Trap code

The *Setup code* does three things: (1) reserve space on the stack that will be used by the signal handler to store the address of the trap instruction; (2) set a flag that informs the signal handler that the coming trap is from obfuscated code, not the original program itself; and (3) save registers and status flags so the original program's state can later be restored to exactly what it was before the jump instruction. In our implementation on an Intel IA-32 architecture, space is reserved by pushing a four-byte word on the stack, the flag is set by moving an arbitrary non-zero value into a global memory location reserved by the obfuscator, and registers and flags are saved by executing `pusha` and `pushf` instructions.

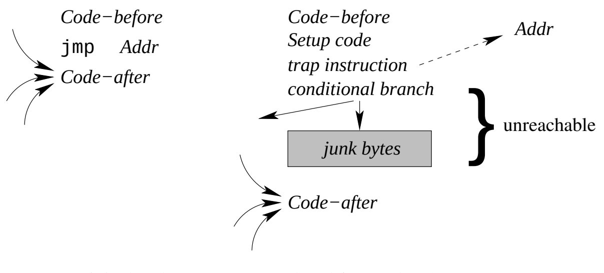
The *Trap code* generates a trap, which in turn raises a signal. We only raise signals for which the default action is to dump core and terminate the program in order to avoid interfering with signals or signal handlers that might be in the original program (see the next section for details). In particular we use illegal instruction (SIGILL), floating point exception (SIGFPE), and segmentation violation (SIGSEGV). For each obfuscation point, the obfuscator randomly chooses one of these types of signals. For an illegal instruction on the IA-32 architecture, we use three bytes "`0F 3E 08`." To cause a floating point exception, we move a zero into register `ebx`, then divide by that register. To generate a segmentation faults, we move a zero into register `eax`, then dereference that register in a move instruction.[4]

[4]Although we randomize the use of these code sequences, the latter

Return instructions are obfuscated in an identical way. For call instructions, the only essential difference is that we need to reserve two stack words so the signal handler has places to store both the target and return address of the original call. The structure of the control flow graph is also different for call and return blocks. In all cases, we retain the ICFG structure of the original program, so that we can use it later to generate the table that maps trap addresses to their original targets.

Inserting Bogus Code

After obfuscating a control transfer instruction, we next insert bogus code—a conditional branch and some junk bytes—to further confuse disassemblers, as shown below.



(a) Original code (b) Obfuscated code

Since the trap instruction has the effect of an unconditional control transfer, the conditional branch immediately following the trap is “bogus code” that will not be reachable in the obfuscated program, and hence it will not be executed. However, since the trap instruction does not look like a control transfer instruction, the bogus conditional branch is not readily identifiable as being unreachable. The conditional branch goes to a junk address that does not contain actual code. The purpose of adding this instruction is to make a disassembler think there is another edge in the control flow graph. The target address is chosen so as not to reveal instructions that a disassembler would not find on its own. A secondary benefit of such bogus conditional branches is that they help improve the stealthiness of the obfuscation, since otherwise the disassembly would produce what appeared to be long sequences of straight-line code without any branches, which would not resemble code commonly encountered in practice.

The junk bytes are the prefix of a legal instruction. The goal is to cause a disassembler to consume the first few bytes of *Code-after* when it completes the legal instruction that starts with the junk bytes. This will ideally cause it to continue to misidentify the true instruction boundaries for at least a while. This technique only works on variable-instruction-length architectures such as the IA-32. Moreover, disassemblers tend to resynchronize relatively quickly, so that on average they are confused for only three or four instructions before again finding the true instruction boundaries. In our implementation, we use one of five randomly chosen 6-byte instructions; we also randomize the register that is used and the order of the operands, so there are 80 different possibilities in all. For each prefix of the chosen instruction (from 1 to 5 bytes long), we calculate how many of the following instructions will be missed by a disassembler, and then insert the prefix that maximizes disassembly error. (See [21] for details on junk byte insertion and self-repairing disassembly.)

two are pretty obviously going to cause traps. We have played with generating random sequences of instructions that have the same effect—and would be much harder to reverse engineer—but we have not yet implemented this scheme.

Building the Mapping Table

After obfuscating control flow and inserting bogus code, our obfuscator computes a memory layout for the obfuscated program and determines final memory addresses. Among these are the addresses of all the trap instructions that have been inserted.

The obfuscator then goes through the control flow graph and gathers the information it needs to build the table that maps trap locations to original targets. The locations of the original targets in the obfuscated program can be determined from the memory layout together with the control flow edges that have been retained in the ICFG. For example, when a jump is replaced by a trap, there is still a jump control flow edge from the code block that ends with the trap to the code block that was the target of the jump.

Suppose that N control transfer instructions have been obfuscated. Then there are N rows in the mapping table, one for each trap point. Each row contains a flag that indicates the type of transfer that was replaced, and zero, one, or two target addresses, depending on the value of the flag. To make it hard to reverse engineer the contents and use of this table, we use two techniques. First, we generate a perfect hash function that maps the N trap addresses to distinct integers from 0 to $N-1$ [14], and we use this function to get indices into the mapping table. This hides the values of the trap addresses, because they do not appear in the obfuscated program itself. We generate the machine code for the perfect hash function after constructing the mapping table. This machine code is quite inscrutable and hence hard to reverse engineer.

Second, to make it hard to discover the target addresses in the mapping table, we do not store them directly. Instead, when constructing the mapping table, in place of each target address T we store a value XT that is the XOR of T and the corresponding trap address S. To later retrieve a target address for trap address S, we (1) use the perfect hash function on S to get the index of the appropriate row in the mapping table, (2) load the appropriate stored value XT into a register, and then (3) XOR the contents of the register with S to get back the real target address. In this way, the stored entries in the mapping table always look like arbitrary strings of bits.

3.3 Signal Handling

When an obfuscated control transfer instruction traps and raises a signal, a sequence of actions occurs, the end result of which is that control is transfered back to the program at the target address of the original control transfer instruction. Below we give an overview of how signal handling is implemented in Linux (and other Unix-based systems), then we describe the actions we take to handle signals and then to restore the execution context of the original program.

Overview

When an instruction raises a signal, the processor stores the address S of the instruction on the stack, then traps into the kernel. If no handler has been installed, the kernel takes the default action for the signal.

If a signal-handling function has been installed, the kernel calls the function. (An application program installs a signal handler by using the `signal` system call.) When the signal handler returns, it goes to a kernel restore function rather than back to the kernel trap handler. This function restores the program's state, and then transfers control back to user code at the trap address (S).

Figure 2 shows the components and control transfers that normally occur when a program raises a signal at address S and has installed a signal handler that returns back to the program at the same address. Figure 3 shows the components and control transfers that occur in our implementation. The essential differences are

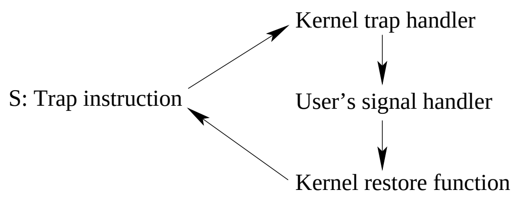


Figure 2: Normal Signal Handling

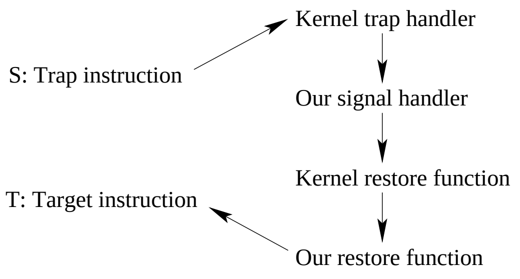


Figure 3: Our Signal Handling Path

that we return control to a different target address T, and we do so by causing the kernel to transfer control to our restore code rather than back to the trap address.

Handler Actions

The main role of our signal handler is to cause the kernel to return control to our restore code, which then returns to the user program at the original target of the obfuscated control transfer instruction, as shown in Figure 3. We do this when a signal is raised from a trap location that we inserted in the binary. However, other instructions in the original program might raise the illegal instruction, floating point exception, or segmentation fault signals. To tell the difference, we use a global memory location `my_signal` that is initialized to zero. In the Setup code before each of the traps we insert in the program, we set this location to a non-zero value, and we later reinitialize it to zero in our signal handler. Thus, we can tell whether a signal was raised by our code or by the original program.

If a signal was raised from one of our locations, we set up the control-flow path shown in Figure 3. If not, we take one of several actions depending on whether the user specified a handler or specified that the signal was to be ignored.

The full algorithm for our signal handler is given in Figure 4. The instances of "reinstall this handler" in the code are needed on Linux to re-inform the kernel that we want to handle these three types of signals.[5]

One subtlety that has to be dealt with, in order to preserve the semantics of the original program, is that of user-defined signal handlers. The issue is that a program may define its own handlers for signals (or specify that the signal is to be ignored), including signals used by our obfuscator, and may in fact dynamically change the handler associated with any given signal as the program executes.

A program installs a user-defined handler for a given signal by calling the `signal()` function in the system-call library. In a binary, this becomes a call to `__bsd_signal()`. To deal with user-installed handlers, we statically rewrite the program binary to intercept all calls to `__bsd_signal()` at runtime. The intercepting code inspects the arguments to the system call to identify the signal s and the handler f. If s is not a signal used by the obfuscator, then we go ahead and execute the call `__bsd_signal`(s, f), which installs the handler f for the signal s. However, if s is a signal used by our obfuscator, then—instead of installing f as a handler—we remember the user-defined association between s and f in a table T. $T[j]$ indicates the action associated with signal j: the default action, ignoring the signal, or the address of a handler function to be invoked when that signal is raised. When a signal is raised from the original program code, the signal handler code in Figure 4 consults this table to determine the appropriate course of action. Since the table T is updated appropriately whenever `__bsd_signal()` is called to (re)define the handler for any signal used by our obfuscator, this allows us to preserve the semantics of the program even if the handler associated with a signal changes dynamically.

[5]Reinstalling the handler also makes our implementation more portable. However, operating systems differ in their semantics for signal handlers. For example, we would not have to reinstall the handler on BSD Unix.

```
if ( my_signal > 0 ) {          /* signal from obfuscation code */
    reinstall this handler;
    my_signal = 0;
    move fault address S into the four bytes reserved on the
            stack right before the trap;
    overwrite kernel restore's return address with
            address of my_restore;
}
else {                          /* signal raised by user program */
    look up how user handles this signal;
    if (no handler installed or user installed default action) {
        install default action;  /* so trap again and terminate */
    }
    else if (user installed ignore action) {
        reinstall this handler;
    }
    else {                      /* user installed own handler */
        call user's handler;    /* and return here */
        reinstall this handler
    }
}
return;
```

Figure 4: Signal Handler Actions

```
my_restore:
    invoke perfect hash function on trap address;
    use returned index to get tag and targets from mapping table;
    if (tag == return) {
        pop an address off the stack;
    }
    else if (tag == jump) {
        overwrite return address with jump target address;
    }
    else {                                  /* tag == call */
        overwrite return address with address of call target;
        overwrite deeper return with address of original return;
    }
    restore saved registers and flags;
    return;
```

Figure 5: Restore Code Actions

Restore Actions

In the normal case where our signal handler is processing one of the traps we inserted in the program, it overwrites the kernel restore function's return address with the address of our restoration code `my_restore`. That code (1) invokes the perfect hash function on the trap address S (which was put in the reserved stack space by our signal handler), (2) looks up the original target address, (3) resets the stack frame as appropriate for the type of control transfer, (4) restores the saved registers and status flags, and finally (5) transfers control (via return) to the original target address.

The full algorithm for our restore code is given in Figure 5. For obfuscated return instructions, the original return address will already be on the stack, one word below the current top of the stack, because it was placed there by the original code. For obfuscated call instructions, we use the two reserved stack locations to store the target of the call and, below it on the stack, the original return address.

Interaction With Other Signals

If the original program raises the kinds of signals that we use for traps, then we handle them as shown in Figure 4. However, the user might install other signal handlers, and these can interact with ours. For example, one of the SPECint-95 benchmark programs, m88ksim, installs a handler for SIGINT, the interrupt signal. If we obfuscate that program, run the code, and repeatedly interrupt the program, eventually it will cause a segmentation fault and crash. The reason is that there is a race condition between our interrupt handler and the user-installed SIGINT handler. In particular, if we interrupt the program while it happens to be executing in our handler, the program crashes.

To solve this type of problem, our signal handler needs to delay processing of other signals that might be raised. On Unix this can be done by having the signal handler call the `sigprocmask` function, or by using `sigaction` when we (re)install the handler. Once our trap processing code gets back to the Restore block of the obfuscated program, it can safely be interrupted because it is through manipulating kernel addresses. Our current implementation does not yet block other signals.

An even worse problem would occur in a multithreaded program, because multiple traps could occur and have to be handled at the same time. Signal handling is not thread safe in general in Unix systems, so our obfuscation method cannot be used in an arbitrary multithreaded program. However, this is a limitation of Unix, not our method.

4. EVALUATION

We measure the efficacy of obfuscation in two ways: by the extent of incorrect disassembly of the input, and by the extent of errors in control flow analysis of the disassembled input. These quantities are related, in the sense that an incorrect disassembly of a control transfer instruction will result in a corresponding error in the control flow graph obtained for the program. However, it is possible, in principle, to have a perfect disassembly and yet have errors in control flow analysis because control transfer instructions have been disguised as innocuous arithmetic instructions or bogus control transfers have been inserted.

4.1 Evaluating Disassembly Errors

We measure the extent of disassembly errors using a measure we call the *confusion factor* 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. More formally, let A be the set of all *actual* instruction addresses, i.e., those that would be encountered when the program is executed, and P the set of all *perceived* instruction addresses, i.e., those addresses produced by a static disassembly. $A - P$ is the set of addresses that are not correctly identified as instruction addresses by the disassembler. We define the *confusion factor* CF to be the fraction of instruction addresses that the disassembler fails to identify correctly:[6]

$$CF = |A - P|/|A|.$$

Confusion factors for functions and basic blocks are calculated analogously: 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.

4.2 Evaluating Control Flow Errors

When comparing the control flow structure of the disassembled program P_{disasm} with that of the original program P_{orig}, there are two possible kinds of errors that can occur. First, P_{disasm} may contain some edge that does not appear in P_{orig}, i.e., the disassembler may mistakenly find a control flow edge where the original program did not have one. Second, P_{disasm} may not contain some edge that appears in P_{orig}, i.e., the disassembler may fail to find an edge that was present in the original program. We term the first kind of error *overestimation errors* (written Δ_{over}) and the second kind *underestimation errors* (written Δ_{under}), and express them relative to the number of edges in the original program. Let E_{orig} be the set of control flow edges in the original program and E_{disasm} the set of control flow edges identified by the disassembler, then:

$$\begin{aligned}\Delta_{over} &= |E_{disasm} - E_{orig}|/|E_{orig}| \\ \Delta_{under} &= |E_{orig} - E_{disasm}|/|E_{orig}|\end{aligned}$$

Even if we assume a perfect "attack disassembler" that does not incur any disassembly errors, the disassembly produced will contain control flow errors arising from two sources. First, control transfers that have been transformed to trap-raising instructions will fail to be identified as control transfers. Second, bogus control transfers introduced by the obfuscator will be identified as control transfers. We can use this to bound the control flow errors even for a perfect disassembly. Suppose that n_{trap} control flow edges are lost from a program due to control transfer instructions being converted to traps, and n_{bogus} bogus control flow edges are added by the obfuscator. Then, a lower bound on underestimation errors $\min \Delta_{under}$ is obtained when the only control transfers that the attack disassembler fails to find are those that were lost due to conversion to trap instructions:

$$\min \Delta_{under} = n_{trap}/E_{orig}.$$

An upper bound on overestimation errors $\max \Delta_{over}$ is obtained when every bogus conditional branch inserted by the obfuscator is reported by the disassembler:

$$\max \Delta_{over} = n_{bogus}/E_{orig}.$$

4.3 Defining Hot Code

[6] We also considered taking into account the set $P - A$ of addresses that are erroneously identified as instruction addresses by the disassembler, but rejected this approach because it "double counts" the effects of disassembly errors.

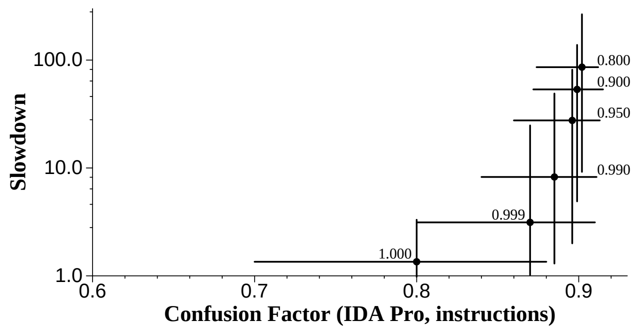


Figure 6: The tradeoff between obfuscation and speed

As mentioned in Section 3, we transform jumps to signal-raising instructions only if the jump does not occur in a "hot" basic block. To evaluate the efficacy of obfuscation, therefore, we have to specify what it means for a basic block to be "hot," and then examine the effect of how the definition of "hot" blocks affects 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.

Note that at $\theta = 1.0$, this implies that any basic block with nonzero execution count, i.e., which is executed at least once on the profiling input, will be considered "hot."

5. EXPERIMENTAL RESULTS

We evaluated the efficacy of our techniques using ten programs from the SPECint-2000 benchmark suite.[7] Our experiments were run on an otherwise unloaded 2.4 GHz Pentium IV system with 1 GB of main memory running RedHat Linux (Fedora Core 3). The programs were compiled with *gcc* version 3.2.2 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 derived by running seven trials, removing the highest and lowest times from the sampling, and averaging the remaining five.

We experimented with three different "attack disassemblers" to evaluate our techniques: GNU `objdump` [24]; IDA Pro [13], a commercially available disassembly tool that is generally regarded to be among the best disassemblers available;[8] and an exhaustive disassembler by Kruegel *et al.*, engineered specifically to handle obfuscated binaries [18]. `Objdump` uses a straightforward linear sweep algorithm, while IDA Pro uses recursive traversal. The exhaustive disassembler of Kruegel *et al.* explicitly takes into account the possibility that the input binary may be obfuscated by not making any assumptions about instruction boundaries. Instead, it considers alternative disassemblies starting at every byte in the code region of the program, then examines these alternatives using a variety of statistical and heuristic analyses to discard those that are unlikely or impossible. Kruegel *et al.* report that this approach yields significantly better disassemblies on obfuscated inputs than other existing disassemblers [18]; to our knowledge, the exhaustive disassembler is the most sophisticated disassembler currently available.

Obfuscation/Speed Tradeoff

Given the definition of "hot code" in Section 4.3, as the hot code threshold θ is reduced, less and less of the code is counted as "hot," and more and more of it remains available for obfuscation, which in turn implies an increase in runtime overheads due to the obfuscation. Intuitively this means that as θ is decreased, we should expect an increase in the extent of obfuscation seen, accompanied by a drop in execution speed.

The effect of varying the hot code threshold θ on performance (both obfuscation and speed) is shown in Figure 6 for a number of different threshold values ranging from $0.80 \leq \theta \leq 1.00$. For each threshold value, Figure 6 shows the geometric mean of the confusion factor, with a horizontal bar showing the range of values encountered for the confusion factor for our benchmarks at that threshold and a vertical bar showing the range of values for the slowdown. (To reduce clutter, we show only the instruction confusion factor for IDA Pro; the results for the other disassemblers are analogous.) Ideally we want to have high confusion factors and low slowdowns, i.e., points close to the x-axis and far from the y-axis.

We can see that as the threshold is decreased from $\theta = 1.00$ to $\theta = 0.80$, the mean confusion factor goes from about 0.8 to about 0.9: an increase of roughly 10%. At the same time, however, the performance of the obfuscated code drops dramatically, with the mean slowdown increasing from 1.35 at $\theta = 1.00$ to 86.0 at $\theta = 0.80$. Thus, choosing a threshold θ of 1.0 still results in a considerable amount of obfuscation being achieved, but without excessive performance penalty. For the purposes of this paper, therefore, we give measurements for $\theta = 1.0$.

Disassembly Error

The extent of disassembly error, as measured by confusion factors (Section 4.1) is shown in Figure 7(a). The work described here focused primarily on disguising control transfer instructions by transforming them into signal-raising instructions, so it does not come as a surprise that the straightforward linear sweep algorithm used by the `objdump` disassembler has the best performance. Even here, however, over 30% of the instructions in the program, 66% of the basic blocks, and 86% of the functions, on average, are not disassembled correctly. The other disassemblers, which rely on control flow information to guide their disassembly, fare much worse: Kruegel *et al.*'s exhaustive disassembler fails on 57% of the instructions, 75% of the basic blocks and almost 90% of the functions on

[7]We did not use two programs from this benchmark suite: *eon* and *vpr*. We had problems building *eon*, and *vpr* did not run properly because of a bug in our profiling code that had nothing to do with obfuscation.

[8]We used IDA Pro version 4.7 for the results reported here, except for the *gcc* program, on which IDA Pro version 4.7 did not terminate after three days and had to be killed; for the *gcc* benchmark, the numbers reported are using IDA Pro version 4.3. For the remaining benchmarks, the data reported by these two versions of IDA Pro typically differed by less than 0.1%.

PROGRAM	OBJDUMP			EXHAUSTIVE			IDA PRO		
	Instructions	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*
bzip2	33.50	70.90	85.40	58.48	78.82	92.10	87.07	89.05	84.19
crafty	30.71	66.05	84.68	53.62	73.82	83.87	78.98	82.02	69.35
gap	31.10	64.14	87.72	57.93	74.88	91.27	82.24	82.37	81.75
gcc	29.30	60.47	86.81	56.16	71.21	90.48	69.84	66.96	80.66
gzip	33.68	70.99	86.77	57.59	78.62	92.28	87.94	89.62	83.62
mcf	34.20	70.93	84.15	71.47	85.62	88.87	88.01	89.79	81.32
parser	31.50	64.77	83.46	54.60	72.22	87.03	78.96	78.67	73.32
perlbmk	31.25	65.08	86.68	54.77	73.30	89.34	73.31	73.03	80.78
twolf	30.71	64.84	84.24	54.63	73.55	86.68	81.13	82.40	77.79
vortex	28.68	63.86	92.01	53.60	73.57	94.49	71.31	73.30	90.46
GEOM. MEAN:	31.41	66.12	86.16	57.09	75.45	89.59	79.62	80.37	80.13

(a) Disassembly Errors (Confusion Factor, %)

PROGRAM	E_{orig}	n_{trap}	n_{bogus}	max	min	OBJDUMP		EXHAUSTIVE		IDA PRO	
				Δ_{over}	Δ_{under}	Δ_{over}	Δ_{under}	Δ_{over}	Δ_{under}	Δ_{over}	Δ_{under}
bzip2	47331	21227	40420	85.39	44.84	58.00	72.39	32.66	81.18	5.87	90.48
crafty	54954	22506	43288	78.77	40.95	53.40	65.52	29.86	74.70	7.67	82.91
gap	92803	36759	71468	77.01	39.60	52.26	64.21	28.02	75.65	7.45	83.20
gcc	204756	77288	140154	68.44	37.74	46.57	58.26	23.65	70.22	12.23	68.65
gzip	47765	21564	41038	85.91	45.14	58.06	72.74	33.97	81.06	5.35	91.22
mcf	41807	18435	35612	85.18	44.09	57.41	72.49	22.57	87.49	5.74	91.28
parser	54524	21569	41954	76.94	39.55	52.39	64.13	29.27	72.63	8.31	79.08
perlbmk	100341	41243	77702	77.43	41.10	52.86	64.37	29.30	73.82	12.47	75.39
twolf	57210	22874	44730	78.18	39.98	53.01	64.33	30.07	74.07	7.53	83.36
vortex	87799	36077	65044	74.08	41.09	50.54	61.25	27.69	72.59	10.23	74.04
GEOM. MEAN:				78.56	41.34	53.34	65.80	28.50	76.18	7.94	81.65

(b) Control Flow Error (%)

Key:

E_{orig}:	no. of edges in original program	max Δ_{over}:	upper bound on overestimation errors
n_{trap}:	no. of control flow edges lost due to trap conversion	min Δ_{under}:	lower bound on underestimation errors
n_{bogus}:	no. of bogus control flow edges added		

Figure 7: Efficacy of obfuscation ($\theta = 1.0$)

average; IDA Pro fails to disassemble close to 80% of the instructions and just over 80% of the basic blocks and functions in the programs. Part of the reason for IDA Pro's high degree of failure (especially compared to the other two disassemblers) is that it only disassembles addresses that (it believes) can be guaranteed to be instruction addresses. Because of this, it abandons disassembly of a function from the point where it encounters an illegal instruction up to the next point that is identifiably the target of a control transfer from already-disassembled code: the intervening bytes—which in our case is very often the remainder of the function, since most control transfers have been obfuscated to resemble other instructions—is simply presented to the user as a jumble of raw hex data.

Overall, the instruction confusion factors show that a significant portion of the binaries is not correctly disassembled; the basic block and function confusion factors show that the errors in disassembly are distributed over most of the program. Taken together, these data show that our techniques are effective even against state-of-the-art disassembly tools.

Control Flow Obfuscation

Figure 7(b) shows the effect of our transformations in obfuscating the control flow graph of the program. The second column gives, for each program, the actual number of control flow edges in the obfuscated program, counted as follows: each conditional branch gives rise to two control flow edges; each unconditional branch (direct or indirect) gives rise to a single edge; and each function call gives two control flow edges—one corresponding to a "call edge" to the callee's entry point, the other to a "return edge" from the callee back to the caller. Column 3 gives the number of control flow edges removed due to the conversion of control flow instructions to traps, while column 4 gives the number of bogus control flow edges added by the obfuscator. Columns 5 and 6 give, respectively, an upper bound on the overestimation error and a lower bound on the underestimation error. The remaining columns give, for each attack disassembler, the extent to which it incurs errors in constructing the control flow graph of the program, as discussed in Section 4.2.

It can be seen, from Figure 7(b), that none of the three attack disassemblers tested fares very well at constructing the control flow graph of the program. `Objdump` fails to find almost 66% of the control flow edges in the original program; at the same time, it reports over 53% spurious edges (relative to the number of original edges in the program) that are not actually present in the program. The exhaustive disassembler fails to find over 76% of the edges in the original program, and reports over 28% spurious edges. IDA Pro fails to find close to 82% of the control flow edges in the original program, and reports almost 8% spurious edges. The last of these numbers—which is significantly smaller than the corresponding values for the other two disassemblers—is misleadingly low: the reason IDA Pro does not report many of the spurious conditional branches introduced by our obfuscator is not that it is particularly good at identifying spurious control flow, but rather that it

PROGRAM	EXECUTION TIME (SECS)		
	Original (T_0)	*Obfuscated* (T_1)	*Slowdown* (T_1/T_0)
bzip2	283.070	327.491	1.156
crafty	139.115	180.038	1.294
gap	146.357	485.215	3.315
gcc	148.169	314.121	2.120
gzip	217.923	218.939	1.004
mcf	428.486	427.377	0.997
parser	294.760	292.528	0.992
perlbmk	201.289	383.002	1.902
twolf	571.445	575.719	1.007
vortex	234.530	232.944	0.993
GEOM. MEAN			1.348

Figure 8: Effect of obfuscation on execution speed ($\theta = 1.0$)

simply fails to disassemble large portions of the program.

Also significant are the error bounds reported in columns 5 and 6 of Figure 7(b). These numbers indicate that, even if we suppose perfect disassembly, the result would incur up to 78.6% overestimation error and at least 41.3% underestimation error.

Execution Speed

Figure 8 shows the effect of obfuscation on execution speed. For some programs, such as *gzip*, *mcf*, *parser*, *twolf*, and *vortex*, for which the execution characteristics on profiling input(s) closely match those on the reference input, there is essentially no slowdown (a few benchmarks run very slightly faster after obfuscation; we believe this effect is due to a combination of cache effects and experimental errors resulting from clock granularity). For others, such as *gap* and *gcc*, the profiling inputs are not as good predictors of the runtime characteristics of the program on the reference inputs, and this results in significant slowdowns: a factor of 3.3 for *gap* and 2.1 for *gcc*. The mean slowdown seen, for all ten benchmarks, is just under 35%.

Program Size

Figure 9 shows the impact of obfuscation on the size of the text and initialized data sections. It can be seen that the size of the text section increases by factors ranging from 2.33 (*crafty* and *vortex*) to almost 2.7 (*bzip2*, *gzip*, *mcf*), with a mean increase of a factor of 2.5. The relative growth in the size of the initialized data section is considerably larger, ranging from a factor of about 13 (*crafty*) to a factor of just over 67 (*twolf*), with a mean growth of a factor of 32. The growth in the size of the initialized data is due to the addition of the mapping tables used to compute the type of each branch as well as its target address. However, this apparent large relative growth in the data section size is due mainly because the initial size of this section is not very large. When we consider the total increase in memory requirements due to our technique, obtained as the sum of the text and initialized data sections, we see that it ranges from a factor of 2.8 (*crafty*, *vortex*) to about 3.4 (*gzip*, *mcf*), with a mean growth of a factor of about 3.

The increase in the size of the text section arises from three sources. The first of these is the code required to set up and raise the trap for each obfuscated control transfer instruction. The second is the junk bytes and bogus conditional branch inserted after a trap instruction. Finally, there is the signal handler and restore code. In our current implementation, the first two of these sources—the setup code for a trap and bogus code inserted after a trap—introduces on average an additional 30 bytes of memory for each obfuscated control transfer instruction. This accounts for over 99% of the total increase in the text section size. Each obfuscated control transfer also adds three memory words (12 bytes) to the initialized data section, accounting for the increase in the size of this section.

6. RELATED WORK

The earliest work on the topic of binary obfuscation that we are aware of is by Cohen, who proposes overlapping adjacent instructions to fool a disassembler [7]. We are not aware of any actual implementations of this proposal, and our own experiments with this idea proved to be disappointing. More recently, we proposed an approach to make binaries harder to disassemble using a combination of two techniques: the judicious insertion of "junk bytes" to throw off disassembly; and the use of a device called "branch functions" to make it harder to identify branch targets [21]. These techniques proved effective at thwarting most disassemblers, including the commercial IDA Pro system.

There has been some recent work by Kapoor [17] and Kruegel *et al.*[18] focusing on disassembly techniques aimed specifically at obfuscated binaries. They work around the possibility of "junk bytes" inserted in the instruction stream by producing an *exhaustive disassembly* for each function, i.e., where a recursive disassembly is produced starting at every byte in the code for that function. This results in a set of alternative disassemblies, not all of which are viable. The disassembler then uses a variety of heuristic and statistical reasoning to rule out alternatives that are unlikely or impossible. To our knowledge, these exhaustive disassemblers are the most sophisticated disassemblers currently available. One of the "attack disassemblers" used for the experiments described in this paper is an implementation of Kruegel *et al.*'s exhaustive disassembler.

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, 25, 30, 31]. 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. 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.* [20]. 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.

7. CONCLUSIONS

This paper has described a new approach to obfuscating executable binary programs and evaluated its effectiveness on programs in the SPECint-2000 benchmark suite. Our goals are to make it hard for disassemblers to find the real instructions in a binary and to give them a mistaken notion of the actual control flow in the program. To accomplish these goals, we replace many control transfer instructions by traps that cause signals, inject signal handling code that actually effects the original transfers of control, and insert bogus code that further confuses disassemblers.

These obfuscations confuse even the best disassemblers. On average, the GNU `objdump` program [24] misunderstands over 30%

PROGRAM	TEXT SECTION (BYTES)			INITIALIZED DATA SECTION (bytes)			COMBINED: TEXT+DATA (bytes)		
	Original (T_0)	Obfuscated (T_1)	Change (T_1/T_0)	Original (D_0)	Obfuscated (D_1)	Change (D_1/D_0)	Original ($C_0 = T_0 + D_0$)	Obfuscated ($C_1 = T_1 + D_1$)	Change (C_1/C_0)
bzip2	347,363	929,500	2.68	6984	245560	35.16	354347	1175060	3.32
crafty	458,163	1,067,502	2.33	20,420	267,252	13.08	478583	1334754	2.79
gap	678,579	1,697,748	2.50	7,380	424,212	57.48	685959	2121960	3.09
gcc	1,402,235	3,376,181	2.41	22,924	814,592	35.53	1425159	4190773	2.94
gzip	349,411	939,630	2.69	6,604	249,044	37.71	356015	1188674	3.34
mcf	300,723	809,448	2.69	3,796	213,332	56.19	304519	1022780	3.36
parser	395,203	987,618	2.50	6,340	246,368	38.85	401543	1233986	3.07
perlbmk	731,659	1,822,820	2.49	33,572	472,628	14.07	765231	2295448	3.00
twolf	458,383	1,089,504	2.38	3,876	260,428	67.18	462259	1349932	2.92
vortex	689,575	1,605,416	2.33	24,332	396,764	16.30	713907	2002180	2.80
GEOM. MEAN			2.50			32.18			3.06

Figure 9: Effect of obfuscation on text and data section sizes ($\theta = 1.0$)

of the original instructions, over-reports the control flow edges by 50%, and misses nearly 66% of the original control flow edges. The IDA Pro system [13], which is considered the best commercial disassembler, fails to disassemble nearly 80% of the original instructions, over-reports control flow edges by about 8%, and under-reports control flow edges by nearly 82%. A recent disassembler [18] that has been designed to deal with obfuscated programs fails to disassemble over 57% of the instructions, over-reports control flow edges by 28%, and under-reports control flow edges by over 76%.

These results indicate that we successfully make it hard to disassemble programs, even when we only obfuscate code that is in cold code blocks—i.e., code that was not executed on a profiling run with training input. If we obfuscate more of the code, we can confuse disassemblers even more. However, our obfuscation method slows down program execution, so there is a tradeoff between the degree of obfuscation and execution time. When we obfuscate only cold code blocks, the average slow-down is about 35%, but this result is skewed by three benchmarks for which the training input is not a very good predictor for execution on the reference input. On programs where the training input is a good predictor, the slow-down is negligible. An interesting possibility, which we have not explored, would be selectively to obfuscate some of the hot code, in particular, that which the creator of the code especially wants to conceal.

Acknowledgements

We are grateful to Christopher Kruegel for the use of the code for his exhaustive disassembler for our experiments.

8. 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] J. P. Bowen and P. T. Breuer. Decompilation. In *The REDO Compendium: Reverse Engineering for Software Maintenance*, chapter 10, pages 131–138. 1993.

[3] W. Cho, I. Lee, and S. Park. Against 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. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, Australia, July 1994.

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

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

[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 protection. *IEEE Transactions on Software Engineering*, 28(8), August 2002.

[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] C. R. Hollander. *Decompilation of object programs*. PhD thesis, Stanford University, 1973.

[16] G. L. Hopwood. *Decompilation*. PhD thesis, University of California, Irvine, 1978.

[17] A. Kapoor. An approach towards disassembly of malicious binaries. Master's thesis, University of Louisiana at Lafayette, 2004.

[18] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. 13th USENIX Security Symposium*, August 2004.

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

[20] 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.

[21] C. Linn and S.K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, October 2003.

[22] 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.

[23] A. Mycroft. Type-based decompilation. In *Proc. European Symposium on Programming*, volume 1576 of *Springer-Verlag LNCS*, 1999.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, Dept. of Computer Science, University of Virginia, 12 2000.