Enhancing Software Tamper-Resistance via Stealthy Address Computations [*]

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

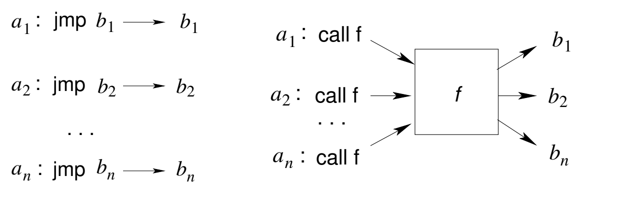
1 Motivation

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 [9]. Techniques such as watermarking and fingerprinting have been developed to discourage piracy [4, 12], however, if no protective measures are taken, an attacker may be able to remove and/or destroy watermarks and fingerprints with relative ease once they have been identified. For this reason, methods such as source code obfuscation [4, 11, 3, 15], code encryption [1, 13] and self verifying code[1, 7] have been developed to help achieve some measure of tamper-resistance.

It is, of course, necessary for an attacker to gain a reliable disassembly of some portion of executable code before any intelligent tampering can take place. In fact, even a reliable disassembly in the absence of some sort of control flow graph is not sufficient for serious tampering[15]. Coupled with other methods [9] we propose one method of obfuscating address computations in which the targets of control transfers are made difficult to determine statically. We describe this method in Section 2.

Assuming an attacker is able to gain a reliable disassembly of a binary, it is quite possible for a malicious host to compromise any and all watermarks and/or fingerprints such that they no longer serve their intended purpose [5]. Code verification schemes that employ techniques such as check-sums have been introduced to "tamper-proof" code [2]. Such schemes must, by definition, examine the code in an executable's text section. Done in a straightforward manner, the check-sum code will contain *load* instructions that reference addresses that are clearly in the text section, allowing attackers to easily identify the check-sum code and potentially disable them ([1, 2] propose ways to raise the difficulty involved in doing so). To counter such attacks we propose one method by which address computations involving text section references can be made less obvious. We discuss this in Section 3.

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



(a) Original code (b) Code using a branch function

Figure 1: Branch functions

2 Branch Functions

One application of obfuscating address computations is to hide from an attacker the true target of some control transfer. This idea is not new, as others such as [16] propose similar ideas but instead focus their efforts at the source code level. At the assembly level, however, even such modifications are translated into unambigous control transfers such as direct and conditional jumps. We propose a method of indirection in which direct control transfers such as call and jump instructions are replaced with calls to specialized functions that are responsible for directing control to the intended targets in some stealthy manner. The intent is that the successors of each transformed basic block will be difficult to discover.

The specialized functions, which we have termed *branch functions*, are illustrated in Figure 1. Given a finite map

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

over locations in a program, 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, $1 \le i \le n$. In our current implementation of branch functions, an unconditional branch at a location a that jumps to an address b is replaced by a call to the branch function that passes, as an argument, the value $b - a$,[1] giving the offset to the target b. The branch function simply adds its argument to the return address stored on the stack and returns, via code (on the Intel IA-32 architecture) of the form:

[1]Strictly speaking the argument passed is $b - a - 5$, to account for the size of the `call` instruction.

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

The effect is to transfer control to the modified return address, i.e., the original target address *b*.

While control transfers effected using branch functions are less obvious than the original unconditional jumps with their target addresses hard-wired into them (as immediate or PC-relative operands), the simple scheme sketched above uses only a single straightforward arithmetic operation, and so is not as robust against reverse engineering as we would like. We are currently working on more complex implementations that use tables and and perfect hashing. We chose to implement branch functions using perfect hashing, which guarantees $O(1)$ access to any element, in order to reduce runtime overhead and also because of the somewhat cryptic computations involved in achieving a perfect hash. Space limitations do not allow for a more detailed description of perfect hashing, but the interested reader is referred to [10, p.318].

During the final stages of binary rewriting, after final code layout has been done and all a_i's have been determined, we create a perfect hash function h using code derived from [8]. We then construct a table t, in the data section of the binary, that lists offsets for each (a_i, b_i) pair: t is then populated in the following fashion:

$$t[h(a_i)] \leftarrow a_i - b_i, \text{ for each } i$$

Next, a function f is created and injected into the binary. f's primary function is the same as the simple branch function, ie given some a_i as a return address on the stack, it adds some offset to that value, achieving the corresponding b_i then transfers control to b_i, the original target address. The difference between the two approaches is the *way* in which the offset is computed. Below is a very basic representation of f:

```
$x \leftarrow h(a_i)$ [2]
$x \leftarrow t[x]$
$x \leftarrow a_i + x$
return to $x$
```

The binary is then modified such that the control transfer (jump or call) at each a_i is replaced with a call to f (Figure 1).

Since the hash function h is guaranteed to run in $O(1)$ time, f will be some constant number of instructions. Moreover, evaluating h involves arithmetic operations with large precomputed constants, whose results are difficult to follow.

3 Disguising Text Section References

Obfuscating address computations can also be used to conceal the presence of self-verifying code. For example, given some checksumming operation performed on the text section of a binary (e.g., see [2]), the fact that the checksum operations load from an absolute address in the text section, and then carry out arithmetic on the results, makes it easier for an attacker to identify such code and, potentially, tamper with it. Along the same lines as methods proposed by [7] who suggests splitting text address references between multiple registers to make them more obscure, we can camouflage such code by replacing text section references with appropriate data section (or other non-text section) references. To do this we must know the final size of each section in the executable program; this cannot be done at compile time, but has to be done post-link-time, after all the program components have been linked together and the final size of each section in the file has been determined.

Given a load from an absolute text section address, e.g.,

```
mov 0x80482ff, %eax
```

we can replace the text section reference by instructions that instead refer to the data section, e.g.:

```
mov $0x1ffffde3, %eax          #I1
mov 0x8049410(,%eax,8), %eax   #I2
```

In the resulting code we disguise the load from the text section as an immediate load of some constant (`#I1`) and a load of a large offset from the data section (`#I2`). The instruction at `#I2` computes 8 times the value in `%eax`, adds that value to `0x8049410`, which is the start address of the data section, and loads the word that resides at the resulting address into `%eax`. After this conversion it appears that some piece of the data section has been loaded, although scaling and overflow actually allow loading from the text section. Obviously there are many other ways in which the desired address can be computed from the address of the data section (or any other section) such as various bit and arithmetic operations.

4 Current Status

We have incorporated the "simple" branch function scheme described in Section 2 into a binary obfuscation tool based on PLTO [14]. We currently use this capability to disguise address computations in order to mislead static disassemblers. Our experiments indicate that once the executables obtained from a compiler have been obfuscated using our techniques, even state of the art industrial disassemblers such as IDA Pro [6] are unable to successfully disassemble them. For example, on the SPECint-95 benchmark suite, IDA Pro is able to correctly disassemble only about 1.5%–12.7% of the instructions (mean: 6.2%); the remainder are either incorrectly disassembled, or else simply presented to the user as large blocks of uninterpreted hex dumps.

We are currently working on extending this tool to enhance branch function capabilities using perfect hashing techniques (Section 2) as well as adding support for disguising text address computations (Section 3).

[2] a_i can be derived from the return address on the stack by simply subtracting 5 for the length of the call at a_i

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] H. Chang and M.J. Atallah. Protecting software code by guards. In *Security and Privacy in Digital Rights Management, ACM CCS-8 01, Philadelphia, PA, USA, November 5, 2001, Revised Papers*, pages 160–175, 2001.

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

[5] 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.

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

[7] Bill Horne, Lesley R. Matheson, Casey Sheehan, and Robert Endre Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Digital Rights Management Workshop*, pages 141–159, 2001.

[8] Bob Jenkins. Minimal perfect hashing. `http://burtleburtle.net/bob/hash/perfect.html`.

[9] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 290–299. ACM Press, 2003.

[10] K. Mehlhorn and A. K. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 301–341. MIT Press, 1990.

[11] 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.

[12] Brigit Pfitzmann and Matthias Schunter. Asymmetric fingerprinting. *Lecture Notes in Computer Science*, 1070:84–??, 1996.

[13] Tomas Sander and Christian F. Tschudin. On software protection via function hiding. *Lecture Notes in Computer Science*, 1525:111–123, 1998.

[14] B. Schwarz, Saumya 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.

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

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