Hybrid Static-Dynamic Attacks against Software Protection Mechanisms

Matias Madou Bertrand Anckaert Bjorn De Sutter Koen De Bosschere

Electronics and Information Systems Department
Ghent University
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium
{*mmadou,banckaer,brdsutte,kdb*}*@elis.UGent.be*

ABSTRACT

Advances in reverse engineering and program analyses have made software extremely vulnerable to malicious host attacks. These attacks typically take the form of intellectual property violations, against which the software needs to be protected. The intellectual property that needs to be protected can take on different forms. The software might, e.g., consist itself of proprietary algorithms and datastructures or it could provide controlled access to copyrighted material. Therefore, in recent years, a number of techniques have been explored to protect software. Many of these techniques provide a reasonable level of security against static-only attacks. Many of them however fail to address the problem of dynamic or hybrid static-dynamic attacks. While this type of attack is already commonly used by black-hats, this is one of the first scientific papers to discuss the potential of these attacks through which an attacker can analyze, control and modify a program extensively. The concepts are illustrated through a case study of a recently proposed algorithm for software watermarking [6].

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Intellectual property rights*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Intellectual property; Security*; K.5.1 [**Legal Aspects Of Computing**]: Hardware/Software Protection—*copyrights; proprietary rights*

General Terms

Economics, Security

Keywords

Intellectual Property, Software Protection, Attacks, Obfuscation, Watermarking, Tamper-resistance

DRM'05, November 7, 2005, Alexandria, Virginia, USA.

1. INTRODUCTION

The intellectual property contained within and protected by software is enormous. The cost of software piracy alone is estimated at 29 billion dollars for the year 2003 [18]. Besides the copying of the entire application, software developers are also faced with the threat that valuable parts of their application could be included in a competitor's code [1]. Furthermore, software may provide a controlled access to other copyrighted material [37]. If the software is tampered with, illegal access to this material may be obtained.

This paper focuses on pure software-based mechanisms for self-defense. We do not consider approaches that require changes to the hardware, operating system, or any other part of the computing system, other than a program itself.

A number of approaches to protect software intellectual property have recently gained increased interest [9, 26]. Software watermarking [8, 30, 33] has been proposed as a defense against software piracy, in which the embedding of a copyright notice in the software allows to prove ownership of the code. Software fingerprinting is a related technique that embeds a unique message into each distributed copy to facilitate the tracking and prosecution of copyright infringers. Another technique, obfuscation [10, 21, 35], attempts to transform an original program into a program that is harder to understand. Any attack against software requires an (at least partial) understanding of the program. Obfuscation is a technique to prevent the attacker from acquiring such an understanding, thereby making attacks more difficult. Tamper resistance [2, 3, 16] is a technique that is targeted specifically at making the program unmodifiable. As such, obfuscation and tamper-resistance can be used to reinforce other techniques.

In recent years, many techniques for watermarking, obfuscation and tamper-resistance have been published. However, little research seems to have been done into the vulnerability of these methods. While most of the existing techniques are fairly robust against static attacks, many fail in the presence of dynamic or hybrid static-dynamic attacks. Although the authors of the aforementioned techniques acknowledge that it is not possible to protect software against all conceivable attacks, they do claim to make possible attacks "expensive enough" –in time, effort, or resources– that for most attackers, an attack it is not worthwhile. The hybrid static-dynamic attacks proposed in this paper refute this claim, however, as they can be implemented without significant requirements as to computation time, effort or resources.

As any attack against software requires some understanding of the software, we will first show how a partial view of the program can always be obtained through the use of advanced, dynamic instrumentation tools, even if obfuscation has been extensively applied. Departing from the thus obtained view of the program, we will show how to remove a recently proposed software watermark [6]. We will not present attacks against techniques for tamper-resistance because such attacks have recently been proposed by others [36].

Despite the major advances that have been made in the protection of software, we show in this paper that the attack model against which existing techniques provide protection is not realistic. By introducing a more severe attack pattern than usually considered, we hope to raise the bar for future techniques.

This paper is structured as follows: Section 2 presents a classification of attacks along three axes. The more severe attack model is introduced in Section 3. In Section 4, we show how a partial understanding of the program can always be obtained through the use of advanced dynamic instrumentation techniques. In Section 5 we show how to remove a dynamic path-based watermark. Related work is the topic of Section 6 and conclusions are drawn in Section 7.

2. CLASSIFYING ATTACKS

While a lot of research has targeted the development of new software protection techniques, little effort has been made to bring structure into the attacks against these techniques. In this section we present a classification of attacks.

2.1 Static versus Dynamic

Similar to the distinction between static and dynamic analyses [15], we can distinguish between static and dynamic attacks, based upon the information they use. When an attack is based solely upon static information, we will call this a *static attack*. Static information is obtained by examining program code and reasoning about possible behaviors that might arise at run time, without actually executing a program.

We define a *dynamic attack* as an attack that is based solely on dynamic information, which is obtained by executing the program and observing one or more executions. When an attack is based upon both static and dynamic information, we call this a *hybrid static-dynamic attack*.

Static and dynamic information can complement one another. By first collecting one type of information and then using this information to collect the other type, perhaps iterating, the understanding of a program can be increased.

2.2 Conservative versus Approximative

Ideally, the information we have about the program is exact, meaning that it takes into account every possible execution, but no other executions. In practice, exact information is often unattainable.

Static information is obtained by building a model of the state of the program and by determining how the program reacts to this state. If all possible executions and associated program states are considered, then the obtained information is exact. In practice, we cannot keep track of all possible executions, as there may be infinitely many, while the program state can be extremely large.

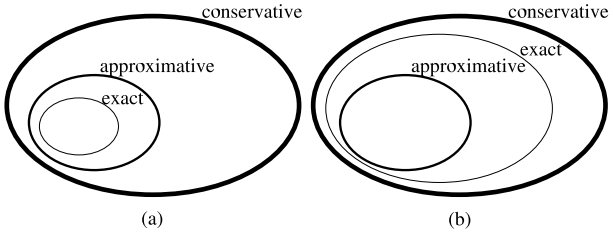

Figure 1: Real set of executions and set of executions considered by a conservative attack versus set of executions considered by (a) a sound approximative attack and (b) an unsound approximative attack

Dynamic information is obtained by executing the program and by observing the executions. If every possible input (and every possible environmental interaction) is examined, then the obtained information will also be exact. Unfortunately, we cannot expect to build a test suite that contains every possible input and environmental interaction.

We can deal with the missing information pessimistically or optimistically. If the attack is designed to work no matter what the missing information might be, we call this attack a *conservative attack*. If the attack may fail for certain values of the missing information, the attack is called an *approximative attack*.

2.3 Sound versus Unsound

As a conservative attack works for every possible execution, it is guaranteed to be sound. Whether an approximative attack is sound depends on whether the assumptions made during the attack are valid for all real executions. This is, in general, undecidable.

An example might help to clarify this issue. Let's consider the fact whether two pointers do or do not alias (point to the same location) as a piece of information. This information is known to be hard to obtain statically. Therefore, we might not have this piece of information. If we try to obtain this piece of information dynamically, and we have not encountered an execution where these pointers alias, then we do not have this piece of information either, because there might be another input for which the pointers alias. A conservative attack would in this case consider both possibilities: they alias or they do not and is therefore guaranteed to be sound for a superset of the real executions. This is illustrated in Figure 1. An approximative attack might assume that they do not alias. Whether this assumption is sound, depends on whether or not there exists an execution in which these pointers alias. In Figure 1(a) we are not as conservative as we theoretically should be, but we still consider a superset of all the possible executions, while in Figure 1(b) the set of executions for which the attack is sound is only a subset of all the possible executions.

This classification is important. In theory, many attacks are unsound, while in practice, they are sound, or at least sound under the conditions an attacker is interested in. For example, a cracked program's behavior might differ when illegal inputs are fed to it, but who cares? In practice, the theoretical argument that no sound attacks exist against a defense mechanism, is hence void.

2.4 Stand-alone versus Environment-dependent

An attack that results in a fully functional program in which the software protection mechanism has been circumvented, is called a *stand-alone* attack. This kind of attack results in a program that can be executed without further requirements to the computing environment of the original program. On the other hand, attacks which require changes to the computing environment during subsequent executions of the attacked program, for example changes to the operating system or the attachment of a dynamic instrumentor each time the cracked program is run, are called *environment-dependent* attacks.

3. A REALISTIC ATTACK MODEL

When considering the strength of any software protection mechanism, it is important to have a realistic attack model. In the past, many proposals have –silently or explicitly– assumed that an attacker mounts an attack based solely upon static information. This assumption is reflected in the titles of some publications. For example, Wang's technical report "Software tamper resistance: Obstructing static analysis of programs" [34], and "Obfuscation of executable code to improve resistance to static disassembly" by Linn *et al.* [21].

In practice, advanced debuggers, such as SoftICE [12] and IDA Pro [13] are already commonly used. These tools allow for the incorporation of dynamically obtained information to form a view of the program. Therefore, it is not realistic to assume that an attacker will limit his attacks to static-only attacks.

We furthermore believe that an attacker will not be overly conservative. Rather, he might be conservative under realistic assumptions. He might for example assume that the calling conventions will be respected or that instructions will not overlap. The attacker may even be nonconservative, but approximative and practically sound. If we, e.g., look at a situation where an attacker tries to modify a program to circumvent a copy protection mechanism, will it bother him that, in theory, the program might contain an additional bug because he was not able to model the entire program? An attack is often performed by trial and error. If an attack results in a program that performs all the actions required by 99% of the user base of a program correctly, it will still pose a serious threat to any software provider. As these attacks have long been applied in practice, we believe that future proposals should also provide a reasonable level of protection against this kind of attack.

Any software protection proposal should take into account that an attacker can:

- inspect every instruction or data value at every program point;
- modify every instruction or data value at every program point;
- start and stop execution of the program at every program point.

In the next section, we describe how an attacker may use dynamic instrumentation to perform these actions automatically, and to use the thus collected information to construct a control flow graph of a program, thus thwarting many of the proposed software protection techniques.

4. CONTROL FLOW GRAPH CONSTRUCTION

As any attack requires at least some understanding of the program, a first line of defense is often to prevent an attacker from obtaining such an understanding. A very useful, and omnipresent, aid in program understanding and analysis is the control flow graph. This graph consists, in its most simple form, of the set of nodes, representing the instructions in the program, and a set of edges between these nodes, representing possible control flow paths. [24].

As such, this graph represents a superset of all the possible paths through the program. Clearly, when analyzing the program, it is important to have a control flow graph that is as complete and correct as possible. A possible defense against attacks therefore consists of making it hard to construct a control flow graph. This can be done by hiding the two types of information needed to construct a control flow graph: instructions and control transfers.

These types of information hiding (control transfer and instruction) can be combined to achieve a higher level of obfuscation. This is illustrated in Figure 2. On the left-hand side, we can see the original control flow graph. Through an obfuscating transformation (b) the control transfers are hidden and the instructions themselves are hidden as well (b). This complicates the construction of the flow graph.

In practice, however, it is not easy to hide information, as the attack model is not a mild one: the attacker can inspect the program at will as the program should still perform the required functionality when processed by a computer.

It is exactly this latter demand that empowers the attacker: sooner or later, the program has to be executed and at that point, we will monitor the program to retrieve parts of the hidden information.

4.1 Collecting Information

Collecting a program's run-time behavior information is usually done through instrumentation. From an attacker's perspective, instrumentation can unfortunately be made more difficult through tamper-resistance and/or obfuscation.

Instrumentation can be performed at different levels of abstraction: the hardware level, the library level, the source code level, and the machine code level. Instrumentation at the hardware level is expensive, instrumentation at the library level is not flexible enough to collect the required information. Furthermore, the source code may not be available.

The only option left is therefore the machine code level. Static machine code level instrumentation techniques cannot be applied to programs that have been obfuscated, as they require a certain level of program understanding that might not be available. For example, it might be unknown which computations depend on code addresses. In that case, it is unknown which computations need to be adapted if instrumentation code is inserted into a program, thus requiring the original code to be moved. Furthermore, instrumentation at this level can obviously not be applied to programs that have been made tamper-resistant, as they require the insertion of instrumentation code.

In compiler-generated programs, two major difficulties arise when inserting code into programs: (i) correctly distinguishing between code and data and (ii) correctly relocating the code and data after the insertion. When the program is made self-modifying or tamper-resistant, additional difficulties need to be circumvented. In traditional systems,

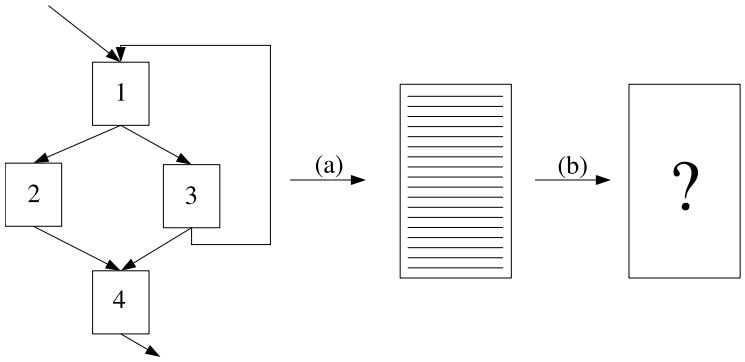

Figure 2: Hiding Information: (a) control transfers, (b) instructions

(i) and (ii) are solved by applying a sophisticated analysis (flow graph construction) of the program, using assumptions about the origin of the code. Most systems can be broken by unconventional code. The other problems are basically ignored as they does not occur in practice very often.

We, on the other hand, need a technique that does not rely on assumptions, reverse engineering or program understanding to instrument the program. Furthermore, it must work in every situation, even when the code has been obfuscated, is self-modifying or is made tamper-resistant.

Therefore, we will use DIOTA [23], a dynamic instrumentor which has none of the limitations just discussed. The basic idea behind DIOTA is to keep a running program unaltered at its original location and to generate instrumented code on the fly somewhere else. The generated code is constructed in such a way that it will use the original program for its data accesses and address computations, while the generated code is used only for actual execution. Hence instrumenting data in the code does not pose any problems as data is always taken from the original program. Code in data is also treated correctly because (executed) data is also instrumented as if it were code. Self-modifying code is processed elegantly by instrumenting the store-operations in such a way that they are not limited to writing the data in the original program, but that they also (re)generate an instrumented version of the data. As the original program is not modified and it is the unmodified version that is inspected by tamper-resistant code, such code does not pose a problem either.

4.2 Collecting Instructions

The first step in analyzing a program is to retrieve the instructions that make up the program, i.e., disassembling the program. The main problem here derives from the representation of data and instructions in the stored-program computer. Distinguishing code from data in a binary file is in general an undecidable problem [17].

This problem is even more apparent on architectures with variable instruction length and where instructions do not need to be aligned, such as the IA-32 architecture. It is considered to be one of the main problems in reverse engineering [4]. In practice, however, heuristics such as linear sweep [21] and recursive traversal [29] often suffice.

These algorithms rely on assumptions which are often valid when no attempts have been made to thwart disassembly. As could be expected, obfuscators have developed techniques to undermine these assumptions. We will summarize these algorithms and associated exploits.

Table 1: An instruction begins at every byte of the program

byte	instruction	instruction size	executable
8b	`mov 4(%exp),%eax`	4	yes
44	`inc %esp`	1	no
24	`and $4,%al`	2	no
04	`add $3,%al`	2	no
03	`add 12(%esp),%eax`	4	yes
44	`inc %esp`	1	no
24	`and $12, %al`	2	no
0c	...	...	no

Linear sweep basically begins to disassemble at the program's entry point, and sweeps through the entire text section, disassembling each instruction as it is encountered. As such it cannot distinguish data embedded in the text section. This weakness can be exploited by inserting "junk" bytes at selected locations in the instruction stream [21]. Another approach consists of overlapping adjacent instructions [5].

Recursive traversal tries to overcome some of these limitations by dealing more intelligently with control flow. As such, it can disassemble around data in the text segment. To this end, it assumes that control transfers behave "reasonably". We will delay the discussion of these assumptions and techniques that exploit these assumption to the next section as they mainly obfuscate control transfers.

If we cannot rely on these heuristics, we must assume that an instruction can begin at every byte. As such, we will need to consider a lot of instructions that can never occur in any trace of the execution of the program. In Table 1, we have illustrated what this would mean for an example code fragment of eight bytes [21]. In the program from which this code was taken, only two of these instructions will ever be executed, but if we cannot detect this, we must conservatively assume that they can all be executed. As such, the real instructions are hidden amongst a large set of unexecutable instructions.

As we noted earlier, the main difficulty with obfuscating transformations is that they need to preserve program

semantics, i.e. the input/output behavior of the program needs to be identical to that of the original program. This means that an attacker is still able to execute the program without any additional efforts. Therefore, the instructions that are actually executed will sooner or later be visible (in the absence of tamper-resistant hardware).

Using the dynamic instrumentation techniques discussed in Section 4.1, we can record the instructions that are executed on a large set of training input sets. This process is highly automated: we instrument the program in such a way that the information is collected during normal execution. This way, the user can continue to use the program as he normally would, except for a small slowdown. Meanwhile, dynamic information is collected.

As opposed to the results returned by static disassemblers, there is no uncertainty about which instructions were executed. Therefore, the dynamic information is guaranteed to be correct. Unfortunately, as known from the literature on code coverage, it is not possible to achieve full coverage of the program. However, a coverage of 80% is realistic [11].

The instructions that are identified through this dynamic information can be seen as a skeleton of the program, which is passed to another static disassembler. This disassembler is an extension of an existing recursive traversal disassembler: instead of starting from scratch, it starts from the skeleton and fills in the missing holes. If the detected instructions conflict with the dynamically found instructions, they are ignored, otherwise, they are added to the disassembly. This way, we obtain a disassembly of the program in which we are certain about the largest part of the program (the dynamically observed instructions), and have approximative information about the rest of the program (static extension of the dynamic information). This process is illustrated in Figure 3.

4.3 Collecting Control Transfers

As noted earlier, we need two types of information to construct the control flow graph: the instructions and the control transfers. The previous section discussed how the instructions could be retrieved. This section will discuss the problems involved with detecting the control transfers.

A large number of tools are confronted with this problem, including binary rewriters [20], link-time optimizers [14, 28] and reverse engineering tools [4, 13]. Once again they rely on a number of assumptions to detect the possible control transfers. One of these assumptions is that a conditional branch has two possible targets: one to the branch target and one fall-through to the next instruction. Similarly, a function call is assumed to return to the instruction immediately following the call instruction.

Indirect control transfers pose an additional challenge. In this case tools usually 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.

Once again the obfuscator will try to undermine these assumptions. The assumption that conditional branches have two possible targets can be undermined using opaque predicates [10]. An opaque predicate is a boolean expression whose value is constant and known to the obfuscator. However, it is hard for a deobfuscator to prove that this value is constant. This means that a deobfuscator has to assume that both targets are valid, while only one is in practice.

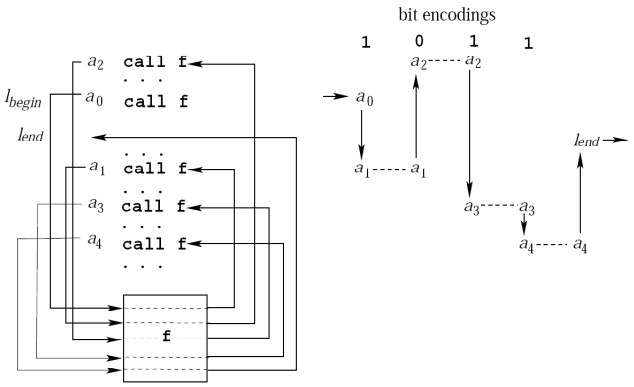

Figure 4: An example of branch-function-based watermarking (watermark = 1011)

The assumption that calls return to the instruction following the call can be undermined by replacing direct jumps with calls to a function which redirects control to the target of the original jump instruction. And finally, new jump tables can be introduced artificially or existing jump tables can be enlarged with branch targets that will never be taken [21]. A conservative static analysis has to assume that an indirect jump can jump to every byte of the program unless it can prove otherwise.

This way, the real control transfers are hidden amongst a larger set of control transfers that will never be executed.

By executing the instrumented program, we can collect control transfers that are executed in practice. This information is again guaranteed to be correct and it is extended by the information returned by static analyses as long as it does not conflict with the dynamically obtained information.

5. CASE STUDY

In this section, we will give a detailed explanation on how to attack a recently proposed algorithm for software watermarking [6] through a hybrid static-dynamic attack. The idea of this watermarking algorithm is to encode a watermark in the dynamic branching behavior of the program.

In the remainder of this section, we will first explain the watermarking algorithm. Next, we show how an attacker is able to detect whether a watermark is present in a program. Finally, we will remove the watermark from the program, while maintaining the input/output behavior.

5.1 Watermarking a program

The technique described by Collberg *et al.*[6] operates by extending a program with a branch function. A branch function is a function that is called in the normal manner, but which manipulates the return address such that control is transferred to an address different from the call site.

The watermark is then embedded in the correspondence between the original return address and the address to which control is actually transferred. A forward jump encodes a '1', while a backward jump encodes a '0'. This concept is illustrated in Figure 4 ([6]). Starting at a_0, the first bit is 1 and is encoded by the forward branch $a_0 \rightarrow a_1$, which is realized via a call to the branch function $f()$; the next bit, 0, is encoded by the backward branch $a_1 \rightarrow a_2$, etc.

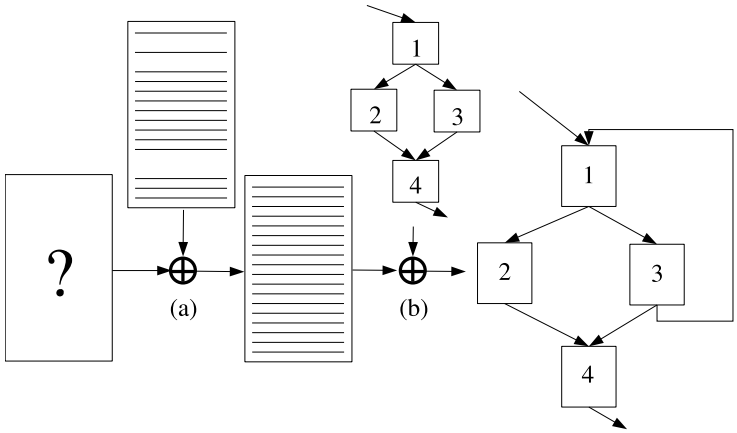

Figure 3: Extending dynamic information with static information to retrieve (a) instructions and (b) control transfers

If the mapping between the original return address and the address to which control will actually be transfered is known by an attacker, he can remove the indirection. This would successfully remove the watermark. In order to hide this mapping, perfect hashing is used to compute the correspondence at run time.

Furthermore, the watermark is made tamper-resistant by introducing side-effects in the branch function. These side-effects influence the behavior of the program. Therefore, the branch-function cannot simple be bypassed in a cracked version of the program, as this bypassing would also remove the required side-effects.

5.2 Locating the watermark

In general, it should be as hard as possible for a cracker to know if the program he's interested in contains a watermark. The presence of a watermark should hence not be discovered by some simple general truth checking, such as simple questions like *Could the assembly code be the output of a regular compiler?* Some unrealistic behavior of a program could be the presence of instructions that a compiler does not generate, the presence of a variable that is used in each function of the program, a lot of jumps with an unrealistic large offset, and many others.

The first phase of the attack consists of building an approximate control flow graph based upon dynamic information as described in Section 4. Immediately, some real unusual behavior can be noticed: there are a lot of calls to one function, and the return never goes back to the caller function. First of all it is quite strange that a function does not return to the caller function, but returns to a piece of code that is located somewhere else. The only possibility to explain the behavior as regular could be the use of tail call optimizations during compilation. A detailed inspection reveals that the instruction before the target is not yet executed and is even not a call. Secondly, the function has an unusually high in- and out-degree. For example: in the benchmark gcc, there are 15336 call sides to this single branch function. In this control flow graph, it can't be denied that this single function sticks out like a sore thumb.

Detailed inspection of the now highly suspicious branch function reveals a lot more, even without knowledge about the algorithm used to compute the branch address. When dynamically stepping through the branch function, the original return address is used for several calculations and a new return address is pushed on the stack. The function seems to compute a new target by use of a unique key, the old return address. Dynamically, there are exactly two differences between the start and the end of the function: the top of the stack (the return address) is changed and two entries in the data are changed which are important side effects.

Before changing the callers to the branch function into regular jumps, we have to be sure that the side effects do not affect the functionality of the program. Entries in memory that are changed by the branch function are monitored before and after they are changed. If the cracker is sure that the memory locations are only used after they are changed, the correct values could be written out in the binary before execution. Otherwise, a new piece of code could be added to the program that takes care of the side effects.

5.3 Removing the watermark

Once we have successfully identified the branch functions, we statically detect all the calls to branch functions. Subsequently, we start the execution of the program under the control of `gdb`, a known, open-source debugger (http://www.gnu.org/software/gdb/). We are now in full control of the program, we can modify every register, instruction and memory location and we can jump to anywhere in the code. Below, we have listed the gdb command-line commands that allow us to retrieve the correspondences between original return addresses and the addresses to which control will be transferred.

```
1 break return_of_branch_function
2 set *((int *) $esp = 5+address_of_call
3 set $pc = start_of_branch_function
4 continue
5 print /x $esp
```

The idea is both simple and powerful. We put a breakpoint at the return instruction of the branch function (1). The address of the instruction immediately following the call to the branch function is pushed on the stack (2). This is the address that would be placed on the stack when executing the call instruction. Next, we set the program counter to the

Table 2: Number of calls to the branch function

	branch function calls
bzip2	4963
crafty	5150
gap	8705
gcc	15336
gzip	5043
mcf	4360
parser	4878
twolf	5361
vortex	7432
vpr	4861
average	**6608.9**

beginning of the branch function (3) and continue execution at that point (4). The execution will stop when the branch function has finished execution and we can simple read the value on top of the stack to know where control would have been transferred to (5). This way we can detect the correspondence between the original return addresses and the addresses to which the branch function will transfer control. Clearly, the security cannot be improved by using a stronger hash function: an attacker can simply call the function, he does not need to analyze it. Furthermore, the hash function should not consume a lot of resources as it will be executed during normal execution as well.

The approach is extended to undo the tamper-resistance of the watermark by putting an additional breakpoint at the instruction of the branch function that modifies a memory address. When the breakpoint is met, the address where and the value that would be written is printed.

This process is automated and repeated for all calls to branch functions. This way, we acquire enough information to replace all the indirections through the branch-function by jumps to the corresponding addresses. The tamper-resistance is circumvented by simply writing out the correct addresses in the binary program, instead of the random values that were previously there.

We have applied this attack to a number of benchmarks we received from the authors of the original paper. Table 2 lists the number of calls to the branch function, all of which are removed by our attack. The watermark was 128 bit in each benchmark and we found 129 memory locations in each benchmark, used for tamper-proofing. The attack takes about one hour on a 2.8Ghz Pentium.

An attack against a watermarked program is successful if the watermark cannot be reliably extracted from the attacked program, and if the performance of the attacked program is such that it still has value to the attacker. Our attack does not decrease the performance of the program, rather it removes a number of indirections, thereby slightly improving the performance. As the watermark is no longer present, we have successfully broken the protection.

6. RELATED WORK

Any form of software protection can be seen as a battle between two parties: the defenders, who try to protect intellectual property contained within or protected by the software, and the attackers, who try to circumvent the protection mechanisms.

We will limit our discussion to mechanisms for self-defense, meaning that we will not consider approaches that require changes to the hardware, operating system, or any other part of the computing system other than the program itself. Most of the publications in academic literature have looked at the problem of software protection from the side of the defenders. A great variety of defenses have been proposed, including software watermarking [8, 30, 33], code obfuscation [10, 21, 22, 35] and tamper-resistance [2, 3, 16]. We refer to a number of recent survey papers [9, 26, 32] for a more extensive list of techniques and publications.

Publications that look at the problem from the side of the attackers are however rare. The group of Collberg has performed an evaluation of some existing watermarking techniques [7, 25, 27], Kruegel has proposed a disassembler that is more resistant against obfuscating transformations [19] and Wurster *et al.* have proposed an attack against techniques for tamper-resistance based on checksumming [36]. To the best of our knowledge, only one paper discusses a hybrid dynamic-static attack [31]. In this paper, the control-flattening of Wang [35] is attacked.

7. CONCLUSIONS

In this paper, we have discussed the power of hybrid static-dynamic attacks. While this kind of attack has long been used in practice, recent proposals fail to address it, perhaps because the problem was not fully understood. This is one of the first scientific papers to clearly pinpoint the associated threat and the potential of this kind of attack. An attacker can inspect the program at will and control it in any way he likes. This enables him to collect a lot of information upon which an attack can be based. It is our hope that a better understanding of this type of attack will lead to stronger software protection mechanisms in the future.

The strength and potential of this attack has furthermore been illustrated by attacking a recent technique for software watermarking that was presented by highly respected authors in the domain and at one of the leading conferences.

Acknowledgments

The authors would like to thank the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT), the Fund for Scientific Research - Belgium - Flanders (FWO) and Ghent University, member of the HiPEAC network, for their financial support.

8. REFERENCES

[1] Atari Games Corp. vs. Nintendo of America Inc., U.S. Court of Appeals, Federal Circuit, September 10,1992.

[2] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the first ACM Workshop on Digital Rights Management*, pages 160–175, 2001.

[3] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. In *Proceedings of the 5th International Workshop on Information Hiding*, pages 400–414, 2002.

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

[5] F. Cohen. Operating system evolution through program evolution, 1992.

[6] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 107–118, 2004.

[7] C. Collberg, A. Huntwork, E. Carter, and G. Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In *Proceedings of the 6th Workshop on Information Hiding*, pages 192–207, 2004.

[8] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Proceedings of the 26th Conference on Principles of Programming Languages*, pages 311–324, 1999.

[9] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[10] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *The 25th Conference on Principles of Programming Languages*, pages 184–196, 1998.

[11] S. Cornett. Code coverage analysis, 2004. http://www.bullseye.com/coverage.html.

[12] C. Corporation. SoftICE. http://www.compuware.com/.

[13] Datarescue. IDA Pro. http://www.datarescue.com/.

[14] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. *SIGPLAN Notices*, 39(7):211–220, 2004.

[15] M. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[16] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Proceedings of the first ACM Workshop on Digital Rights Management*, pages 141–159, 2002.

[17] R. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[18] International Planning and Research Corporation. *First Annual BSA and IDC Global Software Piracy Study*, 2004.

[19] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13the USENIX Security Symposium*, pages 255–270, 2004.

[20] J. R. Larus and E. Schnarr. Eel: machine-independent executable editing. In *Proceedings of the Conference on Programming Languages Design and Implementation*, pages 291–300. ACM Press, 1995.

[21] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and Communication Security*, pages 290–299, 2003.

[22] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *Proceedings of the 6th International Workshop on Information Security Applications*, pages 371–385, 2005.

[23] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of the 4th Workshop on Binary Translation*, 2002.

[24] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publischers, Inc., 1997.

[25] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. In *Proceedings of the 7th International Conference on Electronic Commerce Research*, 2004.

[26] G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[27] T. Sahoo and C. Collberg. Software watermarking in the frequency domain: Implementation, analysis, and attacks. Technical Report TR04-07, Dept. of Computer Science, Univ. of Arizona, 2004.

[28] B. Schwarz, G. Andrews, M. Legendre, and S. Debray. PLTO: A link-time optimizer for the intel ia-32 architecture. In *Proceedings of the 3rd Workshop on Binary Rewriting*, 2001.

[29] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 45–54, 2002.

[30] J. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In *Proceedings of the 5th International Workshop on Information Hiding*, pages 368–378, 1999.

[31] S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*. IEEE Computer Society, 2005.

[32] P. van Oorschot. Revisiting software protection. In *Proceedings of the 6th International Information Security Conference*, pages 1–13, 2003.

[33] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Workshop on Information Hiding*, pages 157–168, 2001.

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

[35] C. Wang, J. Hill, J. Knight, and J. Davidson. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 193–202. IEEE Computer Society, 2001.

[36] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 127–138. IEEE Computer Society, 2005.

[37] C. Xiao. Delayed secure data retrieval, May 2003. International Business Machines Corporation, US Patent 6571337.