Software Protection through Dynamic Code Mutation

Matias Madou⋆, Bertrand Anckaert⋆, Patrick Moseley‡, Saumya Debray‡, Bjorn De Sutter⋆ and Koen De Bosschere⋆

⋆ Department of Electronics and Information Systems
Ghent University
B-9000 Ghent, Belgium
{mmadou, banckaer, brdsutte, kdb}@elis.UGent.be

‡Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
{moseley,debray}@cs.arizona.edu

Abstract. Reverse engineering of executable programs, by disassembling them and then using program analyses to recover high level semantic information, plays an important role in attacks against software systems, and can facilitate software piracy. This paper introduces a novel technique to complicate reverse engineering. The idea is to change the program code repeatedly as it executes, thereby thwarting correct disassembly. The technique can be made as secure as the least secure component of opaque variables and pseudorandom number generators.

1 Introduction

To reverse-engineer software systems, i.e., to obtain an (at least partial) understanding of the higher-level structure of an executable program, a malicious attacker can subvert many recent advantages in program analysis technology and software engineering tools. Thus, the existing technology can help an attacker to discover software vulnerabilities, to make unauthorized modifications such as bypassing password protection or identifying and deleting copyright notices or watermarks within the program, or to steal intellectual property.

One way to address this problem is to maintain the software in encrypted form and decrypt it is as needed during execution, using software decryption [1], or specialized hardware [18]. Such approaches have the disadvantages of high performance overhead or loss of flexibility, because software can no longer be run on stock hardware.

To avoid these disadvantages, this paper instead focuses on an alternative approach using code obfuscation techniques to enhance software security. The goal is to deter attackers by making the cost of reverse engineering programs prohibitively high.

The seminal paper on decompilation and reverse engineering [4] considers two major difficulties in the process of reverse engineering programs. The first problem is that data and code are indistinguishable, as code on a Von Neumann computer is nothing more than a specific type of (binary) data. The second problem relates to self-modifying code, which does not follow the convention of static code that there is a one-to-one mapping between instructions and memory addresses.

In this paper, we propose a novel technique to automatically aggravate and/or introduce these problems in existing programs. The basic idea is to mutate a program as it executes, so that a region of memory is occupied by many different code sequences during the course of execution. We show how this technique undermines assumptions made by existing analyses for reverse engineering. Furthermore, we claim that our technique can be made as secure as the least secure component of opaque variables [5] and pseudorandom number generators [24].

The goal of this research is to deter "ordinary attackers" by making it substantially more difficult to reverse engineer the obfuscated code; it is consistent with the prior work on code obfuscation, which aims primarily to raise the bar against reverse engineering high enough so as to deter all but the most determined of attackers.

The remainder of this paper is structured as follows: Section 2 discusses related work. Our technique is introduced in Section 3. The security of this technique is the topic of Section 4. An evaluation of the impact on the size and execution time of the program is discussed in Section 5. Finally, conclusions are drawn in Section 6.

2 Related Work

The only other paper we are aware of that proposes dynamic code modifications for obfuscation purposes is that of Kanzaki *et al.* [16], which describes a straightforward scheme for dynamically modifying executable code. The central idea is to scramble a selected number of instructions in the program at obfuscation time, and to restore the scrambled instructions into the original instructions at run time. This restoration process is done through modifier instructions that are put along every possible execution path leading to the scrambled instructions. Once the restored instructions are executed, they are scrambled again. It is however not clear how the modifier instructions pose problems for a static analysis targeted at restoring the original program.

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 [6, 7, 21, 25]. 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 understand the statically disassembled program. Our work is complementary to these proposals: we aim to make a program harder to disassemble correctly to begin with, let alone recover high level information. If a 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.

Researchers have looked into run-time code generation and modification, including high-level languages and APIs for specifying dynamic code generation [3, 12, 13] and its application to run-time code specialization and optimization [2, 17, 20]. Because that work focuses primarily on improving or extending a program's performance or functionality, rather than hindering reverse engineering, the developed transformations and techniques are considerably different from those described in this paper.

A run-time code generation techniques that to some extent resembles the technique proposed in this paper was proposed by Debray and Evans [11] for applying profile-guided code compression. To reduce the memory footprint of applications, infrequently executed code is stored in compressed format, and decompressed when it needs to be executed. At any point, only a small fraction of the infrequently executed code is in decompressed form. Because of the large decompression overhead however, the frequently executed code is always available in decompressed, i.e., the original, form. Hence this compression technique does not hide the frequently executed portions of a program, which are generally also likely to contain the code one might wish to protect.

3 Dynamic Software Mutation

This section discusses the introduction of dynamic software mutation into a program. We consider two types of mutation: one-pass mutation, where a procedure is generated once just before its first execution, and cluster-based mutations, where the same region of memory is shared by a cluster of "similar" procedures, and where we will reconstruct procedures (and thus overwrite other procedures) as required during the execution. We first discuss our novel approach to run-time code editing (Sec. 3.1). This will enable us to treat the one-pass mutations (Sec. 3.2). Next, we look at how "similar" procedures are selected (Sec. 3.3) and clustered (Sec. 3.4). Finally, we propose a protection method for the edit scripts against attacks (Sec. 3.5) and discuss our technique's applicability (Sec. 3.6).

3.1 The Run-time Edit Process

Our approach is built on top of two basic components: an editing engine and edit scripts. When some procedure, say f, is to be generated at run-time, it is statically replaced by a template: a copy of the procedure in which some instructions have been replaced by random, nonsensical, or deliberately misleading instructions. All references to the procedure are replaced by references to a stub that will invoke the editing engine, passing it the location of the edit script and the entry point of the procedure. Based upon the information in the edit script, the editing engine will reconstruct the required procedure and jump to its entry point.

Edit Script. The edit script must contain all the necessary information to convert the instructions in the template to the instructions of the original procedure. This information includes the location of the template and a specification of the bytes that need to be changed and to what value. The format we used to encode this information is the following:

```
editscript = address <editblock>_1 <editblock>_2 ...<editblock>_l $
editblock  = m <edit>_1 <edit>_2 ...<edit>_m
edit       = offset n byte_1 byte_2 ...byte_n
```

An edit script starts with the address of the template, i.e., the code address where the editing should start. It is followed by a variable sequence of edit blocks, each of which specifies the number of edits it holds and the sequence thereof, and is terminated by the stop symbol $. An edit specifies an offset, i.e., a number of bytes that can be skipped without editing, followed by the number of bytes that should be written and the bytes to write. As all the values in the edit script, except the address, are bytes, this allows us to specify the modifications compactly, while still maintaining enough generality to specify every possible modification.

Editing Engine. The editing engine will be passed the address of the edit script by the stub. It will save appropriate program state, such as the register contents, interpret the edit script, flush the instruction cache if necessary, restore the saved program state and finally branch to the entry point of the procedure, passed as the second argument. Note that the necessity of flushing the instruction cache depends on the architecture: on some architectures, such as the Intel IA-32 architecture used for our current implementation, an explicit cache flush is not necessary.

Our approach to dynamic code editing modifies the template code *in situ*. This is an important departure from classical sequence alignment and editing algorithms [9], which scan a read-only source sequence, copying it over to a new area of memory and applying modifications along the way where dictated by the edit script. With *in situ* modifications this copying can be avoided, thereby increasing performance. Insertion operations are however still expensive, as they require moving the remainder of the source. Consequently, we do not support insertion operations in our edit scripts. Instead only substitution operations are supported. Deletion operations may be implemented by overwriting instructions with no-op instructions, but as this introduces inefficiencies, we will avoid this as much as possible.

3.2 One-Pass Mutations

We are now ready to discuss one-pass modifications. With this technique, we scramble procedures separately, meaning that each procedure will have its own template. Consequently, different procedures are not mapped to the same memory location. The idea at obfuscation time is to alter portions of a procedure in the program. At run-time, these alterations are undone via a single round of editing, just before the procedure is executed for the first time. To achieve this, we place the stub at the entry point of the procedure. At the first invocation of the editing engine, this stub will be overwritten with the original code of the procedure. This way, the call to the editor will be bypassed on subsequent calls to the procedure.

3.3 Cluster-Based Mutations

The general idea behind clustering is to group procedures of which the instruction sequences are sufficiently similar to enable the reconstruction of the code of each of them from a single template without requiring too many edits. The procedures in a cluster will then be mapped to the same memory area, the cluster template. Each call to a clustered procedure is replaced by a stub that invokes the editing engine with appropriate arguments to guide the edit process, as illustrated in Figure 1.

To avoid reconstructing a procedure that is already present, the editing engine will rewrite the stub of a constructed procedure in such a way that it branches directly to that procedure instead of calling the editing engine. The stub of the procedure that has been overwritten, will be updated to call the editing engine the next time it needs to be executed.

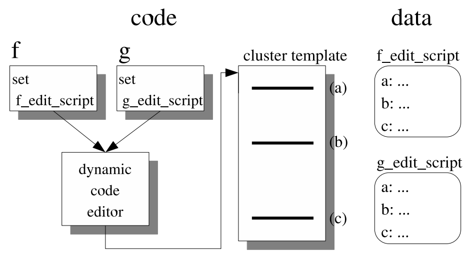


Fig. 1. Run-time code mutation with clustered procedures

Clustering. Clustering is performed through a node-merging algorithm on a fully-connected undirected weighted graph in which each vertex is a cluster of procedures and the weight of an edge (A, B) represents (an estimate of) the additional run-time overhead (i.e., the cost of the edits) required when clusters A and B are merged.

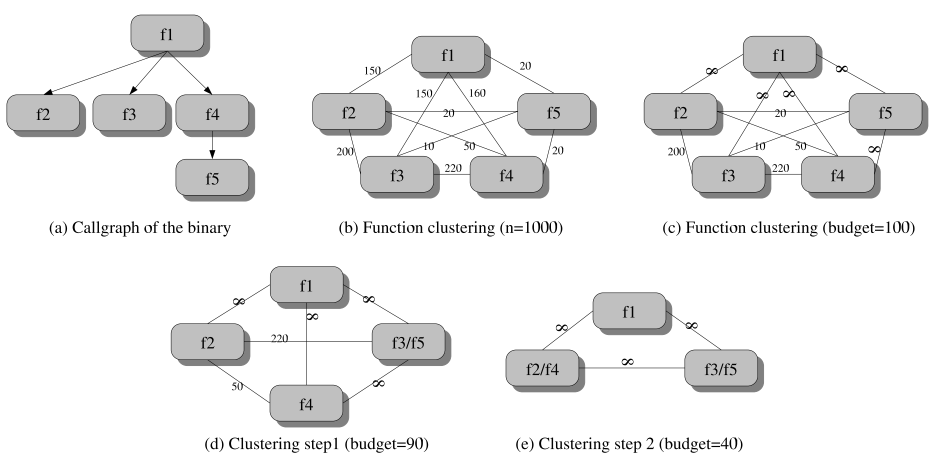
The number of run-time edits required by a cluster, i.e., the number of control flow transfers between two members of that cluster, is estimated based on profiling information drawn from a set of training inputs.

As usual, the clustering process has to deal with a performance trade-off. On the one hand, we would like every procedure to be in an as large as possible cluster. The larger we make individual clusters –and therefore, the fewer clusters we have overall– the greater the degree of obfuscation we will achieve, since more different instructions will map to the same addresses, thus moving further away from the conventional one-to-one mapping of instructions and memory addresses.

On the other hand, the larger a cluster, the more differences there will likely be between cluster members, resulting in a larger set of edit locations, and hence a greater run-time overhead. Furthermore, this will result in an increasing number of transitions between members within a cluster. With transition, we mean the execution of one member of a cluster after the execution of another member. Clearly, each transition requires editing the next procedure to be executed. Both these factors increase the total run-time cost of the dynamic modification.

When our greedy clustering algorithm starts, each cluster consists of a single procedure. The user needs to specify a run-time overhead "budget" (specified as a fraction ϕ of the number of procedure calls n that can be preceded by a call to the editing engine, i.e, budget=$n \times \phi$). As we want all procedures to be in an as large as possible cluster, we proceed as follows. First we try to create two-procedure clusters by only considering single-procedure clusters for merging. The greedy selection heuristic chooses the edge with the lowest weight and this weight is subtracted from the budget. We then recompute edge weights by summing their respective weights to account for the merge. When no more two-procedure clusters can be created, we try to create three-procedure clusters, using the same heuristic, and so on.

Merging clusters is implemented as node coalescing. This sets an upper bound to the actual cost and hence is conservative with regard to our budget. This is repeated until no further merging is possible. A low value for the threshold ϕ produces smaller clusters and less run-time overhead, while a high value results in larger clusters and greater obfuscation at the cost of higher overhead. It is important to note that two procedures that can be active together should not be clustered. Otherwise, their common template would need to be converted into two different procedures at the same time, which obviously is not possible.



(a) Callgraph of the binary (b) Function clustering (n=1000) (c) Function clustering (budget=100)

(d) Clustering step1 (budget=90) (e) Clustering step 2 (budget=40)

Fig. 2. The creation of clusters, ϕ=0.1

These concepts are illustrated in Figure 2. The call graph is shown in Figure 2(a). It is transformed into a fully connected new graph, where the initial nodes are clusters consisting of exactly one procedure. The weight given to the other edges between two clusters is the number of transitions between the respective procedures in those clusters, i.e., the number of calls to the editor that would result from merging these two procedures. These values are collected from a set of training inputs. The resulting graph is shown in Figure 2(b). We furthermore assume that ϕ=0.1 and as the maximum number of procedure calls to the editing engine n is 1000 (10+3*20+50+2*150+160+200+220), a budget of 100 calls is passed to the clustering algorithm. To avoid clustering procedures that can be active at the same time, the edges between such procedures are assigned the value infinity, as illustrated in Figure 2(c).

As our clustering algorithm starts with clusters consisting of a single procedure, the algorithm looks for the edge with the smallest value, which is $(f3, f5)$. The weights of the edges of the merged cluster to the other clusters are updated accordingly. Our graph now consists of three clusters consisting of single procedure ($f1$, $f2$, and $f4$) and one cluster consisting of two procedures (Figure 2(d)). As it is still possible to make clusters of two procedures, the edge with the smallest weight between the three clusters consisting of a single procedure will be chosen (if its weight is smaller than our budget). This way, procedure $f2$ and $f4$ are clustered (Figure 2(e)). As we can no longer make clusters of two procedures, the algorithm now tries to make clusters of size three. This is impossible, however, and so the algorithm terminates.

3.4 Minimizing the edit cost

In this section, we will discuss how the template for a cluster is generated. This is done in such a way that the number of edits required to construct a procedure in the cluster from the template is limited.

This is achieved through a layout algorithm which maximizes the overlap between two procedures. First of all, basic blocks connected by fall-through edges are merged into a single block, as they need to be placed consecutively in the final program. In the example of Figure 3, fall-through edges are represented by dashed lines. Therefore, basic blocks 1 and 2 are merged. This process is repeated for all procedures in the cluster. In our example, there are three procedures in the cluster and the procedures each have two blocks. These blocks are placed such that the number of edits at run-time is minimized, as illustrated in Figure 3. The cluster template consists of sequences of instructions that are common to all the

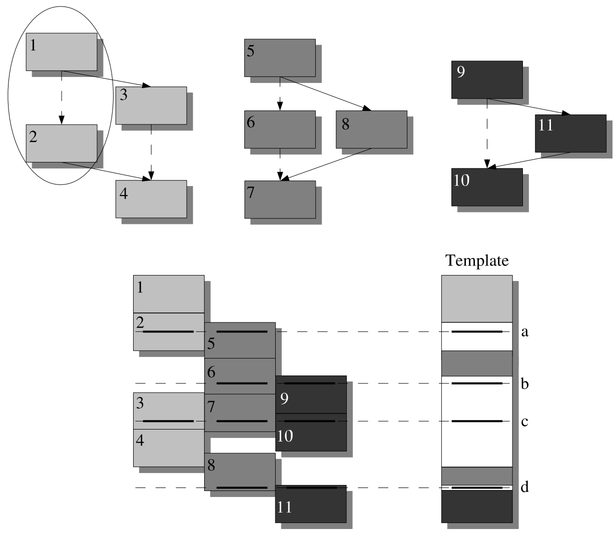


Fig. 3.

procedures and locations that are not constant for the cluster. The locations that are not constant are indicated by the black bars labeled a, b, c, and d. These locations will be edited by the editing engine.

3.5 Protecting Edit Scripts

With the code mutation scheme described thus far, it is possible, at least in principle, for an attacker to statically analyze an edit script, together with the code for the editor, to figure out the changes effected when the editor is invoked with that edit script. To overcome this problem, we will use a pseudorandom number generator seeded with an opaque variable [5]. A variable is opaque at point p in a program, if it has a property at p which is known at obfuscation time, but which is computationally difficult to determine analytically.

The basic idea is to combine the values statically present in the edit script with a value generated by the pseudorandom number generator. As we know the value of the seed (opaque variable) at obfuscation time, we can predict the values that will be generated by the pseudorandom number generator. Therefore, it is possible to write values in the edit script which will produce the needed values when combined with the pseudorandom numbers. Every byte in the edit script is then xor'ed with a byte created by the pseudorandom number generator before it is passed to the editing engine.

3.6 Applicability

Dynamic code mutation relies fundamentally on statically constructing edit scripts that can be used to carry out run-time code mutation. This presumes that a program's code is statically available for analysis and edit script construction. Because of this, the technique is not applicable to code that is already self-modifying. Dynamic code mutation also causes instruction opcodes and displacements to change. New instructions are inserted in procedure stubs, and displacements in branch and call instructions may change as a result of code movement. This precludes the application of dynamic code mutation to programs that rely on the actual binary values of code locations (as opposed to simply their instruction semantics), e.g., programs that compute a hash value of their instructions for tamper-proofing.

Finally, the contents of the code locations change as dynamically mutating code executes. This means that the technique cannot be applied to reentrant code such as shared libraries. Note that while this is an issue for multi-threaded programs as well, we can deal with multi-threading using static concurrency analyses to identify code regions that can be executed concurrently in multiple threads [19], and use this information to modify clustering to ensure that code regions that can execute concurrently in multiple threads are not placed in the same cluster for mutation.

4 Security Evaluation

In this section we will discuss the security of our technique against attacks.

4.1 Broken assumptions

While the omnipresent concept of the stored program computer allows for self-modifying code, in practice, self-modifying code is largely limited to the realm of viruses and the like. Because self-modifying code is rare nowadays, many analyses and tools are based upon the assumption that the code does not change during the execution.

Static disassemblers, e.g., examine the contents of the code sections of an executable, decoding successive instructions one after another until no further disassembly is possible [22]. Clearly these approaches fail if the instructions are not present in the static image of the program.

Dynamic disassemblers by contrast, examine a program as it executes. Dynamic disassemblers are more accurate than static disassemblers for the code that is actually executed. However, they do not give disassemblies for any code that is not executed on the particular input(s) used.

In order to reduce the runtime overheads incurred, dynamic disassembly and analysis tools commonly "cache" information about code regions that have already been processed. This reduces the runtime overhead of repeatedly disassembling the same code. However, it assumes that the intervening code does not change during execution.

Many other tools for program analysis and reverse engineering cannot deal with dynamically mutating code either. For example, a large number of analyses, such as constant propagation or liveness analysis require a conservative control flow graph of the program. It is not yet fully understood how this control flow graph can be constructed for dynamically mutating code without being overly conservative. Through the use of self-modifying code, we cripple the attacker by making his tools insufficient.

4.2 Inherent Security

While undermining assumptions made by existing analyses and tools adds a level of protection to the program and will slow down reverse engineering, its security is ad-hoc. However, no matter how good reverse engineering tools will become, a certain level of security will remain. As long as the opaque variable or the pseudorandom number generator are not broken, an attacker cannot deduce any other information than guessing from the edit script. Assuming that the opaque variable and pseudorandom number generator are secure, it corresponds to a one-time pad.

Depending on the class of expressions considered, the complexity of statically determining whether an opaque variable always takes on a particular value can range from NP-complete or co-NP-complete[8], through PSPACE-complete[23], to EXPTIME-complete[14].

A lot of research has gone into the creation of secure pseudorandom number generators. For our purposes, we need a fast pseudorandom number generator. ISAAC [15] for example meets this requirement and, in practice, the results are uniformly distributed, unbiased and unpredictable unless the seed is known.

5 Experimental Results

We built a prototype of our dynamic software mutation technique using Diablo, a retargetable link-time binary rewriting framework[10]. We evaluated our system using the 11 C benchmarks from the SPECint-2000 benchmark suite. All our experiments were conducted on a 2.80GHz Pentium 4 system with 1 GiB of main memory running RedHat Fedora Core 2. The programs were compiled with gcc version 3.3.2 at optimization level -03 and obfuscated using profiles obtained using the SPEC training inputs. The effects of obfuscation on performance were evaluated using the (significantly different) SPEC reference inputs.

The prototype obfuscator is implemented on top of the tool Diablo, which only handles statically linked programs. In real-life however, most programs are dynamically linked. To mimic this in our experiments, and obtain realistic results, our prototype obfuscator does not obfuscate library procedures.

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	**Mean**
Nr of functions	31	105	848	1272	56	16	176	891	165	655	91	
No protection	3.23%	5.71%	6.01%	20.68%	17.86%	0.00%	5.68%	11.78%	3.03%	1.22%	13.19%	8.04%
One-pass protection	6.45%	6.67%	75.12%	46.15%	46.43%	25.00%	6.82%	80.13%	4.85%	41.07%	14.29%	32.09%
Cluster protection	90.32%	87.62%	18.87%	33.18%	35.71%	75.00%	87.50%	8.08%	92.12%	57.71%	72.53%	59.88%
total protected	96.77%	94.29%	93.99%	79.32%	82.14%	100.00%	94.32%	88.22%	96.97%	98.78%	86.81%	91.96%

Table 1. Number of procedures that can be protected

Table 1 shows the number of procedures that are scrambled by applying our new obfuscation technique. The value of ϕ was set to 0.0005. Procedures containing escaping edges[1] can't be made self-modifying in our prototype, as it is impossible to make sure that the targeted procedure of the escaping edge is present in memory. On all other procedures, we first applied the clustering mutation. After this first pass, we scrambled the remaining procedures with the one-pass mutation. On average this combined application of the two mutation technique is capable of protecting 92% of all (non-library) procedures in the programs.

In Figure 4 the distribution of the number of procedures per cluster is shown. The value of ϕ was set to 0.0005. On average, there are 3.61 procedures per cluster.

Table 2 shows the run-time effects of our transformations. On average, our benchmarks experience a slowdown of 17.7%; the effects on individual benchmarks range between slight speedups (for gzip and vpr), to an almost 2x slowdown (for vortex). This slight speedup experience is due

[1] Escaping edges are edges where control jumps from one procedure into another without using the normal call/return mechanism for interprocedural control transfers. They are rare in compiler generated code, and can most often be avoided by disabling tail-call optimization.

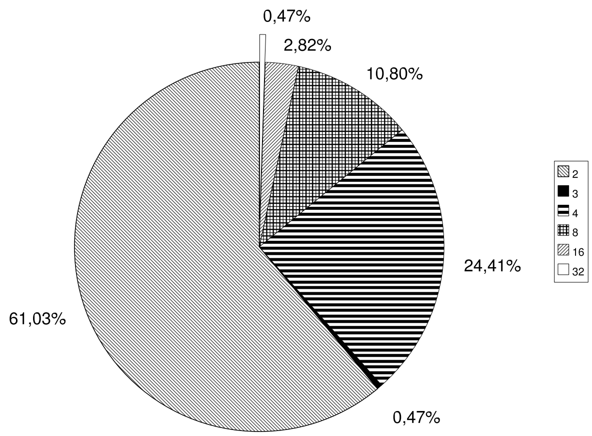


Fig. 4. Number of procedures per cluster

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	**geo. mean**
Original (T_0)	89.140	159.303	128.227	36.623	42.757	429.057	305.060	1.423	611.513	87.820	90.369	
Obfuscated(T_1)	88.037	229.567	150.853	39.697	43.753	429.583	317.573	1.183	618.850	168.170	173.340	
Slowdown (T_1/T_0)	0.988	1.441	1.176	1.084	1.023	1.001	1.041	0.831	1.012	1.915	1.918	**1.177**

Table 2. Relative execution time, ϕ=0.0005

to cache effects. In general, frequently executed procedures, and especially frequently executed procedures that form hot call chains, will be put in separate clusters. Hence these procedures will be mapped to different memory regions. If the combined size of the templates of all clusters becomes smaller than the instruction cache size, the result is that all hot call chains consist of procedures at different locations in the cache. Hence few or none hot procedures will throw each other out of the instruction cache. For gzip and vpr, the resulting gain in cache behavior more than compensates for the, already small, overhead of executing the edit scripts.

Figure 5 summarizes the run-time overhead of our transformations for different ϕ's. On average benchmarks are 31.1% slower with a ϕ=0.005 and 5.9% slower with ϕ=0.00005.

6 Conclusion

This paper introduces an approach to dynamic software protection, where the code for the program changes repeatedly as it executes. As a result, a number of assumptions made by existing tools and analyses for reverse engineering are undermined. We have further argued that the technique

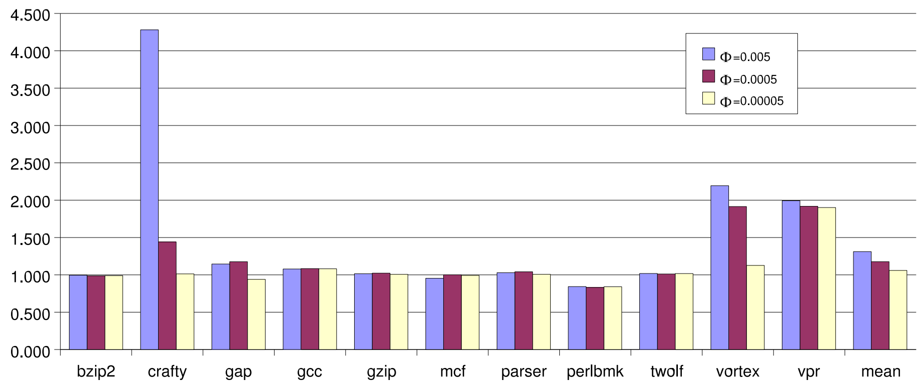


Fig. 5. Execution time slowdown for different values of ϕ

is secure as long as the opaque variables or random number generator have not been broken.

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 for their financial support. The work of Debray and Moseley was supported in part by NSF Grants EIA-0080123, CCR-0113633, and CNS-0410918.

References

1. D. Aucsmith. Tamper resistant software: an implementation. *Information Hiding, Lecture Notes in Computer Science*, 1174:317–333, 1996.
2. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.
3. B. Buck and J. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
4. C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software - Practice & Experience*, pages 811–829, July 1995.
5. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.
6. C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, Aug. 2002.

7. C. S. Collberg, C. D. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.
8. S. A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.
9. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 1991.
10. B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 211–220, 2004.
11. S. K. Debray and W. Evans. Profile-guided code compression. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, pages 95–105, June 2002.
12. D. Engler, W. Hsieh, and F. Kaashoek. ‘c: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, pages 131–144, 1996.
13. M. Hicks, J. Moore, and S. Nettles. Dynamic software updating. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.
14. P. Hudak and J. Young. Higher-order strictness analysis in the untyped lambda calculus. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, pages 97–109, Jan. 1986.
15. R. Jenkins. Isaac. In *Fast Software Encryption*, pages 41–49, 1996.
16. Y. Kanzaki, A. Monden, M. Nakamura, and K. ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *Proc. of the 27th Annual International Computer Software and Applications Conference*.
17. M. Leone and P. Lee. A Declarative Approach to Run-Time Code Generation. In *Workshop on Compiler Support for System Software (WCSSS)*, 1996.
18. D. Lie *et al.* 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, 2000.
19. S. Masticola and B. Ryder. Non-concurrency analysis. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–138. ACM Press, 1993.
20. F. Noel, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 132–142, 1998.
21. T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. In *IEICE Transactions on Fundamentals*, pages 176–186, 2003.
22. B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 45–54. IEEE Computer Society, 2002.
23. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. 5th ACM Symposium on Theory of Computing*, pages 1–9, 1973.
24. J. Viega. Practical random number generation in software. In *Proc. 19th Annual Computer Security Applications Conference*, pages 129–141, 2003.
25. C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *International Conference of Dependable Systems and Networks*, Goteborg, Sweden, July 2001.