A RELATIONAL MODEL FOR SOFTWARE PROTECTION METRICS

Qing Ju, Tapasya Patki, Christian Collberg

Department of Computer Science
University of Arizona

ABSTRACT

Software obfuscation and tamper-proofing algorithms have been widely deployed to protect the intellectual property associated with distributed software and to guard against unintended usage. It has been found that these algorithms tend to make programs larger, slower and raise caching concerns. This paper addresses the relative performance aspects of a few popular software protection algorithms and attempts to define security metrics.

Index Terms— obfuscation, tamperproofing, performance metrics

1. INTRODUCTION

The holy grail of software protection research is to be able to determine, for every algorithm proposed, the amount of security it adds. It is conjectured that no algorithm will protect a program for an indefinite length of time. The security metric we instead would like is *time-to-crack* — i.e. how much longer will it take a cracker to get to the asset we want to protect after a particular algorithm has been applied, compared to the amount of time he would need to crack an unprotected program? Unfortunately, such a metric would require a model of cracker behavior and abilities, something we don't have. Even given such a model, there is such a wide range of cracker abilities that such a model would likely be useless for practical purposes.

Even though time-to-crack has remained an illusive metric, it's been observed in practice that the more powerful a protection algorithm is, the more performance penalty it incurs. The literature is replete with algorithms which incur anywhere from a factor 2 to a factor 2000 slowdown. These papers still are not able to weigh this significant performance overhead against some reasonable metric of the security the algorithm affords.

In this paper we will take a step towards a *relational* model of software protection metrics. We won't pretend we can say "algorithm A slows down an attacker by 10% while adding a performance overhead of 30%". Instead, our goal is to be able to say "Algorithm A is better than algorithm B." In particular, we start by identifying some protection primitives that common algorithms make use of in order to confuse or slow down an attacker. We will then say "Algorithms A and B both make use of primitive P. If we adjust the parameters of A and B so that they perform the same number of P operations, then they ought to afford the same level of protection. If A has lower performance penalty than B we say that A is better than B with respect to P."

A consequence of this strategy is that there may be many cases where two algorithms A and B are not comparable, simply because they make use of completely different kinds of protection primitives. If we *still* want to compare A and B we are going to have to analyze the primitives they use to determine which ones afford the most protection. That will sometimes be feasible, in particular when we expect crackers to employ a particular type of attack. So, for example, we might be able to say "Algorithm A makes use of primitive P_1 and algorithm B makes use of primitive P_2. P_1 makes the attack we're expecting harder to perform than P_2. A and B have similar performance overhead. Thus, A is better than B."

So, in order to embark on building a relational software protection metric we need to do the following:

1. settle on a set of interesting protection algorithms,
2. examine their implementation to identify the basic protection primitives they employ,
3. settle on simple benchmarks,
4. implement the algorithms and apply them to the benchmarks,
5. fix the parameters of the algorithms so that they afford the same level of protection (i.e. execute the same number of protection primitives), and
6. run performance evaluations.

Ideally, we would be able to pick benchmarks that are common in the literature, such as the SPEC benchmarks, and, ideally, we'd have complete implementations of the protection algorithms that can be applied to these benchmarks. Unfortunately, this is neither reasonable nor, ultimately, desirable. Software protection algorithms are notoriously difficult to implement and there exist no freely available reference implementations. Also, the actual performance overhead that

these algorithms incur depend heavily on *how* and *where* in the code they are applied. An algorithm which "happens" to add a heavy-weight protection primitive in the middle of a tight loop will incur a significant overhead, an overhead that another implementation might not. For these reasons, we've chosen to work on trivial, synthetic, benchmarks and to hand-apply the protection algorithms to these benchmarks. The result is that we're measuring the relative performance impact of the basic protection primitives rather than of particular algorithm implementations.

In this paper we are going to consider three algorithms. These are the Horne's algorithm, the Cappaert's algorithm, and the Aucsmith's algorithm. They make use of the following primitives:

- code swap
- hash functions
- xor blocks
- encryption/decryption blocks

The remainder of this paper is organized as follows. Section 2 introduces the obfuscation and tamper-proofing algorithms that we are considering for analysis and discuss implementation issues. Equivalence metrics for obfuscation and tamperproofing algorithms are discussed in section 3. Section 4 discusses the experimental setup and presents the results of our study.

2. SOFTWARE PROTECTION ALGORITHMS

In this paper, we will talk about two main categories of software protection algorithms- tamperproofing and obfuscation. Software tamperproofing is a way to ensure that the program *executes as intended*, even when an adversary tries to change the execution [4]. Obfuscation means to transform a program p into p_i that is functionally equivalent but hard to comprehend and extract logic from. Here, the information that we want to protect may be a new algorithm, a cryptographic key or a license check that the adversary wants to remove. Based on its approach, an obfuscator could be classified as static obfuscator or dynamic obfuscator.

A static obfuscator applies code transformations to the program prior to its execution. This makes it difficult for an adversary to collect information about the program statically, i.e. without running it. Dynamic obfuscators transform programs continuously at runtime, keeping them in constant flux. Dynamic obfuscation tries to counter dynamic attacks by making the code and the execution path change as the program runs. In this paper, we focus on dynamic obfuscation algorithms. These have been discussed in [4, 5]. A dynamic obfuscator typically runs in two phases. In the first phase, at compile time, obfuscators initialize the program and then add a runtime code transformer. In the second phase, at run-time, this transformer is invoked with the correct arguments. As a result, a dynamic obfuscator turns a normal program into a self-modifying one.

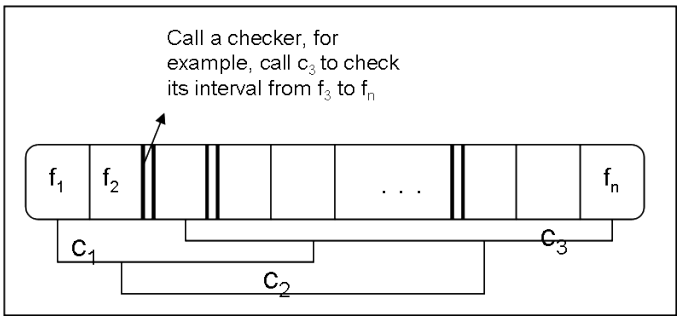

Fig. 1. Horne's Algorithm

2.1. Horne's Algorithm

A technique for software tamperproofing is called introspection, which means the program is augmented to compute a hash over a code region to compare to an expected value. Horne's algorithm is an example of introspection [6, 7]. It uses a number of checkers to check whether the program has been tampered with. Each checker is responsible for a set of functions from the program, referred to as intervals. The checkers hash the value of the interval it is responsible for, and check for tampering. This is shown in figure 1. A disadvantage with hashing is that the hash values are large integer constants that are uncommon in regular programs, and this may not be stealthy enough. Horne's algorithm uses a very clever way of hiding the constants. The idea is to construct the hash function in such a way that, unless the code has been hacked, the function always hashes to zero. This yields much more natural code:

```
h = hash(start, end);
if (h) abort ();

uint32 hash(addr_t start, addr_t end, uint32 C){
  uint32 h = 0;
  while (start < end) {
    h = C * (*start + h);
    start ++;
    }
  return h;
}
```

A code segment of our implementation of Horne's Algorithm is shown below. Here, the `asm` intructions serve as a synthetic benchmark, and these have been used in all our implementations. Here, func1() first calls func2(), which would eventually call func3(). And interval1START and interval1END are the starting and ending address of func2(). intervalK is a constant 3. If the hash value is not 0, the program will exit.

```
void func1() {
 asm("push %ax");
 asm("inc %ax");
 asm("inc %ax");
```

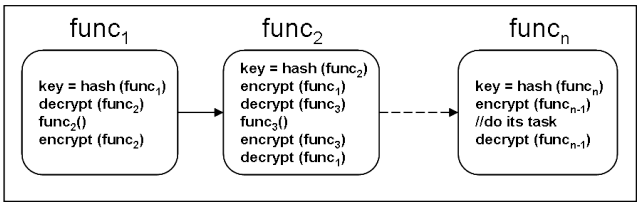

Fig. 2. Cappaert's Algorithm

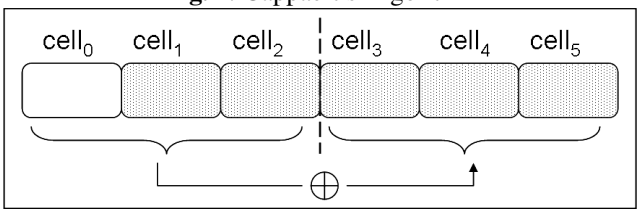

Fig. 3. Aucsmith's Algorithm

```
      ...
 asm("inc %ax");
 asm("pop %ax");
   if (hash(interval1START,interval1END,
    intervalK))
    {exit();}
   func2();
}
```

2.2. Cappaert's Algorithm

A common technique for dynamic obfuscation is through *encryption*. The basic idea is simple: encrypt the program at obfuscation time, store the key in the executable, decrypt and execute at runtime. In this technique, where we store the key in the executable is more important than the strength of the encryption algorithm itself. Cappaert's algorithm [3] is a good example of this. To begin with, the whole program except the main function is encrypted. When a function call is made, the function is decrypted, executed, and re-encrypted at the time of return. However, this would leave the current call chain in clear. To avoid this, the first thing that a function has to do once it has been called is to encrypt the routine that called it. In this way, Cappaert's algorithm makes sure that at any point there are at most two functions in cleartext. Figure 2 depicts how the algorithm works.

2.3. Aucsmith's Algorithm

Another technique for dynamic obfuscation is through *code splitting*. Obfuscators *move code around* to alter the control flow at run-time. Aucsmith's algorithm [1, 2] is such an example. It is also the first known dynamic obfuscation algorithm, which was published in 1996. Aucsmith's algorithm consists of three steps. First, the algorithm splits the program into pieces, cyclically moves them around, and xors them with each other. Second, the algorithm uses simple encryption to keep fewer pieces of the program in the clear. Finally, the algorithm builds up a network of protected routines checking each other. Figure 3 gives an overview of this algorithm.

The sample code that we implemented for Aucsmith's Algorithm is shown below. The section of code of *firsttime* could be executed at obfuscation time, it uses *xor* and *swap* to set up the initial configuration. *ALIGN* is a macro that makes sure that every cell is the same size. The construct &&cell0 is a *gcc* extension to C called *labels-as-values*. It takes the address of a local label that can be stored in a variable. We use this construct here to make the jump-array next. After executing *cell0*, it *xor* the left half cells to the right half cells and then jump to the corresponding cell in the right half.

```
if (firsttime) {
   //Initial Configuration
   xor(&&cell5,&&cell2,CELLSIZE);
   xor(&&cell0,&&cell3,CELLSIZE);
   swap(&&cell1,&&cell4,CELLSIZE);
}
   char* next[] ={&&cell0,&&cell1,&&cell2,
 &&cell3,&&cell4,&&cell5};
goto *next[0];
align0: ALIGN
cell0:
        printf("cell0\n");
        xor(&&cell0,&&cell3,3*CELLSIZE);
        goto *next[3];
```

For each of our algorithms, we had a common setup in which a generator converted an original input file into a protected output file based upon a set of input parameters and the algorithm. This protected output file was compiled, executed and profiled. Table 1 depicts the input parameters for each of the algorithms.

Table 1. Metrics

Algorithms	Input Parameters
Horne	#checkers, progsz
Cappaert	#fns, progsz
Aucsmith	#cells, cellsz

3. EQUIVALENCE METRICS

To be able to compare the performance of these algorithms, there is a need to establish equivalence in terms of security. Possible metrics include the amount of data that is touched, the amount of data that is protected by the algorithm (i.e. amount of data not in the clear), or the number of operations carried out. We discuss these ideas in this section. Table 2 shows the values corresponding to each of these algorithms for these metrics.

3.1. Number of Operations

As shown in the table, Cappaert's algorithm has approximately 5 operations per function. These include the methods to obtain the key, and the encryption and decryption routines. The hash routine in the Horne's algorithm and the xor block in

Table 2. Metrics

	Horne	Cappaert	Aucsmith
#reads	$progsz \times \frac{\#checkers}{2}$	$5 \times progsz - \frac{4 \times progsz}{\#fns}$	$2 \times \#cells(\frac{\#cells \times cellsz}{2})$
#writes	0	$4 \times progsz - \frac{4 \times progsz}{\#fns}$	$\#cells(\frac{\#cells \times cellsz}{2})$
#operations	$\#checkers$	$5 \times \#fns - 4$	$\#cells$
data not in the clear	0	$(\#fns - 1) \times \frac{progsz}{\#fns}$	$(\#cells - 1) \times cellsz$

the Aucsmith's algorithm are the other operations considered. All these operations have the same cost, hence equivalence can easily be achieved by setting number of checkers to the number of cells, and the number of functions to approximately one-fifth of the number of cells. All three algorithms can be compared on the basis of this metric.

3.2. Data touched

The amount of data touched can be defined as the number of memory reads plus the number of memory writes. These have been shown in the table. Now, equivalence can be easily established by fixing a program size, adding the reads and writes and equating them. For Horne's and Aucsmith's algorithms, this is achieved by setting number of checkers to the number of cells. For the Cappaert's algorithm, as the number of functions increases, the choices to establish equivalence become very limited. Only Horne's anf Aucsmith's algorithms can be compared based upon this metric.

3.3. Data not in the clear

Another possible metric is the amount of data that is not in the clear, i.e. the amount of data that has been protected from the adversary. Note that for the Horne's algorithm, this is 0. Thus, by setting the number of functions to be equal to the number of cells, we can use this metric to compare the Aucsmith's and the Cappaert's algorithms.

4. EXPERIMENTAL SETUP AND RESULTS

We carried out our profiling on an Intel dual-core, 1.86 GHz processor with Fedora Core 9, with a cache size of 2048 KB. To configure OProfile for this setup to account for cache misses, we used the LLC_MISSES event [8].

4.1. Observations

Figures 4 - 6 show the results that we obtained. It can be noted that our implementation of Aucsmith's algorithm was slower. This was expected as Aucsmith's algorithm moves a lot of code around as opposed to the Horne's algorithm which hashes across intervals and checks for tamperproofing. When compared on the basis of number of operations, Cappaert's algorithm was quite fast, as the desired security could be obtained only by using one-fifth of functions than the number of cells or checkers. This trend is also observed when we compare Aucsmith and Cappaert on the basis of data not in the clear. Aucsmith is much slower due to poor caching involved and because of jump statements. Cappaert's algorithm tends to be faster due to a significant amount of cache reuse across functions.

5. CONCLUSIONS AND FUTURE WORK

Section 4 presented our performance results based on two possible metrics. Although Aucsmith's algorithm appears to be slow based on these metrics, it is important to note that these metrics may not be sufficient to determine and compare the performance of the algorithms. Future work includes exploring other metrics that would capture the notion of security more naturally. Also, it is important to compare the same set of algorithms based on different metrics, leading to better insight into relative performance. Authors are already exploring some of these ideas.

6. REFERENCES

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

[2] D. Aucsmith and G. Graunke. Tamper resistant methods and apparatus. *United States Patent 5892899, Assigned to Intel Corporation (Santa Clara, CA)*, Apr. 1999.

[3] J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel. Self-encrypting code to protect against analysis and tampering. *In 1st Benelux Workshop on Information and System Security*, 2006.

[4] C. Collberg, G. Myles, and J. Nagra. *Surreptitious Software*. 2008.

[5] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.

[6] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic selfchecking techniques for improved tamper

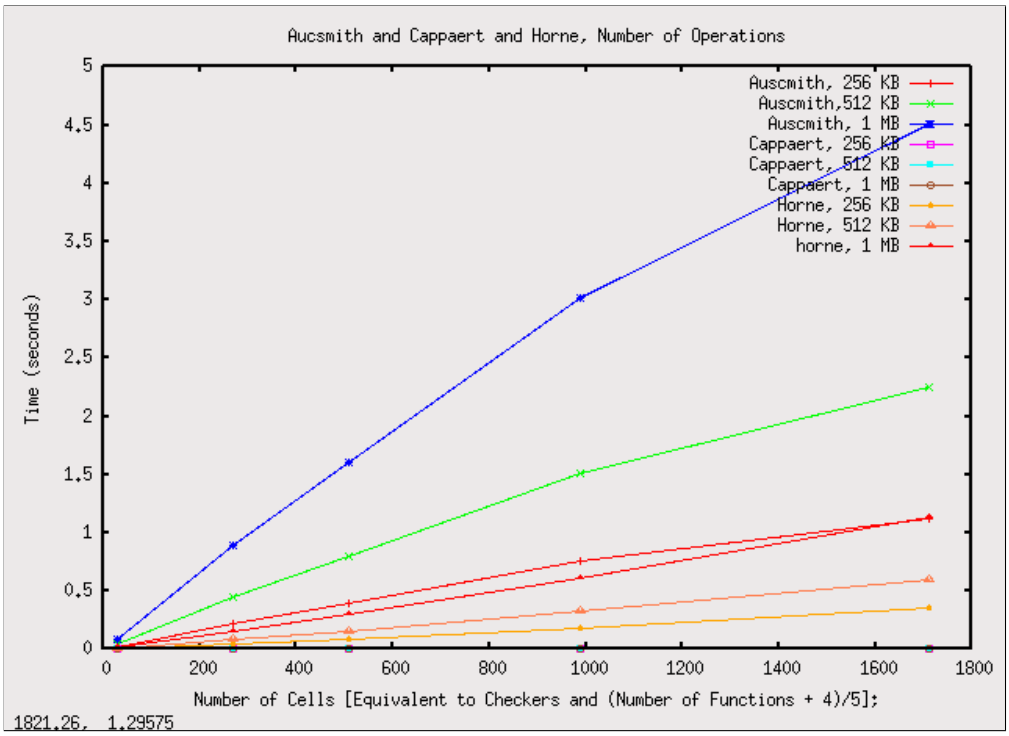

Fig. 4. Timing based on Number of Operations

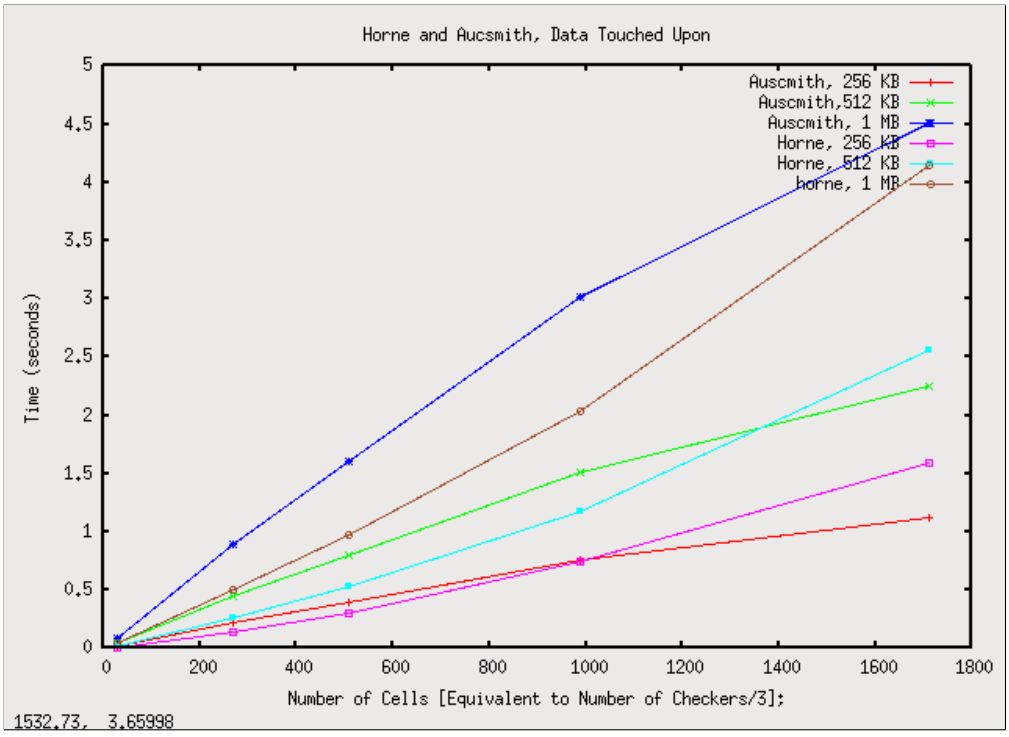

Fig. 5. Timing based on Data touched

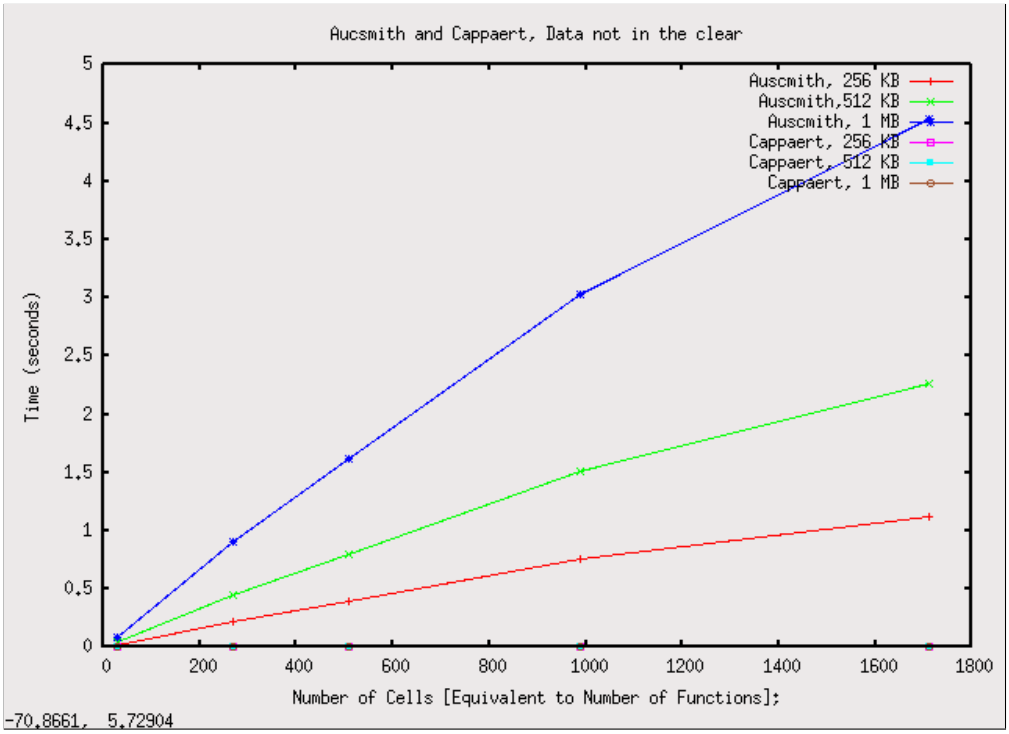

Fig. 6. Timing based on Data Not in the Clear

resistance. *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001, Philadelphia, PA, USA*, Nov. 2001.

[7] W. G. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Software selfchecking systems and methods. *United States Application 20030023856, Assigned to InterTrust Technologies Corporation*, Jan. 2003.

[8] IA-32 Intel Architecture Software Developers Manual. *Intel Corporation*, 2001.