A Metric-Based Scheme for Evaluating Tamper Resistant Software Systems

Gideon Myles, Hongxia Jin

A Metric-Based Scheme for Evaluating Tamper Resistant Software Systems

Gideon Myles[1] ⋆ and Hongxia Jin[2]

[1] Novak Druce + Quigg LLP, San Francisco, CA
[2] IBM Almaden Research Center, San Jose, CA

Abstract. The increase use of software tamper resistance techniques to protect software against undesired attacks comes an increased need to understand more about the strength of these tamper resistance techniques. Currently the understanding is rather general. In this paper we propose a new software tamper resistance evaluation technique. Our main contribution is to identify a set of issues that a tamper resistant system must deal with and show why these issues must be dealt with in order to secure a software system. Using the identified issues as criteria, we can measure the actual protection capability of a TRS system implementation and provide guidance on potential improvements on the implementation. We can also enable developers to compare the protection strength between differently implemented tamper resistance systems. While the set of criteria we identified in this paper is by no means complete, our framework allows easy extension of adding new criteria in future.

keywords: Software Tamper Resistance, Evaluation, Metrics

1 Introduction

Tamper resistant software system is increasingly needed to protect copyrighted materials. Software tamper resistance technique usually consists of two components: *tamper detection* and *tamper response.* The first component, tamper detection, is responsible for detecting undesired changes to the program or environment. For example, an adversary may actually alter bytes in the program to circumvent a license check or he may run the program under a debugger to observe how a protection mechanism works. In response to a tamper event, the tamper response component takes action. This can range from fixing the altered code or degrading the performance of the program to causing the program to terminate. This is also commonly referred to as software tamper proofing.

A variety of tamper resistance techniques have been proposed. One of the first publications in this area was by Aucsmith [2], which provides protection by using the idea of interlocking trust. This is accomplished by verifying the sum of the hashes of all previously executed blocks to ensure they were executed correctly and in the proper order. Another technique was proposed by Chang and Atallah [3] and establishes a check and guard system through a network of guards. Each guard is responsible for monitoring or repairing a section of code. Horne et al. [6] proposed a similar technique based on testers and correctors. A third approach to tamper resistance, oblivious hashing, was proposed by Chen et al. [4]. With oblivious hashing it is possible to compute the hash value of the actual execution instead of just static code. Additional tamper resistance techniques have been proposed by Mambo et al. [8], Jin et al. [7], and Dedic et al. [5]. In this paper we use the term software tamper resistance to encompass a broader range of software protection techniques. We are interested in those techniques that inhibit an adversary's ability to understand and alter the program.

As can be seen, quite a bit of work has been done in the software tamper resistance space; however, there is little work done on evaluating their strength. No quantitative method currently exists that makes it possible to really say something meaningful about the strength of a tamper resistance algorithm, let alone the strength of a particular implementation of that algorithm. Indeed, it is very difficult to make comparisons between two proposed algorithms.

⋆ This work was done when the author was at IBM Almaden Research Center.

In this paper we propose a TRS system evaluation method which begins to address these important issues. The evaluation method provides developers with a way to quantitatively evaluate the strength of a particular implementation of their TRS system through the use of one or more numeric ratings. In general, the technique works by breaking the desired rating down into a set of metrics that are relevant to the specific measurement. For each metric, we calculate a score. Optionally these metric scores can also be combined into an overall score for the rating. The calculation of a score gives the developer a concrete idea as to the strength of his implementation. Furthermore it provides a common base to compare the strength of different TRS systems.

2 Metric-Based Evaluation

Whether a developer is consciously aware of it or not, he most likely has a set of questions in mind that guide the development and implementation of the TRS system. These are questions like "Is essential functionality and data protected," "Is the detection code stealthy," and "Can we detect different types of debuggers." By asking these questions the developer is attempting to "evaluate" the protection capabilities of the TRS system. While the developer's evaluation in this scenario is rather informal, we can use the same type of questions to formalize a quantitative evaluation method.

In general, the TRS system evaluation is comprised of three steps. First, we break the desired rating down into a set of metrics that are relevant to the specific measurement. Then for each metric we calculate a score. Finally, we can derive a overall score for the rating by combining the individual metric scores or simply using the minimum of each individual score.

One of the unique aspects of this process is that we use questions to guide the evaluation process. In essence each metric is based on a guiding question like "is essential data and functionality protected." We phrase each question such that for the ideal TRS system the answer would be "yes." To answer the question and assign a quantitative value to the metric we construct an appropriate model of the protection system. For example, we may be able to answer the question "is essential data and functionality protected," by building a graphical representation of the relationship between the functions in the program.

Using the metric-based TRS system evaluation method, we have devised four categories of TRS system evaluation ratings: *protection coverage rating*, *system complexity rating*, *auxiliary protection rating*, and *overall system protection rating*. We believe that these ratings provide a more comprehensive evaluation method for tamper resistant implementations than any of the previous work in this space.

2.1 Protection Coverage Rating

The protection coverage rating (PCR) evaluates the degree to which the program is covered by the protection mechanism(s). It is important to note that the PCR does not (and should not) say anything about the quality of the protection or how easily it can be subverted. The idea is to convey a sense of the distribution of the protection mechanisms and how they overlap.

To illustrate how the protection coverage rating is calculated we will rely on a running example in which the factorial program shown below is protected using a very simple implementation of the Branch-Based tamper resistance technique [7].

```
void main(int argc, char *argv[]){
   int x = atoi(argv[1]);
   printf("%d! = %d\n", x, factorial(x));
}

int factorial(int x){
   if(x == 1)
      return x;
   return(x * factorial(x-1));
}
```

The Branch-Based tamper resistance technique converts branch instructions to calls to a branch function. The branch function then performs an integrity check of the program and calculates the proper branch target instruction. Below illustrates what the factorial program could look like after the Branch-Based tamper resistance protection has been applied.

```
long key = seed;

void main(int argc, char *argv[]){
   int x = branchFunction1(argv[1]);
   branchFunction2("%d! = %d\n", x,
      branchFunction3(x));
}

int factorial(int x){
   if(x == 1)
      return x;
   return(x * factorial(x-1));
}

void branchFunction1(void *x){
   //perform anti-debugging check
   //evolve the key
   //compute return address
   return;
}

void branchFunction2(void *x){
   //compute checksum over main and factorial
   //evolve the key
   //compute return address
   return;
}

void branchFunction3(void *x){
   //compute checksum over factorial
   //evolve the key
   //compute return address
   return;
}
```

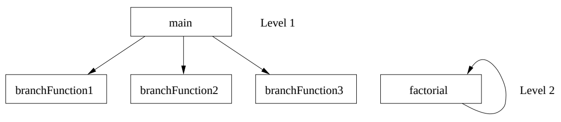


Fig. 1. The function-instance graph for the factorial program

Protection Coverage Model In order to calculate a protection coverage rating we need a method of modeling the protection capabilities of the TRS system. We do this by building two different graphs both of which are based on the call graph for the program. The first graph we construct is the *function-instance graph.* Using a depth first traversal we transform the call graph into the function-instance graph. This construction is illustrated in Figure 1 for our protected program.

The second graph is the *protection coverage graph.* Construction of this graph requires that we first augment the call graph by adding a block for each element of the program that requires protection but is not a function, for instance a memory segment or a secret key. Then to represent protection mechanisms like obfuscation or encryption we insert another place holder block. When multiple obfuscation techniques are used, we insert multiple place holder blocks. Finally, we add a directed edge between two blocks A and B when A provides protection for B. Following this procedure, we arrive at the protection coverage graph in Figure 2.

Protection Coverage Metrics The protection coverage metrics are guided by questions that reveal the full scope of the TRS system's defense network. We have identified six questions that we feel provide a comprehensive view of the system. Using the protection coverage model we are able to develop a metric and calculate a score for each of the questions below. (Notation used in the metrics can be seen in Table 1.) Below we will show the six criteria together with the rationale behind choosing that criteria.

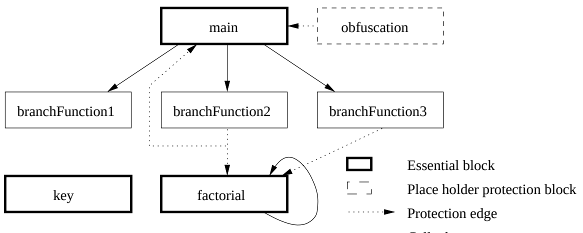


Fig. 2. The protection coverage graph for the factorial program

---

- *Is essential functionality and data protected?* The Essential Coverage Metric (ECM) indicates to the developer whether all of the critical, must be protected elements have in fact been protected. This is a very important measurement for any TRS system.
  - $ECM = \frac{|B_{ep}|}{|B_e|}$
- *Do anti-debugging checks occur throughout the entire execution of the program?* The Anti-Debugging Metric (ADM) gives the developer a sense of how successful the TRS system would be at preventing the execution of the program under a debugger. This criteria is important because a low score indicates that at least part of the program can be executed and observed by an attacker, which could result in the attacker discovering secret information.
  - $ADM = \frac{\sum_{l \in L} \frac{|out_{cp}(l)|}{|out_c(l)|}}{|L|}$
- *Is each integrity verification method invoked at multiple places throughout the program?* Suppose the TRS system has an integrity check which performs a checksum of the program, but that integrity check is only invoked a single time when the program starts. Once the program has started executing, the attacker can make any changes he wishes and they will not be detected. It is important to measure the degree to which the program is vulnerable to scenarios like this. This degree can be measured using the Multiple Invocation Metric (MIM).
  - $MIM = \frac{\sum_{b \in B_{iv}} \frac{|in_c(b)|}{|E_c|}}{|B_{iv}|}$
- *Is there cross-checking of integrity verification methods?* That is, are the integrity verification methods themselves protected? The Cross-Check Metric (CCM) is important because if the integrity verification methods are left vulnerable then an attacker can remove them and the remainder of the program is left vulnerable.
  - $CCM = \frac{\sum_{b \in B_{iv}} \frac{|in_p(b)|}{|E_p|}}{|B_{iv}|}$
- *Are the protection methods overlapping?* When a sensitive section of code is protected using only one means of protection all the attacker has to do is defeat that one mechanism. By increasing the protection on that section, the amount of work the attacker has to do is also increased. So this measurement is also important to the security of a TRS system. The Protection Overlap Metric (POM) lets the developer know whether more layers of protection need to be added.
  - $POM = \frac{\sum_{b \in B_f} \frac{|in_p(b)|}{|E_p|}}{|B_f|}$
- *Are there multiple methods of protecting the integrity of the program?* Again suppose the TRS system has an integrity check which performs a checksum over the program. If this is the only integrity check used to verify the integrity of the program, the attacker only has one protection mechanism to analyze. Obviously, by increasing the number of protection mechanisms, we increase the amount of work the attacker has to do thereby strengthening the TRS system. The Multiple Protection Metric (MPM) indicates to the developer if greater diversity is need.
  - $MPM = \frac{|B_p|}{|B|}$

To construct the overall protection coverage rating (PCR) we combine the individual metric scores by multiplying each component by a constant representing that ratings importance and adding the values together. The sum of the constants is 1.

Protection Coverage Graph Notation

B	set of all blocks.
B_e	set of essential blocks.
B_{ep}	set of essential blocks which are protected.
E_c	set of call edges in the graph.
E_p	set of protection edges in the graph.
B_p	set of all protection blocks.
B_{iv}	set of integrity verification protection blocks.
B_{ad}	set of anti-debug protection blocks.
B_{pp}	set of place holder protection blocks.
B_f	set of blocks which are not protection blocks.
$in_c(b)$	incoming call edges for block b.
$in_p(b)$	incoming protection edges for block b.

Function-Instance Graph Notation

L	set of levels in the function-instance graph.
$out_c(l)$	out going call edges for the block(s) on level l.
$out_{cp}(l)$	out going call edges for the block(s) on level l whose sink is a protection block.

Table 1. The notation used in the protection coverage metrics.

– $PCR = (a)ECM + (b)ADM + (c)MIM + (d)CCM + (e)POM + (f)MPM$ where $a + ... + f = 1$

For example, in general ECM (Essential Coverage Metric) and POM (Protection Overlapping Metric) seem to be relatively more important than other metrics. Therefore it makes sense to give more weights on these two metrics than others. However, for different purposed TRS system, it is possible that different weights may need to be assigned for the same metric. Another more general option to obtain the overall rating is to simply choose the minimum value among the set of metrics.

When we apply the metrics to our example we get:

$$ECM = \frac{2}{3} = .67$$
$$ADM = \frac{\frac{1}{3}+\frac{0}{1}}{2} = \frac{1}{6} = .17$$
$$MIM = \frac{\frac{1}{4}+\frac{1}{4}+\frac{1}{4}}{3} = \frac{1}{4} = .25$$
$$CCM = 0$$
$$POM = \frac{\frac{2}{4}+\frac{2}{4}+\frac{0}{4}}{3} = \frac{1}{3} = .33$$
$$MPM = \frac{4}{7} = .57$$

Note that regardless of the overall rating value and how one calculates the overall rating, each metric alone provides some value in the evaluation. For example, two TRS systems can be evaluated against each of these metrics to see which one is stronger for the protection. Moreover, those metrics can help guide developers improve the security of their software.

In the above example, based on the calculated results, a developer could see that there are aspects of the protection coverage that need further improvement. First, a cross-check metric score of zero reveals that the TRS system's integrity verification methods are completely vulnerable to attack. Second, the anti-debugging check metric score is very low which indicates the anti-debug checks are not very well distributed in the program. This means that certain portions of the program could be executed under a debugger without being detected which could ultimately lead to the attacker discovering sensitive information.

On the other hand, for a same-functional software, if the developers come up with another design of the system, they can similarly use these metrics to see the protection capability for that system. Our evaluation method enables the developers to compare the strength of the differently implemented systems and then make design choice accordingly.

It is also worthy mention that because our evaluation method is based on sets of metrics, it is easily extensible. As protection mechanisms evolve and new evaluation method are developed they can easily be incorporated into the list.

2.2 System Complexity Rating

The system complexity rating (SCR) is used to evaluate the level of difficulty associated with understanding and disabling the protection mechanisms. The focus of this rating is on the topological configuration of the system and the strength of the atomic protection mechanisms that make up the system. The rating is calculated by first recursively breaking down the TRS system's compound protection mechanisms until the atomic protection mechanisms are isolated. The atomic protection mechanisms are then evaluated using the various metrics and the calculated scores are plugged into the graphical model and combined based on the topological configuration.

System Complexity Model Part of being able to properly evaluate a TRS system is being able to properly model its behavior. The modeling approach we use is partially driven by an important system complexity question: "Is it impossible to disable the TRS system in stages?" This is motivated by the belief that a tightly linked set of protection mechanisms is harder to disable than a set of disjoint mechanisms because more analysis required. The system complexity model enables us to answer this question by transforming the tamper resistance capabilities into a graph. We accomplish this as follows:

1. Each code block in the program becomes a node in our graph. A code block can have any level of granularity and the particular composition of a code block will depend of the tamper resistance algorithm.
2. If a code block c_i provides integrity verification for another code block c_j, a directed edge is added from c_i to c_j.
3. If a code block c_i triggers the anti-debugging protection provided by code block c_j, a directed edged is added from c_j to c_i.
4. If a code block c_i repairs another code block c_j, a directed edge is added from c_i to c_j.
5. If a code block c_i contains concealment protection, a new block representing the protection mechanism is added to the graph and a directed edged from the new block to c_i is added.
6. If a code block c_i provides protection for more than one block, a super block is added encompassing the protected blocks. A directed edge is then added from c_i to the super block.

Figure 3 illustrates the graphical model of the protected factorial program that is constructed by following these steps.

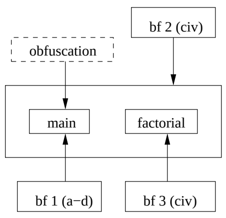


Fig. 3. Graphical model used to calculate the complexity rating for the factorial program protected using the Branch-Based Tamper Resistance technique.

The graph topology model enables us to analyze and evaluate the way tamper resistance is incorporated into the existing software, while providing a common base for comparing tamper resistance algorithms. The main advantage of this model is that we can break the TRS system down into its atomic protection mechanisms and then associate a complexity score indicating how difficult it would be to defeat that particular mechanism. Based on the topology of the graph we can combine the atomic protection scores to determine the overall system complexity rating for the TRS system.

Protection Mechanism Metrics Calculating the complexity metric score of an atomic protection mechanism is still a rather open question. In this section we propose a variety of metrics which begin to address the question, but that by no means provide the complete answer. Our goal is to lay a foundation of metrics that can be built upon as protection mechanisms evolve. Under our method, the evaluation of protection mechanisms is first based on the particular type of the mechanism. It is then guided by type specific questions. We focus on three categories of protection mechanisms: detection, response, and concealment. Each of these categories have unique characteristics that require evaluation metrics specific to the category. Like the protection coverage metrics these metrics are guided by questions. In this case, the questions reveal the level of difficulty an attacker will have in identifying and understanding the protection mechanisms.

Tamper Detection Metrics A tamper detection mechanism is any section of code which was designed to detect changes in the program or environment. This could be a change in the actual instruction sequence or a change in the execution environment such as the use of a debugger. Below are some questions that should definitely be considered when calculating the complexity rating of a tamper detection mechanism, however, it is possible that other questions and therefore metrics could also be incorporated.

– *Is the detection code stealthy?* Ideally the detection code would be similar to the code around it. One possible way to measure this is to consider the instruction sequences in the original program, the tamper detection mechanism, and the tamper resistant version of the program.

$$\begin{cases} 1, \text{ if } \frac{|\text{inst seq } \in DM \text{ but } \notin P|}{|\text{inst seq } \in P_{TRS}|} < \delta \\ 0, \text{ otherwise} \end{cases}$$

– *Is detection widely separated from response in space?* This could be measured by counting the number of instructions between the detection and response mechanisms.

$$\begin{cases} 1, \text{ if } |\text{insts between detection and response}| > \delta \\ 0, \text{ otherwise} \end{cases}$$

– *Is detection separated in time from a response which induces program failure?* There are a couple different ways this could be measured. One would be to measure the number of seconds between detection and response.

$$\begin{cases} 1, \text{ if } |\text{secs between detection and response}| > \delta \\ 0, \text{ otherwise} \end{cases}$$

Another would be to use a call graph to model the time between detection and response.

$$\begin{cases} 1, \text{ if } |\text{calls between detection and response}| > \delta \\ 0, \text{ otherwise} \end{cases}$$

Tamper Response Metrics A tamper response mechanism is any section of code which has been designed to respond to an attack on the program. The response could be triggered by a change in the instruction sequence or the detection that the program is being run in a debugger. The response action taken can vary. The mechanism could alter some portion of the program which eventually leads to program failure or it could repair a section of code which has been altered. Below are some questions that should be considered when calculating the complexity rating of a response mechanism. As with the tamper detection complexity, this is not an exhaustive list, these are simply the questions that are common to all tamper response mechanisms.

– *Is the response code stealthy?* Ideally the response code is similar to the code around it. Response code will often rely on self-modifying code to either cause program failure or to repair a section of code. This type of code is not routinely used in programs, so it is crucial this code is not easily detected by the attacker.

$$\begin{cases} 1, \text{ if } \frac{|\text{inst seq } \in RM \text{ but } \notin P|}{|\text{inst seq that occur in } P_{TRS}|} < \delta \\ 0, \text{ otherwise} \end{cases}$$

– *Does a program that has been tampered with eventually fail or repair itself?* In the event of tampering, it is critical that some type of response occurs. One way to evaluate this is to use the program control flow graph to determine if the failure inducing or repair code is on a possible execution path.

$$\begin{cases} 1, \text{ if code is on possible future path} \\ 0, \text{ otherwise} \end{cases}$$

– *Does a program that has been tampered with, initially proceed seemingly normally so as to hide the location of the response mechanism?* This is of particular concern for failure inducing response mechanisms. If the failure occurs immediately after the response, it will be very easy for an attacker to identify the response mechanism.

$$\begin{cases} 1, \text{ if } |\text{secs between response and failure}| > \delta \\ 0, \text{ otherwise} \end{cases}$$

Concealment Metrics A concealment mechanism is any protection technique which is used to disguise the true purpose of a section of code. This could come in the form of code obfuscation or even encryption.

– *Is the concealment code stealthy?* Ideally even if a technique like obfuscation is used, the obfuscated code should still blend in the with code around it. That is, we do not want to alert the attacker to the fact that we used obfuscation because it indicates that the section of code is important.

$$\begin{cases} 1, \text{ if } \frac{|\text{inst seq } \in CM \text{ but } \notin P|}{|\text{inst seq that occur in } P_{TRS}|} < \delta \\ 0, \text{ otherwise} \end{cases}$$

– *Can the protection thwart static disassembly or decompilation of the operational code and its basic structure?* By preventing proper disassembly or decompilation the attacker is forced to revert to the often more difficult dynamic analysis techniques. A possible method of evaluating this protection is to compare the disassembly code of the original program and the protected program. Additionally, it is often obvious when a program has been disassembled incorrectly because of the instruction sequences generated.

$$\begin{cases} 1, \text{ if } sim(dis(P), dis(P_{TRS})) < \delta \\ 0, \text{ otherwise} \end{cases}$$

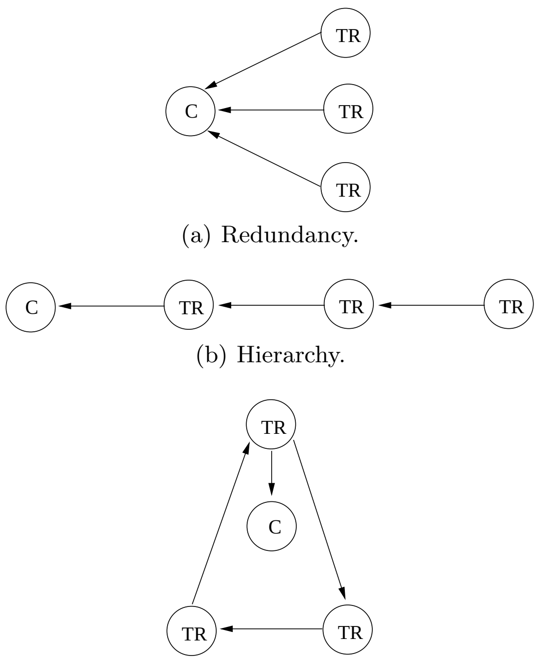
– *If encryption is used, are any decryption keys hidden?* When encryption is used to protect all or part of the program, the code has to be decrypted in order to be executed. This decryption requires a key that is often hidden directly in the software.

It has been suggested that the strength of concealment mechanisms could be evaluated using software complexity metrics [1]. Such evaluation metrics would fit nicely within our framework and further expand the evaluation capabilities.

System Complexity Rating Calculation Using the system complexity model and the system complexity metrics for the atomic protection mechanisms we are able to develop an overall score for the system complexity rating. One of the advantages of the system complexity rating is that even without a complete understanding of the strength of the individual protection mechanisms, through the complexity model we are still able to provide the developer with valuable feedback. As we will see, the topological configuration of the protection mechanisms say a lot about the strength of the TRS system.

There are a variety of different topological arrangements that can be used in designing the TRS system. To illustrate how the system complexity rating can be calculated we investigate three configurations: redundancy, hierarchy, and cluster. In reality a TRS system will be a combination of these configurations, in which case a score for each sub-configuration can be calculated and then combined to form the rating score.

As we mentioned, one of the motivating questions in the system complexity rating is "Is it possible to disable the protection mechanisms in stages?" Because of this we are interested in two different scores: the per-stage complexity rating (PCR) and the total complexity rating (TCR). The PCR is associated with subverting a subset of the protection mechanisms without triggering any detection mechanism. The TCR is the complexity rating associated with disabling the entire TRS system.



(a) Redundancy.

(b) Hierarchy.

(c) Cluster.

Fig. 4. Possible TRS system complexity model configurations.

Redundancy Model In the redundancy configuration tamper resistance mechanisms are incorporated throughout the program without dependencies between the different mechanisms. Figure 4(a) illustrates a possible configuration using redundancy. In this case the TRS system can be subverted in stages. If one of the protection mechanisms is disabled, it will not be detected by any of the others.

In the ideal situation the total complexity rating for a redundancy based TRS system would be the sum of the complexity ratings for the atomic protection mechanisms, $\sum_{i=1}^{n} CR(TR_i)$. However, there is at least one factor that can decrease the overall effectiveness of a TRS system. This factor relates to the similarity of the protection mechanisms used. In the general sense, we have three similarity categories for protection mechanisms: duplicated, derived, and unique.

In the duplicated scenario, one protection mechanism is used in multiple places. While this does make it possible to protect different parts of the program, the extra work required to disable more than one of these mechanisms is negligible. This leads to the per-stage and total complexity ratings being equal:

- PCR = TCR = $CR(TR)$.

A derived classification occurs when one or more protection mechanisms is a derivative of another mechanism in the TRS system. Because the mechanisms are similar, information learned from attacking one can be used to defeat the others. This has the effect of increasing the total complexity rating over the duplicated scenario, but the complexity level is still sub-optimal for the given configuration.

- PCR $= max\{CR(TR_i)|i \in n\}$
- TCR $= max\{CR(TR_i)|i \in n\} + \sum_{(j=1)\backslash i}^{n} CR(TR_j)[1 - sim(TR_i, TR_j)]$ where $sim(TR_i, TR_j) \in [0, 1]$

A variety of different methods have been developed to measure the similarity between sections of code or programs [9]. The similarity measure could be based on one of these ideas or a new measure could be developed specifically for tamper resistance protection mechanisms.

The maximum complexity is achieved when the tamper resistance mechanisms are classified as unique and defeating one does not aid an attacker in defeating any of the others. In this case we achieve the following complexity ratings:

- PCR $= max\{CR(TR_i)|i \in n\}$
- TCR $= \sum_{i=1}^{n} CR(TR_i)$

Of course, the TRS system could be comprised of a mixture of these three categories.

Hierarchy Model The hierarchy configuration consists of n layered protection mechanisms. Each layer provides tamper resistance protection for the mechanism in the lower layer. The innermost layer protects the code block. Figure 4(b) illustrates this configuration. As with the redundancy configuration, a TRS system configured as a hierarchy can be subverted in stages by starting with the outermost tamper resistance mechanism. Theoretically, this configuration is marginally stronger than the redundancy configuration since an attacker has to identify the outermost layer.

The hierarchy configuration can also be classified by duplicated, derived, and unique protection mechanisms. Using these categories we obtain the following complexity ratings:

- Duplicated
  - PCR $= CR(TR)$+(cost to find outermost)
  - TCR $= CR(TR)$+(cost to order units)
- Derived
  - PCR $= CR($outermost $TR)$+(cost to find outermost)
  - TCR $= CR($outermost $TR) + \sum_{j=2}^{n} CR(TR_j)[1 - sim(TR_i, TR_j)]$+(cost to order units)
- Unique
  - PCR $= CR($outermost $TR)$+(cost to find outermost)
  - TCR $= \sum_{i=1}^{n} CR(TR_i)$+(cost to order units)

Cluster Model In the cluster configuration the tamper resistance mechanisms are arranged in a strongly connected digraph topology with some degree of incidence d. The degree of incidence is the in-degree of any particular code block in the graph. It measures the number of protection mechanisms protecting a particular code block. The degree of incidence for the cluster is the minimum degree of incidence within the cluster. Theoretically, the cluster configuration provides the highest level of protection. In this configuration, subverting any of the tamper resistance mechanisms within the cluster requires subverting all of the mechanisms in the cluster simultaneously. Figure 4(c) illustrates a trivial cluster based TRS system which has a degree of incidence $d = 1$.

In order to disable the entire cluster each protection mechanism along with its parent(s) must be subverted prior to a parent mechanism detecting an incidence of tampering. For single threaded programs, it may still be possible to identify a sequence in which a cluster with a degree of incidence $d = 1$ can be subverted. However, identifying this order is more difficult than in the hierarchy configuration. For programs in which code blocks can be executed in parallel, identifying the necessary sequence is even more challenging. Furthermore, as the degree of incidence increases, disabling a single protection mechanism will be detected by several other mechanisms, thus increasing the number of mechanisms that must be subverted simultaneously.

Like the previous two configurations, the cluster configuration can be classified by duplicated, derived, and unique protection mechanisms. Using these classifications we have the following complexity ratings:

– Duplicated
  - PCR = TCR = $(2 - \frac{1}{d})(CR(TR))$+(cost to identify all units in cluster)
– Derived
  - PCR = TCR = $(2 - \frac{1}{d})[max\{CR(TR_i)|i \in n\} + \sum_{(j=1)\backslash 1}^{n} CR(TR_j)(1 - sim(TR_i, TR_j))]$+(cost to identify all units in cluster)
– Unique
  - PCR = TCR = $(2 - \frac{1}{d})[\sum_{i=1}^{n} CR(TR_i)]$+(cost to identify all units in cluster)

2.3 Auxiliary Protection Ratings

The auxiliary protection (AP) rating evaluates the degree to which additional efforts have been made to aid the embedded protection mechanisms. Generally, the protection mechanisms alone are not sufficient to protect a program. This is because there are many aspects of a system which can leak information. For example, strings in a program can often provide an attacker with insight about functionality. Additionally, the protection mechanisms, due to unusual behavior8 like self-modifying code, can reveal themselves to attackers thus requiring the TRS system to disguise the mechanisms or employ misleading code which draws attention away from the real protection code. Again we will use question-guided approaches to obtain the metrics. Below we show two sample measurements only to illustrate the idea.

– *Are common or well-known code sequences avoided?* Many protection systems will leverage off the self cryptographic implementations. Strong attackers will be able to easily recognize the common code sequences without much analysis, thus reducing the effort required to attack the TRS system.

$$\begin{cases} 0, \text{ if } |\text{known code sequence } \in P_{TRS}| > \delta \\ 1, \text{ otherwise} \end{cases}$$

– *Are revealing names, strings, constants, etc. avoided?* Constant values in a program decrease the amount of analysis required by an attacker by providing insight regarding functionality.

$$\begin{cases} 0, \text{ if } |\text{revealing value } \in P_{TRS}| > \delta \\ 1, \text{ otherwise} \end{cases}$$

2.4 Overall System Protection Rating

The overall system protection rating (OSP) is used to evaluate the overall strength of the entire TRS system. This rating can be used to calculate two different values. The first we call the *probability of subversion* which indicates how likely it is that an attacker will be able to circumvent the TRS system. The second value is the *difficulty of subversion.* This value does not tell the developer how much it will cost an attacker to circumvent the system in time, money, or resources, but instead indicates a level of difficulty.

The rating score is still driven by a set of questions:

1. *Is the entire software system protected?*
2. *Is it hard to understand and disable the embedded protection mechanisms?*
3. *Are additional protection efforts being made to aid the embedded protection mechanisms?*

Question 1 corresponds to the protection coverage rating, Question 2 to the system complexity rating, and Question 3 to the auxiliary protection rating. Then to derive the OSP rating we combine the sub-rating scores. The manner in which we combine the sub-ratings determines whether we calculate the probability of subversion or the cost of subversion. In either case, the sub-rating score is multiplied by a constant representing the rating's importance. If we then use multiplication in combining the values we will get the probability of subversion. Using addition will yield the difficulty of subversion.

As one of our future work direction, we would like to expand the OSP evaluation so that it can tell how much it costs to circumvent the system in time, money, or resources. In order to do this, we need to take into consideration what kind of attackers we are dealing with. we would like to be able to identify classes of attackers, based on resources, skills, and attack types, and then map the classes to difficulty levels. This will tell a developer that if they have a TRS system with a certain difficulty of subversion then it can protect against attackers below the corresponding class. This will also tell how many men time it takes to circumvent the system for different classes of attackers. The attacker class may also make the probability of subversion evaluation more accurate, because the probability of subversion can be different for different class of attackers.

3 Conclusion

In this paper we presented a metric-based evaluation method for tamper resistant software system implementations. Our work makes several important contributions. First, it provides what we believe to be the first comprehensive, quantitative method for evaluating the strength of TRS system implementations. Second, the quantitative score not only provides the developer with insight as to the strength of the implementation, it can provide a common base to compare the strength of different TRS systems. Note that it is not critical to verify the validity of each score we obtain in the evaluation. But comparing two scores is sufficient to tell which TRS system is stronger protected. This is especially advantageous for standards-based content protection systems were a guaranteed level of robustness is required. Because most companies are reluctant to release their software to an outside evaluation team for fear of leaking their intellectual property, the robustness guarantee is achieved through the manufacturer's self-certification. This self-certification holds the manufacturer liable in the event of an attack, but it does nothing to truly guarantee the robustness of the system. Our evaluation method could be used to address this issue. A tool based on this method would produce a report that can be publicly shared without leaking the confidential information contained in the software. Finally, because the evaluation method is based on sets of metrics, it is easily extensible. As protection mechanisms evolve and new evaluation method are developed they can easily be incorporated.

There are several directions we want to research further and deeper as future work. Are the metrics sufficiently mature in the sense that they capture the key issues relevant to tamper resistance? What if some of the metrics are contradicting with each other or related to each other? Are there techniques/practices that can result in high ratings of our metrics? This would lead to best practices.

References

1. B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. Program obfuscation: A quantitative approach. In *Proceedings of 3rd ACM Workshop on Quality of Protection*, 2007.
2. D. Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, London, UK, 1996. Springer-Verlag.
3. H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings ACM Workshop on Digital Rights Management (DRM 2001)*, volume 2320 of *Lecture Notes in Compter Science*, pages 160–175. Springer Berling/Heidelberg, 2002.
4. Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Proceedings of 5th Information Hiding Workshop*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414. Springer Berlin/Heidelberg, 2003.
5. N. Dedic, M. Jakubowski, and R. Venkatesan. A graph game model for software tamper protection. In *Proceedings of 9th Information Hiding Workshop*, 2007.
6. B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 141–159, London, UK, 2002. Springer-Verlag.
7. H. Jin, G. Myles, and J. Lotspiech. Towards better software tamper resistance. In *Information Security: 8th International Conference*, volume 3650 of *Lecture Notes in Computer Science*, pages 417–430. Springer-Verlag, 2005.
8. M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Proceedings of 1997 New Security Paradigms Workshop*, pages 23–33. ACM Press, 1998.
9. G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of ACM Symposium on Applied Computing*, pages 314–318, 2005.