Delayed and Controlled Failures in Tamper-Resistant Software

Gang Tan[1], Yuqun Chen[2], and Mariusz H. Jakubowski[2]

[1] Computer Science Department, Boston College. gtan@cs.bc.edu
[2] Microsoft Corporation. {yuqunc,mariuszj}@microsoft.com

Abstract. Tamper-resistant software (TRS) consists of two functional components: tamper detection and tamper response. Although both are equally critical to the effectiveness of a TRS system, past research has focused primarily on the former, while giving little thought to the latter. Not surprisingly, many successful breaks of commercial TRS systems found their first breaches at the relatively naïve tamper-response modules. In this paper, we describe a novel tamper-response system that evades hacker detection by introducing delayed, probabilistic failures in a program. This is accomplished by corrupting the program's internal state at well-chosen locations. Our tamper-response system smoothly blends in with the program and leaves no noticeable traces behind, making it very difficult for a hacker to detect its existence. The paper also presents empirical results to demonstrate the efficacy of our system.

1 Introduction

Software tampering continues to be a major threat to software vendors and consumers: Billions of dollars are lost every year to piracy[3]; tampered software, appearing legitimate to untrained consumers, also threatens their financial security and privacy. As the main countermeasure, the software industry has invested heavily in Tamper-Resistant Software (TRS) with varying degree of success. This paper focuses on a neglected aspect of tamper resistance, namely how the TRS should respond to tampering.

Software tampering is often conducted on a malicious host that is under a hacker's complete control: the hacker is free to monitor the hardware, as well as modify and observe the system software (i.e., OS). On current PC platform, without dedicated hardware support such as provided by NGSCB [6, 17], TRS must rely on software obfuscation to evade detection and defeat hacking attempts [8–11, 19]. Stealth, or the art of hiding code in the host program, is the first and the *primary* defense that most TRS systems deploy against hackers. Ideally, the code pertaining to tamper resistance should be seamlessly intertwined with the host program's code, so that a hacker cannot discover its location(s) by either inspecting the program's code or monitoring its runtime behavior [7].

[3] According to studies [1] by Business Software Alliance (BSA) and International Data Corporation (IDC), the retail value of pirated software globally is 29 billions, 33 billions, and 34 billions, in 2003, 2004, and 2005, respectively.

A TRS system consists of two functional components: *tamper detection* and *tamper response*; each can be made of multiple distinct modules. Both components are equally important to the effectiveness of a TRS system. In practice, however, most R&D work has gone into hiding the tamper-detection code, which verifies the host program's integrity [5, 7, 13]; surprisingly little has been done to improve the stealth of the tamper-response component. Since hackers tend to look for the weakest link to crack the defense perimeter of a TRS system, inadequate tamper-response mechanisms have often become the Achilles' heel of commercial TRS systems [4].

While some TRS systems can be effective if properly applied, software authors have often used only simple or default TRS features. For example, certain dongle- and CD-based copy protections perform just one or a few boolean checks, which may be easily patched out [14]. Thus, it is highly useful to automate the process of separating checks from responses.

In this paper, we describe a novel tamper-response system that evades hacker detection by introducing delayed, probabilistic failures in a program. The main technique is to corrupt certain parts of the host program's internal state at well-chosen locations so that the program either fails or exhibits degraded performance. One can also plug other failure-inducing techniques into our framework; some of them can be found in Section 6. Our tamper-response system smoothly blends in with the program and leaves no noticeable traces behind, making it very difficult for a hacker to detect its existence.

The rest of this paper is organized as follows. We describe some prior art and related work in Section 2. In Section 3, we introduce principles for effective tamper-resistant software. We describe our tamper-response system in Section 4. Implementation details and system evaluation are presented in Section 5. We discuss interesting extensions in Section 6, and conclude in Section 7.

2 Related work

As informal advice, the idea of separating tamper detection from response has long been familiar to programmers of software-protection schemes [4]. The concept of "graceful degradation", or slow decay of a program's functionality after tamper detection, is a closely related technique, which has been widely reported to be used commercially [16]. Software authors typically have not revealed how specific implementations achieve these effects; in general, manual and application-specific techniques have been used. Our work provides systematic, automated methods of separating detection from response in general programs.

Commercial copy protection, licensing, and DRM systems have employed many unpublished techniques, which have been described by hackers on a large number of Internet sites and discussion boards. Such methods have often relied on "security by obscurity," which may be a valid tactic when only limited protection strength is desired or expected, as in the case of certain copy protections.

This work belongs to the general category of tamper-resistance, software obfuscation, and software watermarking. Representative examples in this category

include runtime code encryption and decryption based on a visibility schedule [2]; taxonomies of generic obfuscating transformations and opaque predicates [9–11]; complication of pointer-aliasing and control-flow analysis [8, 19]; and integrity verification of both static program code [5, 13] and dynamic execution traces [7].

Theoretical treatment of obfuscation [3] has revealed that a general obfuscator cannot exist for arbitrary software under a specific model. This shows only the existence of certain contrived programs that cannot be obfuscated against a polynomial-time adversary, and thus does not necessarily block practical solutions. Furthermore, some forms of secret hiding, which include Unix-style password hashing, have been proven secure even in this framework [15, 20]. An earlier, somewhat different model [12] showed that obfuscation is possible in the sense of randomizing memory accesses of certain programs, albeit at a performance cost impractical for typical applications.

3 Tamper-resistant software model and principles

Before describing our system, we first define a simple model of tamper-resistant software and lay out a set of principles to which an effective TRS system must adhere. In the following discussion, we consider a threat model with these participants: software vendors, legitimate users, and software pirates. *Software vendors* produce software, have the source code, and sell software in the form of executable code. *Legitimate users* and *software pirates* buy software (in the form of executable code) from the vendors. Software pirates try to tamper with the software to bypass its copyright-protection system.

In its simplest incarnation, a tamper-resistant software module resides in and protects another software module. The module being protected (or the *host module*) can be an application program, a library (either statically linked or dynamically loaded), an operating system or a device driver. In practice, multiple TRS modules are spread amongst several modules to create a complex web of defenses; in this paper, however, we concentrate on the simplified case of a single host module. This is to simplify the discussion without loss of generality.

The TRS module can be functionally decomposed into two components: tamper detection and tamper response. As the names imply, the former is responsible for detecting whether the host module, including the TRS module itself, has been (or is being) tampered with; the latter generates an appropriate response to either thwart such tampering or render the tampered host module unusable. More specifically,

Detection. We assume one or more detection-code instances exist in the host module. They communicate with the response code via *covert flags*: upon detecting tampering, the detection code sets one or more flags to inform the response module as opposed to calling the latter directly. A covert flag need not (and should not) be a normal boolean variable. It can take the form of a complex data structure, such as advocated by Collberg et al. [9–11].
Researchers have been putting a fair amount of effort into building detection systems. A *static checksum* based on either the static program code [5, 13] or

dynamic execution traces [7] of the code is computed and stored in a secret place. The detection system computes the new checksum when programs are running in malicious hosts, and check whether the new checksum is identical to the old one.

Response. When tampering is detected, an unusual event must happen to either stop the program from functioning (in the case of standalone applications) or informing the appropriate authority (in the case of network-centric applications). In this work, we restrict our attention to standalone applications in which a program failure is often a desirable event post tamper-detection.
We expect the TRS module to have multiple response code instances in place. Ideally they should be mutually independent so that uncovering of one does not easily lead to uncovering of others. In theory, the responses should be so crafted that the hacker cannot easily locate the code and disable it, nor backtrack to the detection code from it. However, in practice the detection mechanism can often be located by inspecting the code statically or back-tracing from the response that the tamper-resistant code generates.

We note that our work is about separating tamper response from detection, but not about choosing the detection sites in the first place. We assume that some list of detection locations is provided to our algorithm. For example, a programmer may choose such locations manually; alternately, a tool may generate a list of sites semi-randomly, possibly influenced by performance and security requirements, as well as by static and dynamic analysis. Related to the checking mechanisms themselves, such methods are beyond the scope of this paper.

3.1 Principles of effective tamper-response mechanisms

Let us first look at a naïve response system (an example also used by Collberg and Thomborson [10]) and see what kind of attacks adversaries can apply:

```
if tampered_with() then i=1/0
```

Upon detecting tampering, the above response code causes a divide-by-zero error and then the program stops. Since the program fails right at the place where detection happens, an adversary, with the ability to locate the failure point[4], can trivially trace back to the detection code and remove it. Alternatively, since divide-by-zero is an unusual operation, an adversary can statically scan the program to locate the detection code fragments and then remove it.

The naïve response reveals many information of the TRS module to an adversary. An ideal response system, in contrast, should not reveal information of the TRS module. Based on this guideline, we next propose a set of principles[5] for effective tamper-response mechanisms.

[4] A debugger is sufficient.
[5] The principle of spatial/temporal separation has also been briefly discussed by Collberg and Thomborson [10].

Spatial separation. Tamper responses and the corresponding failures should be widely separated in space: While the response is performed in one part of the program, its effect (failure) becomes only apparent in other parts. This way, even an adversary can identify the failure point, he cannot trace back to the response point.

One question is that what is a good metric for spatial separation. One metric is the number of function calls invoked between tamper responses and program failures. By increasing the number of function calls, we hope that little trace has been left for an adversary to perform any analysis. In addition, the function where the response code resides is better not in the current call stack when the failure happens, because debuggers can give adversaries the information of the current call stack.

Temporal separation. If a response system can cause enough amount of delay before failure, it can effectively thwart the process of tampering. Imagine an attack whereby an adversary tries a number of tampering options. The adversary tries one option and starts to observe the program's behavior to see if the tampering works. Suppose our response system will not fail the program until after a large amount of time, say one day. Then only after one day, the poor adversary will discover that his trick is not working and he needs to spend another night to try another option. This is psychologically frustrating for the adversary and will certainly slow down the tampering process. The strategy of delayed failure is analogous to injecting extra delay between two consecutive password tries in a password protection system.

The metric for temporal separation is obviously the time or the number of instructions executed between response and failure.

Stealth. The code in a tamper-response system should blend in with the program being protected so that an automatic scanning tool will not identify the tamper-response code easily. A response system involving division-by-zero is definitely not a good idea.

Stealth is a highly context-sensitive quality. Response code that is stealthy in one program may not be so in another. Any metric for stealth has to be with respect to the context, or the program. One possible metric is the statistical similarity[6] between the program being protected and the response code.

Predictability. A program that has been tampered with should eventually fail, with high probability. We also want to control when and where the failure (damage) can happen. A failure that happens during sensitive operations is probably undesirable.

In addition, any available obfuscation should be used to protect the tamper-detection and response code. Ideally, neither observation nor tampering should

[6] E.g., the percentage of each kind of machine instructions.

easily reveal patterns useful for determining where detection and response occur. In practice, both generic and application-specific obfuscation methods should be devised to maximize an attacker's workload.

4 System description

We now describe a response mechanism we have built following the principles in Section 3.1. Our starting insight is that by corrupting a program's internal state, a response system can cause the program to fail. If we carefully choose which part of state to corrupt and when to corrupt, we may achieve the aforementioned spatial and temporal separation. This deliberate injection of "programming bugs" also satisfies our stealth principle because these bugs look just like normal programming bugs and are thus hard to pick out by static scanning.[7] Bugs due to programming errors are hard to locate. Some of these bugs cause delayed failure. As an example, an early system called HUW [18] appeared to run successfully, but crashed after about one hour. This was due to an elusive bug in its string handling module, which corrupted the system's global buffer. The data structures inside the corrupted buffer, however, were not used until about an hour later. Therefore, the system ran OK until the corrupted data structures were accessed.

As we can see from this example, corrupting programs' internal state might produce the effect of delayed failure. For clarity, we assume there are three kinds of sites: detection sites (where tamper detection happens), response sites (where corrupting the internal state happens), and failure sites (where failure happens). Response sites are also called corruption sites in our system. In the rest of this paper, for simplicity, we will identify detection sites with corruption sites. In practice, detection sites and corruption sites should be separated (and communicate via covert flags), and the techniques and implementation that we will introduce applies as well.

There are many ways to corrupt a program's internal state. Our system chooses the straightforward way: corrupt the program' own variables. By deliberately corrupting the program' variables, we hope to achieve the following results:

- Predictable failure of the program, due to the corruption of the program's internal state.
- Stealthy response code, since the response code is just an ordinary variable assignment.
- Spatial and temporal separation, if we carefully choose when and where to corrupt the variables.

Not all variables are good candidates. Suppose the value of an integer variable ranges between 3 and 10. Then what would be the behavior of the program when

[7] If there is regularity in the type of bugs we introduce, an attacker may be able to employ static analysis to increase his/her likelihood of locating them.

the variable is changed to 100? Would the program fail? Where and when would it fail? We have to answer these questions to achieve some predictability in our response system. We suspect that for an arbitrary program variable, the result of any analysis is highly imprecise. However, one observation can be made about pointer variables, which are ubiquitous in C-style programs. If a pointer is corrupted by setting it to a NULL pointer or a value out of the program's address space, dereference of this pointer definitely crashes the program. Moreover, if the next dereference happens only after some time, we achieve the effect of delayed failure.

Corruption of local pointers (declared in a function body) is unlikely to achieve much delay. The corruption of local pointers has to happen locally, because their storage is in the run-time stack, Their values are also used locally, which means the corruption and usage would be very close if we had chosen to corrupt a local pointer.

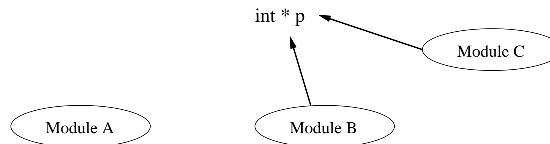


Fig. 1. Global pointers.

For global pointers, the scenario is different and one example is depicted in Figure 1. Suppose there is a global pointer `p` which is used by modules B and C, but not touched by module A. If we choose to corrupt this pointer in module A, then the program will keep running until module A has finished and the program switches to module B or C. Based on this example, intuition is there that delayed failure can be achieved by corrupting a global pointer.

But what if the program has not many global pointers? Our solution is to perform transformations on the program to create new global pointers. One way to achieve this is to add a level of indirection to the existing global variables. The idea is illustrated by the example in Figure 2.

On the left of Figure 2 is the original program; the code after transformation is on the right. For the global variable `a`, we create a new pointer variable `p_a`, whose value is initialized to the address of `a`. Then we replace all uses, or some uses, of variable `a` by the dereference of the newly created pointer variable `p_a`.

Below are the benefits of the extra level of indirection to global variables:

- Any global variables can be used to create new pointers, alleviating the possible shortage of global pointers.
- The failure behavior of the new program is easily predictable. After `p_a` is corrupted, any subsequent use of the variable, `p_a`, would be a failure site.

```c
int a;
void f() {
  a = 3;
}
void main() {
  f();
  printf(``a = %i\n'', a);
}
```

```c
int a;
int *p_a = &a;
void f() {
  *p_a = 3;
}
void main() {
  f();
  printf(``a = %i\n'', *p_a);
}
```

Fig. 2. Example: Creating a layer of indirection to global pointers.

– We can also control where the program fails. For example, if we do not want the program to fail inside the `main` function, we just do not replace `a` with `p_a` in `main`.

Note that the extra level of indirection to global variables does slow down the program because of the cost of extra dereferences. On the other hand, this performance hit is controllable since we can control how many uses of global variables are replaced by their pointer counterparts.

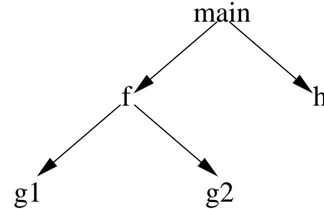
4.1 Choosing corruption sites

As we have explained, global pointer variables are the targets to corrupt in our system. The remaining question is where to corrupt those pointers? Since we could corrupt a global pointer anywhere in the program, the search space is the whole program.

To make our search algorithm scalable for large programs, we use functions as the basic search units instead of, say, statements. Based on this, we state the searching problem more rigorously. Corruption of global pointer variables can happen inside any function body; thus a function body is a possible *corruption site*. A *failure site* is a function where the program fails when the program reaches the function after pointer corruption. In our setting, failure sites correspond to those places where corrupted pointers are dereferenced. To find good corruption sites, we want to search for functions to embed pointer corruptions, to achieve wide spatial and temporal separation between corruption and failure.

First, we should make sure the function where corruption happens is not in the current call stack when the program fails. Otherwise, attacker could use a debugger to back-trace from the failure site to the corruption site. To avoid such attacks, we use a static-analysis tool called call graphs. Below is an example program and its call graph.

```
int a;
int *p_a = &a;
void g1();
void g2();
void h ();
void f() {
  g1(); g2();
  *p_a = 3;
}
void main() {
  f(); h();
}
```



In the example, the `main` function calls the `f` function; thus there is a directed edge from `main` to `f` in its call graph. Similarly, since `f` calls `g1` and `g2`, the call graph has the directed edges from `f` to `g1`, and from `f` to `g2`.

In the example program, suppose our system decides to corrupt the pointer variable `p_a`. Then the function `f` is a failure site since it dereferences `p_a`. Obviously, the corruption should not happen inside `f` because otherwise the program would fail in the function where the corruption happened. Furthermore, the `main` function should not be the corruption site since otherwise the `main` function would be in the current call stack when the program fails in the function `f`. In general, our system excludes all functions where the corrupted pointer variable is dereferenced; furthermore, it excludes all functions who in the call graph are ancestors of those functions where failure can happen. This heuristic guarantees that when the program fails the corruption site is not in the call stack.

Additionally, we want to achieve wide spatial and temporal separation between corruption and failure. We first present some experimental numbers, which show the spectrum of spatial and temporal separation. We conducted the experiment on a C program called Winboard. We picked 800 functions in Winboard, planted the corruption of a selected pointer variable into each function, and recorded the temporal and spatial separation between corruption and failure. Figure 3 shows the temporal separation, and Figure 4 shows the spatial separation. In both figures, the horizontal axis is the function ID where the corruption happens. In Figure 3, the vertical axis is the elapsing time between corruption and failure in microseconds. In the Figure 4, the vertical axis is the number of function calls happened between corruption and failure.

As we can see from the figures, the spread is over several orders of magnitude. The functions in the upper portion are the ones we want to search for. However, a static heuristic may be hard to succeed because essentially an estimate of time between two function calls is needed; we are not aware of any static-analysis techniques that can give us this information.

Our solution is to measure the average distance between two function calls in a dynamic function-call and time trace. This information estimates how far a function is from the nearest failure sites (functions that dereferences the pointer) in terms of both the number of function calls and time. Only functions that are far from failure sites will be selected as corruption sites. We experimented this

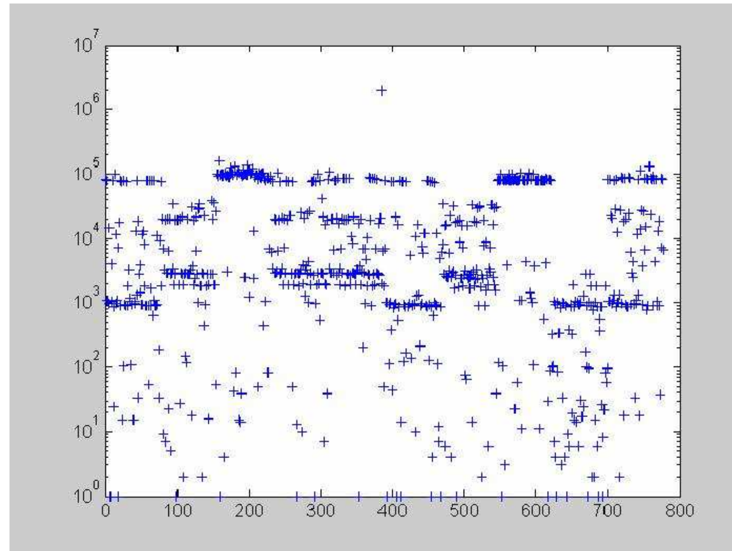


Fig. 3. Temporal separation between corruption and failure. The horizontal axis represents function IDs, and the vertical axis represents the elapsing time between corruption and failure in microseconds.

heuristic on the Winboard program and the results showed that those functions in the upper portion of Figure 3 and 4 are most likely to be selected. The shortcoming of this approach is that it depends on dynamic traces, which may be correlated to user inputs and other random events.

To make it more precise, we outline our algorithm for selecting good corruption sites in Figure 5. For a simple presentation, the algorithm processes a single C file, called example.c (Our implementation can process multiple files at once).

The algorithm takes three inputs. The first is the source file. The second is a function-distance matrix T, which tells distances between functions. The value $T[f_1, f_2]$ is the distance between functions f_1 and f_2. In our system, the matrix is computed from a typical dynamic trace of the program. The last input is a threshold parameter δ to dictate the minimal distance between corruption sites and failure sites.

For each global variable g_i, the algorithm first identifies those functions that use the value of g_i (line 4). A function f in this set is a failure site for g_i, because if we had created an indirect pointer to g_i, say p_{g_i}, and replaced the use of g_i in f by $*(p_{g_i})$, then the program would fail inside f after p_{g_i} had been corrupted. The algorithm then proceeds to rule out all functions that are ancestors of the failure sites in the call graph (line 6), so that when program fails, the function where corruption happens will not be in the call stack. Finally, the algorithm rules out those functions that are too close to failure sites (line 8 and 9).

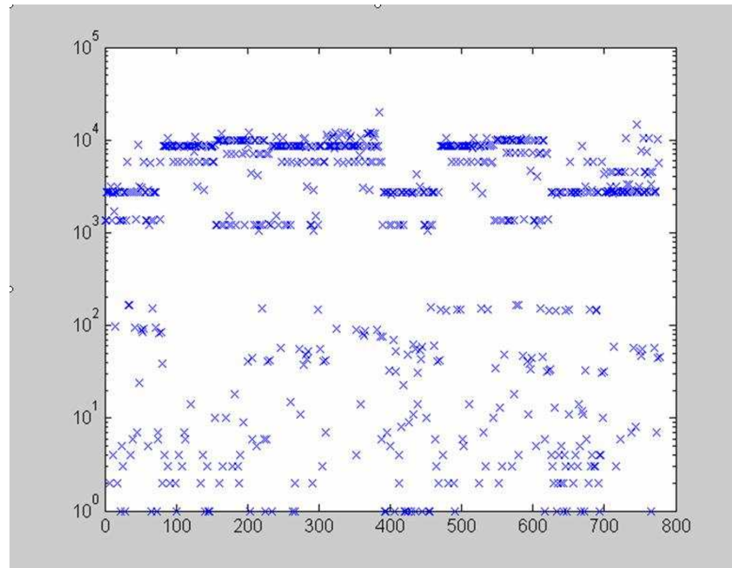


Fig. 4. Spatial separation between corruption and failure. The horizontal axis represents function IDs, and the vertical axis represents the number of function calls between corruption and failure.

5 Implementation and evaluation

We have built a prototype system, which takes C programs as inputs and automatically inserts tamper-response code. The system identifies good global variables as target variables to corrupt when tampering is detected; it also selects good corruption sites according to the heuristics we explained in section 4.

The flow of our implemented system is depicted in Figure 6. In the figure and also in the following paragraphs, we use the Winboard program as the example application to explain our system. Winboard is a chess program written in C. It has totally 27,000 lines of C code, and contains 297 global variables, which are potential target variables to corrupt.

Winboard consists of a bunch of C files: `winboard.c`, `backend.c`, *etc.* In our system, these files are first fed into a `varusage` module. For each source file, the `varusage` module produces a `.use` file, which identifies places that global variables are used. Separate `.use` files are linked by the `uselink` module to produce the global `.use` file. Source C files are also fed into the `callgraph` module, which produces `.cg` files, or call graph files. Separate `.cg` files are linked together by the `cglink` to produce a global call graph.

We also run profiling tools on the program to produce a dynamic trace. The trace records the order of entering and exiting functions and also the corresponding timestamps. This trace is the input to the `trmatrix` module. The module

Input: a) example.c, with global variables $g_1, g_2, \ldots, g_n$;
b) Function-distance matrix T;
c) δ: Threshold for the distance between corruption and failure sites.

Output: The set of good corruption sites C_i, for each g_i.

```
 1: Compute the call graph G of example.c
 2: for each global variable g_i, 1 ≤ i ≤ n do
 3:    C_i ← the set of all functions in example.c
 4:    Identify the set of functions where the value of g_i is used, say {f_i1, ..., f_im}
 5:    for each f_ij, 1 ≤ j ≤ m do
 6:       Remove from C_i all the ancestors of f_ij in the call graph G.
 7:       for each f remaining in C_i do
 8:          if T[f, f_ij] < δ then
 9:             remove f from C_i
10:          end if
11:       end for
12:    end for
13:    Output C_i for the global variable g_i
14: end for
```

Fig. 5. Algorithm for selecting good corruption sites

measures the average distance between two functions in terms of both elapsing time and the number of function calls, records the information into a matrix, and writes the matrix into a `.tr` file.

At this point, we have `winboard.use` (recording where global variables are used), `winboard.cg` (the global call graph), and `winboard.tr` (the trace matrix). These files are inputs to the `delayedfailure` module. The module first computes the set of good corruption sites for each global variable, following the algorithm in Figure 5, and then randomly selects some global variables and good corruption sites. Finally, the `corrupt` module performs source-to-source transformation to first create a layer of indirection to selected global variables, and then plant the corruption of the newly-created pointers into selected corruption sites (on the condition that tampering is detected).

5.1 System evaluation/threat analysis

Overall, our system protects software by making them exhibit the effect of delayed failure after tampering is detected. To remove our tamper-response code, the attacker has to trace back from the crash site to analyze what is corrupted and where the corruption happens. Since we corrupt pointer variables, the attacker essentially has to debug programs with elusive pointer-related bugs, which many programmers know can be extremely hard; the situation is actually worse for the attacker, because he has no source code. Next, we evaluate our system in more detail in terms of the principles we laid out in section 3.1.

Spatial separation. Our system can guarantee wide spatial separation between the corruption site and the failure site. We achieved the separation

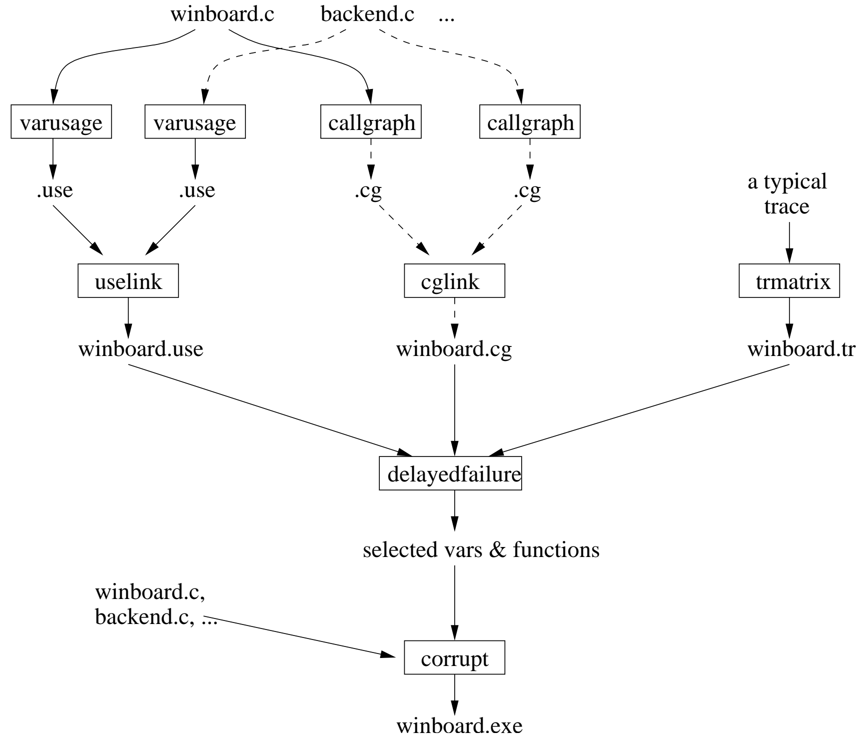


Fig. 6. System Implementation.

on the order of 10^4 functions calls in our experiment. With the help of call graphs, it can further guarantee that the corruption site will not be in the call stack when failure happens.

Temporal separation. Using the dynamic-trace approach, we achieved seconds of delay in our experiment, which is much better than immediate failure. Further delay can be achieved with the following techniques. First, our experiments were conducted by setting the pointer variable to NULL. Our system can be easily configured so that pointer corruption means adding random offsets to the pointer. In this case, the cumulative effect of several consecutive corruptions will most likely crash the program, and the delay will be boosted by this technique. Second, since we wanted automatic testing in our experiment, we avoided those functions which need human interaction to invoke, e.g., the functions that will be invoked only if certain buttons are clicked. User-behavior models can inform us of those functions that are called occasionally. For example, if we know that a certain function is called only once in an hour, then we can plant the corruption into that function to achieve long delay.

Stealth. Since our response system only manipulates pointers, it should be fairly stealthy in programs that have lots of pointer manipulations. For programs in which pointers are scarce, one possible attack for an adversary is to track the pointers of a program dynamically, identify the instructions that make pointers nonsensical, and then remove those instructions.

To counter this attack, our system should be combined with various types of obfuscation and integrity checks. For example, dynamic computation of global-variable pointers, including techniques such as temporary pointer corruption and runtime relocation of global data, should complicate attacks that track pointer usage via breakpoints or traces. We also embed multiple pointer corruptions, so that even one has been removed, others can still work. Finally, we are experimenting with the idea of corrupting data structures through pointers. In these kinds of corruption, pointer values always stay meaningful; only certain invariants of the data structure are destroyed.

Predictability. Our response system is predictable and controllable. Pointer dereferences after its corruption will surely fail the program. Any dereference becomes a failure site and we can control where failure happens.

6 Extensions

Safe languages. In safe, strongly typed languages such as C# and Java, pointers and global variables may be either unavailable or limited to atypical usage. While our pointer-corruption method does not have an immediate analogue in safe code, various other techniques can achieve similar results. In general, the main idea of separating tamper response from detection applies just as well to safe languages as to C/C++.

Below are some examples of delayed corruption possible via a safe language:

- Array out-of-bounds errors: Set up an array index to fall beyond the array's limits.
- Infinite loops: Change a variable in a loop-condition test to result in an infinite (or at least very time-consuming) loop.

Such techniques require implicit data-based links between the code at corruption and failure sites. While global variables in C/C++ serve to create such connections, proper object-oriented design stipulates object isolation and tightly controlled dataflow. Nonetheless, some object fields (e.g., public static members in C#) serve essentially as global variables. Some applications also use dedicated namespaces and classes that encapsulate global data, which can also substitute for true global variables.

To increase the number of opportunities for delayed responses, we can perform various semantically-equivalent code transformations that break object isolation, similar to how we create global pointer variables. As an example, we can convert a constant loop endpoint or API-function argument to a public static

variable that can be modified to effect a tamper response. If a good response location contains no suitable code, we can inject new code that references such variables (e.g., a new loop or system-API call). Randomly generated and tightly integrated, such code should have no operational effects if tampering is not detected.

Graceful degradation. Some of the above techniques do not cause failures as predictably as pointer corruption. However, *graceful degradation* can be more stealthy and difficult to analyze than definite failures. Any particular response might not terminate the program, but if one or more checks continue to fail, the cumulative effect should eventually make the program unusable. Both the checks and responses can also be made probabilistic in terms of spatial/temporal separation and response action.

A program run could degrade its functioning via slowdown, high resource usage, and arbitrary incorrect operation (e.g., file corruption or graphics distortion). Such techniques may be generic and automated; for example, we can transform program loops to include conditions that take increasingly longer time to satisfy (e.g., via gradually incremented counters). While application-specific techniques require manual design and implementation, these could be quite effective (e.g., a game where the player's movements and aim become increasingly erratic [16]).

7 Conclusions

A tamper-resistant system consists of tamper detection and tamper response. Inadequate tamper response can become the weakest link of the whole system. In this paper, we have proposed a tamper-response mechanism that evades hacker removal by introducing delayed and controlled failures, accomplished by corrupting the program's internal state at well-chosen locations.

Acknowledgment

We thank Stephen Adams and Manuvir Das for providing some static-analysis tools, and Matthew Cary for helpful conversations.

References

1. Business Software Alliance and International Data Corporation. Annual BSA and IDC global software piracy study. http://www.bsa.org/globalstudy, 2004-2006.
2. David Aucsmith. Tamper resistant software: An implementation. In *First Information Hiding Workshop*, pages 317–333, 1996.
3. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Advances in Cryptology - CRYPTO '01, Lecture Notes in Computer Science*, 2139:1–18, 2001.

4. Pavol Cerven. *Crackproof Your Software*. No Starch Press, Inc., 2002.
5. Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *Digital Rights Management Workshop*, pages 160–175, 2001.
6. Yuqun Chen, Paul England, Marcus Peinado, and Bryan Willman. High assurance computing on open hardware architectures. Research Report MSR-TR-2003-20, Microsoft Research, Microsoft Corporation, Redmond, Washington, USA, March 2003.
7. Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding Workshop*, pages 400–414, 2002.
8. Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Information Security, 4th International Conference*, pages 144–155, 2001.
9. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
10. Christian S. Collberg and Clark D. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Trans. Software Eng.*, 28(8):735–746, 2002.
11. Christian S. Collberg, Clark D. Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–196, 1998.
12. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
13. Bill Horne, Lesley R. Matheson, Casey Sheehan, and Robert Endre Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Digital Rights Management Workshop*, pages 141–159, 2001.
14. `http://cdfreaks.com`, 2006.
15. Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT 2004*, pages 20–39, 2004.
16. Macrovision. FADE, SafeDisc and SafeDVD copy protection, 2002.
17. Marcus Peinado, Yuqun Chen, Paul England, and John Manferdelli. NGSCB: A trusted open system. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP*, volume 3108 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 2004.
18. I. C. Pyle, R. C. F. McLatchie, and B. Grandage. A second-order bug with delayed effect. *Software – Practice and Experience*, 1(3):231–233, 1971.
19. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, December 2000.
20. Hoeteck Wee. On obfuscating point functions. Cryptology ePrint Archive, Report 2005/001, 2005. `http://eprint.iacr.org/`.