A Framework for Understanding Dynamic Anti-Analysis Defenses

Jing Qiu
Harbin Institute of Technology
Harbin, China
topmint@hit.edu.cn

Babak Yadegari
The University of Arizona
Tucson, USA
yadegari.babak@gmail.com

Brian Johannesmeyer
The University of Arizona
Tucson, USA
bjohannesmeyer
@cs.arizona.edu

Saumya Debray
The University of Arizona
Tucson, USA
saumya.debray@gmail.com

Xiaohong Su
Harbin Institute of Technology
Harbin, China
sxh@hit.edu.cn

Abstract

Malicious code often use a variety of anti-analysis and anti-tampering defenses to hinder analysis. Researchers trying to understand the internal logic of the malware have to penetrate these defenses. Existing research on such anti-analysis defenses tend to study them in isolation, thereby failing to see underlying conceptual similarities between different kinds of anti-analysis defenses. This paper proposes an information-flow-based framework that encompasses a wide variety of anti-analysis defenses. We illustrate the utility of our approach using two different instances of this framework: self-checksumming-based anti-tampering defenses and timing-based emulator detection. Our approach can provide insights into the underlying structure of various anti-analysis defenses and thereby help devise techniques for neutralizing them.

Keywords

Anti-analysis Defense, Taint analysis, Sefl-checksumming, Timing defense

1. INTRODUCTION

Malicious binaries typically combine a variety of defenses in order to avoid detection and hinder analysis. For example, a program may use runtime unpacking to thwart static analyses; anti-analysis defenses to thwart dynamic analyses; code obfuscation to make it difficult to locate and understand these defenses; and anti-tampering checks to prevent the defenses from being disabled even if they are located [9, 13, 16, 32, 35, 36]. An analyst wishing to understand the malware code has to first get past such defenses; this task can be tedious and time-consuming. [1] This makes it important and useful to develop frameworks that help us understand the overall structure of such anti-analysis defenses.

Numerous authors have investigated techniques for neutralizing anti-analysis defenses [3, 10, 11, 18]. To the best of our knowledge, these works all focus on specific kinds of defenses, and do not identify or take advantage of underlying conceptual similarities between different kinds of anti-analysis defenses. However, identifying such conceptual similarities can be helpful for devising strategies for neutralizing them. For example, Cappaert *et al.* [4] and Wang *et al.* [41] discuss how checksum values can be used as code unpacking keys in anti-tampering defenses; Lu and Debray [24] discuss the use of timing information in anti-analysis defenses in web-based malware that use emulation-based obfuscation. While these may seem, superficially, to be very different kinds of defenses, they share underlying structural similarities; understanding these similarities can be useful in adapting defenses from one to the other.

This paper describes a general information-flow-based framework for such defenses and shows how several kinds of anti-tamper and anti-analysis defenses can be seen as instances of this framework. Such an approach can help improve our understanding of such defenses and be useful for devising methods for identifying and neutralizing them.

A key assumption we make is that of *observability*: namely, that the attacker, i.e., the person or system attempting to analyze a program, has complete access to the host and is able to observe the program as it executes, including the instructions executed and the values of registers and memory. This assumption is appropriate for most malware analysis scenarios, but does preclude some systems, like Conqueror [26] that rely on an external entity (a remote host) for some portions of the defense.

2. BACKGROUND

PPREW-4, December 09 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-60558-637-3/14/12...$15.00
http://dx.doi.org/10.1145/2689702.2689704.

[1]In some cases, e.g., involving simple execution-time-based defenses aimed at discovering whether the program is being emulated, it may be possible to fool the defenses using an untruthful clock. However, not all anti-analysis defenses are amenable to such straightforward solutions.

2.1 Taint Analysis

Dynamic taint analysis, which finds a wide variety of uses in software security applications, observes information flow connecting sources and sinks based on an execution trace. The predefined taint sources consist of input to an application – whether it be from a user, network read, etc. Any value within the trace whose calculation depends on data acquired from a taint source is deemed tainted, and any other value is deemed untainted. [34] points out that the exact way the taint flows during a program execution, the kinds of operations that initiate new taint, and the kinds of checks that are performed on tainted values can vary per specific implementation of taint analysis. The flow of input to output values is known as forward tainting.

An analogous technique known as backward tainting exists in which the sinks are predefined – whether through an output to a user, network write, etc. Rather than observe the program values that are dependent upon the taint sources, backward tainting observes the values which the sinks are dependent upon. [43] introduces a technique of combining forward and backward tainting approaches in order to eliminate program obfuscations and in turn simplify an execution trace. Their approach utilizes different levels of granularity during the taint in order to bypass devious defenses such as those used in [37].

2.2 Non-Explicit Flow

However, a limitation of taint analysis is its code coverage. Taint analysis uses assignment statements to propagate explicit flows; however explicit assignments are not the only way of disseminating information flow. [5] examines two kinds of non-explicit flow: control dependencies and implicit flows. Control dependencies occur when a tainted value flows to the conditional of an if-then-else statement whose body contains an assignment.

Another form of non-explicit flow – implicit flow – arises when the assignment within a tainted if-then-else conditional is then used as the predicate for a later if conditional. An example can be drawn from [5] where a piece of tainted data `y` flows to `w` without explicit flows or control dependencies:

```
x=0; z=0;
if(y=0) then x=1; else z=1; endif
if(x=0) then w=0; endif
if(z=0) then w=1; endif
```

If an instruction trace were to be taken of this snippet, the implicit flow from y to w would not be mapped. In order to be able to handle this kind of implicit flow, we should consider control dependencies to propagate taint through control transfers [17]. In the example above, the values of `x` and `z` are control dependent on the value of `y` so there is an implicit data dependency between values of `x,z` and `y`. Similarly, there is an implicit dependency between `x` and `w` in the second `if`; and between `z` and `w` in the third `if`. So the variables `y` and `w` are dependent transitively.

As evident, there are caveats to different kinds of dynamic analysis – specifically, taint analysis. This paper presents a more general approach involving information flow with taint analysis.

3. UNDERSTANDING ANTI-ANALYSIS DEFENSES

3.1 A Framework for Anti-Analysis Defenses

Broadly speaking, anti-tampering and anti-analysis defenses involve making one or more observations about some aspect of the program or its execution environment and using these observations to determine whether the program is executing in a "friendly" environment or a "hostile" environment; depending on the outcome of this check, the program may then continue executing normally if the environment is deemed to be friendly, and take some appropriate evasive action if the execution environment is deemed to be hostile. Furthermore, while the various computations involved may be implemented and/or obfuscated in different ways, it must necessarily be the case that there is a flow of information from the observations to the code that checks the observed values. This structure forms the basis of our framework for anti-tamper/anti-analysis defenses, which has the following general structure:

1. **Observation.** Observe some aspect of the program and/or its execution environment.

2. **Validation.** Check whether the observed value indicates a benign execution environment.

3. **Response.** If the environment is deemed to be benign, continue with normal execution; otherwise, invoke an appropriate response.

4. **Information Flow.** There is a flow of information from the Observation to the Validation to the Response steps.

While the specifics of the code can vary depending on the kind of observations being made, the way in which the observations are validated and the manner in which any necessary response action is invoked, this general structure seems to describe a wide variety of defenses. For example:

- In code anti-tampering defenses based on self-checksumming, the observation consists of computing a checksum of (some portion of) the program's code [4,6,15,39]. Validation can take many forms, including simple comparisons against some expected checksum value and checksum-based code unpacking schemes where the checksum computed is used as a key for unpacking subsequently executed code (i.e., any tampering causes subsequent code to be decrypted with an incorrect key and thereby results in garbage code being executed) [4,39]. The connections between the observation, validation, and response code can be made stealthy to make the defenses harder to identify [38].

- In timing-based anti-emulation checks, the value observed is the time taken to execute some piece of code. Validation involves checking whether the execution speed so determined is above some threshold. Options for implementing responses are similar to those for checksumming.
- In targeted malware, the observation may focus on highly specific characteristics of the intended target. For example, the Stuxnet worm checked for the presence of specific hardware from Siemens [22]. Some kinds of espionage malware will first check a victim computer's IP address and only distribute secondary malware to the victim if it belongs to a particular set of targeted IP addresses [19]; otherwise, it distributes benign content.

As the examples above show, the various components of the anti-analysis defense framework can be implemented in a number of different ways. Nevertheless, the key notion underlying our approach is that there is a flow of information from an observation (either of the program or of its environment) to some code that uses the result of the observation (Validation) to control the subsequent behavior of the program (Response). This flow of information may not always be obvious, e.g., it may be stealthy and widely separated in te program [38], or may use implicit flows for concealment [5]. However, some such information flow is essential for the defense to work.

It should be noted that this kind of information flow is by no means unique to anti-analysis defenses: there are many computations where environmental factors influence the computational behavior of the program. Nevertheless, understanding this conceptual structure of anti-analysis defenses and the role of information flow, and developing and using tools that can help track such information flow, can be helpful in identifying and understanding anti-analysis defenses in programs. For example, a security analyst who suspects that a program under analysis may be using anti-analysis defenses to conceal some of its functionality may be able to use such information flows to discover the defenses being used and possible countermeasures to such defenses.

3.2 Self-Checksumming-based Anti-Tamper Defenses

Code self-checksumming is commonly used as an anti-tampering defense [4, 6, 14, 15]. The idea is to have the program compute a hash value over its code and use this value in the subsequent computation in such a way that the program executes normally if and only if the hash value computed is as expected. The observation step is thus the computation of the checksum, which may be computed either on the code that is actually executed or on memory locations containing a packed version the code. The validation and response steps can be carried out in several different ways. For example, the program can branch to the response code, using either a straightforward compare-and-conditional-branch logic or using the checksum value to compute the target address of an indirect jump instruction [38]. Alternatively, the checksum value can be used as the decryption key for runtime-unpacked code [4, 41], such that an incorrect checksum value results in incorrect unpacking results and therefore incorrect program execution. As the second alternative illustrates, the validation step of a defense need not involve an explicit comparison against some expected value; rather, it can simply be a computation that produces correct output if and only if the observation produces the observed value. Lu and Debray use a similar approach in a construct called "implicit conditionals" that branch to the correct targets only if certain environmental observations yield expected values [24].

Since malware often use runtime code unpacking, we do not rely on static analysis which is unable to examine dynamically created code; instead, we use dynamic analysis. Our approach consists of the following steps:

1. **Backward taint analysis**. Starting from code addresses, i.e., addresses of executed memory locations, this step propagates taint in a backward direction, i.e., from uses to definitions, to identify memory locations that are used to create code.
2. **Forward taint analysis**. This step identifies the flow of values computed from locations that are tainted in the backward taint analysis step (i.e., which either contain executed instructions or are used to create instructions to executed instructions) to code that affects the program's execution. This is done by starting from the locations tainted in the backward taint analysis step and then propagating taint forwards (i.e., from definitions to uses).
3. **Checksumming detection**. This step identifies the validation step, i.e. where the code is performing checksum verification. Instructions I can be identified as performing checksum verification having one of the following effects:
   (a) affecting control flow of execution depending on a tainted value; or
   (b) writing a tainted value to a location that is going to be executed later; or
   (c) uses a tainted value to pass it as an argument to an output system call.

The combination of backward and forward taint computation is necessary because it is possible to set up the checksum computation so that it considers, not the locations that are actually executed, but locations from which the instructions at those executed locations were created. For example, a piece of sensitive code—say, a license check or anti-analysis defense—may be stored in encrypted form in a memory region R, and decrypted as needed into some other memory region S from which it is executed; meanwhile the checksum computation can be applied to the memory region R, which is not itself executed. The backward taint analysis starts with the executed code in S, goes backward to taint R, then propagate this taint forward to the instruction(s) that perform checksum verification.

Once the checksum verification instruction(s) have been identified, the taint information gathered from the forward taint propagation step can be used, possibly in conjunction with some additional analysis of the execution trace, to extract a variety of information about the self-checksumming protections deployed by the program under study. This information can be useful in guiding efforts to defeat or bypass the program's self-checksumming anti-tamper defenses. The remainder of this section briefly describes some information that can be obtained in this way.

I. Checksum Computation.

The forward taint step identifies all code section which access executed locations, but these are not necessarily checksum computations routines as the program might be doing run-time unpacking. In order to obtain checksum computation routines, we can perform a backward slice [21] starting from checksum verification instructions. This needs to be done carefully since the slicing algorithm should be done with respect to unstructured nature of the executable code [20]. Second, the control flow graph of the program should be constructed using the execution trace and the portions of the code that were executed. Since it is possible for an executed memory location to contain different byte codes throughout a program's execution via dynamic unpacking, another parameter in addition to its address is necessary to uniquely identify an instruction. However, it is possible for different checksum computations to use the same verification code; thus a slice is computed whenever a verifier is observed. Although it might not be practical to share a verifier in this way from a security standpoint, multiple checksumming instances can be detected even if they share a verifier. Even if different checksums are computed concurrently and interleaved, e.g. using multiple threads, our technique can identify each one (this involves using the thread-id to keep track of the instructions in the different threads when collecting the trace).

We can also use the dynamic slice to determine where the checksum computation code originated. For example, if the memory locations corresponding to the checksumming code were written to before the checksum was computed, it means that the checksumming code was the result of runtime unpacking. Further, if the locations where the unpacked values originated were derived from an external source (e.g., via a socket read), it means that the checksumming code was obtained from an external source (e.g., as in Conqueror [26]). Similarly, if the checksum computed by the program is written out, e.g., via a network write, it suggests that the validation step may be being performed by a remote agent. Thus, the flow of taint into or out of the program can be used to identify violations of the observability assumption mentioned in Section 1.

II. Checksum Verification.

One way by which an attacker can try to defeat self-checksumming is to change the validation code to transfer control unconditionally to the "normal execution" code. It turns out that this simple attack may not always work, and we can determine whether it will work from examining the flow of tainted and untainted values into the validation code. The following is a list of conditions under which this simple attack will fail:

1. If the validation code is invoked with tainted inputs at some points and with untainted inputs at others.
2. If a backward-tainted location has a forward-tainted value written into it (indicating that the checksum is used as an unpacking key), or passed to an output operation in the program as an argument.

The first situation arises in emulation-obfuscated code (e.g., we observed it in code protected using Themida [30]), where the program logic is embedded into the byte-code of a custom virtual machine and program executable is simply the emulator for this virtual machine. Each different operation of the virtual machine has a different handler in the emulator, e.g.:

```
handle_if_EQ:  /* if_EQ op1, op2, target */
  op1 = fetch_op1();
  op2 = fetch_op2();
  target = fetch_op3();
  if (op1 == op2) {
    ip = target;
  }
  else {
    ip++;
  }
  goto emulator_dispatch;
```

This code fragment will be executed whenever an '`if_EQ`' operation is encountered in the byte code, including for example the checksum validation code; in some of these uses the handler code will have tainted operands; in other cases the operands will be untainted. Modifying this code to always branch to some fixed location will therefore alter the behavior of all such conditional branches in the byte code and therefore result in an incorrect computation.

3.3 Timing-Based Anti-Analysis Defenses

Timing defenses are used to detect whether a program's execution is being monitored via dynamic analysis. The assumption is that code being monitored runs much slower due to the overhead arising from the dynamic monitoring. The idea behind a timing defense is to measure the time taken to execute a (small) piece of code and then check whether the execution time is within a range of execution times on a friendly environment. A hostile environment usually has a high overhead because of the analysis tools trying to observe and record the running program's behavior. This overhead is often high enough to make the execution time out of the range for a friendly environment and thus can be detected; the program can avoid normal execution by observing this abnormal environment.

One solution for getting around timing defenses is by having the underlying operating system return fake time values so as to give the program the illusion that it is running on a friendly machine. This modification can happen in the hardware or the operating system, such as redirecting system calls – which are not trivial to implement. However, it will not defeat timing defenses which receive the values from an outside source, such as a network time server.

Our approach to identify timing based defenses is general in a sense that it can be done either statically and/or dynamically. Static analysis has already shown to be ineffective against run-time unpacking and/or polymorphic or metamorphic techniques [28] commonly used in malicious code, but unless the program being analyzed incorporates a defense to thwart the static analysis (e.g. by using meta-

```
t0 = clock();
// execute some code
t1 = clock();
...
if (t1 - t0 > threshold){
    response();
}
// continue normal execution
```

Figure 1: An example of timing anti-analysis defense

morphism or run-time unpacking), it can be statically analyzed using our technique. Our approach uses a combination of *taint analysis* and *control dependence* analysis which can either be done statically or dynamically. In a case where dynamic analysis is more suitable, we need to first collect a trace of the program by watching the execution of the program and then performing dynamic taint analysis on the trace. Afterwards a control flow graph can be generated from the trace, which is then used to extract control dependencies. There is a great deal of research on these topics from both dynamic and static perspectives [29, 31, 34].

The goal is to identify whether a program uses time values so that these observations can somehow impact the control flow of the program. Intuitively, using taint analysis will be able to identify the flow of input time values and whether they affect a control transfer in the program or not, simply by observing if a conditional and/or indirect control transfer is based on a tainted value. This can be effective against simple time defenses where the time values collected by the program flow directly to any control transfer instruction. However, there are cases where the observed value does not directly affect the control flow and the effect is implicit. We can use control dependency analysis to identify implicit data flow through control transfers, thus addressing this limitation.

The code sample in Figure 1 shows an example of timing defense used for anti-analysis purposes. The program observes the execution time of a piece of code where the code is expected to take no longer than `threshold` to execute, otherwise the environment is not normal as expected and so the program will abort the normal execution. The execution time is then validated by being compared with the expected value, `threshold`. If it is smaller than the expected value then it continues the normal execution; otherwise it elicits an appropriate response. This is the case where there is a direct dependency between observed values and the control transfers of the program.

Taint analysis consists of the following steps:

(*i*) *Introducing Taint:* The first step for taint analysis is introducing or identifying taint sources, which in our case is the time values collected by the program in the execution. There are many ways a program can collect such time values, either through executing the `RDTSC` instruction which returns the processor time, or through different system calls provided by the operating system. For example, Windows has a variety of system calls which return a time value such as `GetSystemTime`, `GetTickCount`, `timeGetTime` which all have different outputs for different uses. Such calls read from specific locations in the system's memory in order to collect a time value to output; these memory locations would be read by any invocation – either inlined or otherwise. Often the protected program only wants to calculate the execution time spent between two different points in the program, so which method used is not so important. That being said, all these sources then become a potential source for a time defense. As opposed to checksum detection where an execution trace is needed to find the taint sources, we know the taint sources for timing-based anti-analysis defenses without preprocessing which makes it even more general.

(*ii*) *Taint Propagation:* After taint sources are identified, we need to propagate taint through the program execution. This step will identify program statements which affect the tainted values and/or are affected by these values. Meaning if a statement in the program uses some tainted value, the values that the statement modifies or affects will be tainted as well. We are interested in finding control transfers which are affected by time values. This step will identify those control transfers which taint directly flows into, and will thus be considered as candidates for timing-defense anti-analysis techniques.

The code example in Figure 1 shows a simple case where the time values are directly used for the validation step. In this case, simply doing the taint propagation on the values returned by the `clock()` function will identify the control transfer used for validation. However, a more complex case is when the time values are not used directly and the effect is implicit through control transfers, for example when the execution of a statement depends on the evaluation of a tainted value through an `if` statement (see Section 2). Identifying control dependencies straightforward using control flow graphs (CFG) and dominance information when the program has a structured CFG. However, for obfuscated code where the structure of the CFG is obstructed, e.g. by using CFG flattening or virtualization techniques, getting control dependencies may become tricky and we are still working to find a solution to handle unstructured CFGs.

Figure 2 shows a code example that can not be detected by only using taint propagation. In the code example, the while loop at line 4 is executed until there are at least 10 pairs of calls to `clock()` which return different time values. The number of calls to `clock()` is also recorded and compared to 10. If the number of calls is less than 10 – meaning that no pair of consecutive calls to `clock()` returned the same time value – then the environment is too slow and the program refuses to execute normally. The difference between this and the previous example is that there is no direct data flow between observation and validation parts. In order to handle this we need to incorporate *implicit flow.* We still need to propagate taint from calls to `clock()` function. The first implicit flow is at line 7 where the execution of `c++` depends on a tainted value. By marking variable `c` as tainted, it makes the evaluation of the `while` loop dependent on a tainted value, so the taint implicitly flows to the `num_calls` variable. The latter will cause the control flow at line `i` to become tainted.

```
num_calls = 1;
c = 0;
$t_0$ = clock();
while (c < 0xa){
    tnew = clock();
    num_calls ++;
    if ($t_0$ != $t_1$){
        c++;
    }
}
...
if (num_calls <= 0xa){
    response();
}
// continue normal execution
```

Figure 2: An example of timing anti-analysis defense through implicit flow

4. EVALUATION

We have developed a prototype tool using C++. Instruction traces are obtained by an Intel Pin tool [25]. The evaluation is performed on a 2.67GHz Intel Xeon E5640 processor with 96 GB of main memory running Ubuntu 12.04.

4.1 Setup

We evaluate our tool using two sets of programs. All these programs compute MD5 for a simple string.

The first set consists of programs with three kinds of checksumming protection and a program with time defense.

The three kinds of checksumming protection are multiple self-checksumming guards [6,15] (labeled as *50-guards*), using a checksum as a code decryption key [4,41] (labeled as *decrypt-key*) and using a checksum to generate a MD5 initialization constant (labeled as *chksum-md5*).

The time defense program fetches time by using system API `timeGetTime()`, `GetSystemTimeAsFileTime()`, `GetTickCount()`, and instruction `rdtsc`. It checks the difference value of two fetched time values to determine whether it has been traced and exits immediately when it found it has been traced. This program is labeled as *time-md5*.

For checksumming programs, we instrument them to report each address range that was checksummed each time a checksum is computed. This is then compared with the results reported by our tool. For time defense program, we verify the result with a debugger loading debug information generated by the compiler.

The second binary set consists of the MD5 computation program packed by Themida 1.8.5.5 and Obsidium 1.3.6.4. We only report the result of time defense in this group because we have not enough time to verify the result of checksumming detection.

4.2 Result and Discussion

The result of the evaluation is given in Table 1. The number of taint source of the two sets are the number of memory locations obtained by backward taint analysis and the number of calls for fetching time values, respectively.

For the program "50-guards", all guards and code they guarded are identified by our tool as expected. For the program "decrypt-key", following code is reported.

```
CODE[0] ^= checksum
CODE[1] ^= checksum
...
...
```

For the program "chksum-md5", the following code is detected as an anti-analysis response by our tool. The number 0x67452301 is a constant initial value of MD5 algorithm.

```
//This value should be 0x67452301;
mdContext->buf[0] = chksum + 0x6740E9CB;
```

For the program "time-md5", all conditional jumps that jump to anti-analysis response are reported. The code of this program is similar to the code in Fig. 1.

Although no false negative is found in the three programs, some guards may not be identified by our approach. For a specific input, not all the protected code is executed, so do the guards. If a guard is not executed, our approach can not identify it.

Results of first binary set indicate that our approach can identify code checksumming and time defense precisely no matter how complex of the relation between taint source and an anti-analysis response. Although no false positive is found, this does not mean our approach will find no false positive in other programs. It is somehow acceptable because in some cases, it is hard to distinguish the usage of taint source from anti-analysis response. For example, the time values can be used for benchmarking.

In the second binary set, time defense is only found in the program packed by Themida. In the tracing Themida with Intel Pin tool, the program pops up a message box and exits after closing the message box. When we hack the system API calls `timeGetTime()` reported by our tool, the program runs normally and outputs the same result as the running without tracing. The hacking is performed by a special debugger as there are anti-debugging techniques used in Themida. This indicates that our approach successfully identifies the time defense in the program packed by Themida.

The work load of our approach roughly equals to the number of instructions processed which approximates to the number of instructions in a trace times the number of taint source. No taint source is found in Obsidium, so its analysis time is the time used for walking instructions in the trace. In Themida, the program exits early as it detected the tracing. Thus, the number of instructions listed in Table 4 is smaller than that of a normal execution. But the activities of time defense is already recorded in the trace before exiting. So the time defense in a trace can be identified by our tool if the code of the time defense is executed.

The analysis times are all acceptable. It indicates that our approach is practical in dealing real world binaries.

Table 1: Evaluation Result

Program	Trace Size		No. of	No. of Guards		Analysis Time
	Mbytes	Instructions	Taint Source	Found	Ground Truth	(sec)
50-guards	823	2,702,679	4,855	50	50	95
decrypt-key	50	165,867	2,352	1	1	8
chksum-md5	88	296,138	2,353	1	1	4
time-md5	67	226,855	8	4	4	5
Obsidium	6,230	27,461,928	0	0	Unknown	347
Themida	2,690	9,304,222	232	2	Unknown	223

5. DISCUSSION

There are several limitations in our approach.

First, the input of our approach is a dynamic analysis approach where the low code coverage is the major limitation. There many other technologies can improve the code coverage, such as multiple path exploration [27] and generating inputs by symbolic execution [7].

Second, it may be not applicable to obtain instruction traces for some programs. For example, a program may need long time initialization or its program logic is very complex. That makes the instruction trace in huge file size. Programs that run in high privilege could also make a tracing tool fails to work.

Third, tracing could be detected by the target program without any time defense. For example, binary instrumentation based tracing tools can be detected at runtime with various tricks [12].

Trace files of modern programs are often in large file size. It is not an easy work for processing large trace files. The work load of our approach depends on the number of instructions in a trace file. There are two feasible approaches will improve the performance of our approach. The first one is only recording relevant instructions in a trace file. For example, it is no need to analyze instructions executed before the entry point of the target program. The second one is using parallel processing to share the work load to multiple processors.

6. RELATED WORK

There is a considerable body of literature on both anti-analysis/anti-tampering defenses in software as well as on the detection and neutralization of such defenses. In the interests of brevity we focus here only on the related work on defeating such defenses.

The only other work we know of that looks at attacks on self-checksumming code is that of Wurster *et al.* [40, 42], who exploit an assumption underlying self-checksumming approaches that the same byte values will be retrieved from a virtual memory address range regardless of whether it is retrieved as code or data. They show that a adversary hardware assisted techniques to violate this assumption and bypass the self-checksumming defense. Giffin *et al.* show that self-modifying code can be used to detect this attack [14]. Unlike Wurster *et al.*'s attack, the work we describe is a pure-software approach that does not rely on hardware assistance.

A number of researchers have investigated the problem of detecting environmentally-dependent behavior in malware. Brumley *et al.* [2] use a combination of dynamic binary instrumentation and mixed symbolic and concrete execution, to identify behavior that is dependent on environmental triggers. Crandall *et al.* use a combination of VM-based timer perturbation and symbolic execution to discover time bombs in malware [8]. Lindorfer *et al.* [23] and Balzarotti *et al.* [1] discuss detecting environment-dependent behavior in native malware by comparing multiple executions in different environments.

There is a wide body of literature on various forms of taint analysis and their applications to software analysis, e.g., see [34]. Cavallarro *et al.* [5] and Sarwar *et al.* [33] discuss approaches for defeating taint analyses, e.g., using implicit information flows.

7. CONCLUSION

Malicious programs often use a variety of anti-analyses defenses to make it harder to analyze their code. Previous work on detecting and neutralizing such defenses has typically focused on specific kinds of defenses, without attention to underlying similarities between different kinds of anti-analysis defenses. This paper describes an information-flow-based framework for understanding a wide variety of anti-analysis defenses, and shows how self-checksumming-based anti-tampering defenses as well as timing-based anti-emulation defenses can be understood as instances of this framework. Experimental results from an initial prototype implementation of our approach shows that taint-based (and, more generally,information-flow-based) dynamic analysis can be effective in identifying and understanding such defenses.

Acknowledgments

This research was supported in part by the Air Force Office of Scientific Research (AFOSR) under grant no. FA9550-11-1-0191 and the National Science Foundation (NSF) under grants CNS-1115829, CNS-1145913, III-1318343, and CNS-1318955. This research was also funded by National Natural Science Foundation of China (NSFC) 61173021. The opinions, findings, and conclusions expressed in this paper are solely those of the authors and do not necessarily reflect the views of AFOSR, NSF or NSFC.

8. REFERENCES

[1] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[2] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. 2008.

[3] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Detection: Countering the Largest Security Threat*, volume 36 of *Advances in Information Security*, pages 65–88. Springer, 2008.

[4] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere. Towards tamper resistant code encryption: Practice and experience. In *Information Security Practice and Experience*, pages 86–100. Springer, 2008.

[5] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*, July 2008.

[6] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[8] J. Crandall, G. Wassermann, D. de Oliveira, Z. Su, S. Wu, and F. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *Proc. 12th. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, Oct. 2006.

[9] A. Danielescu. Anti-debugging and anti-emulation techniques. *CodeBreakers Journal*, 5(1), 2008. `http://www.codebreakers-journal.com/`.

[10] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, Oct. 2008.

[11] _dose. Anti-anti-debugging tricks, Mar. 2000. `http://www.hcunix.net/papers/dose_anti_anti_debug.html`.

[12] F. Falcon and N. Riva. Dynamic binary instrumentation frameworks: I know you are there spying on me. `http://recon.cx/2012/schedule/events/216.en.html`.

[13] P. Ferrie. Anti-unpacker tricks. In *Second CARO Workshop on Packers, Decryptors, and Obfuscators*, May 2008.

[14] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[15] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and privacy in digital rights management*, pages 141–159. Springer, 2002.

[16] F. Howard. Malware with your Mocha? Obfuscation and anti-emulation tricks in malicious JavaScript. Technical report, Sophos Labs, Sept. 2010.

[17] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[18] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security*, Nov. 2009.

[19] Kaspersky Lab. The epic snake: Unraveling the mysteries of the Turla cyber-espionage campaign. 7 Aug 2014. `http://www.kaspersky.com/about/news/virus/2014/Unraveling-mysteries-of-Turla-cyber-espionage-campaign`.

[20] B. Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, Jan. 1997.

[21] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

[22] D. Kushner. The real story of Stuxnet. *IEEE Spectrum*. 26 Feb. 2013.

[23] M. Lindorfer, C. Kolbitsch, and P. Comparetti. Detecting environment-sensitive malware. In *Proc. 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 338–357, Sept. 2011.

[24] G. Lu and S. Debray. Weaknesses in defenses against web-borne malware (extended abstract). In *Proc. 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2013.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, June 2005.

[26] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.

[27] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[28] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. 23rd Annual Computer Security Applications Conference (ACSAC)*, pages 421–430, Dec. 2007.

[29] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[30] Oreans Technologies. Themida: Advanced windows software protection system. `http://www.oreans.com/themida.php`.

[31] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.

[32] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT '09)*, Aug. 2009.

[33] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[35] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. 15th Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.

[36] C. Song, P. Royal, and W. Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proc. 7th USENIX Conference on Hot topics in Security (HotSec'12)*, Aug. 2012.

[37] StrongBit Technology. EXECryptor – bulletproof software protection. `http://www.strongbit.com/execryptor.asp`.

[38] G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In *Information Hiding*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2006.

[39] H.-C. Tsang, M.-C. Lee, and C.-M. Pun. A robust anti-tamper protection scheme. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference*

on, pages 109–118. IEEE, 2011.

[40] P. C. Van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):82–92, 2005.

[41] P. Wang, S. Kim, and K. Kim. Tamper resistant software through dynamic integrity checking. In *Proc. 2005 Symposium on Cryptography and Information Security (SCIS2005)*, Jan. 2005.

[42] G. Wurster, P. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.

[43] B. Yadegari and S. Debray. Bit-level taint analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014.