Emulating Emulation-Resistant Malware

Min Gyung Kang
CMU / UC Berkeley
mgkang@cmu.edu

Heng Yin
UC Berkeley
hyin@cs.berkeley.edu

Steve Hanna
UC Berkeley
sch@cs.berkeley.edu

Stephen McCamant
UC Berkeley
smcc@cs.berkeley.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

ABSTRACT

The authors of malware attempt to frustrate reverse engineering and analysis by creating programs that crash or otherwise behave differently when executed on an emulated platform than when executed on real hardware. In order to defeat such techniques and facilitate automatic and semi-automatic dynamic analysis of malware, we propose an automated technique to dynamically modify the execution of a whole-system emulator to fool a malware sample's anti-emulation checks. Our approach uses a scalable trace matching algorithm to locate the point where emulated execution diverges, and then compares the states of the reference system and the emulator to create a dynamic state modification that repairs the difference. We evaluate our technique by building an implementation into an emulator used for in-depth malware analysis. On case studies that include real samples of malware collected in the wild and an attack that has not yet been exploited, our tool automatically ameliorates the malware sample's anti-emulation checks to enable analysis, and its modifications are robust to system changes.

1. INTRODUCTION

Analyzing malicious software, such as viruses, worms, and botnet clients, whether fully automatically or with human assistance, is a critical step in defending against the threat such malware poses. For instance, knowledge of the possible behaviors of malware and how it chooses among them is key to proactive defense and forensic investigation. Automated analysis based on emulated execution is particularly important for tasks such as unpacking [6, 18, 22] (obtaining de-obfuscated malware code), behavioral signature generation [3] (classifying a malware sample based on the malicious actions it performs), and information flow tracing [13, 31] (for instance, to understand spyware behavior).

However, malware authors are motivated to make analysis difficult: to slow the development of defenses that make their malware obsolete, and also to protect proprietary information from reverse-engineering by competing malware authors. For this reason, anti-debugging and anti-emulation techniques are widespread in modern malware (more than 40% shown in one recent study [9]). A malware sample that detects it is running in an analysis environment can simply terminate, refrain from its usual malicious activity, or take any other action to frustrate analysis. In this work, we propose a new technique towards addressing the problem of how to execute a malware sample in an environment based on software emulation, when the malware incorporates anti-emulation techniques as are now common.

We say that an emulator suffers a failure of *transparency* if there is an aspect of its behavior that allows software running on the emulator to distinguish it from real hardware. Emulators based on binary translation avoid some classes of transparency failures, since they can completely replace instructions, but they are complex enough that making them completely transparent would be impractical. Thus, our goal is to successfully emulate a malware sample, even if an emulator suffers from transparency failures that anti-emulation techniques would normally exploit.

To that end, we formulate the problem of emulating emulation-resistant malware in terms of its environment observations, and propose an approach of *comparison-based state modification* that modifies the execution state of a malware sample to simulate a different environment and so thwart the sample's anti-emulation techniques. In particular, our approach diagnoses the source of anti-emulation behavior by comparing an execution of a malware sample on an emulation-based analysis tool to its execution on a more transparent reference system that would be impractical to use directly for analysis. Based on the differences, our approach constructs a dynamic state modification (or DSM) that indicates how to modify an execution occurring in an emulator to fool anti-emulation checks. We apply the DSM to automatically ameliorate the transparency failures in the emulation tool so that it can be used for automated analysis or tool-supported reverse engineering. The reference system can be any hardware or software platform that matches one aspect of real hardware behavior very closely; our approach allows a general-purpose emulator to enjoy the same fidelity.

We have implemented our technique in the form of enhancements to the popular open-source whole-system emulation system QEMU [26], which is the basis for many malware analysis platforms [2, 4, 22]. As a reference platform, we take an existing virtual execution system based on Intel VT hardware virtualization that achieves high transparency for CPU semantics by executing each instruction directly on a real processor in single-step mode [12]. This reference platform has a high performance overhead, and would not be a convenient platform for building more sophisticated analysis tools, but our technique allows the use of anlyses built using QEMU with the transparency benefits of the reference platform.

Our specific focus is on the emulation-resistance problem as it is most acute for practicing malware analysts working with modern malware. Thus it is important that the technique work in practice and at scale. We evaluate our prototype using real malware samples collected from the wild: complete malicious applications, several of which use other obfuscations like packing or multiple resistance techniques. We ensure that our tool introduces minimal runtime overhead compared to the analysis tools already being used. We also check that the tool's changes are robust: they work not just for a single malware execution but for a range of executions as are often required in analysis.

Achieving transparent execution in a general way is very diffi-

```
functions[0] = &steal_passwords;
long t1 = cycle_count();
long t2 = cycle_count();
long diff = t2 - t1;
long copy = diff;
if (diff < 5)
    abort();
(functions[diff/256])();
/* ... */
assert(diff == copy);
```

Figure 1: C-style pseudo-code for a timing attack (compare to a real sample in Figure 3). `steal_passwords` **will be called only if the time elapsed between the two calls to** `cycle_count` **is between 5 and 255.**

cult; we do not claim that our approach is powerful enough to solve the problem of emulation resistance for all time. Malware authors will constantly be on the lookout for new emulation-resistance techniques. Our approach works automatically against current instances of broad classes of emulation-resistance techniques, and we can be confident it will perform well against novel instances in those classes. This is possible because the technique is not based on the details of specific attacks: it is a general approach that discovers the cause of a failure on its own.

It is harder to predict what further defense techniques may be needed if malware authors shift to entirely new classes of anti-emulation technique, perhaps specifically choosing attacks that would be difficult for a technique like ours. (We suggest some possible next steps attackers might take in Section 5.) Even if our approach is unable to automatically ameliorate an attack, its diagnosis would save a human analyst time by showing its location and cause. Our approach to efficient diagnosis is also of potential independent interest for other applications, such as diagnosing a misconfiguration that interferes with a system's correct operation.

Contributions. In summary, this paper makes the following contributions:

- We give a specific formulation of the problem of how to ameliorate anti-emulation checks in a running program.
- We propose an amelioration approach based on diagnosing differences between execution traces to construct a dynamic state modification addressing the precise observations a malware sample makes.
- We build an implementation of our approach in an open-source emulator widely used for malware analysis.
- We evaluate our technique against a set of real malware samples, analyzing the emulation-resistance techniques they use and verifying that our DSMs are effective and robust against environment changes.

2. PROBLEM FORMULATION

Figure 1 gives an example of a timing attack similar to one we have encountered in the wild, but given in a C-like pseudo-code (for the real example this is based on, see Section 6). When run on real hardware, the number of cycles between the two calls to `cycle_count` will usually be between 5 and 255, so the code will call `steal_passwords`. However, on an emulator, more time is likely to elapse, since each instruction must be translated before it can be first executed. So when run under an emulator, the time difference will be larger, and the malware will execute a different function or crash. This anti-emulation check is typical of the code we would like to emulate automatically.

Problem Formulation. To formulate the problem of ameliorating emulation resistance at a high level, we describe a program and its observations of the environment as simple functions. The behavior of a program P, including its explicit outputs and any other side-effects, is a function of its explicit inputs I, as well as the environment E; we write $P(E, I)$. This environment covers all the aspects of the system the program might query or discover, including the other software installed, the characteristics of hardware, or the time of day. If P uses anti-emulation techniques, then its behavior will be different under an emulated environment E_e than under a real hardware environment E_r, even if those environments are otherwise similar and the program's explicit inputs are the same: $P(E_e, I) \neq P(E_r, I)$.

To understand such differences, we consider more of the program's structure. In general, a program with anti-emulation behavior will first observe an aspect of its environment that differs, then decide whether or not to take a later action based on this observation; we call this complete process an *anti-emulation check*. The program's observations of its environment can be divided into those made in an anti-emulation check, and all other observations. Functionally, we decompose the program P into an anti-emulation check f, other environment observations g, and a balance P', where $P(E, I) = P'(f(E, I), g(E, I), I)$. The function f captures those checks that distinguish the emulated from the real environment, so $f(E_e, I) \neq f(E_r, I)$. For instance, the code shown in Figure 1 implements such an f. By contrast, g contains all the remaining environment observations that are not part of any anti-emulation techniques. For instance, the program in which Figure 1 is embedded might check that it is running an OS no older than Windows 2000. These other aspects of the environment can be controlled independently of whether an emulator is in use or not, so there are some emulated and real environments E_{e_1} and E_{r_1} for which $g(E_{e_1}, I) = g(E_{r_1}, I)$. Note that in practice there will be many incidental differences between the emulated and reference environments; for instance, they might have different sized displays. So in general, $g(E_e, I) \neq g(E_r, I)$ as well; but it is not our goal to change any differences unrelated to an anti-emulation check.

Thus, we formulate the problem of emulating the malware sample as constructing an emulator that behaves as the sample expects with respect to the anti-emulation check, but is otherwise unchanged. In other words, this modification $E_{e'}$ of E_e satisfies $f(E_{e'}, I) = f(E_r, I)$ (the anti-emulator check is fooled), even though $g(E_{e'}, I) = g(E_e, I)$ (the environment is otherwise the same). For instance, an environment would fool the check shown in Figure 1 if the two time values it returned differed by an amount between 5 and 255.

Anti-emulation techniques addressed. We classify the anti-emulation checks f performed by malware into three broad types:

Timing attacks measure the time that elapses during an operation, on the assumption that an operation will take a different amount of time under emulation than on real hardware. The code in Figure 1 is an example.

CPU semantics attacks target CPU instructions whose behavior in an emulated system differs from their behavior in real hardware.

Hardware characteristic attacks query features or identifiers of a computer's hardware, looking for emulated hardware that is different from any real physical hardware.

The techniques we introduce are applicable against all three types of attack, but the first two present more interesting technical challenges, appear to be more prevalent, and are better suited for our reference platform, so we concentrate on them. Also, we exclude from our scope anti-emulation checks that involve an external host; many do not.

3. APPROACH

In an ideal world, the problem of anti-emulation techniques could be eliminated with an emulator whose behavior was exactly like that of real hardware (i.e., was completely transparent): if $E_e = E_r$, then $P(E_e, I) = P(E_r, I)$ for any P and I. However, this approach would not be practical: achieving complete transparency would be an enormous engineering effort and would be incompatible with other practical requirements for an emulation tool. For instance, cycle-accurate timing simulation is impractically slow.

Intuition. Instead, we propose an approach to achieve emulation of emulation-resistant malware samples under a practical emulator by modifying only a few aspects of the emulator's behavior. In particular, the approach should automatically diagnose and ameliorate new attacks that malware authors might devise. To make this approach work, we must answer two fundamental questions: which aspects of the emulator's behavior should we change, and how should the changes be implemented?

For selecting aspects of the emulator's behavior, our approach is guided by the malware sample itself: we ameliorate those aspects of the emulator that the malware sample uses to detect that it is running under an emulator. For modifying the emulator, our approach is also based on the way the malware sample observes its environment: we modify the state of the emulator at the point the malware sample makes an observation, to change the sample's effective observation. Much as gaze-directed rendering in computer graphics only has to draw a detailed image at the place a viewer is looking [21], these observation-based approaches only have to present an accurate model of those system aspects malware observes.

Reference platform. Of course, in order to recognize that malware is behaving differently in an emulated environment and to properly change those observations, our system requires an environment like the real one for comparison. Thus our approach uses what we call a *reference platform*: another execution platform that more closely resembles the behavior of a real hardware system, but allows that behavior to be recorded. The reference platform may not be a good basis for analysis tools, and it may be expensive to run, so it would not be practical to use it repeatedly for analysis, but we presume that we can run it once to obtain a trace from a correct execution. Examples of such accurate but expensive reference platforms include a cycle-accurate simulator for CPU timing attacks, a system with some specific hardware expected by a hardware-characteristics attack, or a hardware-level snooping tool like a logic analyzer or dedicated PCI card [7, 8].

For our experiments, the reference platform we use is a whole-system virtual machine that uses hardware virtualization in single-step mode [12]. This platform provides very good transparency for CPU semantics, at the expense of high runtime overhead. Hardware virtualization is not naturally transparent for timing attacks, but the implementation we use has some support for simulating CPU time independently from real time. But the reference platform uses most of the same virtual hardware as our emulator, so it would not be useful against hardware characteristic attacks. Such attacks could be addressed with a reference platform that allowed real hardware to be accessed directly. Our reference platform would also be impractical to use for detailed analysis, since it provides no support for instruction rewriting.

Approach overview. Our approach is a two-step process. To determine whether the malware sample is detecting the presence of an emulator, our approach looks to points where the behavior of the malware sample differs between the reference and emulated platforms, and determines what differences between the two platforms were the cause of that behavior difference. Then, to modify the emulated platform as observed by the malware sample, our approach changes the malware sample's observations of its environment to match the observations it makes on the reference platform. However, we do not simply wish to modify the behavior of a single malware sample execution to match the reference platform. It is also important that the modifications we obtain are *robust*: that is, they are still effective for other executions of the same malware sample, with different inputs or environment modifications, since repeated experiments are often needed in analysis.

The emulator modifications our technique computes are specific to a particular malware sample: our goal is not to build a perfect emulator once and for all. But for a given malware sample, the technique determines the root cause of differing behavior and a way to ameliorate it automatically, so it is not necessary for an analyst to understand the malware sample's emulator detection code, which might use a previously unknown technique. The emulator can then be used as normally for automated or human-assisted analysis, reverse-engineering, and defense against that sample.

System outline. We propose an approach of *comparison-based state modification* that modifies the execution state of the emulator so that the observations made by the anti-emulation check match the observations made on a *reference platform* where the malware sample executes as it does on real hardware. The changes to the emulator's execution state are represented as a *dynamic state modification* (DSM for short) that gives new values to specified execution state components (e.g., registers, memory locations, or flags) at specified points in execution. We call the DSM dynamic because it is not a change to the code of the malware sample: it represents a transient alteration to values during the execution of the sample. Note we do not remove the anti-emulation check (which would be difficult to do either statically or dynamically); instead the DSM changes the check's environmental observations to force its result.

For instance, for the example of Figure 1, a DSM might specify that the calls to `cycle_count` on lines 2 and 3 should return 3363487834 and 3363487917 respectively (our real DSMs would use an instruction address). Our approach first constructs a DSM by analyzing and comparing a trace of malware executing on a reference platform with one or more traces of its execution on the emulated platform. Then, the DSM can be applied on any future execution of the emulated platform, replacing the value a location would normally have under the emulator with the value specified in the DSM. For instance, in executing the code in Figure 1, the emulator will get to line 2 and realize that the DSM applies, so rather than setting `t1` to the real time (say, 4104148387), it will instead use the DSM value of 3363487834; in the same way, it will substitute 3363487917 for 4104149461 at line 3. The code will then continue its execution normally, but because the DSM has changed its environment observations, it will behave differently: it will compute a difference of 83 cycles rather than 1074, and it will execute the malicious code in `steal_passwords` instead of `functions[4]`.

Our tool constructs a DSM in two steps, shown in Figure 2. It first compares execution traces from the emulated platform and the reference platform to locate a point where emulated execution differed from the reference execution, then constructs a DSM to correct that difference. A *divergence point* is a point in execution such that directly before that point, the emulator and the reference platform executed the same instruction, but directly after that point, they execute different instructions. Therefore we call these two

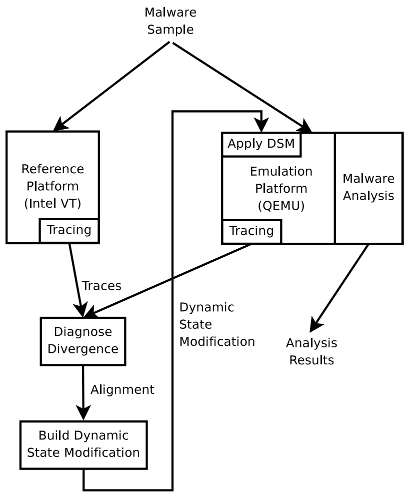
Malware Sample
Reference Platform (Intel VT)
Tracing
Apply DSM
Emulation Platform (QEMU)
Tracing
Malware Analysis
Traces
Dynamic State Modification
Diagnose Divergence
Analysis Results
Alignment
Build Dynamic State Modification

Figure 2: Diagram giving an architecture overview of our system.

steps *diagnosing* a divergence point and *building a DSM* from such a divergence.

The purpose of these steps is to pinpoint the root cause of a behavior difference. The diagnosis step locates a *coverage divergence point* by looking for a difference in code coverage (i.e., an instruction that was executed by the emulator but not the reference platform), and then aligning (matching) the portions of the execution traces before the divergence to isolate any earlier events that may have differed. Then, the DSM-building step uses this divergence point and alignment to trace backwards, linking each differing value either to a previous value, or to a differing environment observation. The environment observations that caused the divergence points are the ones that the DSM will modify. If the malware sample uses multiple anti-emulation techniques, our tool will first generate a DSM from the first divergence point, then re-run the emulator with it, and then repeat these steps as many times as needed to create a DSM that ameliorates them all.

For instance, in the example of Figure 1, suppose that on the unmodified emulator, the timing difference `diff` is 1074 cycles. Our diagnosis algorithm will recognize a divergence after the function pointer call, when the code starts to execute `functions[4]` rather than `steal_passwords`. To construct a DSM, our algorithm will trace backwards from the control-flow divergence to its root cause, which are the two calls to `cycle_count`, and create a DSM that replaces the emulated counts with those from the reference platform.

Trace collection. Because we need traces for diagnosis, we add support for instruction-level traces to our reference and emulated platforms. For each instruction executed, the trace records its address, disassembles it, and records its operands, their values, and any results or side-effects. The trace also records hardware-level exceptions, interrupts, and traps. Both platforms can collect a trace of all instructions executed, both in the operating system and any processes. But they also support collecting traces from a single user-level process, and filtering to include only instructions from the program image itself, excluding system libraries; we usually use this more selective mode. In such a case system calls and library routines are treated as atomic actions.

Applying a DSM. We modify the emulated platform to apply a DSM to the execution of a malware sample. The DSM consists of a list of changes, each of which is identified by an instruction address (potentially along with a calling context or other information to disambiguate a particular execution). In a callback that executes at each instruction, our modified emulator checks whether any changes in the DSM apply to the instruction: if so, it modifies the appropriate part of the emulator state or calls a routine to trigger an exception before the next instruction.

4. TECHNIQUES

In this section we discuss further details of the key components of our technique. For the central steps of divergence diagnosis (Section 4.1) and building a DSM (Section 4.2), we first outline our requirements, and then discuss our technical approach. We also give some further implementation details (Section 4.3).

4.1 Diagnosing a Divergence

4.1.1 Intuition and Requirements

Given two instruction traces, one from the reference platform and one from the unmodified emulator, we wish to diagnose the source of the difference between the two traces by locating the control flow divergence point caused by the code's use of an anti-emulation technique. This divergence is a point in each trace such that just before the point, the traces are executing the same code, but just after, they are executing different code. A divergence can occur for several reasons. A divergence point need not come directly after a branch: many anti-emulation attacks are based on exceptions anywhere. This divergence point also may or may not be the root cause of the divergence: some root causes trigger a divergence right away (for instance, with an exception), while others simply cause a change in the machine state that the malware sample checks later. For instance, in the example of Figure 1, the divergence occurs on line 8, while its root causes are on lines 2 and 3.

The key challenge is to locate this divergence point quickly, but to find only the *relevant* divergence. Instruction traces can be large (up to half a gigabyte in our experiments), so we must be able to locate a divergence with a low computation cost. But traces also contain irrelevant differences that do not result in divergence, caused for instance by loops that process variable-length data, context switches between threads, and system call behavior differences. (Our emulators do not use true parallelism, so a multi-threaded program yields a single interleaved trace. The malware samples we evaluate are single-threaded.) An approach that had to consider every minor difference between two traces would bog down.

Along with finding a divergence point, our diagnosis also aligns the portions of the traces that come before the divergence point. An *alignment* between two sequences (of instructions, in our case), is a maximal matching between occurrences of the same instruction in each trace, such that the matched occurrences occur in the same order in each trace. The alignment provides a definition of which instruction execution in one trace corresponds to an execution of the same instruction in the other trace, which our technique will use later when searching for a root cause.

The classic approach to alignment, as performed by the `diff` utility, is via a dynamic programming algorithm that computes the longest common subsequence of two sequences (e.g., [23]). However, using such an algorithm directly on full traces would be too slow in our context, since the traces are very large and the portion after the divergence may be both long and contain few matches.

4.1.2 Our Approach

Instead of using alignment directly, our diagnosis approach has two steps: it first locates a point of control-flow divergence using a coverage-based heuristic, and then performs a more limited alignment on the portion of the trace prior to the divergence.

Finding a divergence point. To locate a control-flow divergence point that is relevant to an anti-emulation attack, our approach uses code coverage as a heuristic; to be specific, we call the result a *coverage divergence point*. Irrelevant differences generally cause the program to run the same code, just at a different time. By contrast, if the emulated code executes instructions the reference platform never did, that likely indicates anti-emulation behavior. Also, our tool's default behavior is to look for a divergence point only in the malware code itself, not the operating system or standard libraries, since a difference that never causes the malware sample to change its execution is usually not an anti-emulation check.

Thus we locate a coverage divergence point by comparing the two traces, finding the first instruction that appears in the emulated trace but not in the reference trace. Here and elsewhere, we say that two instructions are the same if they had the same program counter location (EIP, in x86 terminology) and the same instruction bytes; we include the latter condition because malware sometimes overwrites its own instructions. Our tool first constructs the sets of instruction locations (PC values) seen in the reference trace. Then, processing instructions from the emulated trace in order, it signals a possible divergence point if the previous instruction occurred in both traces, but the current instruction appeared only in the emulated trace. (If the previous instruction occurred at multiple positions in the reference trace, we consider each occurrence, in order of increasing trace position distance.) For instance, in the example of Figure 1, the divergence point comes after line 8, since after that the reference platform executes the first line of `steal_passwords`, while the emulator executes a different function.

If our algorithm finds multiple divergence points, our tool processes them in trace order. We have found empirically that the first candidate is most often the real divergence point.

Finding an alignment. After locating a divergence point, our approach builds an alignment over the subset of each trace between the beginning of execution and the divergence point, for use in isolating the root cause. Because we expect that this region will have relatively fewer differences, we use an $O(nd)$ algorithm [23] which performs well when the number of differences d is small.

4.2 Building A Dynamic State Modification

4.2.1 Intuition and Requirements

In order to correct an emulation infidelity that allows a malware sample to detect the presence of an emulator, we construct a *dynamic state modification*: information that allows the emulator to correct its behavior when executing the relevant malware code. A DSM is a list of changes, and each change consists of two components: the specification of a location in the malware execution (the address of an instruction), and a set of new values for parts of the emulated machine state, such as registers, processor flags, or memory locations. There may be many DSMs that would correct a divergence, but we have several reasons to prefer one DSM over another. Our intent in creating a DSM is to give the malware sample the illusion that it is running in a different environment, so we attempt to change the program's state at the moment it makes an environment observation such as a time measurement. Failing that, we modify other program state, but we want the changes to be consistent, so that the malware sample cannot detect that its state has been selectively changed. To make a robust DSM that can be used on many executions of the malware sample, we want to make only those changes needed to prevent an anti-emulation check.

To understand the possibilities for DSMs, consider again the example timing attack of Figure 1, which is typical of attacks that extend over a period of execution. First, the malware measures the amount of time some operation requires by asking for the time before and after performing the operation. Next, it computes the time by subtracting those two time values, and finally it makes a control-flow decision based on that difference. The divergence involves all of the data flow from the initial time measurements, which are the root cause of the difference, to the final control-flow check. Any DSM that changed the values on all of these flows would correct the final control-flow decision.

Consistent and minimal DSMs. Modifying the internal values of a program in the middle of its execution has a potentially serious disadvantage: it might leave the program in a state that it could not reach for any environment and inputs. A malware sample might be able to use such a situation to detect that it had been tampered with, and in any case, it could be misleading for further analysis. For instance, in the example of Figure 1, if a DSM corrected the value of `diff` only right before it was used in on line 8, this value would be different from the value saved in `copy`. (This could be fixed with another iteration, but we would like to avoid the need to.)

Thus we create a consistent DSM by fixing differences in machine state at their origins; what refer to as a *root cause*. An instruction (or library routine or system call) is a candidate root cause for the difference between two traces if it was executed in both traces, and all of its inputs were the same between the two traces, but its outputs were different between the two traces. Among the outputs of an instruction, we include its explicit result (i.e., destination operand), but also any side-effects (such as to condition code flags), and any hardware exceptions the instruction might raise. For instance, an instruction can be a root cause if the semantics of the instruction were incorrect in one trace (for instance, by raising an exception in only one trace), or if the instruction takes an implicit input from outside the normal machine state (for instance, querying a timer). If we exclude the operating system kernel or libraries from our traces, then a system call or library routine return value may also be a root cause. However, not all sources of difference between two traces are related to an anti-emulation attack, so our tool searches specifically for causes related to the observed divergence.

It is also important to make only minimal changes to the machine state in a DSM. We do not assume that all aspects of the reference and emulated platforms can be made exactly equal; in addition to the differences that a particular malware sample detects, many other pieces of observable machine state will differ, even between multiple runs of the same software on the same platform. These include the exact time of day the execution occurs, miscellaneous details of the (simulated) hardware on each platform, the timing of user input events that start the malware sample, and many others. Controlling all of these sources of nondeterminism would be a major engineering challenge, and we do not attempt it.

Modifying too many values would also hurt our goal of robustness: if a DSM modifies a program input or an unrelated environmental feature, then the DSM could not be used for analysis experiment that changed that input. In the extreme case, requiring that every value in the emulated trace be the same as the reference platform would completely destroy robustness: it would allow the analysis of that single trace, but any modified experiment would require re-running the reference platform, which we want to avoid. Instead, our approach is to modify only those parts of the state that the malware appears to be using in an anti-emulation technique.

4.2.2 Our Approach

Given an alignment leading to a divergence, our tool next builds a DSM to prevent the divergence. To avoid requiring multiple iterations, our tool tries to create a consistent DSM by modifying the state as close as possible to the root cause of the difference. It finds this root cause by working backward through the dependencies between instructions, much like a backward dynamic slice [30]. But because it performs this slice in parallel on two aligned traces, it can use the similarities and differences between the traces as an additional source of information.

Dependencies. The DSM construction algorithm traces backwards across dependencies between instructions. The most common kind of dependency is a direct *data dependency*, in which a value is written by one instruction and read by a later one. But by using an alignment between the traces, our algorithm can also discover certain *control dependencies*, situations in which a location takes a different value because different code executed earlier in the trace. For instance, in the code `if (c) x = 1; else x = 2;`, we say that `x` has a control dependency on `c`. If the execution of a branch differs between the two traces, this will often show up in the structure of the alignment: instructions will be aligned up until the differing branch, but the two sides of the branch cannot be aligned. Thus in this situation our algorithm treats the value modified on one execution but not the other as control-dependent on the branch condition, so it will consider the branch condition as another possible step on the way to the root cause.

Worklist algorithm. Our algorithm works by maintaining a worklist, consisting of a set of corresponding pairs of values that were different in the two traces. To start, the worklist contains the differing values that caused the divergence (e.g., the branch condition values if the divergence point was a branch). For instance, in the example of Figure 1, if we represent such a difference by a triple consisting of an expression, its value in the reference trace, and its value in the emulated trace, the initial element of the worklist is

$$(\texttt{functions[diff/256]}, \texttt{steal\_passwords}, \texttt{functions[4]})$$

Then, the algorithm repeatedly removes a differing value from the worklist, and finds the instructions in the two traces that produced it. Usually, these will be the same instruction, so the algorithm will compare the corresponding inputs to see if any were different in the two traces. If there were no differing inputs, this instruction is a root cause: for instance, for the difference $(\texttt{cycle\_count()}, 3363487834, 4104148387)$, since the `cycle_count` has no inputs, it must be a root cause. Otherwise, if there were differing inputs, those inputs are added to the worklist: for instance, in Figure 1, the difference $(\texttt{t2 - t1}, 83, 1074)$ is caused by differences in the inputs `t2` and `t1`, so they are both added to the worklist. If the instructions differ, the algorithm will attempt to find control dependencies that connect them to a common branch; failing that, it will compute single-trace slices for each value separately. When the worklist is empty, all the root causes have been identified, and the tool builds a DSM which, for each root cause, replaces the differing value with the value it had in the reference trace. For instance, the root cause $(\texttt{cycle\_count()}, 3363487834, 4104148387)$ yields a DSM specifying that on line 2, the return value of `cycle_count` should be set to 3363487834.

4.3 Implementation

Here we cover some remaining details of the implementation of each step of our technique.

Collect execution traces. We have built our emulated platform as a set of enhancements to QEMU [26], and as a reference platform we use Ether [12], based in Intel VT [16]. We added support to both to collect execution traces, using the Intel XED library to disassemble and obtain an operand list for each instruction, necessary for accurate slicing. Ether's single-step mode already traces the program counter; we extend it to obtain the other information we use.

Diagnose a divergence. We implemented the coverage-based diagnosis algorithm and $O(nd)$ alignment algorithm described in Section 4.1 in C++.

Build a DSM. We implemented the comparison-based back slicing algorithm explained in Section 4.2.2 as a tool in a combination of OCaml and Python that takes two execution traces and alignment information as inputs, and produces an instruction slice. The sources in this slice represent the root causes for which we construct the dynamic state modification.

Apply a DSM. We modify QEMU to apply a DSM. Modifications to registers and CPU flags are simple changes to the emulator's state. To modify a memory value, we translate the virtual address to a physical address using the page table; if the page is not present, we postpone the modification until after a page fault. For interrupts and other hardware exceptions, we set a flag that causes QEMU to deliver the exception before the next instruction execution.

5. DISCUSSION, COUNTERMEASURES

We have found our technique to be very effective against malware currently in the wild. In this section we discuss some further characteristics of our approach, including some potential countermeasures malware authors might take against the techniques we introduce.

Virtual execution for malware analysis. Some kind of virtual execution, such as a debugger, virtual machine, or emulator, is indispensable for malware analysis, but approaches for observing malware execution differ in their performance overhead, the ease of building analysis tools using them, and their transparency. In this work we build on the approach emulation via of software-level binary translation, since it already represents an attractive tradeoff. An emulation-based system resists most anti-debugging techniques, as well as anti-VM techniques based on incomplete virtualization [28]. Systems that perform binary translation also make it relatively easy to build sophisticated dynamic execution analysis tools, since they support code rewriting and have a model of instruction semantics.

Changing non-attack data. Our system attempts to only change those aspects of the program's environment that are used in an anti-emulation attack, but if there are other environmental differences that lead to code coverage differences, our approach may "ameliorate" them unnecessarily, leading to a less robust DSM. We have not observed this to be a problem in practice, but if it became one, several approaches could help.

First, if an analyst anticipates wanting to modify an environment aspect on later emulation runs, the analyst can set that aspect equal to the value it has on the reference platform when generating the DSM. Second, we could extend our DSM creation approach with a new post-processing step that, given a DSM that prevents attacks, simplifies it by removing changes whose removal does not affect its attack-prevention ability.

Attacks for which there is no reference platform. Our technique depends on the existence of suitable reference platform on which the malware behaves correctly. If an attack exists for which no possible reference platform shows the correct behavior, our technique would not be applicable. We are not aware of any such attacks, but there are attacks for which the reference platform our implementation currently uses is unsuitable.

For instance, one such attack involves the string store instruction

Sample Label	# of Samples	Attack Category	Root Cause
W32/Sdbot.worm	2	timing	timestamp counter
Generic.dv	1	timing	timestamp counter
multirep	1	CPU semantics	malformed opcode
Downloader-AFH	1	CPU semantics	FPU register
Proxy-Bypass	1	CPU semantics	undoc. opcode

Table 1: Summary of malware samples. Labels are from the McAfee scanner for examples from the wild; "multirep" is our name for the example of Dinaburg et al. discussed in Section 6.2.1.

`rep stos`, which happens to be executed incorrectly (stopping too soon) by both QEMU and our reference platform when it is used to overwrite its own instruction bytes. But our technique would apply to the attack if given a trace from an accurate reference platform (such as an emulator without this bug, like Bochs [5]).

Too many divergence points. Though our approach uses iteration to correct multiple anti-emulation attacks in a single execution, each iteration is relatively expensive. Therefore, a malware author could frustrate the use of our tool by introducing a very large number of divergence points. To make this countermeasure more difficult, our tool could attempt to correct all the attacks of a particular kind in a single iteration.

Obfuscating data flow. Malware authors could interfere with our diagnosis and slicing algorithms by obscuring the data flow between root causes and branches. Some potential techniques include mixing environment observations with with other inputs (e.g., with XOR), or using code constructs that are difficult for a slicer to analyze. In particular, if our technique cannot locate the root cause of a difference, its changes to intermediate states become more vulnerable to consistency checks.

Interleaving detection with primary behavior. Our approach is most valuable if one DSM, based on a single reference platform execution, is sufficient for all malware executions an analyst performs under differing environmental conditions to test different malware behavior. A malware author could make the DSMs produced by our tool less robust by adding more emulation-detection code that ran interleaved and interdependent with other input dependent malicious behavior, so that a DSM could not be reused on other executions that performed different malicious behavior.

Cover failure code in the non-failure case. Our approach for finding the point of control-flow divergence is based on the heuristic assumption that the code that executes after an emulation failure has been discovered is code that does not execute when the malware runs on the reference platform. Malware could interfere with this heuristic by executing the same failure-case code before the detection point (with some other state difference so that it does not affect behavior). This would allow the malware to postpone the detection point as late as the final termination of the program in the emulation case, which would make backwards slicing more difficult.

6. EVALUATION

In this section, we describe the anti-emulation techniques of several malware samples, and show the results obtained by applying our implementation to them. We selected 5 real malware samples and a proof of concept code sample given in [12]. As shown in Table 1, the samples uses two kinds of attack techniques, based on timing differences and differences in CPU semantics. The following two subsections provide evaluation results organized according to these categories.

The evaluation results include the details of the techniques employed by the samples and show the results of DSM generation and the effectiveness of the DSMs applied to QEMU. To verify the effectiveness of the DSMs, we perform two experiments: first, we take an execution trace from the ameliorated version of QEMU and repeat the divergence diagnosis to see if the sample runs as in the reference system. Second, we apply a QEMU-based analysis tool to record the Windows API calls a sample makes, and compare the calls that a malware sample makes before and after applying the DSM. We verify that the DSM successfully fools the malware's anti-emulation check: the samples execute as they did on the reference platform, and display various malicious behaviors. In the last two subsections, we provide a summary of the performance evaluation and demonstrate the robustness of the DSMs; that is, that they are still applicable if the system environment changes.

Choice of samples. Our technique applies to many kinds of malware, but our evaluation focuses on common automatically propagating malware such as viruses and drive-by downloads, since they are common, can be easily obtained from honeypots, and illustrate the need for fast response from malware analysis. We obtained samples of emulation-resistant malware that were submitted to an online malware analysis service, and from other researchers [identities omitted for anonymous submission].

```
0x0048a000: rdtsc
0x0048a002: mov    %eax,%ebx
0x0048a004: rdtsc
0x0048a006: sub    %ebx,%eax
0x0048a008: cmp    $0x10,%eax
0x0048a00b: jl     0x0048a020
0x0048a00d: shr    $0x8,%eax
0x0048a010: add    %eax,(%esp)
0x0048a013: xchg   %eax,%ecx
0x0048a014: mov    %esp,%esi
0x0048a016: lods   %ds:(%esi),%eax
0x0048a017: dec    %cl
0x0048a019: xorb   $0x10,(%eax,%ecx,1)
0x0048a01d: loop
0x0048a01f: ret
```

Figure 3: Timing check in W32/Sdbot.worm(1)

6.1 Category I: Timing Attacks

A timing attack uses an operation that takes a different amount of time on an emulator than a native environment (most commonly the emulator is slower). Among our samples, three (two samples of W32/Sdbot.worm, and Generic.dv) implement timing attacks by comparing values of the timestamp counter during execution.

6.1.1 Comparing Timestamp Counters

Attack Description: W32/Sdbot.worm (in two samples) and our sample of Generic.dv detect a timing difference by executing the `rdtsc` instruction, which loads a 64-bit timestamp counter into `%edx` and `%eax`. The code for W32/Sdbot.worm(1) is shown in Figure 3; this is the real example on which Figure 1 was based. First, it calculates the difference between two consecutive timestamps. Assuming the difference is at least 16, the code divides it by 256 with a shift, then adds it to the return address on the top of the stack. So if the timing difference was 256 or more, the return instruction goes to a different location; usually this location does not hold valid instructions and leads to a crash. The value of the timing difference on real hardware varies with the processor, but was always less than

```
...
48a019: xorb $16,(%eax,%ecx)
48a01d: loop 0x48a019
48a019: xorb $16,(%eax,%ecx)
48a01d: loop 0x48a019
48a01f: ret
48a026: jmp  0x48a03b
48a03b: xchg %eax,%ecx
48a03c: pushf
...
```

Intel VT

```
...
48a019: xorb $16,(%eax,%ecx)
48a01d: loop 0x48a019
48a019: xorb $16,(%eax,%ecx)
48a01d: loop 0x48a019
48a01f: ret
48a072: fwait
[the sample crashes.]
7c90eaec: mov 0x4(%esp),%ecx
...
```

QEMU

Figure 4: Divergence point in W32/Sdbot.worm(1)

100 in our experiments, while the time difference under emulation is much larger: 26740 for one QEMU run.

Divergence Point: By aligning the execution traces from Ether and QEMU, our tool finds the `ret` instruction at `0x48a01f` as the only divergence point. After the return address is modified, the execution of the malware sample in QEMU jumps to a position that does not appear to be reasonable code, and on the next instruction an exception is raised. The trace then shows that the execution path in QEMU is directed to an exception handler.

Root Cause: The DSM generation algorithm automatically slices the execution traces backward from the divergence point at `ret` to find the two root causes: the `rdtsc` instructions at `0x48a000` and `0x48a004`. This confirms the manual analysis described above. This sample also demonstrates the importance of making a change at the root cause: one could simply change the `ret` instruction to use the return address observed on the reference trace, but that would not be effective. Besides affecting the return address, the timing difference is also used to select the location and size of a code region to decrypt by XORing each byte with 16. If this decryption is not modified as well, the jump to the correct location might find still-encrypted code and fail.

DSM and Verification: The DSMs our tool generates for all three timing samples successfully correct all the divergences with the reference platform when applied in QEMU. We also verified that the ameliorated samples displayed the expected malicious behaviors by observing the Windows APIs and memory regions they execute. After applying the DSMs, the sample tries to copy itself to a different location in the file system and create a new process by executing the copied executable image.

6.2 Category II: CPU Semantics Attacks

Because building a software implementation of an entire CPU instruction set is a large task, emulators tend to contain bugs in which a particular instruction is executed differently than it is by real hardware. Malware samples can exploit such differences in a CPU semantics attack. For our experiment we selected four samples demonstrating three different kinds of attack, and evaluate our tool's effectiveness against them.

6.2.1 Malformed Opcode

Attack Description: As an example of their system, the authors of Ether [12] give a proof of concept code sample that uses a 16-byte-long instruction to detect the presence of QEMU; we use the source code provided in appendix A of their paper for our experiment. An x86 instruction may be lengthened by redundant prefixes (for instance, five uses of the repetition prefix `rep` are equivalent to a single one), but the maximum legal length in 15 bytes. In the execution on the real CPU, a violation of this rule should raise a general protection fault exception (number 13), but QEMU fails to enforce the limit. So, when the sample is executed in Ether, it

```
...
  4013e3: push %eax
  4013e4: pushl %fs:0x0
  4013eb: mov %esp,%fs:0x0
[ 4013f2: rep ... rep nop   ]
[  not present in the trace.]
[  an exception raised here.]
7c90eaec: mov 0x4(%esp),%ecx
7c90eaf0: mov (%esp),%ebx
7c90eaf3: push %ecx
...
```

Intel VT

```
...
 4013e3: push %eax
 4013e4: pushl %fs:0x0
 4013eb: mov %esp,%fs:0x0
 4013f2: rep ... rep nop
 401402: mov %esp,%eax
 401404: mov %eax,%fs:0x0
 40140a: add $0x8,%esp
...
```

QEMU

Figure 5: Divergence point in multirep

```
...
77c50c93: pop %edx
77c50c94: ret
  40920a: test %eax,%eax
  40920c: je 0x409225
  40920e: add $0x8,%esp
  409211: call 0x409216
  409216: pop %esi
...
```

Intel VT

```
...
77c50c93: pop %edx
77c50c94: ret
  40920a: test %eax,%eax
  40920c: je 0x409225
  409225: lods (%esi),%eax
  409226: dec %eax
  409227: xor $0x1633523,%eax
...
```

QEMU

Figure 6: Divergence point in Downloader-AFH

raises an exception; the exception handler prints a message "Not QEMU!" and exits. Under QEMU, the non-exception execution path which prints "QEMU Detected!" is taken.

Divergence Point: Our trace alignment tool finds the point after the instruction at `0x4013eb` as the divergence point between the traces. As Figure 5 shows, the reference platform executes an exception handler after this point, while QEMU proceeds with the abnormally long instruction.

Root Cause: As described in Section 4.2, the DSM generation algorithm treats the raising of an exception as if it were an output of an instruction, and checks if it could be caused by an instruction input difference. But there is no input difference in this case, so the instruction that raises the interrupt itself must be the root cause. Based on the behavior of Ether, the DSM for this root cause is to inject a general protection fault exception at `0x4013f2`.

DSM and Verification: Verifying the effectiveness of the generated DSM is straightforward for this sample, since we know its intended behavior. When the we apply the DSM to QEMU, the sample prints out "Not QEMU!" as expected.

6.2.2 FPU Register (Downloader-AFH)

Attack Description: Downloader-AFH differs from our other samples in that it uses a Windows library function to detect an emulated environment. After an unpacking step, it calls the `asin` (arcsine) function in `msvcrt.dll`. The sample ignores the function return value but instead checks the `%eax` register. The sample uses the `%eax` value to calculate a memory location that it will modify later in execution. For reasons we will discuss below under **Root Cause**, the value of `%eax` is `0x27f` under Ether, but `0` under QEMU.

Divergence Point: As shown in Figure 6, the trace alignment our tool produces in its default mode, which looks for a divergence point in the malware sample's code, finds a divergence at `0x40920c` in Downloader-AFH. This is a conditional jump based on a test of `%eax` after the return from `asin`. If the value is non-zero, as under Ether, it is later used in a local function; different code is executed if the value is zero as under QEMU.

Root Cause: In its default mode, our DSM generation process treats calls to standard libraries as atomic; the call itself can be a

```
...
77c4cb2f: fnstcw (%esp)
77c4cb32: je 0x77c4cba1
77c4cb34: cmpw  $0x27f,(%esp)
77c4cb3a: je 0x77c4cb41
77c4cb41: cmp $0x3ff00000,%eax
77c4cb46: jae 0x77c4cb73
77c4cb48: fld1
...
```

Intel VT

```
...
77c4cb2f: fnstcw (%esp)
77c4cb32: je 0x77c4cba1
77c4cb34: cmpw  $0x27f,(%esp)
77c4cb3a: je 0x77c4cb41
77c4cb3c: call 0x77c50bd5
77c50bd5: mov 0x4(%esp),%edx
77c50bd9: and $0x300,%edx
...
```

QEMU

Figure 7: Divergence point in msvcrt.dll (Downloader-AFH)

```
...
  413a04: xor %edx,%edx
  413a06: pushl %fs:(%edx)
  413a09: mov %esp,%fs:(%edx)
  413a0c: icebp
7c90eaec: mov 0x4(%esp),%ecx
7c90eaf0: mov (%esp),%ebx
7c90eaf3: push %ecx
...
```

Intel VT

```
...
413a04: xor %edx,%edx
413a06: pushl %fs:(%edx)
413a09: mov %esp,%fs:(%edx)
[the guest system halts.]
```

QEMU

Figure 8: Divergence point in Proxy-Bypass

root cause, but the algorithm will not look inside the called code for a root cause instruction. This makes the analysis process more efficient, and also makes the generated DSM more robust, since system libraries can change between OS versions. In this mode, the root cause search stops at the call to `asin` in this example: the root cause is the value of `%eax` after the call.

We also repeated this experiment in a mode where our tool considered library routines to be in scope as possible divergence sites. In this mode, the tool finds a divergence inside `msvcrt.dll` and a root cause at the `fnstcw` instruction at `0x77c4cb2f`, as shown in Figure 7. The effect of the `fnstcw` instruction is to store a copy of the FPU control word (a flags register) on the stack. Under Ether, the value stored is `0x27f`, while under QEMU, it is `0x23f`; the code later compares the value to `0x27f`. These values differ in just a single bit, `0x40`, which is described as reserved in Intel's documentation; QEMU allows the bit to be zero, while on real hardware it is always one. This behavior difference does not appear to contradict the documentation, nor affect benign software, but this anti-emulation check takes advantage of it.

DSM and Verification: Using our tool's two modes, we generate two DSMs, with and without libraries. Both DSMs prevent the divergence, but the DSM generated for the malware binary only is the one we recommend for most applications, because it is robust to changes in library code. In fact, an upgrade to Windows XP SP2, described in Section 6.4, prevents the in-library DSM from applying but does not affect the default one.

When executed under an unmodified QEMU, the malware sample crashes shortly after calling `asin`. But when run with the DSM, we used our QEMU-based analysis to observe a number of kinds of suspicious behavior, including modifying the Windows registry, and creating files and network connections.

6.2.3 Undocumented Opcode (Proxy-Bypass)

Attack Description: The Proxy-Bypass sample uses the `icebp` (`0xf1`) instruction, an undocumented opcode in x86 CPUs. The `icebp` instruction was once used together for hardware-level debugging, but on modern CPUs it simply raises an interrupt with the vector of `0x1`. An unmodified version of QEMU uses the instruction for purposes of debugging QEMU itself, such as providing a point to attach `gdb`, but in production use the effect is to cause the entire emulator to hang. These malware samples trap the interrupt thrown by real hardware; code in the exception handler then triggers another obfuscated control-flow transfer.

Divergence Point: Figure 8 shows the divergence point our tool finds, which comes at the point where the QEMU trace is cut off by the emulator hang. Under Ether, by contrast, execution continues with an exception handler after the `icebp` instruction.

Root Cause: Similarly to the earlier malformed opcode example, our DSM generation algorithm here finds an exception thrown in the reference trace by an instruction that has no inputs, so the root cause search terminates immediately.

	Original Environment	Changed Environment
Memory	512MB	256MB
File Path	Desktop folder	Windows/system32
OS Version	Windows XP	Windows XP SP2
User Privilege	Administrator	Limited User
Locale	USA	Russia

Table 3: Changes in the execution environment for robustness evaluation (Section 6.4)

DSM and Verification: The DSM generation procedure produces a DSM that injects an interrupt with the vector of 0x1 at instruction `0x413a0c`. However, this DSM is not sufficient. On two subsequent iterations, our tool finds two more occurrences of `icebp` at other locations, which it ameliorates in the same way. On four iterations, our tool finds another four divergence points involving single-step exceptions (number 1) at four different instructions in a single basic block. We have not yet been able to determine why these exceptions are thrown, but our tool generates the proper DSMs.

After applying the final DSM based on all seven iterations, we can observe several kinds of suspicious behavior from the malware sample. It tries to copy itself to several locations under the Windows `system32` folder, and it modifies a Windows Registry key to make itself automatically executed at every system start-up.

6.3 Performance

We also evaluate the efficiency of our techniques by measuring the time taken to find divergence points and generate DSMs from them. As shown in Table 2, both steps are quick for most of the samples, though there is significant variation between samples. Most of the cost of diagnosis is due to the alignment algorithm. The most sophisticated sample, Proxy-Bypass, requires 7 iterations to fix all its attacks, and some occur deep in execution making alignment and slicing more expensive. The shorter sizes of the QEMU traces are caused by the samples crashing or terminating because of anti-emulation checks. By contrast, malware authors often intend their programs to run forever on a vulnerable host, so we terminated the Ether traces after 500MB. Our QEMU implementation writes traces at 6.75MB/sec, while Ether averages 2.72MB/sec.

6.4 Robustness

To evaluate the robustness of our DSMs, we created a different execution environment in QEMU by changing configuration options and the OS version (see Table 3 for details). By comparing these executions to the original ones, we check both whether the DSM is still effective when the sample runs in a modified environment, and whether the malware sample itself runs differently when the environment is different.

For each malware sample, we verified that it still behaved maliciously in the changed environment, even though the DSM used was the one generated using the original environment. This satisfies our criterion for robustness: a DSM generated using a sin-

Sample Label	Trace Size (file / instructions)		Time Taken (sec)	
	Ether	QEMU	Divergence Diagnosis	DSM Generation
W32/Sdbot.worm(1)	500MB/11923410	12KB/168	24.56	0.32
W32/Sdbot.worm(2)	500MB/11848928	24KB/344	30.89	0.33
Generic.dv	500MB/11945744	22KB/312	31.49	1.27
multirep	2.1MB/36179	1.5MB/26419	0.23	0.95
Downloader-AFH	500MB/8386352	206MB/3444232	32.82	3.12
Proxy-Bypass	335MB/6732837	3.4MB/58370	14.22 + 11.27 + 296 + 297 + 293 + 309 + 21.23	1.85 + 3.15 + 2.86 + 2.87 + 3.11 + 16.82 + 69.83

Table 2: Performance results

gle configuration can be reused in others. For 5 of the 6 samples, the malware executed exactly the same code with the DSM in the changed emulated environment as it did with the DSM in the original emulated environment. Thus for these samples, the environment change did not affect either the malware sample or the DSM. For the remaining sample, Proxy-Bypass, the environment change caused the malware sample to behave differently, but the DSM was still effective: it did not hang as it did on the original emulator. By examining the sample's Windows API calls, we confirm that it still behaves maliciously (for instance, making similar accesses to the file system and Registry), but some of the behavior is thwarted by having fewer privileges. For instance, it fails in an attempt to copy itself to a certain directory and then execute that copy.

7. RELATED WORK

Platforms providing isolation and instrumentation are key to malware analysis, and whole-system emulators are particularly useful. Many analysis tools have been built based on existing open-source emulators including QEMU [26] ([3, 22, 31]) and Bochs [5] ([6, 10]). Automatic support for ameliorating anti-emulation attacks would be particularly valuable in fully automated online malware analysis services [11, 24], several of which are based on QEMU [2, 4], because a skilled analyst may not be available to disable anti-emulation code manually.

On the other side, there has been significant research on weaknesses in virtual platforms that allow them to be detected. Emulators are generally more difficult to detect than high-performance VMs like VMWare [15], but Raffetseder et al. [27] propose a variety of attacks effective against emulators. Ormandy [25] discovers a number of implementation flaws that make popular emulators and virtual machines vulnerable to detection, denial of service, and in a few cases even code injection. Another line of research looks at the even more difficult challenge of identifying a virtual environment on a remote machine [14, 19], which is beyond our scope.

There has been comparatively less work on defense techniques like ours. Chen et al. [9] measure that more than 40% of malware samples exhibit different behavior in the presence of a debugger or virtual environment. They propose harnessing this by making production systems mimic the presence of a debugger or virtual machine, the opposite deception to what we investigate. VMWatcher [17] explores the use of a virtual machine for malware defense, but focuses more on detection (as in a production environment) than analysis, so VM-aware malware is a less acute problem. Ether [12] is a novel analysis framework with transparency as its main goal, using hardware-based virtualization [16, 1] which is very effective against CPU semantics attacks. Our research shows how to transfer the state-of-the-art transparency of a specialized system like Ether to a general-purpose emulator with more support for detailed analysis. Our approach also generalizes to other reference platforms that could address other transparency issues, such as detailed timing attacks or queries of hardware characteristics [15].

Similar issues of transparency occur, but with the roles of attacker and defender reversed, in malware approaches that use a virtual machine to produce an undetectable rootkit [20]. This use of whole-system virtual machines should also not be confused with more specialized virtual machines which are used by malware authors for obfuscation [29].

8. CONCLUSION

Anti-emulation behavior in malware is a serious practical problem that frustrates the analysis needed for effective malware defense. We have presented an automated approach to emulation in the face of anti-emulation attacks, based on a dynamic state modification to the execution of a whole-system emulator that keeps an attack from distinguishing it from a real system. The techniques that make this approach possible are a scalable trace matching algorithm that locates a control divergence, and a DSM-building algorithm that finds the root cause of that divergence and a small state change that corrects it. We have built a practical implementation of this technique into an emulator used for automatic malware analysis, and applied it to 5 samples of malware collected in the wild and 1 attack that has not yet been exploited. Our tool automatically corrected all of the emulation failures, with robust DSMs, allowing an automated analysis to reveal the malware's malicious activities. These results show that automated diagnosis and state modification are powerful tools for understanding and controlling software; we expect they will also see application in other contexts like software engineering and troubleshooting system misconfigurations.

9. REFERENCES

[1] AMD Virtualization (AMD-V) Technology. `http://www.amd.com/us-en/0,,3715_15781_15785,00.html`.

[2] Anubis. `http://anubis.iseclab.org/`.

[3] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *15th EICAR Conference*, pages 180–192, Hamburg, Germany, May 2006.

[4] BitBlaze Online. `https://aerie.cs.berkeley.edu/`.

[5] The Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net`.

[6] L. Böhne. Pandora's Bochs: Automatic unpacking of malware. Diploma thesis, RWTH Aachen University, Jan. 2008.

[7] P. Bosch, A. Carloganu, and D. Etiemble. Complete x86 instruction trace generation from hardware bus collect. In *23rd EUROMICRO Conference*, pages 402–408, Budapest, Hungary, Sept. 1997.

[8] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, Feb. 2004.

[9] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 177–186, Anchorage, AK, USA, June 2008.

[10] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004.

[11] CWSandbox. `http://www.cwsandbox.org`.

[12] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *15th ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, Alexandria, VA, USA, Oct. 2008.

[13] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, June 2007.

[14] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *SIGOPS Oper. Syst. Rev.*, 42(3):83–92, 2008.

[15] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, San Diego, CA, USA, May 2007.

[16] Intel Virtualization Technology. `http://www.intel.com/technology/itj/2006/v10i3/1-hardware/6-vt-x-vt-i-solutions.htm`.

[17] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *ACM Conference on Computer and Communications Security (CCS)*, pages 128–138, Alexandria, VA, USA, Oct. 2007.

[18] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *5th ACM Workshop on Recurring Malcode (WORM)*, Oct. 2007.

[19] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *In 12th USENIX Security Symposium*, pages 295–310, Washington, DC, USA, Aug. 2003.

[20] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, Oakland, CA, USA, May 2006.

[21] M. Levoy and R. Whitaker. Gaze-directed volume rendering. In *Symposium on Interactive 3D Graphics*, pages 217–223, Snowbird, UT, USA, Mar. 1990.

[22] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, pages 431–441, Miami Beach, FL, USA, Dec. 2007.

[23] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[24] Norman SandBox. `http://www.norman.com/microsites/nsic/en-us`.

[25] T. Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. In *8th CanSecWest Applied Technical Security Conference*, Vancouver, BC, Canada, Apr. 2007.

[26] QEMU. `http://fabrice.bellard.free.fr/qemu/`.

[27] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Information Security, 10th International Conference (ISC)*, pages 1–18, Valparaíso, Chile, Oct. 2007.

[28] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, pages 129–144, Denver, CO, USA, Aug. 2000.

[29] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Rotalumè: A tool for automatic reverse engineering of malware emulators. Technical Report GT-CS-09-05, School of Computer Science, Georgia Institute of Technology, 2009. (Related paper to appear at IEEE Symposium on Security and Privacy 2009).

[30] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[31] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, USA, Oct. 2007.