The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code

Clemens Kolbitsch
Vienna University of Technology
Vienna, Austria
ck@iseclab.org

Engin Kirda
Northeastern University
Boston, MA, USA
ek@ccs.neu.edu

Christopher Kruegel
UC Santa Barbara
Santa Barbara, CA, USA
chris@cs.ucsb.edu

Abstract

Malware continues to remain one of the most important security problems on the Internet today. Whenever an anti-malware solution becomes popular, malware authors typically react promptly and modify their programs to evade defense mechanisms. For example, recently, malware authors have increasingly started to create malicious code that can evade dynamic analysis.

One recent form of evasion against dynamic analysis systems is *stalling code*. Stalling code is typically executed before any malicious behavior. The attacker's aim is to delay the execution of the malicious activity long enough so that an automated dynamic analysis system fails to extract the interesting malicious behavior. This paper presents the first approach to detect and mitigate malicious stalling code, and to ensure forward progress within the amount of time allocated for the analysis of a sample. Experimental results show that our system, called HASTEN, works well in practice, and that it is able to detect additional malicious behavior in real-world malware samples.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Malware Analysis, Evasion, Emulation

1. INTRODUCTION

Malicious software (malware) is the driving force behind many security problems on the web. For example, a large fraction of the world's email spam is sent by botnets [16], Trojan programs steal account credentials for online banking sites [27], and malware programs participate in click fraud campaigns and distributed denial of service attacks [15].

Malware research is an arms race. As new anti-malware solutions are introduced, attackers are updating their malicious code to evade analysis and detection. For example, when signature-based anti-virus scanners became widely adopted, attackers started to use code obfuscation and encryption to thwart detection. As a consequence, researchers and security vendors shifted to techniques that focus on the runtime (dynamic) behavior of malware.

An important enabler for behavior-based malware detection are dynamic analysis systems (such as ANUBIS [2], CW-SANDBOX [3], NORMAN SANDBOX [4], and TEMU [25]). These systems execute a captured malware program in a controlled environment and record its actions (such as system calls, API calls, and network traffic). Based on the collected information, one can decide on the malicious nature of a program, prioritize manual analysis efforts [8], and automatically derive models that capture malware behaviors [14, 19]. Such models can then be deployed to protect end-users' machines.

As dynamic analysis systems have become more popular, malware authors have responded by devising techniques to ensure that their programs do not reveal any malicious activity when executed in such an automated analysis environment. Clearly, when malware does not show any unwanted activity during analysis, no detection models can be extracted. For anti-malware researchers, these evasion attempts pose a significant problem in practice.

A common approach to thwart dynamic analysis is to identify the environment in which samples are executed. To this end, a malware program uses checks (so-called red pills) to determine whether it is being executed in a virtual machine [13, 24] or a system emulator such as Qemu [20, 22, 23]. Whenever a malware program is able to detect that it is running inside such an environment, it can simply exit.

Reacting to the evasive checks (red pills), researchers have proposed more transparent analysis environments [11, 12]. Another approach has focused on the detection of differences between the execution of a program in a virtual platform and on real hardware [6, 17]. When such a discrepancy is identified, the checks responsible for the difference can be removed. Finally, systems that perform multi-path exploration [9, 21] or that identify "malicious triggers" [10] can detect and bypass checks that guard malicious activity.

In the next step of the arms race, malware authors have begun to introduce *stalling code* into their malicious programs. This stalling code is executed before any malicious behavior – regardless of the execution environment. The purpose of such evasive code is to delay the execution of malicious activity long enough so that automated analysis systems give up on a sample, incorrectly assuming that the program is non-functional, or does not execute any action of interest. It is important to observe that the problem of stalling code affects *all* analysis systems, even those that are fully transparent. Moreover, stalling code does not have to perform any checks.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-0948-6/11/10 ...$10.00.

Thus, systems that aim to detect malware triggers or that explore multiple execution paths do not reveal any additional behaviors.

With stalling code, attackers exploit two common properties of automated malware analysis systems: First, the time that a system can spend to execute a single sample is limited. Typically, an automated malware analysis system will terminate the analysis of a sample after several minutes. This is because the system has to make a trade-off between the information that can be obtained from a single sample, and the total number of samples that can be analyzed every day. Second, malware authors can craft their code so that the execution takes much longer inside the analysis environment than on an actual victim host. Thus, even though a sample might stall and not execute any malicious activity in an analysis environment for a long time (many minutes), the delay perceived on the victim host is only a few seconds. This is important because malware authors consider delays on a victim's machine as risky. The reason is that the malicious process is more likely to be detected or terminated by anti-virus software, an attentive user, or a system reboot.

In this paper, we present the first approach to detect and evade malicious stalling code, and to ensure forward progress within the amount of time allocated for the analysis of a sample. To this end, we introduce techniques to detect when a malware sample is not making sufficient progress during analysis. When such a situation is encountered, our system automatically examines the sample to identify the code regions that are likely responsible for stalling the execution. For these code regions (and these regions only), costly logging is disabled. When this is not sufficient, we force the execution to take a path that skips (exits) the previously-identified stalling code.

We implemented our approach in HASTEN, an extension for ANUBIS, our dynamic analysis system. Our experimental evaluation shows that HASTEN is effective in practice and can reveal additional behavior in real-world malware.

This paper makes the following contributions:

- We present the first approach to detect and deal with malicious stalling code in real-world malware, and to ensure forward progress within the amount of time allocated for the analysis of a malware sample.

- We propose HASTEN, a dynamic analysis system extension to detect and *passively* mitigate the impact of stalling code. To this end, we identify stalling code regions and execute them with reduced logging.

- We introduce an *active* extension to HASTEN that forces the execution of the malware to take a path that skips the previously-identified stalling code. This helps in cases where reduced logging is not sufficient.

2. PROBLEM DESCRIPTION

In this section, we discuss the problem of stalling code in more detail, and we show a real-world code example that implements such a mechanism.

2.1 Definition

We define a *stalling code region* as a sequence of instructions that fulfills two properties: First, the sequence of instructions runs considerably slower inside the analysis environment than on a real (native) host. In this context, "considerably slower" means that the slowdown is large compared to the average slowdown that the sandbox incurs for normal, benign programs. Examples of slow operations are system call invocations (because of additional logging overhead) and machine instructions that are particularly costly to emulate (e.g., floating point operations or MMX code).

```
unsigned count, t;

void helper() {
 t = GetTickCount();
 t++;
 t++;
 t = GetTickCount();
}
void delay() {
 count=0x1;
 do {
  helper();
  count++;
 } while
  (count!=0xe4e1c1);
}
```

Figure 1: Stalling code in W32.DelfInj

The second property of stalling code is that its entire execution must take a non-negligible amount of time. Here, non-negligible has to be put into relation with the total time allocated for the automated analysis of a program. For example, for a dynamic analysis system, this could be in the order of a few minutes. Thus, we expect the stalling code to run for at least several seconds. Otherwise, the analysis results would not be significantly affected. That is, when an instruction sequence finishes within a few milliseconds instead of microseconds, we do not consider this as stalling code.

Clearly, an attacker could create stalling code that stalls execution on the real host in the same way it does in the analysis environment. For example, the attacker could use sleep calls, or create high amounts of activity to delay execution. However, in practice, execution delays using sleep-like functions can be easily skipped, and delaying execution (for example, by raising the volume of activity) increases chances of being detected and terminated on a victim host.

Intuitively, our definitions imply that stalling code contains "slow" operations (to satisfy the first property), and that these operations are repeated many times (to satisfy the second property). As a result, attackers typically implement stalling code as loops that contain slow operations (and we will sometimes refer to stalling code as *stalling loops* in this paper).

2.2 Example of Stalling Code

Figure 1 shows a stalling loop implemented by real-world malware. This code was taken from a sample that was first submitted to the Anubis analysis system in February 2010. As the sample was only available in binary format, we reverse engineered the malware program and manually produced equivalent C code. Since the executable did not contain symbol information, we introduced names for variables and functions to make the code more readable. To determine the malware family that this sample belongs to, we submitted it to the popular VirusTotal service. VirusTotal uses around 40 anti-virus (AV) tools to scan each uploaded sample. All 40 AV scanners considered the file to be malicious, and most tools labeled our sample as *Trojan Win32.DelfInj*.

As can be seen in Figure 1, the code contains a number of unnecessary calculations involving the return value of the Windows `GetTickCount` function. According to documentation provided by Microsoft [5], `GetTickCount` "retrieves the number of milliseconds that have elapsed since the system was started, up to 49.7 days." It is typically used to measure the time that elapses between two events. In this scenario, however, the repeated invocations are not useful. In fact, the value of the variable `tick` is simply overwritten by subsequent calls to `GetTickCount`. Moreover, this loop is executed 15 million times. This strongly supports the intuition that the purpose of this code is to stall execution.

When the code shown in Figure 1 is run natively on a real processor, the loop executes and terminates in a matter of milliseconds.

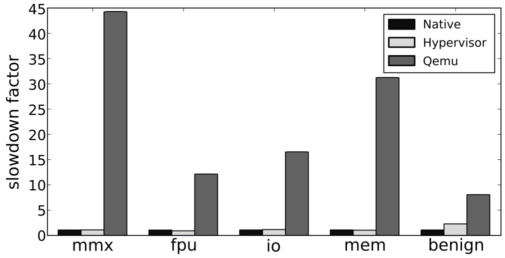


Figure 2: Slowdown for native, hypervisor-based, and emulated (Qemu) execution (native = baseline).

Interestingly, on the Qemu system emulator, the execution time is not significantly higher. This is because of the way in which the `GetTickCount` function is implemented on Windows. Instead of invoking an expensive system call to retrieve the number of milliseconds elapsed since system start-up, this value can be read directly from a kernel memory page that is mapped (read-only) into the address space of all user programs (the counter is updated periodically by the operating system).

Analyzing the stalling code inside ANUBIS, our Qemu-based dynamic analysis environment, yields a completely different result. In ANUBIS, `GetTickCount` is monitored and, thus, the system invokes a pair of log functions for each call to `GetTickCount` (one before the call, and one after returning). These log functions extract and write detailed information about the monitored function call. For the stalling code in Figure 1, this means that the logging functions of our analysis system are called 60 million times. Our estimates show that this would take about ten hours, which means that the attacker can successfully evade the analysis.

Note that excluding `GetTickCount` from the list of monitored functions would provide a solution in this specific example. However, the general problem that we address in this paper would still remain. We discovered various other samples in the wild that invoke different functions that are frequently monitored (such as `WriteFile`), and we cannot generally ignore them all. Furthermore, attackers can implement stalling code without targeting the overhead introduced by the monitoring system and instead make use of instructions that are particularly slow to emulate. Figure 2 shows four examples of programs that make heavy use of FPU, MMX, IO, and memory operations, each experiencing a significant slowdown (up to a factor of 45). In comparison, for benign applications average overheads stay below a factor of 10.

3. SYSTEM OVERVIEW

In this section, we present a high-level overview of HASTEN. The goal of our system is to detect stalling loops and to mitigate their effect on the analysis results produced by a sandbox (such as ANUBIS). In particular, our system must ensure that the analysis makes sufficient progress during the allocated analysis timeframe so that we can expose as much malicious activity as possible. To this end, HASTEN can operate in three modes: monitoring, passive, and active mode.

Monitoring mode. When the analysis of a malware sample is launched, HASTEN operates in monitoring mode. In this mode, the system performs lightweight observation of all threads of the process under analysis. The goal is to measure the *progress* of each thread, and to identify instances in which the execution might have entered a stalling region.

Passive mode. When the monitoring mode detects insufficient progress, this is typically due to slow operations that are executed many times. Thus, as a first step, HASTEN attempts to identify the code region that is repeatedly executed. To this end, our system starts to dynamically record information about the addresses of instructions (code blocks) that are executed. Using these addresses, we build a (partial) control flow graph (CFG) of the non-progressing thread. This CFG is then searched for loops.

```
// H4X0r: make sure delay loop was not interrupted
void check() {
 if (count!=0xe4e1c1) exit();
}
```

Figure 3: Malware code to check that the delay loop has not been interrupted.

Intuitively, the code that is identified by this process represents the stalling loop(s). For example, in the code snippet in Figure 1, our tool would identify the *do-while* loop in the `delay` function (Line 11-15). The stalling code region would also include the `helper` function (L. 3-8), since it is invoked inside the body of the loop (L. 12). Note that our system uses inter-procedural control flow analysis for loop detection. This allows us to handle cases in which a loop is not part of a single function, but the result of recursive function calls.

Once the stalling loop is identified, HASTEN adapts its analysis for this code region. More precisely, we first whitelist the code that is part of the stalling region. Note that this whitelist covers *only* those instructions (basic blocks) that have been executed previously by the malware. Thus, parts of a stalling loop that have not been executed before are not whitelisted. In the next step, the system limits (or turns off) detailed malware introspection for the whitelisted code regions. This approach significantly reduces (or removes) the overhead that is introduced by the analysis environment.

The passive mode is called "passive" because we do not interfere with the malware execution; we only modify the analysis environment to accelerate stalling loops.

Active mode. When HASTEN operates in active mode, it actively interferes with the execution of the binary. In particular, the tool attempts to force a stalling loop to exit (or, more precisely, to follow a code path that exits the whitelisted region that the previous analysis has identified; this might force execution into parts of a stalling loop that have not yet been executed). To this end, our system uses the previously-constructed CFG and identifies all nodes associated with conditional jumps that are *(i)* part of the stalling loop and that *(ii)* have one successor node that is not part of the whitelisted code region. That is, we identify all nodes through which there exists a path that exits the stalling code. At the next loop iteration, when such a conditional jump is encountered, HASTEN flips the check that this instruction implements (e.g., a *less-than* would be converted into a *greater-or-equal*). Hence, the thread is forced to continue execution along a path outside the stalling region.

Altering the flow of execution of a program (such as prematurely preempting a loop or following an "unexplored" path within the loop) can leave this program in an inconsistent state. Referring back to our example in Figure 1, one can see that the variable `count` will not have the expected value (`0xe4e1c1`) when HASTEN forces the loop to exit. This might be irrelevant, but at the same time, there is a possibility that it could also lead to program crashes. Malware authors could leverage these inconsistencies to expose the analysis system. For example, consider an attacker that calls `check`, shown in Figure 3, after the `delay` function (from Figure 1). If our system would preempt the loop, this comparison would succeed, and the malware would terminate without revealing any additional malicious behavior.

To overcome this problem, our approach is as follows: Before we exit a whitelisted code region, we first analyze this region for all variables (memory locations) that the code writes as part of a computation (logic and arithmetic instructions). These memory locations are then marked with a special label (tainted), indicating that their true value is *unknown*. Whenever a machine instruction uses a tainted value as source operand, the destination is tainted as well. For this, we leverage the fact that we have implemented our prototype solution on an emulator-based analysis platform that already supports data flow (taint) tracking.

Whenever a tainted variable is used in a comparison operation or in an indirect memory access (using this variable as part of the address computation), HASTEN temporarily halts the execution of the malware process. It then extracts a backward slice that ends at the comparison instruction and that, when executed, will compute the correct value for this variable. To obtain this slice, we leverage a tool that we previously developed [18]. Once the slice is extracted, it is executed on a native machine. As a result, this computation does not incur any overhead compared to the execution on a real victim host.

4. MONITORING MODE

The goal of the monitoring mode is to determine whether a malware process has entered a stalling code region. To this end, HASTEN monitors the *progress* of a program under analysis. More precisely, the system monitors, for each thread, the frequencies, types, and return codes of all (native) system calls that this thread invokes.

System calls are the mechanism through which a process interacts with its environment and the operating system. Also, to perform security-critical tasks (such as spawning new processes, modifying files, or contacting network services), a program *must* invoke at least one system call. Thus, we believe that it is reasonable to monitor the progress of a running program by inspecting the system calls that it invokes. Moreover, all system calls use the same convention to report errors. That is, there are well-specified values that indicate a successful, or erroneous, invocation. This makes it easy to determine whether a system call has been successful or not (which is useful knowledge when reasoning about the progress of a process). Finally, almost all analysis systems already support the monitoring of system calls, and typically, this data can be collected efficiently.

Detecting progress. To measure the progress of a process, HASTEN periodically evaluates the system calls that each of its threads invokes. More precisely, after a thread has been scheduled for a time interval of length t, the system employs five simple detectors to evaluate the system calls that have been observed during that last period. Whenever one or more of these detectors reports insufficient progress, the system is switched into passive mode (explained in more detail in the following Section 5).

When choosing a concrete value for the time interval t, we need to balance two ends: Large values are more resistant against short sequences of unusual, repeated behavior; small values allow faster detection of abnormal activity. For our experiments, we have chosen a value of $t = 5$ seconds. This allows us to evaluate the progress multiple times during an analysis run (which is typically a few minutes) and yielded good results in our experiments. Our five detectors are discussed in the paragraphs below:

Detector for too few successful system calls: We expect to see a reasonably large number of successful system calls that reflect the normal interactions of a process with its environment and the OS. Thus, when our system does not observe any system calls (or too few of them), this is an indication of insufficient progress. This detector simply computes the total number of *successful* system calls S_s within the time interval t. An alarm is raised when this number is lower than a threshold $S_s < S_{min,s}$.

Detector for too many successful system calls: Interestingly, there is also the possibility that a malware process invokes too many system calls, even though no progress is made. Usually, the number of system calls a program executes is limited by the necessary data processing in between two calls, or delays introduced while waiting for IO completion. With some stalling loops, however, malware authors try to exploit the logging overhead of the analysis system and rapidly issue many calls. For this, one would typically select system calls that finish very quickly (e.g., NTCREATESEMAPHORE, as it does not need to wait for IO completion). This detector recognizes such attempts. An alarm is raised when the number of successful system calls exceeds a certain threshold $S_s > S_{max,s}$.

Detector for too many failed system calls: Of course, it is also possible to overload a sandbox with many invalid system calls (i.e., system calls with incorrect parameters). Invalid calls are quickly caught by the kernel, which checks the validity of the parameters (e.g., pointers should not be NULL) early on. Hence, they execute very fast on a victim host. Unfortunately, they still incur the full logging overhead on the analysis platform. Thus, this detector raises an alarm when the number of failed system calls exceeds a certain threshold $S_f > S_{max,f}$.

Detector for too many identical system calls: The previous two detectors identify cases in which stalling code attempts to overload the sandbox with excessive numbers of system calls. A more cautious attacker could attempt to hide stalling code by issuing fewer system calls (that stay under our thresholds) and inject additional computations between the call sites. However, even in this case, it is likely that we will observe the same (small) set of calls repeated many times (executed in a loop). This is different from normal program execution, where a program invokes more diverse system calls. The detector for identical system calls recognizes cases where most system calls seen within interval t belong to a small set. To this end, we compute the entropy S_e of the set of system calls during t. An alarm is raised when the entropy is below a threshold $S_e < S_{min,e}$.

Detector for too diverse system calls: To thwart the previous detector, a malware author could choose to execute a series of *randomly* chosen system calls to artificially increase the entropy. Although the stealth implementation of such an approach seems difficult (considering the detector for too many failed system calls), we add an additional detector that raises an alarm when the entropy exceeds a threshold that is atypical for normal program executions: $S_e > S_{max,e}$.

In our experiments, we found that the previously-described detectors are successful in identifying stalling code in real-world malware. However, if necessary, it is easy to develop additional heuristics and integrate them into our system.

Parameter computation. The choice of concrete parameter values used by the five detectors ($S_{min,s}$, $S_{max,s}$, $S_{max,f}$, $S_{min,e}$, and $S_{max,e}$) determines how often the system is switched into passive mode. When the thresholds are set too low, HASTEN might enter passive mode more often than necessary. When the thresholds are too high, the system might mistake stalling code for sufficient progress and, as a result, miss interesting activity. Because the cost of unnecessarily switching into passive mode is only a small performance hit, we prefer low thresholds that result in sensitive detectors.

To compute the necessary thresholds, we compiled a training set of 5,000 malware samples submitted to ANUBIS. We only selected samples that performed obvious malware activity (they produced network traffic, spawned additional processes, or opened a UI window). We assume that these samples do not contain stalling code because the analysis successfully exposed malicious behaviors. All samples from the training set were executed in HASTEN, counting the numbers of successful and failed system calls during each time period t, and computing their entropy. Then, the thresholds were set so that the numbers derived for the training samples do not trigger any detector. For more details about the parameter computation, the reader is referred to Appendix A.

5. PASSIVE MODE

When the previous monitoring phase detects that a thread makes insufficient progress, the system switches into passive mode (for this thread). When in passive mode, the system performs two steps.

In the first step, the system attempts to find code blocks that are repeatedly executed (as part of loops or through recursive function calls). To this end, the system starts to build a dynamic control flow graph (CFG) that covers the code that the thread is executing. This CFG is then searched for cycles, and all blocks that are part of a cycle are marked as (potential) stalling code. In the second step, HASTEN reduces (or disables) logging for all blocks that the previous step has marked as stalling code.

5.1 Identifying Stalling Code

Constructing the dynamic control flow graph. To build a dynamic, inter-procedural control flow graph, we need to keep track of which instructions the program executes. For this, we leverage the execution infrastructure of ANUBIS, which is based upon the system emulator Qemu. More precisely, whenever HASTEN is switched into passive mode, the system starts to instrument the first instruction of each translation block (which is roughly the Qemu-equivalent to a basic block) as well as all `call` and `ret` instructions. The first instruction of a basic block simply appends the value of the current instruction pointer `%eip` to a queue. This allows us to efficiently determine the sequence in which a thread has executed its basic blocks. The `call` and `ret` instructions additionally store the current value of the stack pointer. This makes it easier to handle non-standard function calls (when the callee does not return to the calling function).

The queue holding `%eip` information is processed periodically (during scheduling or when switching to kernel code). At this point, the sequence of values $(eip_1, eip_2, ..., eip_n)$ in the queue is used to construct the CFG: Each distinct eip_i value represents a different basic block, and hence, a different node in the control flow graph. Thus, for each $eip_i \; \forall i : 1 \leq i \leq n$, we check whether there exists already a corresponding node in the CFG. If not, it is added. Then, for each pair $(eip_i, eip_{i+1}) \; \forall i : 1 \leq i < n$, we insert an edge from the node associated with eip_i to the node associated with eip_{i+1} (if this edge does not exist). Finally, the queue is flushed, and the last element eip_n (which has not been processed so far) is inserted as the next head of the queue.

Every time the system processes the queue with the `%eip` values, there is potentially more information available that can be used to obtain a more complete picture of the program's CFG. However, we expect that there are diminishing returns. That is, after some time, all code blocks that are part of the stalling code region have been seen, and no new nodes are added to the CFG when the queue is processed. At this point, the CFG can be analyzed for stalling code regions. Note, we also trigger this analysis after a timeout T_r (currently set to 8 seconds). This handles cases where the stalling code contains a loop with many different paths so that a few more nodes are discovered and added every time.

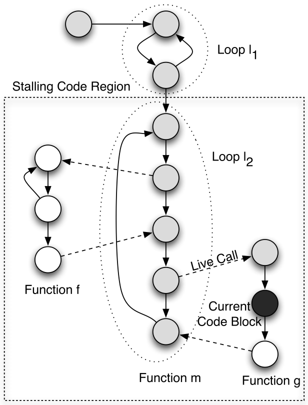


Figure 4: Example CFG for functions m, f, and g.

Finding live loops. To find stalling code, we analyze the CFG for loops. To this end, we use the algorithm proposed by Sreedhar et al. [26]. At this point, we have identified a number of loops in the dynamic control flow graph of the malware program. Some of these loops might be nested, but in most cases, we will find non-overlapping loops that belong to different functions. The question now is: Which of these loops (and which code blocks) should we consider to be part of the execution stalling region?

To answer this question, we first identify all code blocks that are *live* at the current point during the execution. Intuitively, live code blocks represent the code of all functions the program has started to execute, but have not returned yet. More precisely, we define live code blocks as follows:

> *Definition:* A code block b is live when there is a path in the inter-procedural CFG from the node that corresponds to block b to the node c that corresponds to the *current code block*. Moreover, this path can traverse only intra-procedural edges and *live call edges*.

The current code block is the code block that holds the currently-executing instruction (i.e., the code block that the instruction pointer `%eip` currently points to). A live call edge is an edge that has been introduced into the graph by a function call that has not returned yet. Consider a function f that invokes function g. This introduces an edge into the CFG from a node in f to a node in g. Initially, this edge is live, because the program executes instructions in g. When g returns, g is no longer live and its frame is removed from the stack. As a result, the call edge between f and g is no longer live. Of course, the edge in the CFG still remains.

Consider an example where a main function m first invokes f and then, after f has returned, a second function g. Figure 4 shows a CFG for this example. Let us further assume that the current code block is in function g (node in black). It can be seen that only the call edge between m and g is live. The live blocks for this program are represented as gray nodes. Note that the live blocks cover the loops in m and the first nodes in g.

To determine live call edges, we use a stack to track call and return instructions that the malware program executes. As mentioned previously, the stack pointer information associated with `call` and `return` instructions helps us to handle cases in which functions perform non-standard returns.

Finding stalling loops. Once we have determined all live code blocks, the next step is to identify *active loops*. Intuitively, an active loop is a loop that is still in progress; that is, it has not finished yet. We are only interested in active loops as potential stalling loops, since we assume that the stalling code is still running when we perform our analysis.

A necessary precondition for an active loop is that it contains code blocks that are live. However, this is not sufficient: Consider, again, the example in Figure 4. One can see that function m contains two sequential (non-nested) loops; l_1 followed by l_2. The current code block is in g, which is called from a node in l_2. This indicates that l_1 has finished. Thus, even though l_1 is live, it is not active. In our example, only l_2 is active.

To determine active loops, we first identify all live loops. This is straightforward: A live loop is a loop in the CFG with at least one node that corresponds to a live block (actually, because there is a path from all nodes inside a loop to all other nodes in that loop, either all nodes in a loop will be live, or none). Then, we mark as active all live loops where either *(i)* the current code block is part of the loop, or *(ii)*, a node with an outgoing, live call edge is part of the loop.

Note that all active loops within the same function (instance) are nested. To see this, consider all active loops within one function. They must contain either the node associated with the current code block or the node associated with the live call edge. In the case of recursive function calls, we need to distinguish between different instances of the same function, but for each instance, there can be at most one live call node.

Any active loop (or a combination thereof) is potentially responsible for stalling the execution. Thus, we can use different heuristics to pick the one(s) that is (are) most likely stalling. For example, we could pick the innermost active loop. This assumes that the execution is stalled inside the loop that is currently executing, and the code region for which logging is reduced is as small as possible (which is desirable). However, if the inner loop is only a part of a larger stalling loop, accelerating only this region is not sufficient. As an alternative, it is also possible to simply take the outermost loop. In this case, it is very likely that the stalling code is covered in its entirety. On the downside, reduced logging will be enabled for a larger portion of the malicious code. Thus, in our system, we pick the innermost, active loop as the stalling loop. This ensures the most accurate analysis results. In the following subsection, we discuss how we handle the situation when this innermost loop does not cover the entire stalling code region.

5.2 Reducing Analysis Overhead

Once the system has selected the active loop that is likely causing the execution to stall, we have to find the entire code that is part of the stalling code. Of course, the code blocks that are part of the selected, active loop are part of the stalling code, but this is not everything. We also need to add to the stalling region all code blocks that are reachable from any block in the loop body, even when these blocks are not live at the point in time. This is necessary because the stalling region must cover the entire code that is executed as part of the loop; including other functions that can be invoked by the loop, not only those parts that are currently live. However, it is important to remember that the stalling code region can only contain basic blocks that the program has executed previously (others are not part of our CFG).

To find the stalling code region, we find all code blocks b in the CFG so that there is a path from any active loop block to b. This is very similar to the analysis that finds live code blocks, but with the main difference that a path can pass through any call edge (and not only live call edges). All code blocks b that are found by this analysis are added to the stalling region. In the example shown in Figure 4, this means that the stalling code also includes code blocks in function f (since they are reachable from l_2 in m).

Once we have found all blocks that are part of the stalling code, we reduce the amount of analysis that is performed while this code is executing. More precisely, we temporarily disable logging of all functions that are not critical for the proper operation of the analysis system or that we consider highly security-critical. Examples for functions that are continued to be analyzed even in these situations are related to loading of libraries or spawning of new processes. Of course, this is done only for the selected thread and limited to the identified stalling code.

Whenever the execution exits the stalling region, we switch back to monitoring mode. To this end, we whitelist all blocks that are part of the stalling loop. Whenever execution encounters a non-whitelisted code block, the system is reverted back to monitoring mode, and the full analysis is re-enabled.

In case a stalling region contains multiple, nested loops, we first whitelist the innermost loop. When we later observe that this is not sufficient, the whitelisted region is successively extended to cover also the enclosing, parent loops. The details of the process are described in Appendix B.

6. ACTIVE MODE

In passive mode, HASTEN attempts to accelerate the execution of stalling loops by reducing the overhead associated with logging security-relevant functions. Unfortunately, this might not be sufficient to obtain progress. In particular, malware authors can write stalling code that exploits machine instructions that are very slow to emulate (such as floating point or MMX instructions). To address such types of stalling code, we introduce HASTEN's active mode. In active mode, the system forces the termination of the whitelisted stalling code. The purpose of this is to continue to explore the program execution further.

6.1 Modifying the Flow of Execution

When entering active mode, we first have to identify suitable nodes in the CFG that have an outgoing edge that exits the stalling loop. To this end, HASTEN searches all whitelisted code blocks that are part of the stalling region for those that end in a conditional branch. Then, the system checks whether any of these conditional branch instructions has a successor node that is *not* part of the stalling region. These instructions are marked as exit points. If the system fails to find at least one suitable exit point, HASTEN stops analysis and flags the analysis report for subsequent human analysis. Finally, we resume the execution of the malware.

Whenever the process subsequently attempts to execute an exit point instruction, HASTEN examines the operands of the branch operation and determines the path that the execution is about to take. When this path exits the stalling region, the process is allowed to continue. Otherwise, our system will dynamically invert the branch (for example, a *greater-than* operation is changed into *less-or-equal*). This means that HASTEN will take the first exit point that the execution encounters after switching into active mode. See Section 8 for a discussion of the implications of this approach. Once execution leaves the whitelisted code region, HASTEN re-enables full monitoring.

6.2 Handling Inconsistent Variables

When HASTEN changes the control flow and, thus, the execution of a process, the state of this process might become inconsistent. This is a problem when the program later tries to access or use variables that are modified by the stalling code (but that hold incorrect values). For example, the program could compute a magic value inside a stalling loop that is checked afterwards; a simple mechanism to detect whether an analysis system has skipped the stalling code.

To handle the problem of *inconsistent variables* (which are variables that hold potentially incorrect values), we use taint analysis to track memory locations that might hold incorrect values (as well as the variables that are derived from these memory locations). When the program later uses a potentially inconsistent variable, we extract a slice that efficiently computes the correct value.

In a first step, we need to determine all memory locations that can potentially be written by the stalling code. We can then assign a taint label to these variables that marks them as inconsistent. Our underlying analysis platform already supports the tracking of taint labels. That is, whenever a computation (or data transfer operation) involves at least one tainted source operand, the destination is tainted as well. As a result, at any later point in time, the system can determine whether the value of a memory location was derived from an inconsistent variable.

To approximate all locations that can be written by the stalling code, we use the following technique: While the program is executing in passive mode, we label the destinations of all arithmetic and logic operations. Given that the stalling code is executed many times, we assume that we see all write targets at least once.

Whenever a labeled (tainted) value is used as an operand in a control flow operation (a conditional or indirect branch), an indirect memory access, or an argument in a system call, HASTEN needs to step in. This is because using an incorrect value as input to these *fragile operations* might yield visibly incorrect program behavior (such as program crashes, invalid outputs, etc.). This is different from "normal" instructions, which HASTEN handles in a lazy fashion by simply propagating taint labels.

Efficiently computing inconsistent variables. Whenever HASTEN encounters a fragile instruction that uses a tainted operand, we need to efficiently compute its correct, expected value. To this end, we extract a backward slice from the malware program that contains exactly the instructions necessary to compute the inconsistent value.

To extract the slice, we leverage our previous work on INSPECTOR [18], a tool to extract stand-alone programs (gadgets) from malware programs. More precisely, the goal of INSPECTOR is to extract, from a given malware binary, a slice that computes the argument values of interesting (security-relevant) system calls. This is very similar to our problem, where we are interested in a slice that computes the value of an inconsistent variable.

To perform its task, INSPECTOR employs binary backward slicing. This process exploits the fact that detailed runtime information can be gathered from the dynamic execution of malware (such as the targets of indirect memory accesses and control flow instructions). This allows the extraction of complex slices that span multiple functions and involve elaborate memory access patterns. For space reasons, we have to refer the reader to our previous paper [18] for a detailed discussion of INSPECTOR.

Once our system has extracted a slice, it can be executed as a stand-alone program by the *slice player* that comes with INSPECTOR. Note that the slice player performs no instrumentation, and it can execute the code directly on a native host. Thus, the slices execute the stalling code very fast, basically as fast as on the victim's machine. For stalling code that exploits the overhead introduced by slow emulation, this can speed up execution by several orders of magnitude. Once the slice has computed the required value, it replaces the current (incorrect) value of the tainted variable, and HASTEN continues the execution of the malware process.

7. EVALUATION

In this section, we evaluate HASTEN's ability to mitigate stalling code in malware binaries. We show that detecting signs of low progress works on samples found in the wild, and we test the effectiveness of HASTEN's different modes.

7.1 Malware Data Set

To evaluate our system, we randomly selected 29,102 samples from the files that were submitted to ANUBIS between August 2010 and February 2011. When picking malware programs for our data set, we did not include any invalid Windows PE files or files that failed to load in the analysis environment (e.g., due to unresolved library dependencies). Moreover, we did not select any sample that terminated within the time allocated for the original analysis run (in our case, 240 seconds). The reason is that these samples have already revealed their entire malicious activity during the allocated analysis timeframe. Our data set represents a diverse mix of malware currently active on the Internet.

We also retrieved the analysis report generated for each of the samples. We refer to these reports as the *base run*, and we use the observed behavior as one part to evaluate whether HASTEN is successful in revealing additional behavior.

7.2 Measuring Behavior

The goal of HASTEN is to reveal additional behaviors in malware samples that contain stalling loops. To be able to measure this in an automated fashion, it would be tempting to simply re-run each sample in our system and compare the observed behavior to this sample's base run. However, this would not be fair. The reason is that the behavior exhibited by a sample can significantly depend on the date/time of analysis and the availability of remote resources.

Thus, for a fair evaluation of HASTEN, we re-ran each of the samples twice: First, we performed a *redundancy run* with identical settings compared to the base run; in parallel, we conducted a *test run* using HASTEN. In this way, we attempted to minimize external influences on the behavior of a sample by eliminating the time between *test* and *redundancy runs* for each binary.

Whenever a test run reveals added behavior compared to the redundancy run, we also compare the redundancy run to the (initial) base run. If these two runs differ considerably, it is possible that the detected, added behavior is independent of HASTEN. To assess the added behavior produced by HASTEN, we use three different metrics: *Optimistic improvement (OI)* considers any added behavior seen during the test run (over the redundancy run) as an improvement due to HASTEN. *Average improvement (AI)* takes into account randomness in malware executions. To this end, we do not attribute any added behavior to HASTEN when the test run produces less added behavior than the corresponding redundancy run over the base run. *Pessimistic improvement (PI)* counts added behavior in the test run as new only when redundancy and base runs do not differ.

Added behavior. To determine added behavior when comparing two analysis runs, we only take into account *persistent features* of the corresponding behavioral profiles: A persistent feature is an activity performed by the malware that is visible from the outside of the analysis sandbox or that introduces permanent changes to the system. Examples for such features are communicating over network sockets, writing to files in the file system, or opening graphical windows. Table 3 (Appendix C) lists all combinations of resources and actions that we considered persistent features in this paper. In our setting, focusing on persistent features is reasonable, since they are indicative of the typical behavior used to classify and detect malware.

7.3 Evaluation Results

We analyzed each of the 29,102 samples in our test set twice; once without HASTEN (redundancy run), and once with our system enabled (test run). HASTEN has a negligible impact on analysis performance in monitoring mode and only introduces small overheads in the other modes. We considered extending the analysis time for the test runs to compensate for this difference. However, this difference is difficult to predict precisely. Thus, we conservatively used the same timeout (240 seconds) for all evaluation runs.

Monitoring mode results. 9,826 (33.8%) of the analyzed malware programs exhibited insufficient progress at some point during the analysis. That is, at least one of the heuristics triggered. In 98.3% of these cases, observing too few successful system calls was the predominant cause for switching into passive mode. An excessive number of successful and failed system calls were observed in 1.5% and 0.3% of the cases, respectively. Throughout our experiments, the heuristics for identical or suspiciously diverse calls never triggered. This does not come as a surprise, however, as we introduced these to increase the burden to bypass our system. With attackers becoming aware our system, we expect these detectors to become more important.

For the 9,826 low progress samples, HASTEN switched into passive, and, if necessary, also into active mode. The remaining 19,276 programs showed good progress. While this number appears large at first glance, it makes sense. We do not expect that a majority of malware samples in the wild already contains stalling code against sandboxes.

Passive mode results. Whenever HASTEN switches into passive mode, the system starts to record the control flow of the thread that stalls and tries to extract live loops. For the 9,826 samples that exhibited stalling behavior, the tool was able to do so in 6,237 (63.5%) cases.

We manually inspected some of the remaining 3,589 cases to determine why no loop was identified, despite low progress. We found that many samples were in a "mostly waiting" state; that is, the threads were sleeping, occasionally interrupted by the OS that invoked a callback function. In other cases, we found that "stalling loops" were of short duration, and threads made progress before HASTEN could complete the analysis of the loop. We attribute these cases to the conservative thresholds that we selected for the detectors in monitoring mode. That is, the malware processes show signs of low progress but it is not due to malicious stalling. Thus, we cannot expect that HASTEN can extract added behavior. Note that the negative impact of switching into passive mode is minimal (only a small performance loss).

We further investigated the 6,237 samples for which HASTEN discovered insufficient progress *and* extracted a live loop. In 3,770 cases, the system only switched into passive mode. In the remaining 2,467 test runs, the system activated the passive and, subsequently, the active mode. Table 1 details our findings of new, added behavior observed during the test runs, using the average improvement metrics (for a detailed overview of added behaviors using the optimistic and pessimistic metrics, refer to Table 4 in Appendix D). The table also shows the number of malware labels assigned to different sets of malware samples by a popular anti-virus product. We used only the family portion of the malware labels for this analysis (that is, `W32.Allaple.A` becomes `allaple`). Of course, we are aware of the limitations of labels assigned by anti-virus products. We just provide these numbers to underline the heterogeneity of our malware data set.

The left side of Table 1 shows that the passive mode allowed HASTEN to observe new behaviors in 1,003 runs (26.6% of all passive-only runs). With 25.2% and 14.9%, file and registry modifications are the most prevalent behavior detected by HASTEN, respectively. Furthermore, 11.8% of samples in this class participated in new networking behavior, which can lead to the exposure of new command and control servers or drop zones on the Internet.

HASTEN detected added behavior in 1,447 more cases. However, due to the conservative nature of our evaluation, and since the redundancy run also produced additional features, we cannot conclusively attribute this to our approach.

For the remaining 1,320 analysis runs, we did not observe new behavior, but neither did the system advance into active mode. Typically, this means that HASTEN did not manage to build a whitelist in time that covers the entire stalling code. More precisely, while building the CFG, some paths inside the stalling code were initially not observed, or a deeply nested stalling loop is executed. As a result, HASTEN repeatedly hits non-whitelisted code, which switches the system back into monitoring mode. After monitoring progress for another time slot, passive mode is re-enabled, and a more complete whitelist is generated. However, the analysis timeout is encountered before this iterative process can complete. A longer timeout or more aggressive whitelisting could be used to address such situations.

Active mode results. For certain samples, HASTEN did not observe progress despite reduced logging. Thus, the system proceeds to actively interrupt stalling code to force forward progress. In our evaluation set, HASTEN modified the control flow of 2,467 samples, revealing new behavior in 549 analysis runs (22.3%). The right-hand side of Table 1 shows details about the added behaviors.

While new behavior in passive mode always means additional, relevant activity, we need to be more careful in active mode: Since we actively change the flow of execution, we could follow infeasible paths that lead to errors (shown in the last row of Table 1). We handle inconsistent variables as discussed in the previous section, but the program might still show undesired behavior (such as premature program exits or exceptions). The reason is that the program could be forced down an error path that does not reference any tainted variable modified in the loop, but simply performs some clean-up activities and then raises an exception. A prominent example are the checks that are introduced by Microsoft's compiler to guard the return addresses on the stack against overflow corruption. As a result, when HASTEN encounters as exit point the conditional branch that checks for stack corruption and forces execution down this path, the program terminates with an exception (see the next Section 8 for a discussion on how this can be addressed). Of course, we discard all added behaviors that are due to exceptions (such as the start of the debugger).

Overall, HASTEN detected the use of inconsistent variables during 780 analysis runs in active mode. In 778 cases, INSPECTOR needed to be invoked once, for two runs, two invocations were necessary.

Discussion. Table 1 demonstrates that HASTEN was able to reveal interesting added behavior for 1,552 (1,003 + 549) samples, using the average improvement metrics. This number increases to 3,275 when considering the optimistic measure for additional behavior (as shown in Table 4). This behavior relates to significant activity, such as file system accesses or network traffic. Our numbers show that HASTEN successfully found additional behavior in 24.5% (or 52.5%, when using the optimistic metrics) of the 6,237 samples that contain some kind of loop that delays progress in a sandbox. The failure to expose additional behavior for the remaining samples has several reasons: Most often, some necessary resource (e.g., a command and control server) was no longer available, thus, the malware stopped its execution. In other cases, HASTEN switched

Table 1: Additional behavior observed in Hasten (using *average improvement*).

Description	Passive			Active		
	# samples	%	# AV families	# samples	%	# AV families
Runs total	**3,770**	–	319	**2,467**	–	231
Added behavior (any activity)	**1,003**	**26.6%**	119	**549**	**22.3%**	105
- Added file activity	949	25.2%	113	359	14.6%	79
- Added network activity	444	11.8%	52	108	4.4%	31
- Added GUI activity	24	0.6%	15	260	10.5%	51
- Added process activity	499	13.2%	55	90	3.6%	41
- Added registry activity	561	14.9%	82	184	7.5%	52
Ignored (possibly random) activity	**1,447**	**38.4%**	128	**276**	**11.2%**	72
No new behavior	**1,320**	**35.0%**	225	**1,642**	**66.5%**	174
- Exception cases	0	0.0%	0	277	11.2%	63

into passive mode towards the end of the analysis timeframe, and a timeout was encountered before additional behaviors could be exposed. Finally, in some cases, Hasten selected an error path that lead to premature program termination.

We also wanted to understand in more detail the nature of the stalling code introduced by malware authors. To this end, we focused our attention on the 1,552 samples that revealed added behavior. While it is difficult to determine with certainty that a particular piece of code has been deliberately inserted by a malware author to prevent dynamic analysis, we checked for the presence of many repeated calls to API functions that serve no apparent purpose. In particular, during manual analysis of a few samples, we observed repeated calls to `GetTickCount` and `GetCommandLine`. We then checked the prevalence of these calls in the execution traces of the 1,552 samples. We found that more than a third of the samples (543) invoked one of these two functions more than 5,000 times inside a loop.

Even when considering the entire data set of 29,102 malware programs, a surprisingly large number of samples contains stalling code. This demonstrates that execution stalling is not a purely theoretical problem. Instead, a non-trivial fraction of malware in the wild already includes such evasive techniques. Thus, it is crucial that sandboxes are improved to handle this threat.

8. DISCUSSION AND LIMITATIONS

As mentioned previously, the fight against malicious code is an arms race, and malware authors always strive to improve their programs to resist new analysis and detection approaches. In this section, we discuss the robustness of our approach and possible, future improvements to further harden our techniques.

Malware authors can target each of the three modes of Hasten: First, the stalling loop could be crafted so that it delays the execution of malicious activity in the sandbox without triggering any of the detectors in **monitoring mode**. To this end, a malware process would have to issue successful system calls with a frequency and diversity similar to non-stalling programs. To counter such mimicry attacks, we can add additional detectors that measure progress in different ways. For example, we could count, for each time interval, the number of distinct basic blocks that a program executes or the number of new (never before executed) basic blocks that Qemu, our system emulator, has to translate. For stalling code, we expect both values to be low. Moreover, we could opportunistically (and randomly) switch to passive mode every once in a while, even when no detector has raised an alert.

Likewise, it is possible that long-lasting operations (e.g., the decryption of large files) trigger one of our heuristics. Internally, we refer to this as *accidental stalling* code, since the malware author does not do this deliberately. This is similar to opportunistically switching to passive mode and introduces only a minor performance hit.

Passive mode is more difficult to exploit for the attacker. While we reduce the amount of information that is logged, we are careful to preserve all security-relevant information. To this end, we only whitelist code regions (basic blocks) that have previously been executed by the malware. For example, consider a malware author who puts a piece of malicious code into the body of a stalling loop and guards this code with a conditional branch. When the passive mode recognizes the stalling loop, it will only whitelist the part of the loop that has been executed before. The malicious code region, on the other hand, will be excluded from reduced logging. Moreover, certain security-critical system calls are always recorded.

The **active mode** seems most vulnerable to attacks since it modifies the normal flow of program execution, and, thus, could leave the program in an inconsistent state or force it along an impossible path. Hasten addresses the problem of inconsistent program state by tracking the variables (memory locations) that stalling code modifies. Moreover, through taint analysis, all variables derived from these memory locations are tracked as well. Whenever the malware process attempts to check or use a potentially inconsistent variable, its execution is suspended. Then, the system generates a slice that will efficiently compute the variable's correct value, and this value is provided to the program before it is allowed to continue. Of course, a malware author can force our system to extract and execute slices. However, as described in more detail in [18], the slices that Inspector generates are small, stand-alone executables (gadgets) that can be executed directly on a native host. Moreover, slice generation is fast. Thus, the stalling code will be run almost as fast as on a native (victim) machine.

An important limitation is our current, simple technique to determine all variables (memory locations) that stalling code can write to. Since this technique uses only dynamic information, it might miss certain locations. One way to address this problem is to incorporate static analysis to identify and taint such variables. This analysis can be conservative, as "over-tainting" can only result in unnecessary calls to Inspector but will not cause incorrect program executions. Furthermore, it is possible that Inspector cannot extract a proper slice to compute a needed value. This is less of a concern, as our previous work has shown that the tool is able to extract meaningful slices from real-world malware programs that span multiple functions and that contain non-trivial memory accesses. Also, Inspector will report the failure to extract a slice, allowing further manual inspection.

An attacker could also try to force Hasten to skip a stalling loop that contains malicious code (and thus, fail to observe malicious activity). This is difficult for the attacker because *all* conditional branches that exit any whitelisted stalling code region are possible exit points. In particular, when there is previously non-executed, malicious code inside a stalling loop, the branch that guards this malicious code is considered as a possible exit point (in addition to the loop exit). Recall that we only whitelist code that was previously executed; this implies that we will not whitelist the entire body of a stalling loop when there are paths through this loop that were not executed before.

In our current prototype, we force the program execution out of a stalling region through the first exit point that is encountered. This has two drawbacks. First, we might skip malicious code inside the loop (when the wrong exit point is chosen). Second, it is possible that HASTEN picks an incorrect (infeasible) loop exit path that later results in a program exception (as mentioned in the previous section, an example are error paths). To tackle these problems, we can leverage the fact that our system has full control over the analysis environment. More precisely, we can create a snapshot of the CPU and the memory before any active modification to the program's control flow. This way, HASTEN can later revert to previous snapshots and explore multiple exit paths in a more systematic fashion. This is similar in spirit to a multi-path exploration of the binary [21].

9. RELATED WORK

The idea of detecting and evading dynamic malware analyzers is not new. In the past, malware authors have introduced checks, so-called red pills, to detect execution inside a virtualized environment [13, 24]. Other work has proposed techniques to detect execution in system emulators, such as Qemu [20, 22, 23]. Yet other research [1, 7] has discussed practical fingerprinting techniques to recognize public, dynamic malware analyzers (e.g., ANUBIS [2], CWSANDBOX [3], JOEBOX, etc.). Fingerprinting a malware analysis system is particularly easy when the system provides a public interface that accepts submissions from anywhere.

To mitigate the problem of checks that detect malware analyzers, researchers have proposed transparent analysis environments [11, 12]. These systems leverage novel virtualization features of modern processors to implement a minimalistic hypervisor. This makes it more difficult to reveal the presence of the analysis platform.

Others have focused on the detection of differences between the execution of a program in a virtual or emulated platform, and on real hardware [6, 17]. The basic intuition is that any difference between the executions in two different environments (using deterministic replay) must be due to checks in a sample. If one can determine that a sample is trying to evade analysis, the evasive check can either be removed, or the sample can be marked for manual inspection.

Finally, checks that detect the analysis environment can also be bypassed by systems that perform multi-path exploration [9, 21, 28]. The idea is to first identify conditional branches that depend on some input read from the program's environment (file system, registry, network, ...). Once such a branch (check) is found, both alternative execution branches can be explored. When the system explores the branch that is different from the the current program execution, care must be taken to keep the program state consistent. In a related approach [10], the authors try to identify behaviors that are triggered by timer events.

The crucial difference between previous research and this work is that previous techniques focused on the detection and mitigation of evasive checks. Such checks are inserted by malware authors to identify dynamic analysis environments and, subsequently, change the behavior of their programs. In this paper, we present the first solution to the growing problem of dynamically analyzing malware samples that include stalling code. Stalling code does not use any checks, and it does not alter the flow of execution of the malware program in a dynamic analysis environment. As a result, transparent platforms or systems that detect and remove evasive checks are ineffective.

10. CONCLUSIONS

As new malware analysis solutions are introduced, attackers react by adapting their malicious code to evade detection and analysis. One recent form of evasion code for dynamic analysis systems is stalling code. In this paper, we present the first approach to detect and mitigate malicious stalling code, and to ensure forward progress within the amount of time allocated for the analysis of a sample. Our results show that HASTEN works well in practice, and is able to reveal additional behaviors in real-world malware samples that contain stalling code.

11. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n. 257007 (SysSec), the Austrian Research Promotion Agency (FFG) under grant 820854 (TRUDIE), the NSF under grant CNS-1116777, the ONR under grant N000140911042, and the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537. We also acknowledge Secure Business Austria for their support.

This publication reflects the views only of the authors, and the funding agencies cannot be held responsible for any use which may be made of the information contained therein.

12. REFERENCES

[1] Forum Posting - Detection of Sandboxes. `http://www.opensc.ws/snippets/3558-detect-5-different-sandboxes.html`, 2009.

[2] `http://anubis.iseclab.org`, 2010.

[3] `http://www.cwsandbox.org`, 2010.

[4] `http://www.norman.com/enterprise/all_products/malware_analyzer/norman_sandbox_analyzer/en`, 2010.

[5] `http://msdn.microsoft.com/en-us/library/ms724408%28VS.85%29.aspx`, 2010.

[6] BALZAROTTI, D., COVA, M., KARLBERGER, C., KRUEGEL, C., KIRDA, E., AND VIGNA, G. Efficient Detection of Split Personalities in Malware. In *Network and Distributed System Security Symposium (NDSS)* (2010).

[7] BAYER, U., HABIBI, I., BALZAROTTI, D., KIRDA, E., AND KRUEGEL, C. A View on Current Malware Behaviors. In *Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2009).

[8] BAYER, U., MILANI COMPARETTI, P., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium* (2009).

[9] BRUMLEY, D., HARTWIG, C., LIANG, Z., NEWSOME, J., SONG, D., AND YIN, H. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Tech. Rep. CMU-CS-07-105, Carnegie Mellon University, 2007.

[10] CRANDALL, J., WASSERMANN, G., DE OLIVEIRA, D., SU, Z., WU, F., AND CHONG, F. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006).

[11] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security* (2008).

[12] Fattori, A., Paleari, R., Martignoni, L., and Monga, M. Dynamic and Transparent Analysis of Commodity Production Systems. In *International Conference on Automated Software Engineering (ASE)* (2010).

[13] Ferrie, P. Attacks on Virtual Machines. In *Proceedings of the Association of Anti-Virus Asia Researchers Conference* (2007).

[14] Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., and Yan, X. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *IEEE Symposium on Security and Privacy* (2010).

[15] Freiling, F., Holz, T., and Wicherski, G. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symposium On Research In Computer Security (ESORICS)* (2005).

[16] John, J., Moshchuk, A., Gribble, S., and Krishnamurthy, A. Studying Spamming Botnets Using Botlab. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[17] Kang, M., Yin, H., Hanna, S., McCamant, S., and Song, D. Emulating Emulation-Resistant Malware. In *Workshop on Virtual Machine Security (VMSec)* (2010).

[18] Kolbitsch, C., Holz, T., Kruegel, C., and Kirda, E. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *IEEE Symposium on Security and Privacy* (2010).

[19] Kolbitsch, C., Milani Comparetti, P., Kruegel, C., Kirda, E., Zhou, X., and Wang, X. Effective and Efficient Malware Detection at the End Host. In *Usenix Security Symposium* (2009).

[20] Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. Testing CPU Emulators. In *International Symposium on Software Testing and Analysis (ISSTA)* (2009).

[21] Moser, A., Kruegel, C., and Kirda, E. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy* (2007).

[22] Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *usenix-woot* (2009).

[23] Raffetseder, T., Kruegel, C., and Kirda, E. Detecting System Emulators. In *Proceedings of the Information Security Conference* (2007).

[24] Rutkowska, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://www.invisiblethings.org/papers/redpill.html`, 2004.

[25] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. BitBlaze: A new approach to computer security via binary analysis. In *Conference on Information Systems Security (Invited Paper)* (2008).

[26] Sreedhar, V. C., Gao, G. R., and fong Lee, Y. Identifying loops using DJ graphs, 1995.

[27] Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[28] Wilhelm, J., and Chiueh, T.-c. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Recent Advances in Intrusion Detection*. 2007.

APPENDIX

A. MONITORING MODE: PARAMETERS

Figures 5a and 5b show information about the number of successful and failed system calls, respectively, that we observed for the 5K malware samples in our training set (see Section 4). More precisely, the figures show, for a particular number n of system calls (depicted on the x-axis), the (absolute) number of time slots during which we have seen n system calls (depicted on the y-axis). For example, when analyzing the samples, we have recorded 150 successful system call invocations for roughly 1,000 intervals (see Figure 5a). Likewise, Figure 5c shows that the number of times (y-axis) a certain entropy value (x-axis) was encountered. Table 2 shows the thresholds that we derived from the training runs for Hasten's detectors.

Table 2: Threshold values used in monitoring mode (for slot time duration $t = 5$ seconds).

Type	Name	Threshold (min)	Threshold (max)
Successful	$S_{min/max,s}$	3	600
Failed	$S_{max,f}$	–	900
Entropy	$S_{min/max,e}$	0.17	20.0

B. HANDLING NESTED STALLING LOOPS

When Hasten operates in passive mode, its goal is to find and whitelist code that stalls execution. To this end, the system checks the program's dynamic CFG for loops. When a number of active, nested loops are found, Hasten whitelists the innermost one (as discussed in Section 5.2). This is a problem when the innermost, active loop is only a part of the entire stalling code (we have encountered cases in real-world malware where a stalling loop is itself executed many times by an outer loop). In this case, the system reverts back to monitoring mode as soon as the innermost loop is finished. Unfortunately, the entire execution still does not make progress, and the outer loop will simply execute the inner loop again. Of course, the monitoring phase will detect insufficient progress, and the system will be switched into passive mode. However, our analysis will find the same active loops again. At this point, whitelisting the inner loop only would be the wrong solution. This would result in an execution where the system would constantly switch between monitoring and passive mode. To address this problem, the system keeps track of all loops that have been previously whitelisted. When we find that an active loop has been previously whitelisted, but no progress has been made, the system extends the whitelist to include the next enclosing loop. That is, nested loops are incrementally added to the whitelist, starting from the inner-most one, proceeding outwards.

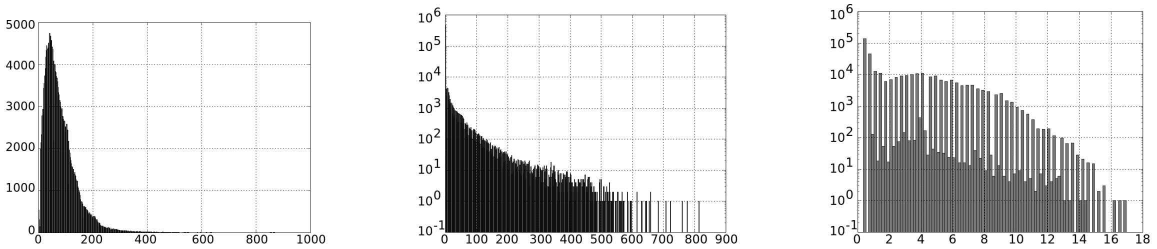
C. PERSISTENT BEHAVIORAL FEATURES

Table 3: Persistent behavioral features.

Resource	Action
file	create, delete, write, open-truncate rename, set-information
network	*any*
registry	keys: create, delete, save, restore; set-value, set-information
process	create, terminate, set-information
driver	load, unload
GUI	open window

D. DETAILED FINDINGS

As mentioned in Section 7.2, we can use different approaches for measuring additional behavior. Table 4 shows additional behaviors detected by Hasten as measured by the optimistic and pessimistic and metrics.



(a) Successful system calls S_s.

(b) Failed system calls S_f.

(c) System call entropy S_e.

Figure 5: Evaluating the progress of active samples by bucketing number of observed system calls (x-axis) and counting observed number of time-slots for each bucket (y-axis).

Table 4: Additional behavior observed in Hasten.

Description	**Passive**			**Active**		
	# samples	**%**	**# AV families**	**# samples**	**%**	**# AV families**
Runs total	3,770	–	319	2,467	–	231
	Optimistic Improvement					
Added behavior (any activity)	2,450	65.0%	188	825	33.5%	139
- Added file activity	1,873	49.7%	169	519	21.0%	100
- Added network activity	906	24.0%	73	128	5.2%	39
- Added GUI activity	28	0.7%	16	367	14.9%	69
- Added process activity	895	23.7%	79	127	5.1%	48
- Added registry activity	795	21.1%	107	224	9.1%	62
No new behavior	1,320	35.0%	225	1,642	66.5%	174
- Exception cases	0	0.0%	0	277	11.2%	63
	Pessimistic Improvement					
Added behavior (any activity)	416	11.0%	86	500	20.3%	94
- Added file activity	390	10.3%	80	314	12.7%	69
- Added network activity	257	6.8%	41	95	3.9%	29
- Added GUI activity	20	0.5%	12	249	10.1%	49
- Added process activity	177	4.7%	42	70	2.8%	34
- Added registry activity	331	8.8%	60	165	6.7%	46
Ignored (possibly random) activity	2,034	54.0%	156	325	13.2%	84
No new behavior	1,320	35.0%	225	1,642	66.5%	174
- Exception cases	0	0.0%	0	277	11.2%	63