Burn Before Reading:

A Stealthy Framework for Combating Live Forensics Examinations

MINA GUIRGUIS JASON VALDEZ BASSAM EL LABABEDI JOSEPH VALDEZ
{msg,jv1150,bl1111,jv29382}@txstate.edu
Computer Science Department
Texas State University
San Marcos, TX 78666, USA

***Abstract*— Malicious Software/programs (Malware) have grown to be quite sophisticated in their design causing significant levels of damage and for prolonged periods of time. Their capabilities encompass a wide range of activities; from simple monitoring/spying programs to more complex, highly destructive tools. Moreover, they typically aim to hide their own existence through a large number of techniques. To that end, this paper demonstrates that the full malicious potentials of malware have not been realized yet. In particular, we present a novel framework – which we term Burn Before Reading (BBR) – that actively aims to detect potential live forensics investigations and adapts the behavior of the malware online. In a nutshell, the BBR framework registers for a set of triggers that typically occur in live forensics investigations. Once a trigger fires, BBR executes actions as dictated by the malware to destroy any evidence. To remain stealthy during the execution of those actions, BBR utilizes control-theoretic actuators that dynamically adjust the timing information for the executing modules, at a very fine time scale (in the order of micro seconds). We study the stability regions for those actuators under different parameters. We believe that this framework can be used by malware to destroy incriminating evidence, their own signatures and their own existence and thus its capabilities should be brought to the attention of the forensics and security communities. We also discuss potential defense mechanisms against BBR.**

I. INTRODUCTION

Motivation: Over the past decade, we have witnessed a major increase in the number of malicious software (malware) on computer systems and networks. Malware – as in viruses, worms, bot programs, trojans, spyware, backdoors, rootkits – are being released at an alarming rate that may exceed that of legitimate software [1]. Moreover, they have mutated from simply "annoying programs" to more complex "financially-aware services" that run a black industry [2]–[4]. The impact of malware has been (and would likely remain for some time) quite significant. For example, the latest CSI report for the year 2008 shows that companies which dealt with "bot" computers within their network reported an average loss of $345,000 per respondent [5], that is in addition to other costs in dealing with viruses, and other exploits.

The software architectures in malware have grown to be quite sophisticated in their design and operation. Many of them are up-to-date with the current security methods used and aim to bypass them. Some of them even update their packages automatically, similar to what major software packages do. Typically, malware utilizes a wide range of stealth techniques to avoid being detected [6]–[8]. However, once a suspicious behavior is detected, an examination is typically performed that is based on the nature of the computer involved and the extent of the damage inflicted. At one end, the examination can be as simple as a regular user seeking to know if his/her computer is compromised through monitoring the process list, along with executing simple tools (e.g., FileMon, RegMon, etc). At the other end, a large investigation could involve multiple computers, possibly across different corporations, and would likely involve the authorities. In both cases a decision has to be made on whether a live response is needed versus turning the system off and carrying an off-line investigation. In many environments, servers cannot be taken off-line as it would have severe economic losses. This paper is mainly focused on live forensics investigation concerning a piece of malware that is found on a major server that cannot be taken off-line.

To date, no malware has been found that reacted online towards ongoing live forensics investigations in a systematic manner. The stealth techniques that they utilize are mainly static in nature and primarily used to prevent them from being noticeable (discussed in Section V). But, what if malware was able to *utilize the footprint from live forensics investigations towards its own benefit and react accordingly?* We argue that the full malicious potentials of malware have not been realized yet. Upon detecting an ongoing live forensics investigations, malware can destroy incriminating evidence, their own signatures and their own existence. That is, in addition to implicating the investigator.

In this paper we present a comprehensive, stealthy framework – which we term Burn Before Reading (BBR) – that aims to prevent a piece of malware, along with its data files and signatures, from being captured/revealed by an ongoing forensics examinations. BBR allows the execution of different actions by the malware upon detecting an ongoing live forensics examination. These actions include encryption, overriding, and complete destruction among other customizable actions. Indeed, in many occasions, destroying the malware along with its signatures may be natural course of action for a smart

attacker who wishes to remain unlinked to the malware. The actions are performed based on particular triggers that indicate a possible live investigation.

The procedures carried out by first responders and investigators are typically well known and their guidelines are publicly available [9], [10]. Thus, it is safe (and even encouraged!) to assume that the malware is aware of those guidelines. If one of those events occurs, the BBR framework will destroy the incriminating data preventing it from being captured. Typical triggers include the insertion of a particular USB device [11], [12], the insertion of CDs [13], or the detection of a live response over the network [14]. As we will explain later, BBR can also allow for more actions to elude investigator and significantly complicate their task. Notice that in such cases, the malware is actually exploiting the knowledge of those procedures towards its own benefit.

The BBR framework utilizes control theoretic tools to adjust its dynamic operation in order to minimize its own footprint on the overall system. This makes detecting the presence of BBR a much harder process. The work in this paper is motivated by several case scenarios, however, we outline here one of the most prevalent ones.

An Illustrative Example: Consider an attacker who infected a major transaction server with a backdoor to steal sensitive information. The backdoor sends the attacker, either directly or indirectly, sensitive information. It is typical that this backdoor would have a footprint on the server (open ports, registry key, etc..). Now consider the case that a forensics live response is requested to be performed on this server due to a suspicious activity. [1] Through the BBR framework, the backdoor running on the server can register for some triggers that typically occur in a live response. Once one of the trigger fires, BBR would, in a stealthy manner, destroy the footprints and even wipe the backdoor completely, so that a link cannot be established between the backdoor and the attacker.

Contributions: We summarize our contributions in the following points:

- We introduce a new concept on how malware can adapt its behavior upon detecting ongoing forensics examinations. This concept will likely be adopted by smart attackers who wish to remain unlinked to the malware distribution and its functions.
- We provide a comprehensive framework that ties events/triggers that occur during forensics examinations to actions that prevent the detection of malware and tracing back to the attacker. The framework is flexible enough to allow attackers to customize the operation of their malware upon detecting forensics examinations.
- We use a control theoretic framework to develop the execution actuators that dynamically control the execution of actions in order to minimize the footprint on the system. This would strengthen BBR against detection. We also analyze its stability margins under different parameters.
- Currently, many forensics tools rely on inserting different forms of media to collect evidence during a live response. The presence of BBR can make such events work in favor of the suspect. We believe that this work gives an early warning to forensic investigators to what they may be up against and would help the forensics community in building better products. Towards this goal, we discuss improvements to the forensics applications in order to hide their footprints during investigations so that they cannot be picked up by BBR and BBR-like tools.

Throughout this paper, we use capital BBR to indicate the main framework, while we use small-letter `bbr` to refer to our prototype implementation of BBR.

Finally, it is important to note here that our motivation in this paper is not to promote the use of BBR by suspects to destroy incriminating evidence and avoid prosecutions, but to improve the state-of-the-art forensics tools and techniques.

Paper Organization: In section II, we describe the overall design of BBR, with all of its goals and components. In Section III, we present experimental results from our evaluation of our prototype implementation of `bbr`. In Section IV, we discuss extensions, limitations and defense mechanisms. We present a taxonomy of related work in Section V. We conclude the paper in Section VI with a summary and ongoing work.

II. The BBR Framework

In this section, we describe the Burn Before Reading (BBR) framework in more details. We start with an overview. Then, we explain its components and execution engine. Finally, we analyze its stability based on a linearized model.

A. An Overview

BBR is a framework that enables malware to adapt its behavior upon detecting an ongoing forensics investigation. The main idea is to provide BBR with actions to be performed, along with a list of triggers to register for. Once a registered trigger fires, BBR executes the actions. The main operation of BBR follows two stages, a preparation stage and an execution stage. A general block diagram for BBR is given in Figure 1.

In the preparation stage, a preprocessor receives an *action* file and a *trigger* file. These files are compiled into a list of modules that are *ready for execution*. BBR would then go into a silent mode waiting for a registered trigger to fire. Once this happens, BBR goes into the execution stage (possibly after a controlled delay period) to execute the actions. Since the malware itself may potentially need to execute an action that may cause a registered trigger to fire, BBR can be disarmed and run in a passive mode. In a passive mode, registered trigger do not cause actions to be executed[2].

In the execution stage, BBR executes the modules through actuators that dynamically adjust the demand on the resources in order to keep BBR's own utilization of the resources at a target level. This would help BBR to evade detection.

[1] Since this is major transaction server, it cannot be taken off-line, and hence a live response is needed as opposed to an off-line analysis.

[2] Notice that selective triggers can be easily employed, where a malware can arm/disarm a subset of triggers.

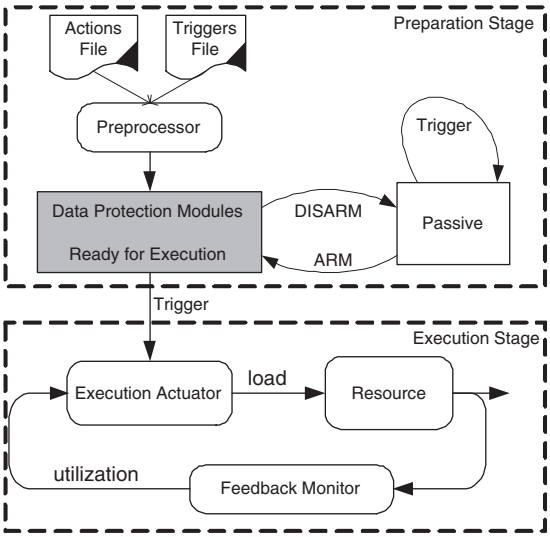

Fig. 1. Basic BBR operation.

B. Actions

For malware to protect their own existence, BBR allows for three main actions to be performed on data: deletion, encryption and overriding. Also, BBR allows for a fourth type of action that enables custom executions of programs. This becomes handy for customizable treatment, and for launching other programs that aid BBR to run in a stealthy mode. We explain those actions in more details below.

1) *Deletion (DEL):* This action securely deletes a file. There are many available tools that can be used within BBR. This option can be used when the data is either extremely sensitive (better to destroy it than have it fall into the hands of someone else) or it is a redundant copy and an original source exists elsewhere.
2) *Encryption (ENC):* This action encrypts a file. Encryption is done with a public key. The associated private key is never shipped with the malware. This action can be used if the attacker believes the he/she may regain access to this computer at a later point (e.g., the forensics investigation is called off).
3) *Over-writing (OVW):* This action overwrites a whole file, a block of a file or a registry key with random data that has the same size. Like DEL, OVR would destroy the data. It can be used on other forms of data containers as registry key.
4) *Running (RUN):* This action executes a particular program. We envision that this option can be used for executing custom programs like overriding certain memory portions, wiping slack spaces, etc. Also, they can alert the malware designer or implicate the investigator.

Each action is specified along with a type, path and a priority. All actions are contained in an action file that is passed to BBR for preprocessing. Executing those actions will follow the priorities specified. Figure 2 gives a sample of an action file.

Action	*Type*	*Path*	*Priority*
ENC	FILE	C:\Data\taxes.xls	
DEL	FILE	C:\Data\medical.xls	
OVW	KEY	HKEY_USERS/…/Last Username	
DEL	FILE	C:\BBR\actions.bbr	
RUN	EXE	C:\wipe_memory_until_crash.exe	

Fig. 2. An example of an action file.

C. Triggers

BBR waits for one trigger to fire before it starts executing the actions. Each trigger is specified with a type, signature, and a delay value. All triggers are contained in a trigger file that is passed to BBR. The delay value makes BBR sleep for the value specified before it starts executing the actions. This can give the malware enough time to disarm BBR, in case the malware was aware of the event happening. We explain the classes of triggers in more details below.

1) *Media Insertion:* This trigger fires once a CD/DVD or a USB device gets plugged into the computer. A signature can be specified to match to known CDs (e.g., HELIX) and USBs (e.g., e-fence) that may be typically used during a live response.
2) *Live Response over the Network:* This trigger fires once BBR detects an ongoing live response over the network. BBR monitors incoming and outgoing traffic to detect interactive ones. This is typically done by monitoring the size and the inter-arrival times of packets as explained in [15], [16]. To prevent false positives (connections that appear to be interactive, but are not) from executing actions, BBR *correlates the times of arrivals of small packets with the times of new processes launched on the system*. BBR seeks to look for processes that are directly related to a forensics investigation, such as top, who, ps, netstat, etc.
3) *Startup/Waking Up:* These triggers fire once the computer starts up or wakes up from hibernation.

D. Execution Stage

When a trigger fires, BBR starts executing the associated actions. This may cause a surge in demand on the resources available and the presence of BBR may be detectable. In order to remain stealthy, BBR dynamically controls the executions of actions. To study this behavior in more detail, the BBR framework is modelled using three main components, the *resources*, the *feedback monitors* and the *execution actuators*.

The Resources: Resources are utilized while executing actions. BBR accounts for different resources, such as CPU, disk, and network.

The Feedback Monitor: The feedback monitor periodically measures the utilization of the resources for BBR's *own actions* and reports the values back to the actuators.

The Execution Actuator: The utilization of each resource (due to the execution of BBR actions), is controlled by an

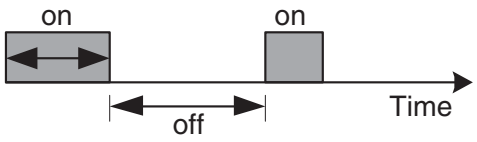

Fig. 3. Dynamically adjusting the on and off periods for each action.

execution actuator. The actuator decides the load through adjusting the on and off periods of each action as depicted in Figure 3. Table I summarizes the parameters used in our model.

Parameter	Description
$\rho(i)$	BBR Resource utilization at time step i
$\bar{\rho}(i)$	BBR Average utilization at time step i
$n(i)$	The ON period at time step i
$f(i)$	The OFF period at time step i
G_{on}	Gain parameter for the ON period
G_{0ff}	Gain parameter for the OFF period
β	Averaging parameter

TABLE I
MODEL PARAMETERS.

In this paper, we use a Proportional Integral (PI) actuator [17] to adjust the on and off periods based on an error signal between the average utilization $\bar{\rho}(.)$ of the resource (from executing BBR actions) and a desirable set value ρ^*. At time instant i, the actuator adjusts the on period value, $n(i)$, based on the following equation:

$$n(i) \quad = \quad n(i-1) + G_{on} \times (\rho^* - \bar{\rho}(i)) \tag{1}$$

Similarly, the off period, $f(i)$, is adjusted based on the following equation:

$$f(i) \quad = \quad f(i-1) - G_{off} \times (\rho^* - \bar{\rho}(i)) \tag{2}$$

The constants G_{on} and G_{off} decide the gain of the system which in turn determines the aggressiveness of the actuator based on the error signal between the average utilization $\bar{\rho}(i)$ and the target utilization ρ^*. The impact of other forms of actuators can also be studied within this context. For example, one can think of an actuator that can increase the on period very slowly (in a linear fashion) and decrease the off period very fast (in an exponential fashion). This would increase the stealthiness of BBR significantly.

Notice that if we directly use the instantaneous utilization, $\rho(.)$, instead of the average utilization $\bar{\rho}(.)$, we may be at the risk of having oscillations since the instantaneous signal may vary significantly. In order to obtain a smoother signal to use for control, we use an Exponential Weighted Moving Average (EWMA) according to the following equation:

$$\bar{\rho}(i) \quad = \quad \beta\bar{\rho}(i-1) + (1-\beta)\rho(i) \tag{3}$$

where β is a smoothing parameter that decides the weight on the history versus the new measurement. The utilization of the resource, $\rho(i)$, represents the percentage of time the resource is busy executing BBR's actions (in the absence of any other processes) and is given by:

$$\rho(i) \quad = \quad \frac{n(i)}{n(i) + f(i)} \tag{4}$$

On a given system, other processes would share the same resource. BBR will continue adjusting the on/off periods so that its own resource utilization is matching the target utilization specified.

E. Stability Analysis

The overall performance of the execution model presented above relies on the choice of parameters (e.g., G_{on}, G_{off} and β). Fortunately, control theory provides a number of techniques to study the impact of these parameters and to guide the process of their selection, so that the steady-state and transient properties can be controlled. Here, we show the effect of parameter choice on the overall stability and performance of the system. We derive our control-theoretic results for the case of a single resource under investigation.

A standard control-theoretic approach to studying a non-linear dynamical system is to linearize it around an operating point. This is done by expanding the equations using Taylor series and ignoring the high order terms. After linearization, the variables in the equations describe *perturbations* around the operating point that we have chosen. The question then becomes whether the system converges to the chosen operating point, i.e. whether perturbations in the system outputs eventually vanish after the system inputs were subjected to small perturbations. Clearly, based on our choice of the operating point and parameters, we would obtain a different set of linear equations.

We consider a continuous time model similar to the one described in Section II. We denote $\dot{x}(.)$ to be the derivative function of $x(.)$. The equivalent continuous-time equations for equations 1 and 2 are given by:

$$\dot{n}(t) \quad = \quad G_{on} \times (\rho^* - \bar{\rho}(t)) \tag{5}$$

$$\dot{f}(t) \quad = \quad -G_{off} \times (\rho^* - \bar{\rho}(t)) \tag{6}$$

Also the averaging continuous-time equation for equation 3 is given by:

$$\dot{\bar{\rho}}(t) \quad = \quad -\beta(\bar{\rho}(t) - \rho(t)) \tag{7}$$

Equation 4 is nonlinear. Linearization around an operating point characterized by an on period of n^* and an off period of f^* is given by:

$$\rho(t) \quad = \quad K_1 n(t) + K_2 f(t) \tag{8}$$

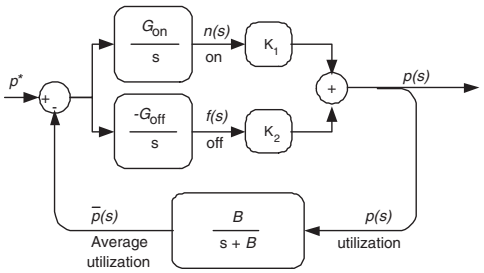

Fig. 4. A linearized block diagram.

where K_1 is given by $\frac{f^*}{(n^*+f^*)^2}$ and K_2 is given by $\frac{-n^*}{(n^*+f^*)^2}$.

Taking the Laplace transforms for the above linear model, we get:

$$n(s) = \frac{G_{on}}{s} \times (\rho^* - \bar{\rho}(s)) \quad (9)$$

$$f(s) = \frac{-G_{off}}{s} \times (\rho^* - \bar{\rho}(s)) \quad (10)$$

$$\bar{\rho}(s) = \frac{\beta}{s+\beta}\rho(s) \quad (11)$$

$$\rho(s) = K_1 n(s) + K_2 f(s) \quad (12)$$

Thus the closed-loop transfer function, $\frac{\rho(s)}{\rho^*}$ is given by:

$$\frac{\rho(s)}{\rho^*} = \frac{(K_1 G_{on} + K_2 G_{off})(s+\beta)}{s^2 + \beta s + \beta(K_1 G_{on} + K_2 G_{off})} \quad (13)$$

The location of the roots of the characteristic equation (denominator of equation 13) in the s-plane determines the stability of the overall system and the nature of its transient behavior [17]. The roots of characteristic equation r_1 and r_2 are given by:

$$r_1, r_2 = \frac{-\beta}{2} \pm \frac{\sqrt{\beta^2 - 4\beta K_3}}{2} \quad (14)$$

where K_3 is equal to $(K_1 G_{on} + K_2 G_{off})$.

As the value of β in equation 14 varies from 0 to $4K_3$, both roots are imaginary and their real components lie inside the *left* half of the s-plane, leading to an oscillatory convergence. If we chose β to be beyond $4K_3$, the roots become real and again they lie inside the left half of the s-plane, ensuring exponential convergence. Notice that there is no positive value for β that would cause the system to be unstable, since β is always larger than $\frac{-1}{4K_3}$, where K_3 is always positive.

One of the diagrams that illustrate the stability regions for a given system is the root locus plot. The root locus plot represents the locations of all possible closed loop poles as we vary the system gain under unity feedback. In our case, the gain of the system is a factor of G_{on} and G_{off}. Figure 5 shows the root locus plot for the case where β is chosen to be 0.5. The operating point was chosen with n^* equals to 5 and f^* equals to 20. One can see that the locations of the roots reside on the left half of the s-plane (ensuring convergence). Moreover, with the proper choice of parameters, one can ensure exponential convergence (when the roots are on the x-axis without an imaginary component).

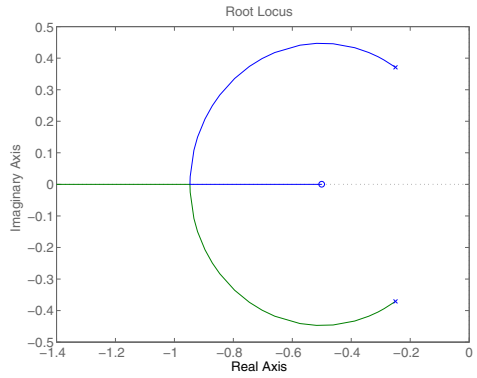

Fig. 5. The root locus plot.

III. Implementation Results

In this section, we report on a prototype implementation of the BBR framework described in Section II. The `bbr` tool is a multithreaded application that is written in C and uses the pthreads library. The version we describe here runs on Linux, however, similar implementations can run on other operating systems.

A. An Overview

`bbr` runs as a command line with different arguments as we explain below:

`-m` *malware*
loads and executes the specified malware.

`-a` *filename*
loads the specified file of actions.

`-t` *filename*
loads the specified file of triggers.

`-s` *utilization_threshold*
sets the stealthiness threshold. `bbr`, while executing the actions, will aim to keep its own utilization of each resource below the specified threshold. For simplicity, we use one threshold value for all resources, however, it is possible to have a different threshold for each resource.

`-d` *password*
disarms `bbr` with the correct password.

`-r`
executes `bbr` in a rehearsal mode. This option is

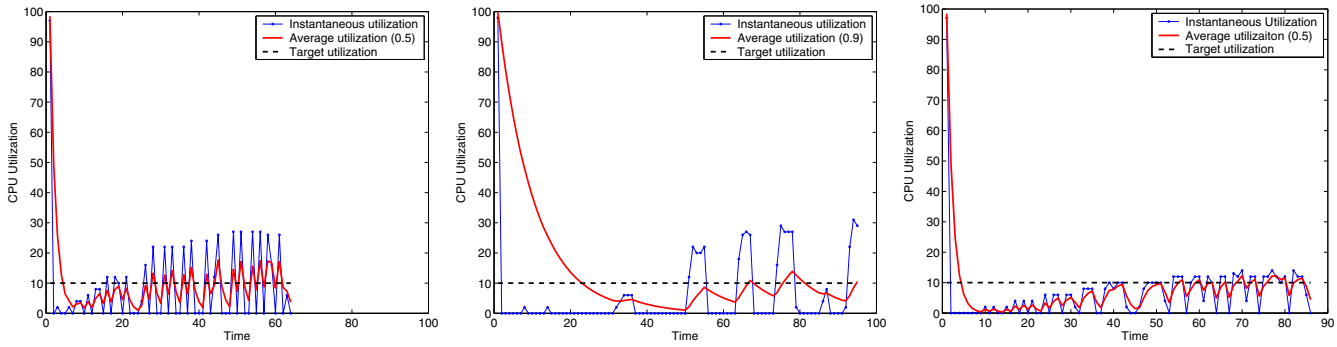

Fig. 6. Performance results for CPU-bound actions. Left plot used a β value of 0.5, with G_{on} and G_{off} chosen to be 500 μ sec, center plot used a β value of 0.9, with the same values for G_{on} and G_{off}, and the right plot used a β value of 0.5, with G_{on} and G_{off} chosen to be 100 μ sec, 1000 μ sec. The target utilization was set to 10%.

used to obtain timing information on how long it takes to execute the actions specified. This information could be used to drive bounds based on different utilization thresholds. When this option is specified, `bbr` creates a copy of the actions and executes those actions on the copy of the data, while keeping track of the time elapsed.

To run `bbr`, the following command is issued:

```
bbr -m malware -a actions.bbr -t
trigger.bbr -s 10
```

This will execute the program malware through `bbr`. The command will also load the actions.bbr file and the trigger.bbr file and will arm `bbr`. If one of the triggers fires, `bbr` will execute the actions while keeping its own utilization of each resource below 10%.

The preprocessor parses the actions file and group the actions based on their resource needs and their priorities. For each resource, `bbr` creates three threads. One thread for executing the actions, one thread for monitoring the utilization of the resource due to the actions being executed and a thread for adjusting the on/off periods for each action. Initially, all these three threads are blocked on a semaphore. In our implementation, we used `SIGSTOP` and `SIGCONT` to enforce the on and off periods computed by the actuators. We have experimented with two resources, CPU and Disk. The preprocessor also parses the trigger file and creates a thread for each trigger. The current implementation uses polling techniques on devices found at specific locations such as `/proc/bus/usb/devices` and `/dev/cdrom` to detect media insertions. Once a device is detected and the signatures match, the thread will release the semaphore after the delay period specified in the trigger file, if any.

To evaluate our implementation, we ran `bbr` on a Dell machine with an Intel Core 2 Duo processor stepping at 2.33 GHz with 4 GB of memory. The machine runs CentOS Linux distribution 2.6.18.

B. CPU Utilization

In this set of experiments, we study the behavior of `bbr` in controlling the CPU utilization for actions that are CPU intensive. We used `gpg` [18] to encrypt 100 MB of data. This action requires 18 seconds and `gpg` will use 100% of the CPU cycles on one of the processors on this machine architecture. We set the CPU utilization threshold to 10%. When setting the on and off periods, the choice of initial values is very important as it may compromise the stealthiness of `bbr`. We start with an on period of 0 μ sec and an off period of 1 sec.

Figure 6 (left and center plots) studies the impact of the parameter β on the CPU utilization used by `gpg`. We ran experiments for two values of β; 0.5 and 0.9. In this experiment, we fixed G_{on} and G_{off} to 500 μ sec. One can see that larger values of β degrades the performance significantly and leads to larger oscillations. Also, a very small β may cause the system to oscillate, if the original signal is bursty (which was not the case here). For example, the instantaneous value of the CPU utilization was bounded below 30%.

Figure 6 (right) shows the results of a similar experiment, with G_{on} and G_{off} chosen to be 100 μ sec, 1000 μ sec, respectively. β was chosen to be 0.5. This choice of constants leads to a smaller increase when the on period is computed and a larger increase when the off period is computed. One can see that the target is well matched at steady state (which starts around time 55).

C. Disk Utilization

In this set of experiments, we study the behavior of `bbr` in controlling the disk utilization for disk-bound actions. We used `shred` to wipe 100 MB of data. This action requires 39 seconds and `shred` will cap the data transfer rate for writing on disk. In order to determine the maximum load sustained by the disk, we ran a different number of shred processes concurrently, and observed the maximum data rate achieved. Figure 7 shows the results from running 1 shred process and 3 shred processes. We found that the maximum data rate is around 68 MB per second. We used this value, so we can measure the disk utilization.

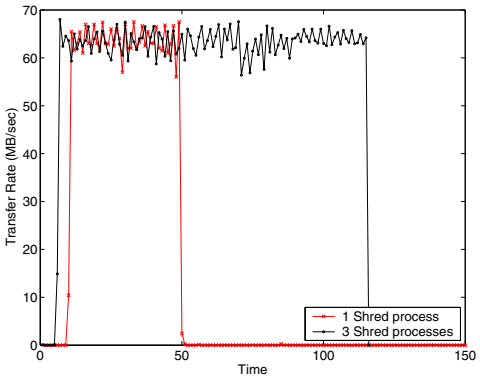

Fig. 7. Disk benchmark. Results obtained from running 1 shred process and 3 shred processes.

Figure 8 shows the results of an experiment with the disk utilization set to 20%. The experiment used a β value of 0.5, with G_{on} and G_{off} chosen to be 5 milliseconds each. In comparison to CPU, disk utilization tend to be more bursty in nature. The instantaneous utilization can go up to 40%.

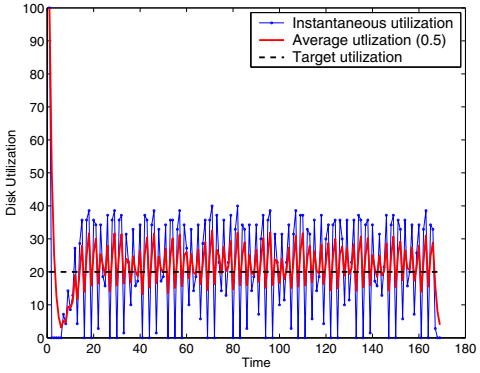

Fig. 8. Performance results for Disk-bound actions. β is chosen to be 0.5, with G_{on} and G_{off} chosen to be 5 milliseconds. The target utilization was set to 20%.

One of the interesting issues we came across is scheduling. From the outside, BBR is treated as a normal task, given CPU cycles by the OS kernel. Within BBR, many threads share those same cycles (since pthreads uses user-mode threads). The threads that implement the actuators are of a particular interest, since they are responsible for controlling the actions. If those threads are not responsive, controlling the actions would be very hard. Fortunately, one can give those threads a higher priority from within the pthreads library. We believe that this option can further improve the results obtained in this section.

D. Live Response over the Network

In the introduction we noted that first responders and investigators typically follow a well known set of guidelines with regard to a live response investigations. Some of those may occur over the network. In this subsection, we address how `bbr` detects an ongoing live response over the network in order to hide itself and manage its processes.

We chose to configure `bbr` to identify live responders by their network traffic footprint correlated with local process execution on the computer compromised by `bbr`. To detect live responders, we seek to identify interactive traffic that is largely dominated by small packets. We follow the algorithm in [16], where a threshold, τ, is defined to be:

$$\tau = \frac{S - G - 1}{N} \tag{15}$$

where S is the number of small packets (less than 72 bytes in our experiments), G is the number of gaps between small packets and N is the total number of packets. A gap is defined as two small packets separated by at least one large packet. We take τ to be 0.2 as recommended in the original paper. To correlate small packets with execution of programs, `bbr` monitors the file system for access attempts on the executables. These access attempts are presumed to be requests to execute the programs. While this method is not the only way to monitor for process execution, it gives `bbr` the possibility to monitor removable storage and FTP folders for executables transferred to the machine, thereby allowing it to respond accordingly. The specific triggers are configurable, making a static defense against `bbr` more difficult that is in addition to the possibility of configuring `bbr` to respond on multiple vectors. To simulate a live response, an ssh session is created in which some executables are run. This is done along with other sources of traffic of HTTP and bulk transfers to test `bbr`'s reaction to other forms of traffic.

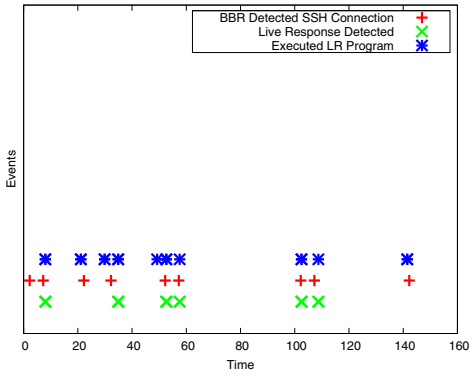

Fig. 9. Correlation of BBR Remote Live Response detection with Live Response activity. Events lined up vertically that occur at the same time.

Figure 9 plots the events detected by `bbr` over time. It demonstrates `bbr`'s ability to equate presumed remote shell access with first response processes execution (e.g. ps, netstat, top, who, etc).

Using Figure 9, we can see remote access and process execution events line up to produce `bbr`'s live response event-response, marked as an "X" on the plot. In some instances we see that a program is executed without being marked as a live response. This occurs for two reasons: 1) The executable file is

accessed by some other means than a remote shell (e.g. locally or through a backdoor) 2) The heuristic used to determine that a process execution event is a live response is forward looking only and does not account for events prior to detected remote shell access attempts.

Once bbr detects a live response is in progress, it can take appropriate action to mitigate the chances it will be found. The exact nature of the response is proportional to the presumed need for stealth. bbr may determine, if configured so, to completely eliminate itself, or it may hide itself. These ideas are more thoroughly examined in Section III-E.

Notice that it is unlikely that a live responder would trust the local tools installed on a computer, if it is suspected of being compromised. This challenges bbr in knowing which processes to monitor for triggered events. In Section IV, we discuss this issue further on how to make forensics tools more robust to defend against bbr.

E. A Case Study

We take the illustrative example, in Section I, and create a scenario where a malicious attacker has installed bbr on a server and wishes to utilize a backdoor without being detected. We do not cover the exact motivation and specifics of how bbr was executed on the machine in the first place, but this can be achieved through known exploits.

Our environment consists of a compromised Linux PC running a web server. It is capable of remote administration and file sharing on a Windows network. The Linux PC is connected to a network with one Windows domain controller and one Windows PC. It is necessary to note the specifics of the environment because spurious traffic from the other computers must be filtered by bbr in order for it to properly detect remote shell connections on uncommon ports (i.e. other than port 22).

Since bbr does not inherently have the ability to open backdoors on a computer, this functionality must be acquired by other means. bbr is designed to load and execute component malware, as described in Section III-A. In Section IV-A there is a discussion about a proposed enhancement to the prototype bbr that would allow it to essentially "wrap" the functionality of other malicious software found on the machine automatically, with a default configuration. In either case, the results would be that bbr would produce a stealthy backdoor for any potential intruder.

In this case study, we acquired a simple bindshell (3vilSh3ll) from the Packet Storm website [19]. The bbr prototype executes this malware on startup, then begins monitoring the environment for live response activity. The bindshell opens a backdoor remote shell access port, of our choosing, when started. bbr hides the process and port activity by starting and stopping the 3vilSh3ll when necessary. It is a fully functional program in its own right, meaning that there is no physical link between bbr and itself. bbr's only interaction with the process is to start and stop it on triggered events.

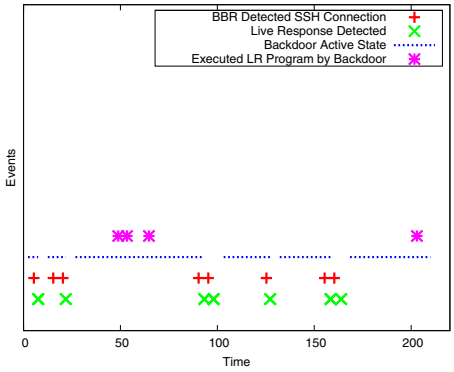

Fig. 10. bbr Live Response event trigger in action. Dashed line indicates when an active malicious backdoor is open. X's mark bbr Remote Live Response detection events. Events lined up vertically occur at the same time.

For this experiment, bbr was configured to close the backdoor and wait 5 seconds. bbr then checks for live response activity over the network. Figure 10 shows that, on every detected live response, bbr closes the backdoor when necessary and then reopens it when the live response threat had passed. While it is an inconvenience to the intruder to be automatically disconnected, they can be assured that the backdoor will be available again. Moreover, the plot shows that the intruder is allowed to execute programs associated with live responders without triggering a response from bbr.

In a more paranoid state, it can be imagined that bbr would eliminate the malicious program and start erasing evidence of its existence under the threat of a live response, or it may simply use a diversionary tactic. One of the major benefits of acquiring "third party" malware is that an investigator who is able to determine the existence of bbr's wrapped malware may be able to cleanse the system of that threat but still remain unaware of the existence of bbr due to it's ability to hide itself in various configurable ways. This layer of indirection is another aspect of bbr's stealthy behavior that would make it more difficult to uncover.

In summary, we showed that a backdoor could be hidden from prying investigators in this scenario. In a similar manner other aspects of bbr's footprint could be hidden. Sections III-B and III-C show experiments, for CPU and Disk utilization respectively, that target the resource usage of bbr's tasks. Those could be stopped on the event of a live response, as well, but resource usage is a flexible medium; bbr can expand and contract the resources it expends with a respectable degree of precision.

IV. Discussion

In this subsection, we discuss extensions, limitations and defense mechanisms against BBR.

A. Rolling in known Signatures

We envision BBR to be able to acquire the signatures of known malware, and automatically generate their corresponding actions file to destroy those signatures upon detecting a

live response. Those signatures are widely available online through many security databases [20], [21]. In this regard, BBR would act as a *wrapper* for any malware. The malware execution would be controlled by BBR to keep its resource usage below a specific target. BBR is distributed and installed on computer systems in a similar manner to malware, through vulnerability exploits, spam emails and packaged with other freeware. Alternatively, if BBR contains the code that does the exploit, then it does not need to acquire the source code of the malware, but would execute it as a child process.

B. Physical Disk Examination

So far, we have focused on BBR taking control after a trigger fires. This means that if a computer gets subjected to a physical examination of its hard disk while it is off-line, BBR's presence may be revealed, specially if the computer gets unplugged before BBR starts to execute. We believe, however, that BBR can be extended to take actions (e.g., encrypt/wipe files) on some events as shutdown/startup/hibernation. These measures depend on the nature of the incident and the risk the attacker is willing to take.

C. Utilizing Rootkits/Renaming the tools

To increase the stealthiness of BBR during the execution of actions, BBR can utilize a number of techniques to avoid detection (which are studied in Section V in more details). For example, BBR can hide itself from the process list through kernel modules on most common operating systems [22], [23]. Moreover, while executing actions, BBR can hide these actions or execute them under different names that are more familiar to users. This technique has been used a lot by malware to rename themselves to common names (e.g., svchost.exe and run32.dll on Windows systems).

D. Defense mechanisms against BBR

Our goal in this section is to provide guidelines on how to secure forensics tools against BBR and BBR-like frameworks. It is important to note here that we are not trying to detect the presence of BBR. The arms race between malware and detection mechanisms will likely continue for a long period of time, and BBR – like other malware - can always utilize the most up-to-date techniques to hide itself, before the cycle repeats. Thus, we seek to develop new approaches that prevent malware from *detecting an ongoing forensics investigations*.

One approach is to prevent disclosures on how forensics tools operate, so malware designers may not know what constitutes a trigger in a forensics investigation. This is probably hard to achieve since many of those tools are open source and their exact functionalities should be proven to be evidence-preserving. Although in some cases, proprietary tools do exist.

Another approach is for forensics tools to operate in a stealthy manner as well. For example, a search for all images on a computer may be giving away the existence of a forensics investigation. Rather, a forensics tool may group different operations – that would be executed anyways – into the same time frame in order to avoid being picked up by the malware. Also forensics tools can execute under different names, perhaps randomized names as well, to prevent their detection. This may prevent BBR from looking for specific executables running from specific locations.

When it comes to a live response over the network, the forensics investigation should aim to avoid the traditional interactive nature of the connection. This can be achieved for example, by sending large dummy packets in between smaller ones, to prevent the detection of interactive traffic [16] or adding extra delay between small packets.

V. Related Work

In this section, we put our work in comparison to other related works. On one dimension, we discuss different mechanisms used by malware to remain stealthy and on another dimension, we discuss anti-forensics techniques.

A. Stealthy Malware and Detection Mechanisms

Malware relies on a number of mechanisms in order to prevent them from being detected. As a first step, many of them hide in privileged directories that ordinary users do not access regularly. Some malware use randomized filenames to avoid being recognized [24], [25]. Some attach themselves to known process, like the famous Rundll32.exe process on Windows Systems that is called to load dll files. Others, replace/add their own dll files to the system to intercept a subset of the system calls. Based on where they hook themselves into the system, in [26], the author presents a taxonomy for stealthy malware ranging from type 0 to type III. Simple kernel modifications can result in preventing the malware from being reported among the process list [22], [23]. Once a piece of malware is in place, it typically tries to cripple the security systems present. This ranges from antivirus programs to auditing and monitoring tools. For example, the Agobot variant can subvert more than a 100 anti-virus processes [27].

The use of encryption has also added to the stealthiness of malware. There are many tools that provide such functionality [28]–[30], among others. It is very common for malware to encrypt its own data files to avoid them from being detected.[3]

Notice that the level of stealthiness in the above techniques is considered static in nature; it is not adjusted based on potential ongoing investigations. To the best of our knowledge, this work is the first to consider dynamically reacting to live forensics investigations where malware can destroy themselves, their signatures, and avoid being traced.

B. Anti-Forensics tools

In the anti-forensics field, tools are often created that aim to complicate the task of the investigator. They range from simple tools that hide/encrypt information, to more sophisticated ones that target specific forensics packages [31]–[35]. For example, the slacker tool allow files to hide in the slack space. The Timestomp tools can modify the date and time stamp for

[3]Stealthy malware have been discovered that encrypt the users' files and ask them for ransoms in order to recover their content [4].

files, effectively attacking the NTFS. The SAM Juicer dumps the hashes from the SAM without writing them on disk. These tools can be obtained from the Metasploit anti-forensics projects [36]. Forensics packages like EnCase and SluethKit are also susceptible to some attacks by anti-forensics tools. For example, the Transmogrify tool targets the file footprint file functionality in the EnCase package, among some others.

The above anti-forensics tools are focused on targeting specific forensics tools or particular technologies and rendering them useless in an off-line setting. This is different from BBR, which is mainly focused on an on-line approach to manipulating evidence during a live response.

VI. Conclusion and Ongoing Work

Many of the tools and techniques used during a live forensics examination create footprints on the system being investigated. Such footprints can be detected by malware causing the destruction of evidence, the malware itself, its signatures, along with the execution of other actions. This paper presents BBR, a framework that enables malware to detect and adapt to live forensics examinations through a set of registered triggers and a set of malware-defined actions. At its core, BBR utilizes control-theoretic actuators to dynamically adjust the timing information of the executing actions to keep BBR running in a stealthy mode where its own utilization of a resource is controlled at a target level. We have presented an implementation prototype of BBR and studied the impact of different parameters on its performance, stability and stealthiness. This work highlights the importance of improving the state-of-the-art forensics tools and to make them aware of potential presence of `bbr` and `bbr`-like tools during live forensics examinations.

As for our ongoing work, we are currently writing a tool that would automatically obtain known malware signatures and generate their corresponding actions files. We are also developing a defense mechanism that can detect the behavior of bbr and bbr-like tools to secure live forensics investigations.

Acknowledgments

The authors would like to thank Wilbon Davis for his fruitful discussions regarding this work. The authors would also like to thank the anonymous reviewers from the Anti-Phishing Working Group (APWG) for their feedback.

References

[1] Symantec, "Symantec Internet Security Threat Report: Trends for July-December 2007 (Executive Summary," `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf`, 2008.

[2] P. Gutmann, "The Commercial Malware Industry," `http://www.cs.auckland.ac.nz/~pgut001/pubs/malware_biz.pdf`.

[3] GPCode, "F-Secure Virus Descriptions: GPCode," `http://www.f-secure.com/v-descs/gpcode.shtml`.

[4] JH. Miller, "A New Wave Of Malware: Ransom-ware," `http://www.securitypronews.com/news/securitynews/spn-45-20050524ANewWaveofMalwareRansomware.html`.

[5] CSI, "CSI Computer Crime & Security Survey," 2008.

[6] NTIllusion, "Hacker Tools: NtIllusion V1.0," `http://www.hacker-soft.net/Soft/Soft_11734.htm`.

[7] Hacker Defender, "Hacker Defender rootkit for Windows," `http://www.rootkit.com/board_project_fused.php?did=proj5`.

[8] Hacker Defender, "API Hook SDK 2.13," `http://www.brothersoft.com/api-hook-sdk-download-25831.html`.

[9] National Institute of Justice, "Electronic Crime Scene Investigation: A Guide for First Responders," *Second Edition*, April 2008.

[10] J. Branson R. Nolan, C. OSullivan and C. Waits, "First Responders Guide to Computer Forensics," *CERT Training and Education*, March 2005.

[11] e-fense, "Reveal the Truth: Volatile Data Collection from a USB Key," `http://www.e-fense.com/live-response.php`.

[12] C. Waits, J. Akinyele, R. Nolan, and L. Rogers, "Computer Forensics: Results of Live Response Inquiry vs. Memory Image Analysis," *CERT publications*, 2008.

[13] e-fense, "Helix3: Incident Response Electronic Recovery Computer Forensics Live CD," `http://www.e-fense.com/products.php`.

[14] EnCase Enterprise, " Incident Response, Internal Investigations and eDiscovery," `http://www.guidancesoftware.com/products/ee_index.aspx`.

[15] Y. Zhang and V. Paxon, "Detecting Stepping Stones," in *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[16] Y. Zhang and V. Paxon, "Detecting Backdoors," in *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[17] K. Ogata, "Modern Control Engineering, Fourth Edition," Prentice Hall, 2002.

[18] GnuPG, "The GNU Privacy Guard," `http://www.gnupg.org/`.

[19] Packet Storm, "Security Professionals, Linux Security, Administration, Exploits, Tools," `http://packetstorm.linuxsecurity.com/UNIX/penetration/rootkits/3vilSh3ll.c`.

[20] Symantec, "AntiVirus, Anti-Spyware, Endpoint Security, Backup, Storage Solutions," `http://www.symantec.com/`.

[21] McAfee, "Antivirus Software and Intrusion Prevention Solutions," `http://www.mcafee.com/`.

[22] J. Butler, J. Undercoffer, and J. Pinkston, "Hidden Processes: The Implication for Intrusion Detection," in *Proceedings of the IEEE Systems, Man and Cybernetics Society Information Assurance Workshop*, West Point, NY, June 2003.

[23] Greg Hoglund and James Butler, "Rootkits - Subverting The Windows Kernel.," *Addison-Wesley*, 2005.

[24] Symantec, "Adware.Look2Me," `http://www.symantec.com/security_response/writeup.jsp?docid=2004-042016-3810-99`.

[25] Symantec, "Adware.VirtuMonde," `http://www.symantec.com/security_response/writeup.jsp?docid=2003-120914-4108-99`.

[26] J. Rutkowska, "Introducing Stealth Malware Taxonomy," *COSEINC Advanced Malware Labs*, 2006.

[27] F-SECURE, "F-Secure Virus Descriptions : Agobot," `http://www.virus.fi/v-descs/agobot.shtml`.

[28] Microsoft, "BitLocker Drive Encryption," `http://www.microsoft.com/windows/windows-vista/features/bitlocker.aspx`.

[29] TrueCrypt, "Free open-source disk encryption software for Windows Vista/XP, Mac OS X, and Linux ," `http://www.truecrypt.org/`.

[30] PGP, "PGP Whole Disk Encryption - Comprehensive Hard Drive Encryption for PCs, Laptops and Removable Media," PGP Whole Disk Encryption.

[31] C. Peron and M. Legary, "Digital Anti-Forensics: Emerging Trends in Data Transformation Techniques," in *Proceedings of E-Crime and Computer Evidence Conference*, 2005.

[32] S. Hilley, "Anti-forensics with a small army of exploits," *Digital Investigation*, vol. 4, no. 1, pp. 13–15, 2007.

[33] R. Harris, "Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem," *Digital Investigation*, vol. 3, pp. 44–49, 2006.

[34] J. Foster and V. Liu, "catch me, if you can," *blackhat Briefings*, 2005.

[35] S. Garfinkel, "Anti-Forensics: Techniques, Detection and Countermeasures," in *Proceedings of 2nd International Conference on Information Warfare and Security*, 2007, p. 77.

[36] Metasploit, "Metasploit Anti-Forensics Project," `http://www.metasploit.com/research/projects/antiforensics/`.