

A dynamic filtering technique for Sebek system monitoring

Edward Balas, Gregory Travis and Camilo Viecco
Advanced Network Management Lab
Indiana University

Abstract— In this paper we investigate the performance limits of system call based monitoring tools using the Linux version of Sebek as a focal point. We quantify the amount of uninteresting data that it collects and illustrate the problems that this creates: detection of Sebek, amount of work to analyze data, and data privacy. To mitigate these problems we propose a dynamic filtering technique. Finally we evaluate the performance of an implementation of this technique.

I. INTRODUCTION

In an effort to better understand the threats facing networked systems, researchers often use high interaction honeypots[1] as a tool to observe the behavior of remote intruders. In not so distant history, packet traces were the sole method of observation for such honeypots. But as cryptographic tools such as OpenSSH[2] and OpenSSL[3] began to popularize, these were used by intruders to avoid eavesdropping by researchers. As a result, network traffic on its own could not longer be used to completely understand the behavior of intruders within a honeypot. Researchers then began to create tools to instrument the honeypot's operating systems to extract the desired data. Sebek[4] is an example of such a tool. The Linux version of Sebek operates by hooking system calls[4] and then recording associated activity, although it could just as easily use other techniques such as modifying the Interrupt Descriptor Table. Sebek has evolved from a tool specifically designed to record keystrokes and encrypted file transfers into a more general purpose system monitoring tool designed to record process heritage, socket activity, file opening, and read activity. The combination of these recorded activities enables the identification of causal relationships in the system activity and the recreation of the intrusion sequences[5].

To be effective, honeypots must observe intruders without their awareness. This leads to the desire to build tools similar to Sebek which are difficult to detect. Indeed, the original version of Sebek was inspired by the Adore rootkit. For system administrators, malicious software such as rootkits need to be detected. Thus we observe that the success of one security tool is dependent on the weakness of the other. As rootkit detection techniques improve it will be easier for intruder's to use those techniques to detect system monitoring tools like Sebek.

Two methodologies to detect Sebek or any other hidden monitoring tool in a system have been proposed: (i) the use of memory analysis to detect kernel memory locations with suspicious contents and (ii) the use of performance analysis to detect the existence of *abnormal* performance bottlenecks under some operations.

Memory based detection of Sebek is possible due to the fact that Sebek modifies kernel memory. Previous work on memory based Sebek detection includes the works of and Tan Chew Keong[6] and of Dornsief, Holz and Klien[7]. In particular, current signature based detection approaches look to see if the system call table has been modified. Generally, any modification to kernel memory can be detected with enough effort, thus while it might be more difficult to detect a kernel patch we feel that it is still possible.

Such detection techniques assume that they are able to retrieve a accurate picture of kernel memory. Recently, research by Sparks and Butler[8] demonstrated a rootkit hiding technique where by they break the assumption that processes can get an accurate picture of kernel memory by reading it. This technique attempts to desynchronize the Instruction and Data Translation Look-aside Buffers such that, if a process executes memory, one physical page will be presented and yet if the same process reads a block of memory a different physical page will result.

Work by Dornsief, Holz and Klien [7] includes not only memory analysis techniques but also some initial musings about performance based detection techniques. Performance based detection is possible due to the fact that system monitoring necessarily generates extra work to be performed by the system. We believe that performance based monitoring detection is the more pressing of the two threats because unprivileged users can use it to detect system monitoring prior to escalating privileges.

In this paper we will examine the shortcomings in Sebek performance, including the delays incurred on monitored systems, as well as the amount of uninteresting data generated. Next we will propose a technique for addressing some of the shortcomings and finally we will evaluate the suitability of the technique.

II. CURRENT STATE OF SEBEK

While Sebek was initially designed to collect minimal amounts of data, the need to have a better understanding of the behavior of intruders has led to collection of additional data types. The idea of “collecting only data of interest” as specified by the honeynet community, has been lost in the process; and while anecdotal evidence is present about this problem, no characterization of it or of its consequences have been done to the date.

A. *The problem of too much data*

Collecting data with the current version of Sebek is similar to trying to drink from a firehose. Much of what comes out may be of little use and can compromise the whole effort. And while the current version of Sebek has rudimentary mechanisms to control what it collects, these are not sufficient to reduce the data to negligible values even on idle systems.

Within the latest version of Sebek for Linux, there are two binary switches to control what data is acquired. The first determines if all read activity is recorded, or if only keystrokes (1 byte reads) are recorded. The second switch activates the recording of all socket activity. This discrimination is insufficient. In particular we identify three resultant problems:

First, there may be a set of system resources for which we know the associated activity will never be interesting. For example, consider the `/dev/zero` device in Linux. Dorsief, Holz and Klien[7] illustrate a potential profile detection technique based on reading large volumes of data from `/dev/null` and then checking Local Area Network for congestion. While we can not deny the fact that the act of recording system activity consumes resources in the form of CPU clock cycles; it seems logical that system monitors provide a way of not recording activity which is known a priori to be uninteresting.

Second, any method used to move data off the honeypot has a specific rate limit such as the rate of packets per second that can be transmitted and the bandwidth of the network we are using. When uninteresting data is collected, it consumes part of this finite resource. Any high volume of uninteresting data not only makes it easier to detect the presence of Sebek, as has been shown, it also increases the likelihood that we will be unable to collect the data of interest. Within Sebek, records are queued for network transmission in a simple FIFO queue. Thus, Sebek is most likely vulnerable to situations where one process can generate sufficient volume of data such that we lose a significant portion of another process’s log data. Furthermore, assuming that the network has the capacity to carry all of the Sebek data, we must store the data and analyze it. The collection of uninteresting data decreases the accuracy of collected data, the efficiency of the analysis system, and increases the data collection and analysis costs.

One common justification for using honeynets was that “all captured activity is assumed to be unauthorized or malicious”[9]. Clearly the addition of Sebek to honeynets significantly weakens this rationale by unconditionally collecting uninteresting data that is a side effect of routine system activity.

Third, there are times where Sebek would be very useful to use in incident response settings. However, because it has no control over what it monitors, it may not always be appropriate to use in this case. For instance if we have Alice and Bob on the same host, and Alice is under investigation for some misdeed, it is desirable to only monitor Alice and not Bob. Further, depending on the context it might also be inappropriate or illegal to inadvertently monitor Bob.

B. *Sebeked Linux Kernel Performance*

In order to appropriately discuss the performance implications of Sebek we set up several tests to understand both the performance implications of Sebek and the amount of data collected by a system running Sebek. For the kernel performance measures we proceeded with two types of tests: Macro benchmarks to study the aggregate latency effects of Sebek using normal user tools, and micro benchmarks to study the individual system call latency effects.

B.1 The benchmark environment

All benchmarks were run on an 900Mhz Celeron processor in a Dell Dimension 2100 system -running Linux 2.6.9. The system was booted and running in multi-user mode with standard background daemons running and the X11 system enabled. This was done to best approximate “real world” conditions.

B.2 Macro Tests

Because background processes and other non-deterministic activity were taking place, several runs of each benchmark were performed. The benchmarks, each of which consisted of millions of individual system calls, were performed six times each. The user, system, and real times of each run were averaged to get the final result. In addition, standard deviations for all of the runs were calculated as a control measure.

In reporting the results, only the sums of the user and system CPU times are presented. Real time was found to be too variable and of limited informational use. Since we are primarily interested in the amount of additional CPU processing Sebek incurs, summed user and system CPU times were considered an adequate measurement benchmark. In practice, the benchmark consistently produced results congruent with the expected impacts of the methods described in this paper.

The benchmarks consisted of the following components: a benchmark to time the overhead of creating files, a benchmark to time the overhead of performing the system fork()

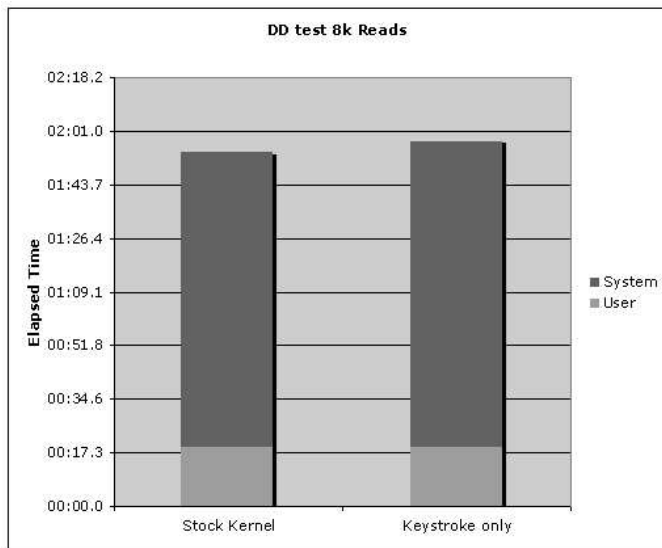


Fig. 1. DD test 8096 byte blocks.

call, a benchmark to time the overhead of performing the system open() call, and then several benchmarks timing the performance of the system read() call, under different conditions.

The file make benchmark was used as a control. Because Sebek does not impact the path for creating files, the make benchmark should be unchanged as different system configurations are used and analyzed. This was confirmed empirically.

The open, fork, and read tests all showed the expected increase in delay due to Sebek. Note however that these were all, to some degree, “worst case” tests and are not representative of actual application performance on either stock Sebek-equipped systems or systems equipped with Sebek modified as described in this paper.

Two charts of these benchmarks are included here: Figure 1 and Figure 2. Figure 1 shows the average expected running times of doing dd for 8096 byte long reads using the keystroke Sebek version. Interestingly, due to the use of keystroke only filtering, there is no appreciable difference between the stock system and the Sebeked version.

Figure 2 illustrates behavior measured when using the 1 byte DD test. It can be seen the large difference in latency, due in part to the process sleeping until the network card is able to accept additional packets for transmission. Notice that the delay is counted as system not user time.

B.3 Micro Tests

To build upon the Macro test we next focused on the sys_read call and examined the elapsed time of individual calls. We were interested not only in the average time for each call, but more importantly the distribution of times for the call. To identify the approximate distribution, we

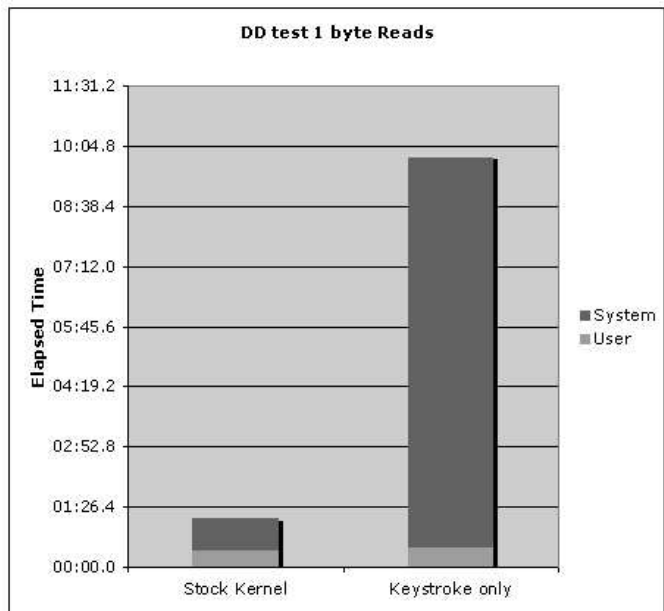


Fig. 2. DD test 1 byte blocks.

created a test application which used the RDTSC instruction [10] to measure the elapsed time as the number of clock cycles required to perform the call. While the number of clock cycles does remove CPU clock speed from the equation, the number of cycles to perform a task varies based on CPU architecture. It is the variability that we feel we can minimize by looking at the distribution rather than absolute value of delay.

We performed two tests. In the first we measured 1,000,000 1 byte reads on our stock test system. In the second test we installed Sebek, configured it to only monitor keystrokes ie reads of length 1 byte, and then measured 1,000,000 1 byte read calls.

Figure 3 illustrates the differences between a stock system and a system running Sebek. This is graph of observed probability density in our two tests. The X axis represents the number of clock cycles to perform a read, in Logarithmic scale, the Y axis represents the probability density for each bucket in our histogram. Because the distribution is highly skewed we used logarithmically sized histogram buckets.

Visually, it appears as if Sebek not only changes the lower bound of the distribution but also the shape of the entire distribution. We suspect that this is the result of having these calls become dependent on a finite bandwidth data path in the system. However we have currently have insufficient data to confirm this notion.

C. Sebek Data volume

Besides the time aspect of data collection, the data volume aspect is also important as data needs to be collected

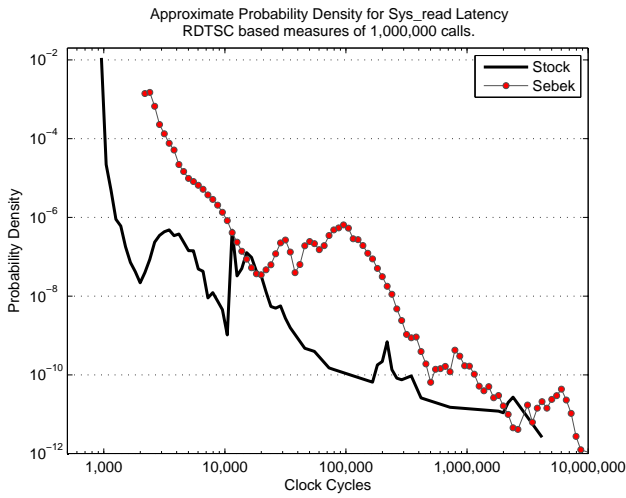


Fig. 3. Comparison of sys_read latencies for stock Linux system and one with Sebek installed.

and interpreted in order to be of use. A different environment was used to generate the data volume tests for Sebek. The environment consisted of a VMware image of a Fedora Core 4 system, running X (gnome) with the latest Sebek version for the 2.6 Linux kernel. The system was connected to a 10 Mbps switch and data was collected by a honeywall in a Pentium III system at 1GHz with 512 MB of RAM.

Two types of tests were done: data volume with an idle system for full logging Sebek vs. keystroke filtered Sebek, and data volume of keystroke filtered Sebek under idle or production conditions.

C.1 Idle system

To observe the difference in data volume, the same system was compared first using Sebek with keystroke filtering enabled and later using Sebek with no filtering. Figure 4 shows the number of Sebek packets collected per hour from the idle system, classified by system call. This figure illustrates filtered behavior from hour 0 to hour 41 and full read recording in hours 42 to 62. When in keylogging only mode, an average of 5,212 packets per hour were being generated and collected by the honeywall. When in full read mode an average of 63,212 packets per hour were being collected by the honeywall. These are packet counts, we expect the throughput statistics to show an even larger difference. Detailed statistics are presented in table I.

The collected statistics show an order of magnitude increase in the volume of data collected, depending on the keystroke filtering parameter. Also, it is important to notice that the volume of non read data is practically constant in the two scenarios, as expected.

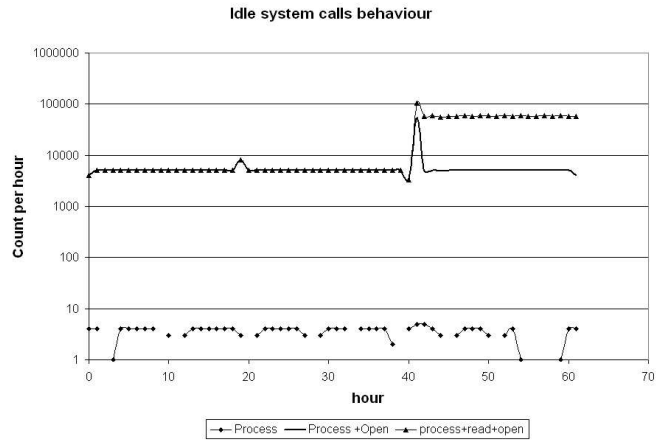


Fig. 4. Per hour count of system call activity.

TABLE I
COMPARISON OF SEBEK CONFIGURATIONS

	Keystroke only			All read		
	Open	Read	Fork	Open	Read	Fork
Average	5209	0.0731	3.00	5016	58194	2.35
Stdev	466.6	0.4685	1.55	248.6	562.1	1.84
Median	5145	0	4	5072	58399	3

C.2 Production vs Idle

While the previous tests only show the amount of uninteresting data collected by running an idle system, we decided to do some testing to estimate what would be the volume of Sebek collected data when using the system in a production environment. Thus a copy of Sebek was installed on one of the author's desktops and we measured the behavior of the system both in idle mode and in production mode. The tested system is a Redhat Enterprise Linux 4 running X (KDE). Table II-C.2 shows the difference in behavior using 10 minute buckets. We can observe how the amount of read calls is 5 times larger when using the system under a "normal" workload, and we also see the open system calls being 25 times larger than idle. The expected number of system and read calls per hour in such environment is of 750,000 packets per hour or 208 packets per second. It is undesirable to collect this volume of data in a production environment.

III. A FILTERING TECHNIQUE TO CONTROL SEBEK

Prior work has demonstrated that OS monitoring tools can collect enough data to track an intrusion sequence[5] by collecting not just read activity, but socket, process creation, and file accesses. In effect, a directed graph of processes is created to which socket events and file activity

TABLE II
SEBEK VOLUME UNDER NORMAL WORKLOAD 10 MINUTE BLOCKS

	open	read
idle keystrokes	849	0
idle all	855	15507
working all	20306	104225

are associated[5]. Using this causal graph one can identify processes related to an incoming flow or perhaps a set of files related to a process tree[11], [12]. We wished to follow a similar approach but to do it at point of capture so that we can avoid generating uninteresting data.

To achieve this goal the system would have to satisfy the following properties: (i) be user configurable, as the needs for each individual installation are probably very different; (ii) be able to express different type of actions, from full recording of system data, to keystroke only logging, to ignore activity completely; (iii) need to follow causal relationships within the system, as this is the key for causal analysis and what will allow the system to effectively follow the actions of an intruder; (iv) have a minimal impact on runtime; (v) to have a sufficiently expressive language which supports desired matching criteria. While this specification is by no means complete, it should be a sufficient starting point to address the Sebek fire-hose problem.

IV. IMPLEMENTATION

The implementation of the previous idea required the design of a sufficiently expressive filtering language and of modification to the Sebek code so that a representation of rules in this language could be used by Sebek at runtime to determine the action for every possibly recorded system call. To simplify our implementation, we constrained our system to accept rules only at module insertion time.

A. Filtering Language

The language used to describe Sebek filtering is inspired by firewall rules. The rules are order dependent and as soon as a system call satisfies the conditions on a rule the action is performed and the remainder of the rule set is ignored.

Each rule has 3 sections, the first determines the action, the second specifies the match logic and the last section provides optional flags. Let's look at an example:

```
action=keystrokes sock=(proto=tcp local_port=22)
opt=(follow_child_proc)
```

In this example rule the action is to "keystroke" monitor matching processes. The match in this cases states that this rule is to fire for any TCP connection where the local port is 22(SSHD). Lastly this rule has one option set, "follow_child_proc";this option tells Sebek to not only keystroke log processes that are associated with the SSH sessions, but to also keystroke log any process created by

the matching process. So if someone logs in remotely and then spawns a shell, we will also record the keystrokes of the shell. This recursive tracking will follow all descendants of the original matched process.

The action section can tell Sebek to ignore a process, keystroke monitor only, or record all sys.read activity. The match portion of the rule expresses the attributes Sebek should examine to determine if it should capture data. There are 3 types of filter matches: User ID, Socket/network, and file system. The User ID match can be used in conjunction with a socket or file match. When done both the User ID and other match must succeed for the match to succeed. Currently only one option is available: follow_child_proc. This option instructs Sebek to monitor not only the process that match the criteria, but also all the descendants of such process. A list of the file options by type is listed in Table III

TABLE III
BREAKDOWN OF AVAILABLE MATCH CRITERIA.

User ID	File Spec	Socket Spec
system uid	Filename Include Subdir Flag Strict Flag	Protocol Local Port Remote Port Local prefix Remote Prefix Client Con. Flag Server Con. Flag Strict Flag

One interesting application of the file based filter is for the use of honeytokens[13]. By using the following filter:

```
action=keystrokes user=bob file(name=/home/secretsquirrel)
opt=(follow_child_proc)
```

The file /home/secretsquirrel/topsecret acts as a honey-token for any user bob. If user bob then opens the topsecretfile, the opening process and all of its children will then be keystroke monitored.

Here are a few more examples:

1. Capture all data read from /dev/random, but ignore all other reads of files in the file systems:

```
action=full file=(name=/dev/random strict )
action=ignore file=(name=/ strict inc_subdirs)
```
2. Keystroke monitor user bar, but if he uses https then get a full capture of that data:

```
action=keystrokes user=bar opt=(follow_child_proc)
action=full user=bar sock=(proto=tcp rem_port=443 client)
```

B. Insertion time and Run time

At insertion time the modified Sebek prototype (code-name Okam) extracts the filter configuration file and generates an internal representation of the filter rules. This rule representation is loaded into kernel memory where it

is later accessed by the runtime modified system calls. By design we have made a trade-off of efficiency versus accuracy in the internal rule representation. While the file system operands expressed in a rule express a location in the file system space, the internal representation of the rule expresses this location as a tuple of device and inode. This allows a much faster rule evaluation at runtime as the system only need to compare two integers instead of a string. However, it also means that the system can only monitor locations that exist at the time of the module insertion and that remapping of the inode device space into file system name space can result in loss of data.

After Sebek has loaded the rules, taken command of the system space, and hidden itself, it starts its regular operation. As sockets calls are created, files are opened, or processes created, Sebek adds markers to key kernel data structures. There are two kernel data structures which we mark: the task structure and the inode structure. Within Linux both files and sockets are treated similarly and both have an associated inode record. For instance, when a process is created Sebek checks the parent's flags. If the parent's filter flags contain the follow_child_proc flag then the child will inherit the parents flags. Within each monitored system call, Sebek then checks the task and inode flags to determine if it should report data about the current system call. This type of flagging is quite efficient. The structures are already in memory and have to be accessed to perform the system call. The determination of whether or not a specific action is recorded only requires at worst the examination of two 32 bit integers per rule.

V. PERFORMANCE AND VOLUME COMPARISONS

A. Micro test revisited

To test the effectiveness of Okam, we performed an additional micro test. In this test we configured Okam to only monitor system activity related to inbound connections or the descendants of the servicing processes. In this mode, if a user walked up and logged into the terminal, Okam would not record any activity. However, if the user were to SSH into the system then Okam would record all keystroke activity. Using this configuration we repeated the measurement of 1,000,000 1 byte reads and compared the results to our prior tests.

Figure 5 illustrates the differences between a stock system, a system running Sebek, and our Okam system.

In visual comparison, Okam appears to follow the same distribution as a stock system, but with slight constant shift to the right.

B. Data volume

Figure 6 shows the history graph for the number of system calls using two instances of Sebek. One includes the keystroke only logging and the second the filtered Sebek version. The figure illustrates the order of magnitude dif-

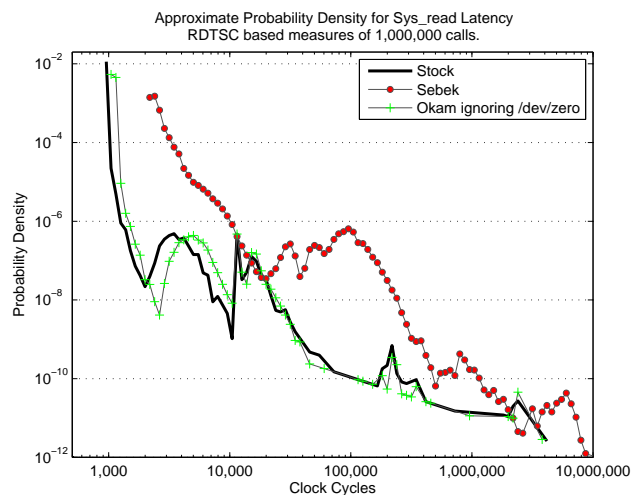


Fig. 5. Comparison of sys_read latencies for stock Linux, Sebek, and Okam installed.

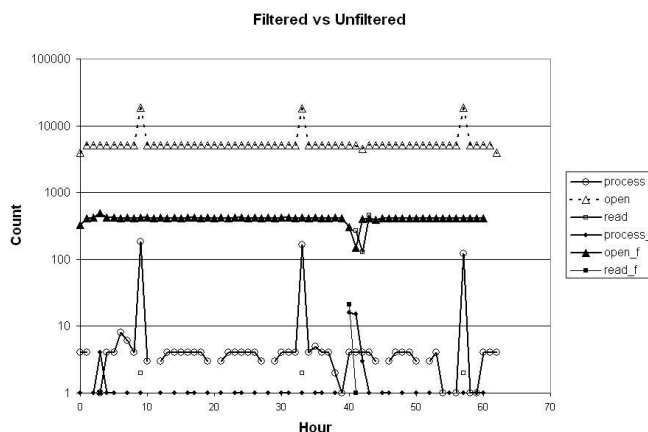


Fig. 6. Filtered and Unfiltered Sebek output volume (packets).

ference of data collected using both systems. While the average number of open system calls is not reduced to zero, it is reduced by an order of magnitude. The three peaks on the number of open calls and processes found on the unfiltered version are due to the initialization of cron jobs every 24 hours.

A detailed comparison of the statistics for one hour intervals can be seen in Table IV.

C. Macro Tests revisited

As hoped, Okam (filtered Sebek) is very similar to the stock version of Sebek for the dd test for both 1 byte reads and 8k reads, as seen in Figure 7 and Figure 8. The sys_open call mimics these performance characteristics as well for Okam and the stock kernel.

TABLE IV
COMPARISON FILTERED AND NON-FILTERED SEBEK

	Keystroke only			Filtered		
	Open	Read	Fork	Open	Read	Fork
Average	5209	0.0731	3.00	396.3	0.3650	1.11
Stdev	466.6	0.4685	1.55	82.44	2.647	2.72
Median	5145	0	4	415	0	1

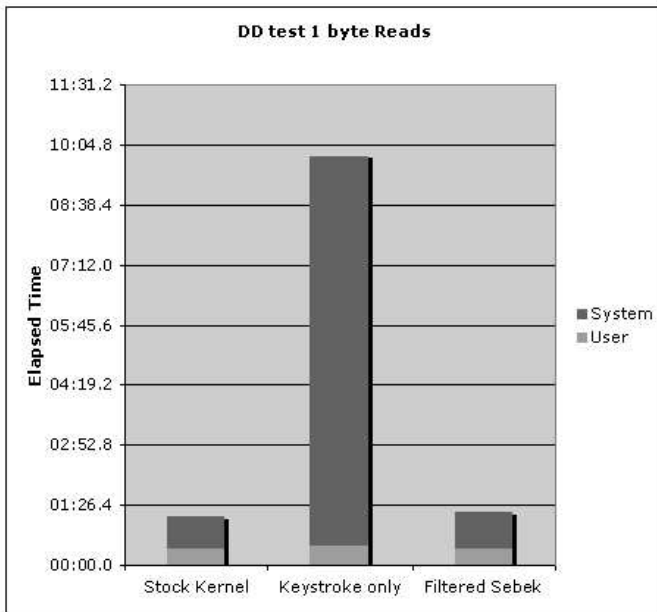


Fig. 7. Filtered and Unfiltered Sebek output volume (packets).

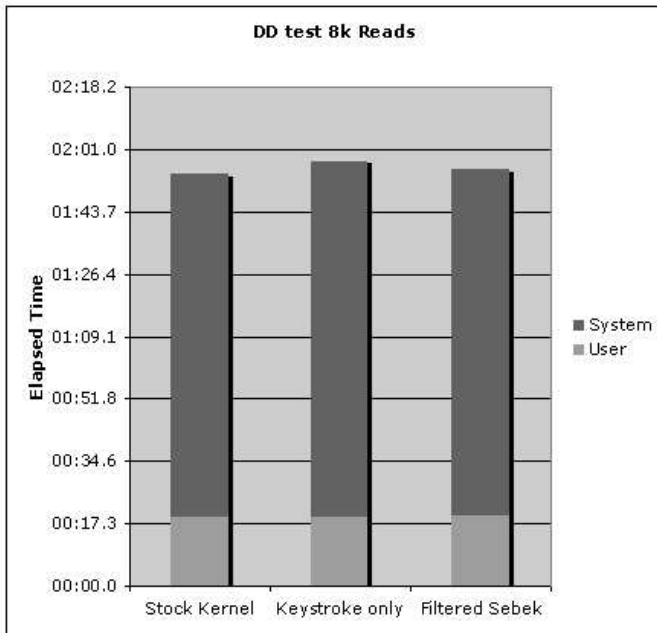


Fig. 8. Filtered and Unfiltered Sebek output volume (packets).

VI. CONCLUSIONS AND FUTURE WORK

We have shown how the presence of Sebek introduces an order of magnitude increase in the number of clock cycles and system time it takes to perform monitored system calls. We have also shown that a non-trivial amount of data is generated by Sebek even when a honeypot is idle and an order of magnitude increase beyond that when the system is in use. These increases in delay and data volume heighten the detection risk and the increase cost of analyzing honeynet data.

To mitigate this problem we proposed and implement a dynamic filtering technique whose objectives are to: (i) reduce Sebek's performance impact and (ii) reduce the amount of uninteresting data collected in an effort to understand the behavior of a system. We demonstrate that our technique clearly addresses these issues assuming the user knows what won't be of interest before installing the filtered version of Sebek.

This technique, while effective, has three limits: First, while the rules can dynamically follow execution in a process tree, the initial rule set is static and defined at install time. Second, any filter is a trade off between completeness of monitoring and system efficiency. If a rule set is overly strict, users may miss critical observations. Third, in the current implementation, file system filters are resolved into their associated inode values at Sebek install time. Thus one weakness is that if a file is deleted and recreated, the files inode value most likely will change and Sebek will no longer monitor the file.

There are three areas of particular interest for future research: Detection, Queuing, and integration into standard systems. For Detection, we feel that with additional investigation a rigorous method to reliably detect Sebek based on latency profiling may be possible. However we need to better understand the impact that different CPU and system designs have on the distribution. Orthogonal to the performance based detection we feel that it may be worth exploring the Spark and Butler technique for rootkit hiding as a technique to further cloak the presence of Sebek.

For Queuing we see a weakness in Sebek that results from its use of shared FIFO queues in packet transmission. We think that a token bucket type rate limiting technique might be useful for reducing risk of local data capture de-

nial of service. Third, we feel that Sebek style system monitors might be useful to include in standard system kernels. This would facilitate incident response based deployments and would further mitigate the memory signature based detection risk.

REFERENCES

- [1] T. H. Project, *Know Your Enemy*. Addison-Wesley, 2nd ed., 2004.
- [2] O. project, "Openssh website." <http://www.openssh.com>, 2006. Last Access: March 2006.
- [3] "Openssl website." <http://www.openssl.org>, 2005. Last Access: March 2006.
- [4] T. H. Project, "Know your enemy: Sebek." <http://project.honeynet.org/papers/sebek.pdf>, November 2003. Last access: Feb 2006.
- [5] E. Balas and C. Viecco, "Towards a third generation data capture architecture for honeynets," in *Proceedings of the 6th Information Assurance Workshop*, IEEE, June 2005.
- [6] T. C. Keong, "Detecting sebek win32 client." <http://www.security.org.sg/vuln/sebek215.html>, 2004.
- [7] C. N. K. Maximillian Dornseif, Thorsten Hoslz, "Nosebreak - attacking honeypots," in *Proceedings of the 5th Information Assurance Workshop*, IEEE, June 2004.
- [8] J. B. Sherri Sparks, "Raising the bar for windows rootkit detection." <http://www.phrack.org/show.php?p=63&a=8>. Last access: March 2006.
- [9] T. H. Project, "Know your enemy: Honeynets." <http://project.honeynet.org/papers/honeynet/index.html>, May 2005. Last access: Feb 2006.
- [10] intel, "Cross intel architecture development tools: Code profiling." <http://www.intel.com/cd/ids/developer/asmo-na/eng/19992.htm?page=10>. Last access: March 2006.
- [11] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, October 2003.
- [12] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality," in *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005.
- [13] L. Spitzner, "Honeytokens: the other honeypot." <http://www.securityfocus.com/infocus/1713>, August 2003. Last Access: March 2006.