The Cost of Observation for Intrusion Detection: Performance Impact of Concurrent Host Observation

Mark M. Seeger*‡, Stephen D. Wolthusen‡§, Christoph Busch*‡ and Harald Baier*

*Department of Secure Services
Center for Advanced Security Research Darmstadt (CASED)
Mornewegstrasse 32, 64293 Darmstadt, Germany
Email: {mark.seeger, christoph.busch, harald.baier}@cased.de

‡Department of Computer Science
Gjøvik University College
N-2818 Gjøvik, Norway

§Information Security Group
Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK
Email: stephen.wolthusen@rhul.ac.uk

***Abstract*—Intrusion detection relies on the ability to obtain reliable and trustworthy measurements, while adversaries will inevitably target such monitoring and security systems to prevent their detection. This has led to a number of proposals for using coprocessors as protected monitoring instances. However, such coprocessors suffer from two problems, namely the ability to perform measurements without relying on the host system and the speed at which such measurements can be performed.**
The availability of smart, high-performance subsystems in commodity computer systems such as graphics processing units (GPU) strongly motivates an investigation into novel ways of achieving the twin objectives of self-protected observation and monitoring systems and sufficient measurement frequency. This, however, gives rise to performance penalties imposed by memory synchronization particularly in non-uniform memory architectures (NUMA) even for the case of direct memory access (DMA) transfers.
Based on prior work detailing a cost model for synchronization of memory access in such advanced architectures, we report an experimental validation of the cost model using an IEEE 1394 DMA bus mastering environment, which provides full access to the measurement target's main memory and involves multiple bus bridges and concomitant synchronization mechanisms. We observed up to 25% performance degradation, highlighting the need for efficient sampling strategies for both, memory size and a preference for quiescent data structures for monitoring executed by off-host devices.

***Index Terms*—Host intrusion detection, asynchronous memory access, coprocessor, DMA, IEEE1394, NUMA**

I. Introduction

Despite the fact that the use of coprocessors for host intrusion detection has been proposed years ago ([1], [2]) they are currently not used in this domain. Even though their practical applicability has been shown by [3][1], commercial host ID software is still being installed on the host and executed by its CPUs.
In contrast to this, coprocessors are applied in the context of network ID ([4], [5]). The ease of accessing data from network traffic in contrast to data on a host system (i.e. the host's main memory) and putting it under the audit of an auxiliary processor has contributed its part to this development.
The use of graphics chipsets or PCI processing devices ([6]) and GPUs in general ([7]) for host intrusion detection has been mentioned in previous work. While [6] mostly deal with self-protection of the observing component itself, [7] introduce a model capable of expressing the *costs* such observations can cause. That is, the authors pay special attention to concurrent memory accesses, particularly in a NUMA architecture as represented by a standard multi-core (and especially multi-processor) system. The actual feasibility of a GPU in order to perform host intrusion detection was the subject of [8]. Although the model proposed in [7] and utilised here is intended mainly for high-speed interconnections such as these offered by GPU and similar components, it is applicable to all non-uniform concurrent memory access architectures. The read-only (observation) access to a state variable of a target process can cross several cascading memory hierarchy layers with the ultimate shared resource in such architectures generally being main memory. As snooping and cache coher-

[1]This included a proof of concept implementation called CoPilot.

ence protocols allow the efficient but forced synchronisation without the possibility of intervention on the part of software components, this provides a mechanism for concurrent state observations that cannot be corrupted or compromised. While [7] is concerned about observations executed by the GPU, similar effects with regard to resource contention and ultimately performance degradation on the host system can be caused by every memory access using DMA. Having in mind the limitations revealed by [8] we thus propose another hardware based approach in this paper: That is, we exploited the ability of the IEEE 1394 technology in order to gain full access to the main memory of a connected computer. In order to make the observation effects as clear as possible, a workload generator produced alterations in a data structure of a pre-defined size. As we measured the CPU cycles consumed by our generator in both cases – with and without observation – we were able to quantify the loss of performance caused by observations executed by a processor other than the CPU of the host system. The results obtained are of prime importance for the design of smart host intrusion detection algorithms running on off-host components. They provide central information with regard to the correlation of factors such as observation frequency, size of the data structure to be observed and the resulting performance degradation.

The remainder of this paper is structured as follows: In Section II we give an overview of the IEEE 1394 serial bus interface and point out the reasons for its applicability in our context. We also provide a historical insight into the main contributions regarding the field of physical attacks using the IEEE 1394 technology. We then present a description of our experimental setup that was used to obtain our results in Section III followed by a presentation of the actual results in Section IV. The paper is closed with a conclusion in section V and a brief description of our future work in Section VI.

II. IEEE 1394

The IEEE 1394 interface, better known as FireWire, was initially developed by Apple Computer, Inc. in the 1980's [9]. Standardized by the Institute of Electrical and Electronics Engineers (IEEE) for the first time in 1995 (IEEE 1394-1995) [10], the latest changes have been published in October 2008 [11]. Due to its high speed and low overhead, today the FireWire technology is mainly used for fast file transfer between external periphery such as mass data storages (e.g. hard drives, secure digital (SD) memory cards, etc.) and a computer. But in contrast to the universal serial bus (USB), the FireWire standard allows for the communication between devices itself. Areas of application are here the communication between a digital camera and a printer, for instance.

In January 2000, version 1.1 of the open host controller interface (OHCI) specification was released by contributors from seven well known computer companies, headed by Apple Computer, Inc. [12]. This interface is an implementation of the link layer protocol of the 1394 serial bus and empowers a broader variety of devices to take advantages of the IEEE 1394 interface. Furthermore, it is the OHCI that has features such as memory-mapping implemented in hardware which allows for a communication between two FireWire devices without the interaction of the operating system (OS) of either of them. Chapter 12 of the OHCI specification defines: *"When a block or quadlet read request or a block or quadlet write request is received, the 1394 Open HCI chip handles the operation automatically without involving software if the offset address in the request packet header meets a specific set of criteria..."* [12]. For us, the relevant criterion to meet is the one that specifies that the address has to fall within the physical range of the target memory space. This range is defined by lower and upper bounds. While the offset is at address `48'h0` the address of the upper bound is either at `48'h000_FFFF_FFFF` or stored in the field `physUpperBoundOffset`[2]. This of course implies a security risk not only in theory but in practice as well, as we will see now.

A. Physical Attacks and Observations Using IEEE 1394

The properties of the IEEE 1394 technology noted above are also attractive in the security context for both, attacks and forensic purposes. Although not within the scope of the present paper, we note that this use has e.g. been documented informally during the MacHack 17 (2002) convention, where a proof-of-concept for overwriting screen memory between two Apple Macintosh computers was demonstrated [13]. This was further elaborated by Dornseif et al. in 2004 using an Apple iPod for reading and writing to arbitrary memory locations in host systems without interacting with the target host operating system [14].

As expected and demonstrated by Boileau ([15]), this is fully independent of any host operating system, although the device class must be known to the host system. This can, however, be easily emulated by setting the appropriate status registers to a known device and class, e.g. a mass storage device. Since all tools necessary for such an attack are freely available, having physical access to an enabled FireWire port on a computer is equal to having full access to its main memory. This includes use cases like changing the password protection code stored in main memory of a screen-locked computer in such a way that it accepts just any input. The intended lack of any access control mechanism or authentication schema between the communicating devices makes this possible.

For the purposes of attacks, performance considerations are largely irrelevant. However, forensic applications will aim to maximize speed. Particularly for the case of forensic memory capture, the fact that IEEE 1394 devices are typically coupled to main memory by one or more bus bridges makes this susceptible to chip-set modifications as e.g. proposed and demonstrated by Rutkowska [16].

We note that the transitioning over multiple bus bridges is also affecting the measurements observed in the present paper as this implies not only matching different memory, bus, and device speeds, but also the use of multiple cache coherence and synchronization protocols.

[2]In order to stay within the range, the value stored here needs to be decremented by one.

III. Experimental Setup

In this section we give a description of the experimental setup that was used in order to measure the impact, host memory observations executed from an off-host component can have on the system being observed. The results will be presented in detail in Section IV. The full project, containing the source code of our workload generator as well as all results, is available at http://sourceforge.net/projects/dmamemoryobserv.

We used two computers: one being the observer and the other one being the target. Each of them provided a S400 FireWire interface.

Observer: Ubuntu Linux 10.04 64-bit, kernel version 2.6.32, 4GB RAM, Intel Core 2 Quad (Q8200, 2.33GHz).

The establishment of the FireWire connection between the two computers, as well as the access to the main memory and the data transfer, is accomplished by the open source tools published by Boileau [17]. The underlying modules raw1394, IEEE1394, sbp2 (serial bus protocol 2) and OHCI1394 are mandatory and usually distributed as part of the kernel but can also be downloaded from the corresponding repository if not. For the observation of a certain amount of data at specific locations, we wrote a wrapper around Adam Boileau's `readWithExclusion()` function which takes the corresponding addresses pointing to the data as an argument.

Target: ARCH Linux 64-bit (a simple and lightweight Linux distribution), kernel version 2.6.33, 2GB RAM, Intel Core 2 Duo (E8400, 3.0GHz).

In our setup, the target's only duty is to run a workload generator and to log the CPU cycles that were consumed while executing it. The workload generator is a C++ console program producing a synthetic workload by continuously writing to a data structure of a pre-defined size. The idea behind this program corresponds to the basic statement of [7]: A forced synchronization between different memory levels (i.e. a cache and the main memory) takes place, whenever two conditions are met: (1) The higher level holds an altered copy $[m]'$ of the original but outdated data $[m]$ which resides in the physical memory. (2) $[m]$ is accessed by another (co)processor. In this case, synchronization must take place in order to serve the accessing (co)processor with the latest data.

The basic pseudo code of our workload generator is shown in Listing 1. The parameters of the `main()` function are:

- *blockSize*: The size (in bytes) of the data structure the workload generator works with.
- *runs*: The number of executions of the `workload()` function.
- *iterations*: The number of times the data structure is written during each execution of the `workload()` function.
- *ratio*: The ratio (i.e. $1 \geq ratio > 0$) expressing the percentage of the data structure that is actually being written.
- *type*: A string, being written to the log file, indicating whether the test ran under observation (o) or normal duty (nd).

As mentioned in previous work, we propose the observation of relations among critical components of the operating system and its security components. Therefore, we deal with rather small data structures compared to intrusion detection systems which use a signature-based approach. One example here is the observation of entries in the system-call-table, which serves as an interface between user and kernel mode. By altering the function pointers of certain kernel functions or by falsifying their return values, an adversary can successfully conceal the presence of malicious software. As the function names, as well as the function numbers and pointers are rather small, we have decided to run our experiments with a data structure between 8 and 64 bytes.

The number of computations (i.e. the execution of line 18 in Listing 1) can be adjusted by the parameters *runs*, *iterations* and *blockSize*. By incrementing the number of *runs* by the same value the number of *iterations* gets decremented (i.e. `runs += y; iterations -= y`), the number of computations would stay same. But it is important to understand the two major side effects this would have: Experiments showed that $(5 \cdot 10^9) \cdot blockSize$ results in a reasonable runtime for the workload generator when the size of the data structure lies between 8 and 64 bytes. That is, the runtime is high enough to obtain meaningful results even when working with small data structures (i.e. 8 bytes) and low enough to allow for an execution of all tests withing one day when working with bigger ones (i.e. 64 bytes). By setting parameter *runs* to $5 \cdot 10^9$ and thus, omitting the first `for` loop in the `workload()` function, we would produce a log file containing $5 \cdot 10^9$ entries; too much data to work with. The second effect is, that by omitting this `for` loop, the time between the two cycle measurements would be very short and therefore, less meaningful.

The parameter *ratio* relates to the equation for calculating the interference ratio for a given amount of data presented in previous work ((1) in this paper). According to [7], the performance loss due to observation depends not only on the amount of data being observed but also on its composition. That is, read-only data cannot be written, therefore, does not need to be synchronized and thus, will not cause resource contention.

$$I = \begin{cases} min, & \text{for } r = 1 \lor x = 0, \\ max, & \text{for } x = 1, \\ undef, & \text{for } x = 1 \land w = 0, \\ \frac{w \cdot x}{1-r} \cdot [t], & \text{for } r < 1 \land 0 < x < 1 \end{cases} \tag{1}$$

In (1), r stands for the ratio of read-only and w for the ratio of writable data ($r + w = 1$). x expresses how much of the processing timespan $[t]$ is actually used for writing data.

The special cases lead to a $minimal$ ($I = 0$) and $maximal$ ($I = 1$) interference rate respectively, while the case of spending all time ($x = 1$) on writing data when there is no writable data available ($w = 0$) is $undefined.$

It is clear to see that the interference ratio I becomes smaller

by decreasing the amount of writable data (w) being written (x). And in our setup, this can be done by defining a stress ratio between 0 and 1 which is expressed by the parameter *ratio*, determining the percentage of the workload that is actually being written.
Before every individual test, we log its characteristic, e.g., whether or not the target process ran under normal duty (nd) or was under observation (o). This information is passed by the parameter *type*.

Listing 1. Pseudo Code of the Workload Generator.

```
main(blockSize, runs, iterations, ratio, type)
{
    log.write(printInfo());
    m = malloc(blockSize);
    for(i < runs)
    {
        startCycles = rdtsc();
        workload(m, iterations, ratio);
        endCycles   = rdtsc();
        log.write(endCycles - startCycles);
    }
}

void workload(m, iterations, ratio)
{
    for(i < iterations)
        for(j < m.size()*ratio)
            m[j] = m[j+1] + 1;
}
```

In order to get reliable results regarding the execution time of each run, we made use of Intel's Read Time-Stamp Counter instruction `rdtsc` [18]. This assembler command is available since the first Pentium model range and returns the value of a 64-bit model specific register that is incremented every CPU cycle [19]. Furthermore, we disabled the multi-core support and the Intel SpeedStep feature to make sure that our program is carried out by one core only and to keep the maximum CPU clock rate static.

The full experiment included three different cases, characterized by different parameters. Each case U_{1-3} comprised four test tuples T_{1-4} and each test tuple consisted of two tests t_{1-2}. The parameters for each case are given by the signature of the `main()` function in Listing 1: While the parameters *runs* (10,000) and *iterations* (0.5mio) stayed the same for all tests making the results comparable, *blockSize* (8, 16, 32, 64), *ratio* (0.5, 1.0) and *type* (o, nd) were adjusted in compliance to the following conditions:
For all cases U_{1-3}:

- The test tuples within the same case are executed with a different *blockSize*: $T_1 = 8$, $T_2 = 16$, $T_3 = 32$ and $T_4 = 64$
- The tests within the same test tuple are executed with a different *type* but derive the *blockSize* from the corresponding test tuple. Therefore, they are executed with the same *blockSize* but with a different *type*.
- Only one instance of the workload generator executing the tests is running on the target, except for U_2: here, two instances of the workload generator are started, executing the exact same tests in parallel.
- The stress ratio for all tests is 1.0 except for the tests within U_3: here, it is set to 0.5.

The lower bound of the parameter *blockSize* was set to 8, in order to assure a reasonable runtime in conjunction with the agreed on values for the parameters *runs* and *iterations*. All subsequent values are multiples of 8, allowing for an easy comparison of the overall results.
As can be seen, all three cases consist of four experiments each where each experiment is conducted with and without concurrent observation. With respect to the conditions above, in the first case we had only one instance of the workload generator running on the target. For the second case we started two instances in parallel and observed both at the same time. In either one of them all bytes allocated were actually written (*ratio* = 1.0). In case three, we again used one instance only but the stress ratio was set to 0.5. In this case only 50% of the allocated data were altered. Once a case was initialized (i.e. all tests were configured and ready for execution), we dumped the full main memory of the target and searched through it, looking for the addresses pointing to our workload. According to the *blockSize* assigned, we then started the observation of the corresponding amount of data whenever parameter *type* = *o*.

IV. Results

Based on the case conditions introduced in Section III, we obtained meaningful results which allow for a direct performance comparison with regard to the CPU cycle consumption of a given process with and without observation. By continuously observing only the actual synthetic workload (i.e. only that amount of data – defined by the parameter *blockSize*– that is actual being worked with) we forced an immediate synchronization between the caches and the main memory and therefore caused a resource contention.
Our results can be seen as a benchmark. As with any other benchmark, it is of prime importance that the results being compared with each other originate from identical experimental setups. Furthermore, the development of CPUs with intrinsic power saving features such as lowering the clock rate or turning off (parts of) certain caches attributes to the fact that reproducing the exact results will most likely be impossible, even if the experiments are carried out by an identical hardware setup. That is, in contrast to, for instance, the Intel SpeedStep feature, which can be seen as a "global" property, future developments will be more fine grained (i.e. instead of a per-socket-feature, a per-core or per-ALU feature will be in place), not allowing for any adjustment from the outside. As a result of this, the number of outliers will increase causing a higher standard deviation.
Our results show several clear and generalizable trends and distinctive technical features, expressed by six characteristic values for each test:

- **Process runtime:** This is the time, measured in minutes, executing an individual test took.

- **Measurements:** During each observation, we measured the number of measurements per minute.
- **Mean Value:** As mentioned before, each test returned 10,000 measurements of the consumed CPU cycles. In order to make a statement regarding the average consumption, we calculated the mean value.
- **Standard Deviation:** The standard deviation gives an idea about the level of heterogeneity of all 10,000 measurements per test.
- **Outliers:** A high standard deviation can be an indicator for both, a very heterogeneous sample space and a reasonable number of outliers. A closer look into the distribution of our measurements revealed the existence of outliers.
- **Performance Degradation:** The most interesting statement is the one regarding the difference between the CPU cycles consumed by the workload generator with and without observation. This performance degradation is expressed in percentages.

Before we present the three generalizable trends according to the characteristic values from above, we want to describe the results of one sample case, namely the one where we observed one process only with a stress ratio of 1.0. Thus, the results of all other cases are of a similar character, the in detail presented example shows the effects of observations carried out by a coprocessor very clear.

A. Sample Results: Observing One Process with a Stress Ratio of 1.0

As mentioned before, one case consisted of eight tests, where each two tests of one tuple are characterized by working on a data structure of the same size but are of a different type. Table I shows the conditioned results obtained from all tests within case one (one process instance, *ratio* = 1.0).

TABLE I
ALL CONDITIONED RESULTS OBTAINED FROM OBSERVING ONE PROCESS WITH A STRESS RATIO OF 1.0.

Runtime in min	Measm./min	Size	Perf. Degr.	Mean Value	Outlier	Std. Div.
3.07	0	8	0.77%	54.99mio	180	36.24%
3.08	130,768			55.41mio	183	36.26%
4.97	0	16	1.28%	88.90mio	294	28.48%
5.02	69,887			90.04mio	298	28.30%
8.15	0	32	1.48%	146.04mio	480	21.94%
8.25	34,076			148.20mio	491	21.84%
15.57	0	64	16.39%	279.41mio	928	15.63%
18.12	16,881			325.19mio	1,085	14.34%

Column one (runtime in minutes) represents the runtime of each test. The figures in the second column (measurements per minute) stand for the frequency with which the observation of the data structure – characterized by the size shown in column three (size) – was executed. The zeros in the second column indicate that this test ran without observation (parameter *type* = nd). Column four (performance degradation) depicts the percentage values of the difference between two tests of the same test tuple, based on the averaged CPU cycles shown in column five (mean value). The number of outliers, stated in column six, will be discussed shortly. They are closely related to the standard deviation shown in column seven.

The high standard deviations shown in the last column of Table I is due to the fact that our data is distinguished by a number of very clear outliers. Using the results that lead to the values shown in the first row of Table I, we created a histogram that clearly reveals the outliers. Figure 1 sharply uncovers that 98.19% of all values lie within the range of 53mio - 54mio CPU cycles. Further investigation showed that the outliers appear almost periodically which strengthens our suspicion that these exceptionally high values are caused by processes belonging to the operating systems running on the target, demanding for processor time. Table II depicts the

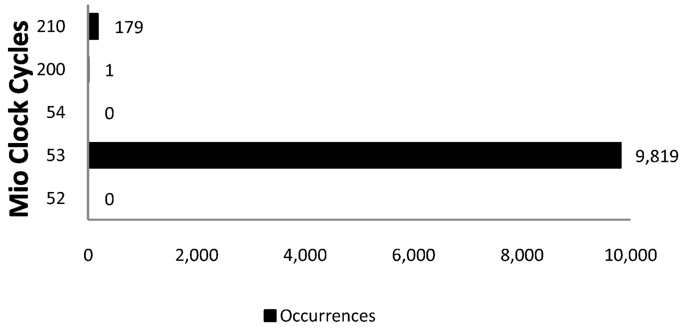


Fig. 1. Histogram of all results from the first test (corresponding to row one in Table I).

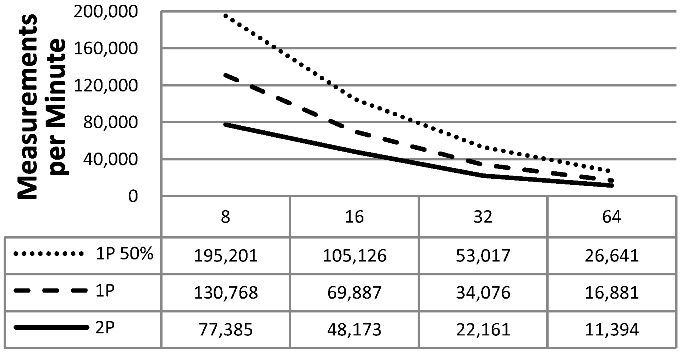
normalized results of the first test. That is, we eliminated all outliers in order to show that the actual result set is quite homogeneous. The still notably higher standard deviation of the normalized results for the last test tuple is due to a higher degree of heterogeneity in the corresponding result set, which increases with the amount of data being processed by the observer. According to the characteristic values in Table I and II, we derived three generalizable trends described in the following subsections.

TABLE II
NORMALIZED CONDITIONED RESULTS FOR OBSERVING ONE PROCESS WITH A STRESS RATIO OF 1.0.

Performance Degradation	Mean Value	Standard Deviation
0.72%	52.29mio	0.04%
	52.67mio	0.10%
1.27%	84.50mio	0.02%
	85.57mio	0.02%
1.44%	138.85mio	0.02%
	140.85mio	0.18%
16.36%	265.45mio	1.47%
	308.94mio	1.26%

B. Observing Frequency vs. Size of the Data Structure

As mentioned earlier, the size of the data structure we observed was set to either 8, 16, 32 or 64 bytes by the parameter *blockSize*. While the amount of data being subject to observation rises, the frequency with which the data can be observed decreases *almost* proportionally: While we were able to observe 8 bytes with an average frequency of 195,201 measurements per minute, twice the amount of data (16 bytes) could only be observed with roughly half of the frequency (105,126 measurements per minute). This behavior is depicted in Figure 2.



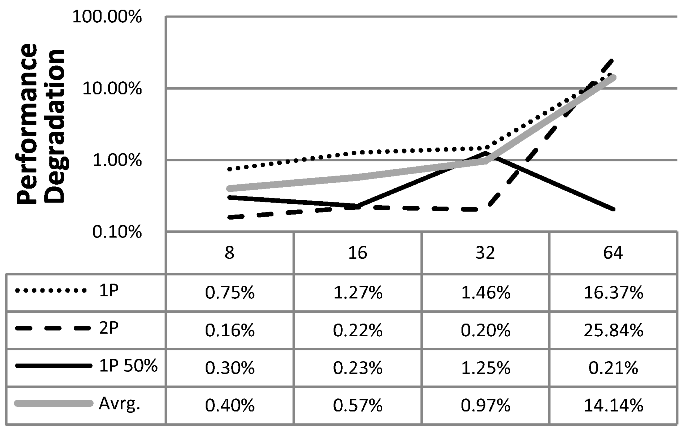
	8	16	32	64
······· 1P 50%	195,201	105,126	53,017	26,641
— — — 1P	130,768	69,887	34,076	16,881
——— 2P	77,385	48,173	22,161	11,394

Fig. 2. Measurement Frequency for Different Data Structure Sizes.

The explanation for this effect is rather straightforward: Observing data from a coprocessor implies the task of copying the concerned data from the host towards a memory closer to the coprocessor. Transferring small chunks of data is just faster than transferring bigger ones. And as we have only one DMA channel between our target and the observer, a new observation can only start after the previous one has finished.
For interpreting Figure 2 correctly, it is important to know that the lines connecting the measuring points have just been added in order to emphasize the trend and to facilitate the readability. The impression that the frequency degradation is almost linear is deceitful.

C. Performance Degradation

Even though the details of Figure 3 may mislead to a slightly different conclusion, the averages calculated from all values (depicted by the solid gray line) clearly show: The larger the data structure we observe, the bigger the resource contention on the host being subject to observation. The percentaged values shown in Figure 3 are the averages calculated from the performance degradation with and without outliers. The message this figure sends out is clear: Observations have a significant performance impact on the host. While the measurements for the case of observing one or two processes with a stress ratio of 1.0 (i.e. the workload generator works on 100% of the initialized data) confirm this message, the curve describing the case of observing one process with a stress ratio of 0.5 seems to contradict it. That is, while executing the tests with a data structure of 64 bytes and a stress ratio of 0.5, the performance impact decreased drastically. This



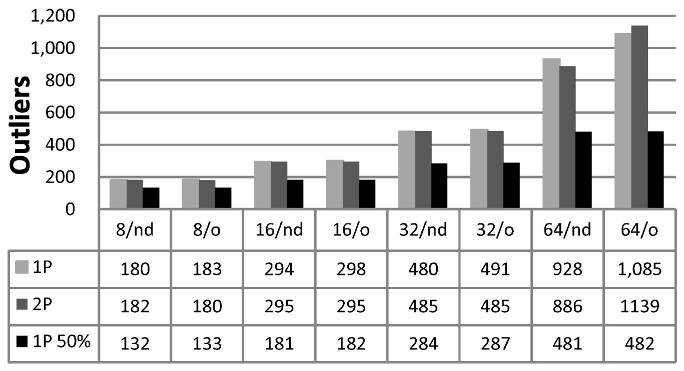
	8	16	32	64
······· 1P	0.75%	1.27%	1.46%	16.37%
— — — 2P	0.16%	0.22%	0.20%	25.84%
——— 1P 50%	0.30%	0.23%	1.25%	0.21%
——— Avrg.	0.40%	0.57%	0.97%	14.14%

Fig. 3. Performance Degradation according to the Size of the Observed Data Structure.

effect is most likely the result of system based algorithms dealing with the assembling of cache lines, working sets, etc. by using techniques such as pre-fetching. Since our way of measuring the performance impact is an end-to-end approach, the explanation of such effects is out of scope. We leave aside the exact diagnosis of intrinsic impacts caused by protocol overheads or diverse bridges and controllers our signal passes as well as the performance boost a process can experience when the executing processor benefits from a good locality of reference within the data it demands. Therefore, we measure what is important in practice: the degree of performance degradation including all side effects.
The downside of this approach is, that we can only speculate regarding the causes of certain effects. Special equipment as used by chip vendors would have been necessary in order to further investigate on these effects. The important remark here: The results stay the same, even if one would be in the position of giving a detailed description on the internal behavior.

D. Increasing Number of Outliers

In Section IV-A we already presented a closer look into the outlier effect and used the results of our first test (row one in Table I) to show that besides the easy identifiable outliers, our result set is heterogeneous. Figure 4 now depicts the growth of the number of outliers with regard to the size of the data structure the workload generator works with.



	8/nd	8/o	16/nd	16/o	32/nd	32/o	64/nd	64/o
■ 1P	180	183	294	298	480	491	928	1,085
■ 2P	182	180	295	295	485	485	886	1139
■ 1P 50%	132	133	181	182	284	287	481	482

Fig. 4. Increasing Number of Outliers.

Three major characteristics are important to point out: The first one is that our observation causes the number of outliers to increase. While for the cases where we observed smaller data structures the difference was marginal, the gap rose when we increased them. Interesting to know here is the fact that the number of outliers has only a very small influence on the performance degradation[3] as one can see by comparing the corresponding values shown in Tables I and II. The second important point regards to the growth of the absolute appearances of outliers. That is, the more data we work with, the more outliers appear. And last but not least, one can see that working with only half the data initialized caused roughly half the amount of outliers. Even though the outliers are clearly to identify in all cases and thus can be eliminated for statistical purposes, they are an intrinsic element of the workload process running on the target. The avoidance of such effects may be desirable but due to several reasons (e.g. multi-tasking on one processor) not always possible.

V. Conclusion

We have shown that memory observation as may arise in concurrent observation for intrusion detection and prevention, implies a non-negligible cost – particularly when executed over a shared memory or non-uniform memory architecture. Validating a model for such NUMA architectures, this paper has shown that data structure size and the fraction of write activities is clearly having effects that cannot be fully masked by even advanced processor and memory architectures.

Memory observations executed by a coprocessor imply *costs* (i.e. performance degradation) on the side of the host being observed. And in this work we were able to quantify these costs for a given hardware. [7] explain that the reasons for this loss of performance can be found in the memory architecture of today's computers: Data most recently used by a processor is copied to a memory level which is much smaller than the systems main memory but also much faster. The probability that data used at time t_0 is needed at time t_1 makes this approach very efficient. Whenever the copied data gets altered, a synchronization between the different memory levels involved (i.e. in most architectures L1-cache, L2-cache and main memory) does not take place immediately. Instead a coherence protocol is implemented, triggering the synchronization according to a set of rules. One of these rules is, that a synchronization must take place when another processor is about to access data residing in the main memory that is marked as *dirty*. That means that there exists an altered copy of this data in a higher memory level. In order to fulfill the requirement of serving every processor with the most current data, synchronization is indispensable. And since the accessing speed of the different memory levels differ potentially by one or more orders of magnitude, each synchronization results in a loss of performance. A cost model for expressing the worst case with regard to the performance degradation such synchronizations can cause has been presented in previous work. Its practical and quantitative validation was missing until now.

[3]Because of this, we were able to average the values for the performance degradation shown in Figure 3.

The results obtained from our experiments fully validate the theoretical statements proposed previously. We implemented a workload generator and executed it on a lightweight Linux operation system (i.e. ARCH Linux) in order to see the effects as clearly as possible. The actual observation was realized using a high-speed bus-mastering DMA channel established between a second computer and the target system. This corresponds to the prerequisite of [7] to have the observation executed by another processor than the one operating the host system (i.e. a coprocessor). By running tests for three different cases (eight tests per case) where each test was distinguished by the parameters with which it has been executed, we clearly unveiled the performance impact. With the cases executed in our test environment (see Section III), we experienced a performance degradation of more than 25% for the case where we observed two instances of our workload generator running in parallel and working on a data structure of 64 bytes each.

The size of the data structure being observed is not the prime factor leading to a well-founded assumption of whether or not the loss of performance is expected to be rather high or low. That is, the composition of the data structure plays a vital role: As already mentioned, synchronizations take place, when altered cached data needs to be written back to the main memory. If the cached data is either read-only or writable but not being written, its value will never change and thus, synchronization is not needed. Together with this statement, an equation capable of calculating the inference rate was published (see (1)), basically saying that the observation of read-only data is *cheap* in comparison to the observation of writable data, that is actually being written. By setting the stress ratio for one case to 0.5, we achieved that the workload generator worked on only 50% of the data initialized.

It can clearly be seen that the results validate the equation presented in previous work, as – compared to the cases with a stress ratio of 1.0 – the performance impact on the target was significantly lower as shown in Table III. Here, we juxtaposed the percentaged performance degradations for the case of observing one instance of the workload generator working on 50%, respectively 100% of its workload. Just as in Figure 3 the percentages shown in Table III are the averages obtained from calculating the performance degradation with and without inclusion of outliers. It is clearly to see that the

TABLE III
Comparison of Performance Degradations Depending on the Amount of Data actually being written.

Performance Degradation	
ratio = 0.5	*ratio* = 1.0
0.30%	0.75%
0.23%	1.27%
1.25%	1.46%
0.21%	16.37%

performance degradation is much lower when not 100% of the data being observed is altered (tested with 50% and 100% in our case).

We have shown that memory observation executed by a coprocessor does not come for free. In fact, depending on the size and composition of the data structures being subject to observation, the performance degradation varies: Bigger data structures and a higher appearance of data being written, result in a more serious loss of performance.

With regard to future work, the results represented in this paper are essential. They prove that an observation strategy according to the busy-wait approach would result in a tremendous loss of performance and is therefore not desirable. This is especially true when we consider that the results presented in this work were obtained using the comparatively slow S400 FireWire technology (400Mbit/s), whose interface is bound to the input/output controller hub (southbridge) while the host's main memory is connected to the memory controller hub (northbridge). This resulted in a signal latency and caused the maximum measurement frequency to be rather low. The maximum measurement frequency that could have been achieved by using a highly parallel and multi-threaded multi-core GPU – connected to the northbridge – would have been much higher. Thus, causing an even bigger performance degradation when following the busy-wait approach is likely.

VI. Future Work

The results reported in the present work were obtained from a single observer and target tuple. In order to rule out anomalies arising from this particular configuration, ongoing experiments are being conducted with further platforms.

In our experimental setup we took advantage of the FireWire S400 technology which served as an easy to use approach to observe the data structures on the target. This was done with a maximum of 195,201 measurements per minute. Besides the bandwidth limitation as such (400Mbit/s), the fact that our signal had to pass two south- and northbridges (target and observer) accounts for the relatively low observation frequency. In our future work, we will overcome the limitations revealed by [8] and use the GPU (up to 16GB/s) for our purposes. By doing so, the measurement frequency will benefit from a higher bandwidth due to the fact that this internal coprocessor is directly connected to the same integrated memory controller the main memory is coupled with.

By designing smart algorithms which are capable of exploiting the parallel design of modern graphics cards, we will propose an observation model that takes advantage of the maximum available observation frequency if necessary but usually tries to work as efficient as possible. That is, keeping the frequency as low as possible while still guarantying a reasonable probability that subversion is detected while taking place.

Acknowledgment

The authors would like to thank Julian Knauer, Pierre Schnarz and Sufyan Siddique for their excellent practical support and Reinhard Riedmüller for his comments on the final draft.

References

[1] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure Coprocessor-Based Intrusion Detection," in *Proceedings of the 10th ACM SIGOPS European Workshop*, G. Muller and E. Jul, Eds. New York, NY, USA: ACM Press, Jul. 2002, pp. 239–242.

[2] P. D. Williams and E. H. Spafford, "CuPIDS: An Exploration of Highly Focused, Co-Processor-based Information System Protection," *Computer Networks*, vol. 51, no. 5, pp. 1284–1298, Apr. 2007.

[3] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot — A Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings of the 13th USENIX Security Symposium*, M. Blaze, Ed. San Diego, CA, USA: USENIX Press, Aug. 2004, pp. 179–194.

[4] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *RAID 2008: Proceedings of the 11th International Symposium of Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, vol. 5230. Springer Berlin / Heidelberg, 2008, pp. 116–134. [Online]. Available: http://www.springerlink.com/content/g2w54q11130r7126/

[5] Y. H. Cho and W. H. Mangione-Smith, "A pattern matching coprocessor for network security," in *DAC '05: Proceedings of the 42nd annual Design Automation Conference*. New York, NY, USA: ACM, 2005, pp. 234–239.

[6] T. R. McEvoy, S. D. Wolthusen, and 2, "Host-based security sensor integrity in multiprocessing environments," in *Information Security, Practice and Experience*, ser. Lecture Notes in Computer Science, vol. 6047. Springer Berlin / Heidelberg, 2010, pp. 138–152.

[7] M. M. Seeger and S. D. Wolthusen, "Observation mechanism and cost model for tightly coupled asymmetric concurrency," in *ICONS 2010: Proceedings of The Fifth International Conference on Systems*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 158–163.

[8] R. Riedmüller, M. M. Seeger, S. D. Wolthusen, H. Baier, and C. Busch, "Constraints on Autonomous Use of Standard GPU Components for Asynchronous Observations and Intrusion Detection," in *IWSCN 2010: Second International Workshop on Security and Communication Networks*. IEEE Computer Society (in press), 2010.

[9] A. Paskins, "The IEEE 1394 bus," in *In Proceedings of: New High Capacity Digital Media and Their Applications*, May 1997, pp. 4/1 – 4/6.

[10] Institute of Electrical and Electronics Engineers, "IEEE Standard for a High-Performance Serial Bus," *IEEE Std 1394-1995*, p. i, 1996.

[11] ——, "IEEE Standard for a High-Performance Serial Bus," *IEEE Std 1394-2008*, pp. 1 –906, 21 2008.

[12] Apple Computer, Inc., Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, National Semiconductor Corporation, Sun Microsystems, Inc. and Texas Instruments, Inc. (2000, Jan.) 1394 – Open Host Controller Interface Specification. pdf. [Online]. Available: http://www.storm.net.nz/static/files/ohci_11.pdf

[13] Quinn. (2002) FireStarter. [Online]. Available: http://www.anarchistturtle.com/Quinn/WWW/Hacks.html

[14] M. Dornseif. (2004, Nov.) 0wned by an iPod. pdf. [Online]. Available: http://md.hudora.de/presentations/firewire/PacSec2004.pdf

[15] A. Boileau. (2006) Hit by a Bus: Physical Access Attacks with Firewire. pdf. [Online]. Available: http://www.storm.net.nz/static/files/ab_firewire_rux2k6-final.pdf

[16] J. Rutkowska. (2007, Feb.) Beyond The CPU: Defeating Hardware Based RAM Acquisition. pdf. [Online]. Available: http://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf

[17] A. Boileau. (2008, Mar.) Firewire, DMA & Windows. [Online]. Available: http://www.storm.net.nz/projects/16

[18] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," Intel Corporation, Tech. Rep., Mar. 2010. [Online]. Available: http://developer.intel.com/assets/pdf/manual/253667.pdf

[19] ——, "Using the RDTSC Instruction for Performance Monitoring," Intel Corporation, Tech. Rep., 1997. [Online]. Available: http://www.ccsl.carleton.ca/ jamuir/rdtscpm1.pdf