Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior

Man-Ki Yoon*, Sibin Mohan†, Jaesik Choi‡, and Lui Sha*
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801
‡School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, Korea
{mkyoon, sibin, lrs}@illinois.edu, jaesik@unist.ac.kr

ABSTRACT

In this paper, we introduce a novel mechanism that identifies abnormal system-wide behaviors using the predictable nature of real-time embedded applications. We introduce *Memory Heat Map* (MHM) to characterize the memory behavior of the operating system. Our machine learning algorithms automatically (a) summarize the information contained in the MHMs and then (b) detect deviations from the normal memory behavior patterns. These methods are implemented on top of a multicore processor architecture to aid in the process of monitoring and detection. The techniques are evaluated using multiple attack scenarios including kernel rootkits and shellcode. To the best of our knowledge, this is the first work that uses aggregated memory behavior for detecting system anomalies especially the concept of memory heat maps.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Intrusion detection, memory heat map, real-time systems

1. INTRODUCTION

Real-time embedded systems are increasingly coming under attack especially due to their increased complexity and connectivity. Applying traditional security mechanisms in real-time systems is much harder due to their limited resources (CPU, memory, *etc.*). On the other hand, these systems are also *predictable* by *design*. Hence, we use this property of real-time embedded systems to improve security.

In this paper, we present techniques to detect *system-wide anomalies* in the execution of real-time embedded systems by monitoring *the behavior of memory accesses* for the operating system. While other behavioral properties have been explored in the past [17, 23, 22], memory access is an important property since it is particularly hard to fake or hide (for malicious tasks). We find that the memory profiles of many real-time applications have a predictable nature and use this property to detect malicious activity.

The memory behavior of a system can be defined in many ways. For instance, one could track the exact sequence of memory addresses that are accessed. However, it requires a prohibitive amount of storage not to mention excessive computation times for lookup, and is also highly sensitive to legitimate variations. On the other hand, we could monitor the amount of memory traffic. However, it could abstract away from the detection of small, abnormal variations.

To avoid such problems we introduce the use of a novel method to profile memory behavior, *viz.,* the *Memory Heat Map* (MHM). The MHM is a concise data structure that represents how many times a particular memory region was accessed (regardless of which component accessed it) during a time interval. Figure 1 presents an example of one such heat map. The key idea is that an MHM is a *composition* of different activities in a certain memory region. Each activity will contribute differently in each MHM. The predictable nature of real-time embedded systems enables us to learn the patterns of usage in such MHMs especially when the system is behaving in a normal, expected fashion. We then apply techniques of image recognition algorithms [21] to transform these memory profiles to a more efficient representation so that analysis and detection become easier.

We demonstrate our techniques on a dual core processor architecture where one core performs the analysis (at run time) for anomaly detection [23]. The other core executes the operating system and applications. The architecture is embellished with certain hardware modifications to ensure that *(a)* the information can be captured in an efficient fashion *without* affecting the main flow of the system and *(b)* the information obtained by the monitoring core can be trusted. Experiments on a prototype show that our approach catches various types of anomalies effectively in an efficient manner.

Hence, our main contributions of this paper are: *(i)* novel monitoring model to characterize the memory behavior for

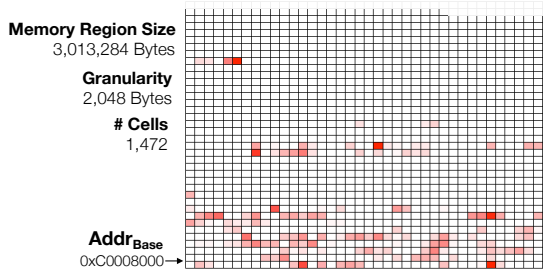

Figure 1: An example memory heat map of Linux kernel `.text` segment measured for $10ms$.

This work is supported in part by grants from NSF CNS 13-02563, NSF CNS 12-19064, NSF CNS 14-23334 and Navy N00014-12-1-0046. Man-Ki Yoon was also supported by Qualcomm Innovation Fellowship and Intel PhD Fellowship. Jaesik Choi was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT & Future Planning (MSIP) (No. NRF-2014R1A1A1002662 and No. NRF-2014M2A8A2074096). Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2744769.2744869

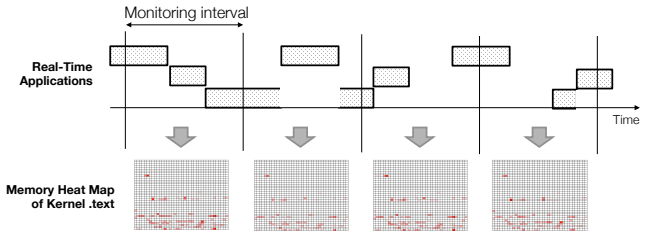

Figure 2: The overall monitoring process.

real-time embedded systems – in the form of *memory heat maps*; *(ii)* the novel combination of the MHM and image recognition algorithms to provide efficient representation and analysis of MHMs that aids in the process of detecting anomalies and *(iii)* a multicore-based architecture to perform the profiling and run-time monitoring.

To the best of our knowledge, this is the first work that uses aggregated memory behavior, especially the concept of memory heat maps, for detecting system-wide anomalies.

2. THE MEMORY HEAT MAP

A memory heat map (MHM) is defined by the following triple: the *base address*, $\mathsf{Addr}_{\mathsf{Base}}$, the *size*, S and the *granularity*, δ. These parameters determine where and at what detail we wish to monitor the memory behavior of the system. Figure 1 shows an MHM example that was profiled from an embedded Linux kernel's `.text` segment for an interval of $10ms$. **Note:** the 2-D plots of MHMs presented throughout the paper are for illustrative purposes only. An MHM is in reality a vector like the actual memory space is.

A memory region is divided into *cells*, each with a size, δ. Each cell counts the number of accesses to a region of size δ for a specified time interval. One can even consider it to be the *temperature* of each cell. The temperature of each cell, on its own, may not reveal useful information; due to variations caused by numerous factors. However, the state of the entire map may reveal important system activities.

An MHM represents a composition of memory accesses from a variety of system activities due to applications and OS. Thus, an MHM can be represented by a weighted combination of the *primary activities*; where the weights represent their contributions to the MHM. A good analogy is that of a Fourier analysis. The key ideas are that *(i)* normal memory behaviors can be grouped into a finite number of sets according to the weights of primary activities and *(ii)* abnormal behavior can be detected by just looking at these weights. The creation and use of MHMs can be quite efficient since it is a vector of numerical counts. Also, MHMs depend only on the size of the memory region that we observe and not on the complexity of the OS and other applications.

Monitoring Kernel Memory Space: In this paper, we focus on monitoring the memory space for *the operating system kernels*. Observations of the kernel memory space provide a good indicator of system-wide behavior since every application has to use kernel services (*e.g.,* system calls) for its operations. From the kernel memory behavior, we can detect certain types of anomalies *e.g.,* unexpected application launch/kill or even suspicious use of kernel services. Furthermore, the hardware design becomes much simpler when compared to monitoring user-level processes. This is because *(i)* the (base) kernel's location in the memory space is fixed and well known and *(ii)* it is contiguous in both the virtual and physical memory spaces (the base kernel's `.text` segment is in the logical address space). Hence, we do not need a complex hardware architecture to deal with the address translation and also memory paging.

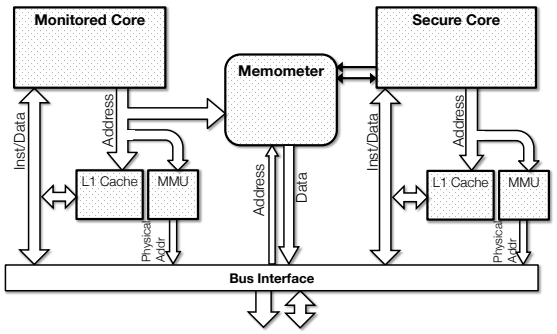

Figure 3: The secure core architecture for memory-behavior monitoring using Memometer.

Overall Process: As shown in Figure 2, our anomaly detection framework periodically checks the MHM of the `.text` region. At each interval, one MHM is created by an on-chip hardware module. The anomaly detector analyzes the MHM at the end of the interval. At that point, we calculate the likelihood of this MHM being part of the normal executions.

Assumptions: We make the following assumptions without loss of generality: *(i)* the system runs a set of real-time applications that execute in a periodic fashion, *(ii)* most of the possible execution contexts can be profiled. This can be justified by the fact that real-time embedded applications have a limited set of execution modes and input data fall within fairly narrow ranges, *(iii)* the system is in its normal (trustworthy) state while being profiled; also, the profiling *is done prior to system deployment*, and *(iv)* we consider certain types of anomalies that make changes in the memory regions that are being monitored; our detection mechanism cannot detect anomalies that access memory segments outside the region under monitoring.

3. MONITORING MEMORY HEAT MAPS

The SecureCore architecture [23] provides the means for a trusted on-chip entity, *viz.,* a *secure core*, to monitor the runtime behavior of another component, the *monitored core*. We adopt and modify the SecureCore architecture to observe the memory behavior of the monitored core through an on-chip module *Memometer*. Figure 3 shows this new architecture. The Memometer continuously snoops upon the memory requests sent from the monitored core. Using this information, the Memometer periodically creates heat maps, which are then analyzed by the secure core to determine if the behavior of the monitored core is normal or abnormal.

3.1 Memometer

The actual implementation of the Memometer depends on the specific processor architecture especially the memory subsystem. In this paper, the Memometer snoops on the *address line* between the monitored core and $L1$ cache because otherwise we would lose memory access information due to cache hit. In addition, we monitor the virtual addresses. In fact, we could monitor physical addresses if the target monitoring region has a linear mapping from virtual to physical address, *e.g.,* kernel logical address. Monitoring the physical addresses is in fact desirable since otherwise, for example, an attacker could potentially execute a malicious code by modifying the memory mappings while making MHMs look normal.

Figure 4 shows the internal structure of the Memometer; we now elaborate on each of the components.

Memometer Controller: The secure core sets the monitoring parameters for the Memometer through control registers. The parameters are *(a)* the base address of the target

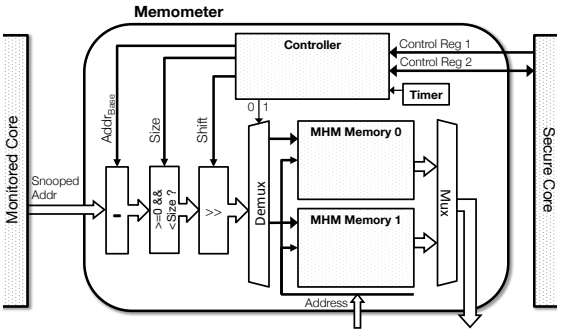

Figure 4: The internal structure of Memometer.

monitoring region; *(b)* the size of the region; *(c)* the granularity (a power of 2) and *(d)* the monitoring interval. Parameters *(a)* – *(c)* are used to filter a snooped address and to calculate the target cell location (explained below). One MHM is created during each monitoring interval, for example, 10 ms. At each interval boundary, the controller informs the secure core of the creation of an MHM and of where it can be retrieved.

Address Filtering and Target Cell Calculation: Let Addr^* be the address that is being accessed by the monitored core. Then, the following steps calculate the target cell index in the current MHM: *(i)* calculate the offset, *i.e.,* $\mathsf{offset} = \mathsf{Addr}^* - \mathsf{Addr}_{\mathsf{Base}}$. *(ii)* Check if it is within the target region, that is, $0 \leq \mathsf{offset} < S$ where S is the region size. The process stops if this is false. *(iii)* Logical right-shift offset by g bits where $g = \log_2 \delta$ and δ is the MHM granularity. The resultant is the target cell index, $\mathsf{idx} = \lfloor \frac{\mathsf{offset}}{2^g} \rfloor = \mathsf{offset} >> g$. The resulting idx is then used to increment the count of the target cell. Note that the MHM memory size determines the maximum number of cells an MHM can have and not necessarily the maximum size of the target memory region. The latter can be determined by the granularity parameter.

MHM Double Buffering: The Memometer includes a *fast on-chip memory* for MHM storage that can only be accessed by the secure core. For uninterrupted monitoring, the Memometer should be able to continue monitoring the memory accesses of the monitored core while a recently completed MHM is being analyzed by the secure core. We achieve this by use of a *double buffering mechanism.* For this purpose, the Memometer has two identical on-chip memory units.

The timing diagram above demonstrates how this double buffering mechanism works. At any time instant, a cell count update is carried out on what we call the *active* on-chip memory unit (say, '0'). Then, at a monitoring interval boundary, say, between the i^{th} and the $(i+1)^{th}$ intervals the second on-chip memory unit ('1') is tagged as being the active one and starts storing the $(i+1)^{th}$ MHM. At the same time, the secure core starts analyzing the i^{th} MHM residing on the first on-chip memory unit ('0'). Once the secure core is done with the analysis, the old MHM is reset.

4. LEARNING MEMORY HEAT MAPS

In this section, we show *(i)* how to learn the normal memory heat maps of untainted systems in an efficient, accurate manner and *(ii)* how to detect anomalies using such profiles.

4.1 Definitions and Overall Learning Process

Let $\mathcal{M} = \{\boldsymbol{M}_1, \boldsymbol{M}_2, \ldots, \boldsymbol{M}_N\}$ be the (training) set of memory heat maps we obtained during normal system executions. Each MHM is represented as $\boldsymbol{M}_n = [m_{n,1}, m_{n,2}, \ldots, m_{n,L}]^T$, where L is the number of cells and $m_{n,k}$ is a non-negative integer that represents the number of memory accesses to the k^{th} cell, $[\mathsf{Addr}_{\mathsf{Base}} + (k-1)\delta, \mathsf{Addr}_{\mathsf{Base}} + k\delta)$.

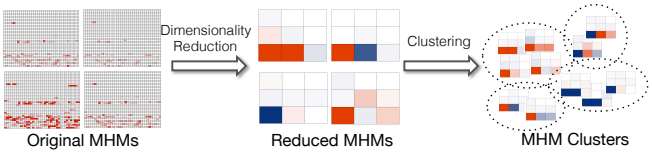

Figure 5: Learning normal MHM patterns.

When a new MHM is presented for classification, it is computationally prohibitive to calculate the similarity against every known MHM in the training set especially if N and L are high. Hence, it is desirable to find *patterns* in the normal MHMs and then calculate the statistical similarity for the new MHM. This reduces the problem into a more tractable scope. The challenges are *(a)* how to handle high dimensionality of MHMs and *(b)* how to find *representative* MHM patterns for efficient classification. Hence, we use a dimensionality reduction method to make the anomaly detection problem more tractable.

4.2 Eigenmemory

Memory heat maps are represented in a high dimensional space especially if we monitor a large memory region at a fine granularity. However, many cells of an MHM are in fact correlated each other. In these cases, given a large set of MHMs, we can represent each MHM by a sum of a small number of components. For purposes of compressing the information contained in an MHM, we use a dimensionality reduction/ feature extraction method, the Principle Component Analysis (PCA) [14], which has widespread uses in image analysis. The PCA transforms data with high dimensionality (in L-dimensions) into low-dimensional features (in L'-dimensions; $L' \ll L$) which of those are called *principal components.* These principal components can compactly represent the original data when many features/dimensions are correlated each other. In the context of image recognition, this is equivalent to the process of extracting a set of basic images, called *eigenfaces* [21]. In our context, the primary activities of the target memory region are mapped to what we call *eigenmemory.*

The following steps transform a training set $\mathcal{M}$ into $\mathcal{M}' = \{\boldsymbol{M}'_1, \boldsymbol{M}'_2, \ldots, \boldsymbol{M}'_N\}$, where each $\boldsymbol{M}'_n$ is an L'-dimensional vector and $L' \ll L$: *(i)* calculate the empirical mean MHM of the training set, $\boldsymbol{\Psi} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{M}_n$, *(ii)* obtain the mean-shifted MHM, $\boldsymbol{\Phi_n} = \boldsymbol{M}_n - \boldsymbol{\Psi}$ for all n, *(iii)* construct the empirical covariance matrix, $\boldsymbol{C} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{\Phi}_n \boldsymbol{\Phi}_n^T = \boldsymbol{A}\boldsymbol{A}^T$, where $\boldsymbol{A} = [\boldsymbol{\Phi}_1 \boldsymbol{\Phi}_2 \cdots \boldsymbol{\Phi}_N]$ is an L by N matrix and the size of $\boldsymbol{C}$ is L by L. Then, find the eigenvectors of $\boldsymbol{C}$ by the Singular Value Decomposition [10]. The extracted eigenvectors are the *eigenmemories* which of those represent the principal components of the MHMs in the training set. Then, *(iv)* order the eigenmemories according to their corresponding eigenvalues in decreasing order. Then, pick the L' best eigenmemories, $\boldsymbol{u} = [\boldsymbol{u}_1 \boldsymbol{u}_2 \cdots \boldsymbol{u}_{L'}]$, with largest eigenvalues. *(v)* Finally, transform each MHM $\boldsymbol{M}_n$ into $\boldsymbol{M}'_n$ by projecting the mean-shifted MHM $\boldsymbol{\Phi}_n$ onto the (L'-dimensional) eigenmemory space, *i.e.,*

$$\boldsymbol{M}'_n = \boldsymbol{u}^T \boldsymbol{\Phi}_n = [w_{n,1}, w_{n,2}, \ldots, w_{n,L'}]^T. \quad (1)$$

It is important to note that the $w_{n,i}$ values, also called *weights*, represent the contribution of the eigenmemory $\boldsymbol{u}_i$ in representing the original (mean-shifted) MHM, $\boldsymbol{\Phi}_n$ of $\boldsymbol{M}_n$. We can view $\boldsymbol{\Phi}_n$ being approximated using a linear combi-

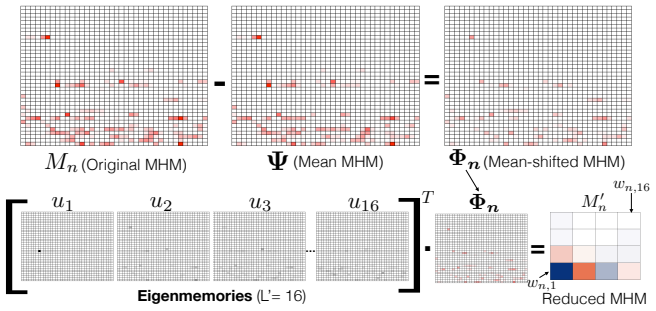

Figure 6: An example of the dimensionality reduction from M_n to M'_n using 16 eigenmemories.

nation of eigenmemories, *i.e.,* $\mathbf{\Phi}_n \approx \sum_{k=1}^{L'} w_{n,k}\boldsymbol{u}_k$. Hence, the more eigenmemories we use, the more accurate the approximation will be. When we use L eigenmemories, we can exactly represent the original input MHMs. Therefore, the best set of L' eigenmemories is one that will retain the best approximation for the original MHMs with regard to the principal components.

Figure 6 shows an example where the above process is applied. $\boldsymbol{M}_n$ is an (original) MHM of length $L = 1472$. In this example, we chose the best 16 eigenmemories. Each eigenmemory represents a primary activity; in this case an activity that touches upon the Linux kernel's `.text` segment. Here, u_1 is the most significant activity, u_2 is the next significant, and so on. Then, the resulting $\boldsymbol{M}'_n$ represents the contribution of each primary activity from the original memory heat map $\boldsymbol{M}_n$. Thus, different MHMs in the original space can be represented by different combinations of the contributions.

These eigenmemories are stored in the secure core and used to transform every newly obtained MHM $\boldsymbol{M}$ into $\boldsymbol{M}'$ (after appropriate mean-shifting) using Eq. (1).

4.3 Finding MHM Patterns

With the reduced MHMs, $\mathcal{M}' = \{\boldsymbol{M}'_1, \boldsymbol{M}'_2, \ldots, \boldsymbol{M}'_N\}$, we now find a small set of *representative* patterns that are significant enough to cover most of the normal MHMs. When a test sample $\boldsymbol{M}$ is provided we check if it is statistically similar to one of them. This is a cluster analysis. In this paper, we use the *Gaussian Mixture Models* (GMMs) for this purpose. GMMs have been widely used in image/signal processing including image clustering, segmentation, retrieval, *etc.* [18, 3] due to its ability to approximate various probability distributions and its computational efficiency. **Note:** in what follows, we will use $\mathcal{M}$, $\boldsymbol{M}$ and L instead of $\mathcal{M}'$, $\boldsymbol{M}'$ and L' for notational convenience. Also, we will use the term memory heat maps to denote the ones in the reduced dimensional space.

In GMMs, the *probability density* of a memory heat map is represented as a weighted sum of J multivariate Gaussian,

$$\Pr(\boldsymbol{M}; \text{GMM parameters}) = \sum_{j=1}^{J} \lambda_j f(\boldsymbol{M}|\boldsymbol{\mu_j}, \boldsymbol{\Sigma_j}), \qquad (2)$$

where λ_j is a mixing parameter ($\sum_{j=1}^{J} \lambda_j = 1$ and $0 \leq \lambda_j \leq 1$) and represents the prior probability that MHMs have been generated from the j^{th} Gaussian probability density, $f(\boldsymbol{M}|\boldsymbol{\mu_j}, \boldsymbol{\Sigma_j}) = \sqrt{(2\pi)^L |\boldsymbol{\Sigma}_j|}^{-1} \exp\{-\frac{1}{2}(\boldsymbol{M} - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j (\boldsymbol{M} - \boldsymbol{\mu}_j)\}$, where $\boldsymbol{\mu}_j$ and $\boldsymbol{\Sigma_j}$ are the mean vector and the covariance matrix.

By modeling the normal memory heat maps as a GMM we treat them as if they have been generated from a set of significant patterns, each of which is modeled as a Gaussian distribution (component). This is a valid model since if the system shows deterministic memory behavior, it can generate only a limited number of patterns; each MHM is then a result of small variations from one or more of these patterns. Intuitively speaking, the MHMs generated from the same basis pattern (a multivariate Gaussian) have similar weights for each eigenmemory (primary activity). Anomalies therefore inherently result in low likelihood because some of their components have not been seen in the normal memory behaviors.

For the estimation of $\boldsymbol{\mu}_j$, $\boldsymbol{\Sigma_j}$ and λ_j, we use the expectation-maximization (EM) algorithm [6]. However, it requires that the number of Gaussian densities, J, must be known. Since these techniques are out of the scope of this work, we employ the standard EM algorithm with a manually chosen J.[1]

In summary, given J and a training set $\mathcal{M}$ (in the reduced-dimensional space), *(i)* we obtain the parameters by applying the EM algorithm to $\mathcal{M}$. *(ii)* When a test MHM is presented, we calculate its probability density using Eq. (2). If it is below a threshold θ, we consider it to be anomalous.

5. EVALUATION

5.1 Prototype Implementation

We implemented a prototype of the memory heat map monitoring mechanism on the Simics full system simulation platform [16] that allows microarchitectural modifications. We used an ARM Cortex-A9 processor that consists of two cores. Each core runs at 1000 MHz and has $L1$ instruction and data caches each of size 32 KB. The cores share a unified $L2$ cache of size 512 KB. The main memory is 512 MB.

The Memometer is implemented as an on-chip hardware module in Simics as shown in Figure 3. The Memometer monitors instruction fetches by snooping on the address bus. The Memometer has two fast, on-chip, memories (Figure 4) each of size 8 KB. Hence, it can support an MHM of at most about $2,000$ cells, each of which counts up to 2^{32}. The memories can be *only read by the secure core.* The monitored core runs embedded Linux kernel 3.4. The Memometer is configured to monitor the kernel's `.text` segment mapped between `0xC0008000` and `0xC02E7AA4` (about $2,943$ KB).

The following MiBench [12] applications run on the monitored core where various kernel threads are running as well:[2]

	Exec. Time	Period	Category
`FFT`	2 ms	10 ms	telecomm
`bitcount`	3 ms	20 ms	automotive
`basicmath`	9 ms	50 ms	automotive
`sha`	25 ms	100 ms	security

A longer *hyper-period* (*i.e.,* the least common multiple of periods) would require a more number of training samples, eigenmemories, and/or GMM components. Due to the space limitation, we leave for future work to evaluate the number of proper training samples, eigenmemories, and GMM components for different settings of application periods.

5.2 Training

To obtain a training set, we ran the system and collected 10 sets of normal MHMs each of which spans 3 seconds. We set the monitoring interval to 10 ms and the granularity, δ, to 2 KB, both of which are arbitrarily chosen. These resulted in a grand total of $3,000$ MHMs each of which has $1,472$ cells.

We then applied the learning method (Section 4.2) on the training set to transform it into a low-dimensional space. We used 9 eigenmemories, since they could account for more than 99.99% of the variances in the original training set. With

[1] Figueiredo *et al.* [8] present methods to deal with these problems.

[2] The applications were not picked for any specific reason other than the fact that they are representative embedded benchmarks. The execution times were measured on Simics. The periods are manually assigned based on the execution times and the system load (78%).

these reduced MHMs, we learn the GMM parameters, $\{\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j, \lambda_j | j = 1, 2, \ldots, J\}$, using the EM algorithm [6]. For the number of Gaussian densities, we arbitrarily chose $J = 5$. Due to the local optimality of EM, we ran the algorithm 10 times and picked the one that resulted in the highest log-likelihood of the training data, *i.e.,* $\sum_{i=1}^{N} \log \Pr(\boldsymbol{M}_i)$. Again, one can apply a deterministic learning method [8].

To find a proper threshold θ for legitimacy tests, we collected another set of normal MHMs. Let $\mathcal{P}$ be the probability densities of this new set calculated by Eq. (2). Then, we set θ to the p-quantile of $\mathcal{P}$ where p can be 0.5%, 1%, and so on. This means the *expected* false positive rate is p. As the θ increases the false positive rate would also increase while we would more likely detect abnormal MHMs. Hereafter, we denote the threshold corresponding to p-quantile as θ_p.

5.3 Anomaly Detection

To demonstrate our method's ability to detect a broad range of anomalies, we tested the following three scenarios:

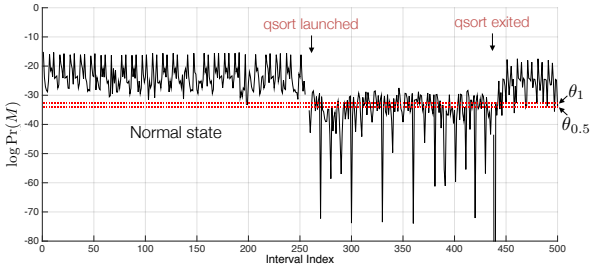

Figure 7: The log probability density of MHMs when `qsort` is launched and exited.

1. Application Addition/Deletion: While the MiBench benchmark applications mentioned above are running, we launched another application, `qsort` (exec time: $6ms$, period: $30ms$). Figure 7 shows the log probability density of the MHMs monitored over 500 intervals. The two horizontal lines show $\theta_{0.5}$ and θ_1. Until the 250^{th} interval, our anomaly detector determined that 0 and 2 MHMs are abnormal according to $\theta_{0.5}$ and θ_1, respectively; these values are the false positive rates of 0% and 0.8%, respectively.

The `qsort` application was launched some moments after the 250^{th} interval. The figure shows that the probability densities drop immediately and stays low afterward. In this particular situation, the abnormality is due to the use of kernel facilities to launch a process. Notice that even after `qsort` is launched some of the MHMs look normal according to threshold θ_1. This is valid since during those intervals `qsort` does not execute. Nevertheless, they are low compared to normal because the timings of the other tasks are affected by `qsort`.

2. Shellcode Execution: A shellcode is a small piece of code that can be executed by exploiting certain vulnerabilities, *e.g.,* buffer overflows or format string vulnerabilites.

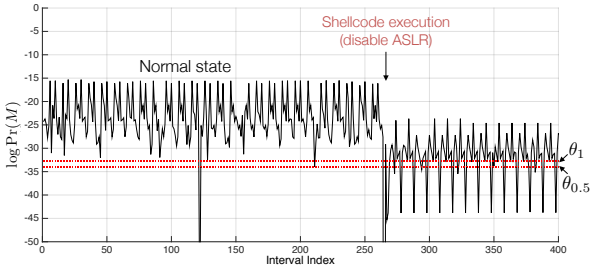

Figure 8: The log probability density when a shellcode disables ASLR.

In our evaluation, we injected a shellcode that disables the address space layout randomization (ASLR) mechanism in Linux/ARM [1] into the `bitcount` application. The result is shown in Figure 8. The shellcode executed some moments after the 250^{th} interval. This shellcode was easily detectable because the shellcode eventually kills its original host, *i.e.,* `bitcount`. In fact, most shellcodes can be detected because they typically kill the host process by spawning a shell.

3. Kernel Rootkit: Most existing kernel rootkits that are publicly available do not work on our Linux kernel version (3.4) for a variety of reasons. Thus, we created a simple loadable kernel module (LKM) that resembles the most representative type of such rootkits, *i.e.,* ones that perform system call hijacking [19]. Our LKM redirects the `read` system call by modifying the corresponding entry in the system call table. The new, malicious, `read` just reads the buffer that is returned by the original handler and nothing else. LKMs in Linux are loaded onto the module memory space that is outside our target region (*i.e.,* `.text`). Thus, the execution of the new `read` handler does not change the MHMs.

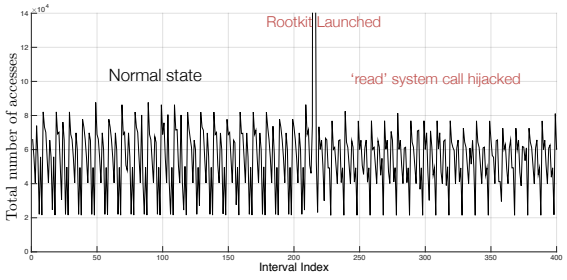

Figure 9: The memory traffic volume when the `read` system call is hijacked by a rootkit.

Figure 9 shows the memory traffic volume of the monitored region. The moment when the rootkit is being loaded is distinguishable as expected. However, after the launch the traffic does not show abnormalities in terms of the volume. This is because the rootkit still calls the original `read` handler which resides in the region being monitored.

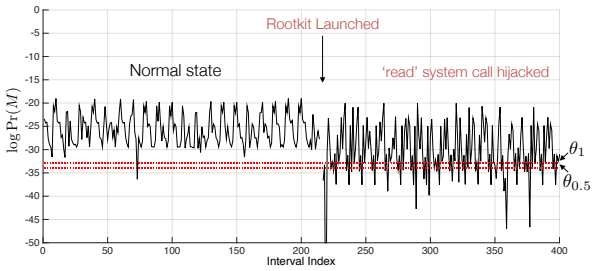

Figure 10: The log probability density when a rootkit hijacks `read` system calls.

Nevertheless, even such stealthy activities (reading the buffer) showed somewhat low probability densities, though not always statistically distinguishable, as shown in Figure 10. Given that many MHMs are normal and the abnormal ones' appear synchronized with `sha` (whose period is 100 ms), it is likely that the delays due to `read` system call hijacking have resulted in timing changes to `sha`'s execution (which uses many `read` system calls) and, as a result, its contributions to the MHMs.

5.4 Analysis Time

We measured the time to perform the analysis on a newly observed MHM, *i.e.,* how long it takes to decide whether it is normal or not. For the parameters used in the evaluations above ($L = 1472$, $L' = 9$, $J = 5$), it took $358\mu s$, on average, on Simics. This time is very short compared to the interval (10 ms). For a coarse cell granularity of 8 KB (results in $L = 368$), it took $100\mu s$ on average. For a smaller number

of eigenmemories ($L' = 5$), the average time is $216\mu s$ since the information is less precise. Each number is based on $1,000$ samples of MHMs. The results show the computational efficiency of our method. **Note:** These are the times spent on the *secure core*. Our method does not impose any overheads on the main system execution (*i.e.,* the monitored core).

5.5 Limitation

For a processor with more than two cores, the proposed architecture requires a hardware change. For AMP (Asymmetric Multiprocessing) architectures on which multiple OSes run, the Memometer should be replicated for each OS instance. However, for SMP (Symmetric Multiprocessing) architectures on which a single OS runs, the Memometer would need only one set of MHM memories (Figure 4) as there is only one OS running on the system. Nevertheless, the address snoop and filtering logic needs to be replicated for each core since the kernel can run at any core at any given time.

One solution to scale the architecture well could be placing the Memometer at a lower part of the memory subsystem such as the shared cache or bus. In this case, we would need only a single Memometer, which much simplifies the architecture. However, we could lose parts of memory access information due to cache hits. Nevertheless, we believe that the accuracy drop would not be significant because of the predictable nature of real-time application executions.

Some systems may exhibit highly unpredictable, but yet legitimate, memory usage caused by, for example, network activities or user interactions. In these cases, our current model may alarm many false positives. To deal with such problems, we plan to build a robust classification algorithm by extracting local features from MHMs in an unsupervised manner as in Deep Learning [13].

6. RELATED WORK

To the best of our knowledge, this is the first work that uses aggregated memory behavior for detecting system anomalies especially the concept of memory heat maps. Hence, we instead present some lines of work that can be adopted to memory behavior monitoring. Barford *et al.* [2] detect abnormal network traffic in terms of volume by using wavelet analysis. Gu *et al.* [11] proposed a method to monitor packet class distributions. One possible way to apply it to memory behavior monitoring is to look at the distributions of memory access types – instruction/data or load/store, *etc.* There exists a line of work on kernel integrity checking such as [5] and virtual machine introspection for intrusion detection [9].

There also exists other work in which a multicore processor is employed as a security measure. Shi *et al.* [20] proposed an architecture in which a monitoring core verifies functional behavior (such as function call/return) of applications on monitored core through buffering of logs on an on-chip memory. Chen *et al.* [4] also employ a hardware logging method that captures program counter values, input/output operands and memory access addresses of instructions that the monitored application executes. There have also been coprocessor-based approaches [15, 7]. Mohan *et al.* [17] use the timing properties of real-time applications as a side-channel monitored by an FPGA-based trusted hardware.

7. CONCLUSION

We showed that the use of memory heat maps can be effective in detecting anomalous system-wide behavior of real-time embedded systems. We demonstrated a novel use of image recognition algorithm and a multicore-based architecture to make the process of detecting anomalous behavior more efficient. Our evaluation using a prototype showed that we are able to detect a wide variety of attacks.

We plan to demonstrate these methods on a real platform that includes a real-time operating system (RTOS). RTOSes have a *more* deterministic memory usage; hence our techniques will be even more effective when applied to such a context. We also plan to extend the architecture to support more than two cores and evaluate the required hardware changes, and to explore Deep Learning-based [13] technique to deal with more complex embedded systems.

8. REFERENCES

[1] http://shell-storm.org/shellcode/files/shellcode-669.php.

[2] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *SIGCOMM Workshop on Internet Measurment*, 2002.

[3] C. Beecks, A. M. Ivanescu, S. Kirchhoff, and T. Seidl. Modeling image similarity by gaussian mixture models and the signature quadratic form distance. In *IEEE International Conference on Computer Vision*, 2011.

[4] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Workshop on architectural and system support for improving software dependability*, 2006.

[5] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy*, 2014.

[6] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[7] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *IEEE/ACM International Symposium on Microarchitecture*, 2010.

[8] M. A. T. Figueiredo and A. Jain. Unsupervised learning of finite mixture models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(3):381–396, 2002.

[9] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium*, 2003.

[10] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970.

[11] Y. Gu, A. McCallum, and D. Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *SIGCOMM Conference on Internet Measurement*, 2005.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characterization*, 2001.

[13] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, 2006.

[14] I. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics, 2002.

[15] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[17] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM Conference on High Confidence Networked Systems*, 2013.

[18] H. Permuter, J. Francos, and I. Jermyn. A study of gaussian mixture models of color and texture features for image classification and segmentation. *Pattern Recognition*, 39(4):695 – 706, 2006.

[19] plaguez. Weakening the linux kernel. *Phrack*, 8(52), 1998.

[20] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *International Symposium on Computer Architecture*, 2006.

[21] M. Turk and A. Pentland. Face recognition using eigenfaces. In *IEEE Coference on Computer Vision and Pattern Recognition*, 1991.

[22] M.-K. Yoon and G. Ciocarlie. Communication pattern monitoring: Improving the utility of anomaly detection for industrial control systems. In *NDSS Workshop on Security of Emerging Networking Technologies*, 2014.

[23] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.