Data mining the memory access stream to detect anomalous application behavior

Francis B. Moreira
Informatics Institute – UFRGS
fbmoreira@inf.ufrgs.br

Matthias Diener
Informatics Institute – UFRGS
mdiener@inf.ufrgs.br

Philippe O. A. Navaux
Informatics Institute – UFRGS
navaux@inf.ufrgs.br

Israel Koren
Department of Electrical & Computer Engineering –
University of Massachusetts
koren@ece.umass.edu

ABSTRACT

Detecting anomalous application executions is a challenging problem, due to the diversity of anomalies that can occur, such as programming bugs, silent data corruption, or even malicious code corruption. Moreover, the similarity to a regular execution that can occur in these cases, especially in silent data corruption, makes distinction from normal executions difficult. In this paper, we develop a mechanism that can detect such anomalous executions based on changes in the memory access pattern of an application. We analyze memory patterns using a two-level machine learning approach. First, we classify the behavior of different memory access periods within applications using Gaussian mixtures. Then, based on these classifications, we construct matrix representations of Markov chains to obtain information regarding the temporal behavior of these memory accesses. Based on metrics of matrix similarity, we can classify whether the application behaves as expected or anomalously. Using gradient boosting on the metrics of matrix similarity, our technique correctly classifies more than 85% of all executions, identifying instances of the same application and different applications. We can also detect a range of faulty executions caused by benign or malicious permanent bit flips in the code section.

CCS CONCEPTS

• **Software and its engineering → Main memory**; • **Dependable and fault-tolerant systems and networks → Reliability**; • **Information systems → Clustering**;

KEYWORDS

Memory access patterns, machine learning, single bit flips

1 INTRODUCTION

Anomalous behavior in program executions encompasses several undesirable phenomena, such as programming bugs, exploits caused by malicious corruption of program code, and silent data corruption caused by single event upsets (SEUs) [14], e.g., bit flips caused by radiation. Common solutions to these problems include tracking system calls to limit the potential to do harm to the operating system [16], and adding redundancy in the code or the hardware to avoid silent data corruption [3, 15, 21]. In reading from and writing to the memory, many programs produce a characteristic pattern [5], which is likely to be disrupted in the case of an anomaly. Therefore, analyzing and comparing the memory access pattern of an application can potentially be used to detect anomalous executions.

Memory access patterns have been extensively studied in previous work. The possibilities of using them for performance improvements are varied, such as prefetching in hardware [13], Microsoft's superfetch [22], predicting disk aggregations [2], among others. However, none of these techniques have been used to detect anomalous application behavior. The advantage of doing so is that, by using an external observer, it is not subject to the same circumstances as the processor, which could be compromised. Therefore, such a mechanism could act effectively as a watchdog for the processor.

It is not possible to find an optimal solution to the problem of detecting anomalous behavior during program execution, as it can be reduced to the halting problem [23]. This work analyzes under which circumstances it is possible to distinguish a normal execution from a faulty one based only on the memory access patterns of the application. To this end, we have analyzed different data sampling methods, memory behavior classification methods, and metrics based on these classifications. Based on the values of these metrics obtained from several executions, we use gradient boosting [4] to determine whether the program behaved normally or not. We present a mechanism that uses the ensemble classifier obtained with gradient boosting to decide whether a new execution of a program is correct or anomalous.

The main contributions of this paper are the following:
Data pattern and metric analysis: Through the analysis of several different memory aggregation patterns, we are able to create a

fingerprint of correct execution and derive metrics which can express a program variation by modeling its execution with a Markov chain.

Highly accurate characterization: Using our mechanism with gradient boosting, we correctly classify 88% of all *miBench* [7] executions, with an average false-positive frequency of less than 13%.

The rest of this paper is organized as follows. The next section provides an overview of the main issues regarding comparison of memory access patterns. In Section 3 we describe our memory access classification and the mechanism to detect anomalous executions. Our experimental methodology is described in Section 4. In Section 5, we analyze and discuss our results. Related work is presented in Section 6. Our conclusions are summarized in Section 7.

2 MOTIVATION

In this section, we describe the problems we focus on. We first provide a detailed description of the target problems: detection of silent data corruption and benchmark identification. We then describe how memory access patterns can be analyzed to address these problems.

2.1 Background and Overview

A multitude of scenarios can occur when a computer program is executed. The common and expected scenario is that the program executes normally and generates the correct output. However, execution can terminate incorrectly for several reasons. Due to wrong inputs or bugs in the software, the application might just crash. Furthermore, malicious attacks may cause the program to behave in an entirely unexpected and even malicious way, such as installing a root kit and allowing undesired, privileged access to remote users. Another incorrect and unexpected behavior may be the result of a Single Event Upset (SEU) due to a particle hit or faulty hardware, which can either have no effect, crash the program, or generate wrong output, that is, silent data corruption [8]. Moreover, silent data corruption can also be used for malicious attacks [20].

Therefore, we are interested in detecting silent data corruptions and, more generally, program behavior that differs from the expected. As the processor is generally unable to detect abnormal behavior of programs and might be compromised itself, we choose to detect these by externally observing memory references. By doing so, we allow other cores to determine whether the behavior is normal, and may in the future allow off-chip co-processors (such as a processor in a Hybrid Memory Cube [17]) to monitor and check the correctness of a program's execution.

Since memory references only allow indirect inferences regarding the program behavior, it is necessary to classify observations of memory accesses into patterns, for which several parameters need to be defined. Such parameters are the number of accesses to be aggregated, the classification method, the access types considered, and what properties of memory accesses are used to compose an observation. For prefetchers, few memory accesses are sufficient to predict a pattern for a given program counter. However, when observing an entire program behavior, there are far too many patterns in the entire sequence of memory accesses to be tracked in practice. Thus we aggregate memory accesses into memory observation periods with a fixed number of accesses, in order to obtain an overall

picture of the execution patterns and filter out small memory access deviations that may be the result of out-of-order execution, multi-threading, and other issues. We show two empirically chosen granularities out of a design space exploration which ranged from aggregations of 128 accesses to aggregations of 32768 accesses; a coarse granularity of 16384 memory accesses per observation, and a fine granularity of 1024 memory accesses per observation, in order to be able to capture small deviations such as silent data corruption. Whenever a request is issued by the processor, we simply increment the corresponding access type counter and, if the address references a page which was still not referenced in this period, we increment the corresponding number of pages counter for that access type. Once the number of accesses required to compose an aggregation is counted, we compose a memory aggregation with that information and reset the counters to begin tracking the next memory period. Superscalar request reordering has a negligible effect on our coarse-grained aggregations, as a possible reordering of requests will only result in a small change in the composition of a memory aggregation.

2.2 Problem Modeling: Composing Observations

Aggregation of memory accesses results in information loss regarding the temporal order of these accesses, and their individual characteristics. However, a period of memory accesses can be differentiated by accesses' type, for example, whether they are reads or writes, whether they are accessing instructions or data; and by the concentration of memory accesses, which we capture by observing how many different pages were touched in a given time period.

In order to be agnostic to operating system memory allocation (which might change for each execution), we do not use any direct spatial information, such as addresses, but rather indirect metrics such as the number of pages touched by each memory access type. Finally, a fixed number of memory accesses might occur within a different number of instructions. We therefore use the total number of instructions that were executed in this period to capture the proportion of memory accesses to instructions in the given observation, and to obtain an indirect metric to execution lenght of the memory aggregation, as relying on number of cycles might introduce undesirable noise due to different caching effects on the same memory aggregation at different times of execution, e.g. sharing cache with another thread that hit a memory-intensive phase.

We thus define the following features of each observation: number of reads, number of pages touched by reads, number of writes, number of pages touched by writes, number of instruction reads, and the total number of pages touched by instructions. A read stream, for instance, would concentrate memory reads in a few sequential pages. Sparse matrix calculations using unoptimized data structures would spread their accesses over several pages. Pointer-based structures would likely spread their accesses depending on operating system memory allocation.

2.3 Classification and Evaluation of Errors due to Bit Flips

During execution of a program, there is a possibility of a fault in several portions of the system. Memory caches are often protected

by error-correcting code (ECC), and can thus be considered safe [24]. Superscalar structures however can suffer from bit flips due to SEUs and can potentially change the execution of the program.

Such a faulty execution may generate one of four cases:

- First, the execution may finish correctly if the bit flip did not change anything relevant to the context, such as a register which was soon updated with another value.
- Second, the execution may finish incorrectly if the bit flip changed the execution path and led the control flow astray, generating silent data corruption and thus incorrect results. Only a few programs have built-in checksums and error checks to detect incorrect results for such cases.
- Third, the execution may not finish if the bit flip resulted in an infinite loop by changing the execution path.
- Fourth, and most likely, the execution may not finish if the bit flip caused either an illegal instruction (thus raising an illegal instruction exception as the processor hardware decode stage detects an invalid opcode), or an illegal memory access, by leading the control flow to access an instruction or data address outside of the allocated area (thus generating a segmentation fault detected by the memory management unit and operating system).

Of these four cases, we are particularly interested in the second case, where a bit flip could be generated with malicious intent to modify the control flow, or extract secret or proprietary data from the executing program. To do so, we look specifically for faulty executions that did not terminate abnormally, which means the program neither crashed nor entered an infinite loop; and had a final result, for the same input, that is different from the correct execution. This is the most interesting case of a fault, as all other faults eventually result in a crash, and are thus easier to detect.

To observe the effect of erroneous executions in the memory trace, we used the *miBench* suite [7]. The inputs were obtained from the work by Fursin et al. [5]. In order to identify such faults, we exhaustively executed each benchmark with every possible bit of the code section of each program flipped. Programs were compiled dynamically, so bits in dynamically linked libraries were not flipped. Silent data corruption faults represented on average 4% of all the altered instructions in each benchmark. Once we knew which bit flips would generate silent data corruption, we used Pin [12] to instrument the benchmarks and collect the memory traces from the regular executions and the faulty executions for a given input.

In Table 1 we show how expressive the memory accesses' difference can be between a correct and a faulty execution. It shows two executions of *quicksort* with the same input, where the first is a correct execution and the second a faulty execution. Although the number of memory accesses stays almost the same, the number of pages touched per operation changes by about 10%, showing that accesses concentrate in fewer pages. We can also see in Table 1 two executions of *dijkstra*, where the opposite happens: the faulty execution dilutes the execution in more pages, even though it has almost the same number of memory accesses as the first. In both executions, there is less than 1% difference between the number of instructions executed, which indicates the program had similar running times and executed instructions, a case in which anomalies are harder to detect.

Table 1: Metrics for a complete benchmark execution of the *quicksort* and *dijkstra* applications.

Metric	quicksort		dijkstra	
	normal	anomalous	normal	anomalous
# reads	34,163,994	37,714,422	31,009,923	30,991,253
# pages read	180,995	158,208	167,082	179,613
# writes	21,314,365	24,706,058	15,040,334	15,030,683
# pages written	67,880	58,277	73,456	85,977
# instructions	172,865,49	174,279,168	162,436,143	162,349,776
# inst. pages	121,890	125,854	64,364	64,303

3 DETECTING ANOMALOUS EXECUTIONS WITH MACHINE LEARNING

In this section, we describe the machine learning solutions applied to the detection of anomalous program executions. We first describe the application of Gaussian Mixture Models as a classification method to cluster similar behaviors in distinct indices. We also explain how using a Markov chain can generate metrics that show whether a program is behaving anomalously. We then show an example of an anomaly detection using these metrics.

3.1 Classifying the Memory Periods with a Gaussian Mixture Model

The problem of classifying the memory access pattern for a program can be divided into two problems. The first is identifying specific patterns to fit each aggregation of memory accesses. The second is identifying in which temporal order these patterns exhibit themselves.

For the first issue, we use clustering methods to cluster observations of memory access periods into similar groups. We chose to use a variation of the expectation-maximization algorithm, the Gaussian Mixture Model (GMM) [19]. This algorithm is resilient against specific structures for clusters, since it does not use the simple euclidean distance like k-means, but rather a statistical distribution of behaviors and deviations is considered for the formation of centroids.

K-means generates a hard classification, it only assigns a cluster for each data point based on proximity to a centroid, that is, an average point for several adjacent data points. This bias towards distance and concentration of points tends to make k-means clusters spherical. Gaussian mixture models, on the other hand, yield soft classifications, by assigning probabilities for each data point to belong to any given cluster. Therefore we chose a soft classifier instead of a hard classifier, reducing sensitivity of the approach as memory behaviors won't always be well-defined and might have variances resulting from side-effects, e.g. out-of-order execution, or data access variation due to structure, e.g. pointer-chasing with varying number of different pages touched for what should be described as the same behavior.

3.2 Learning the Sequential Pattern with Markov Chains

The second issue, finding the temporal order in which patterns occur, is more complex. As a single pattern of memory accesses might identify functions that do different things, it can be followed by different patterns, so there is no single order that happens in a program and can thus be used to differentiate it from other programs. Rather, we can classify a time dependent probability that a pattern is followed by any other pattern. Thus, we represent the program behavior as a Markov chain [6].

A Markov process consists of states and transitions between states. It is commonly represented as a directed graph, and we use a matrix to represent it. Each edge (transition) has a probability, which is the probability that the initial vertex (state) transitions to the end vertex. The sum of probabilities of all outgoing edges of a vertex must be 1, representing every possible transition from that state.
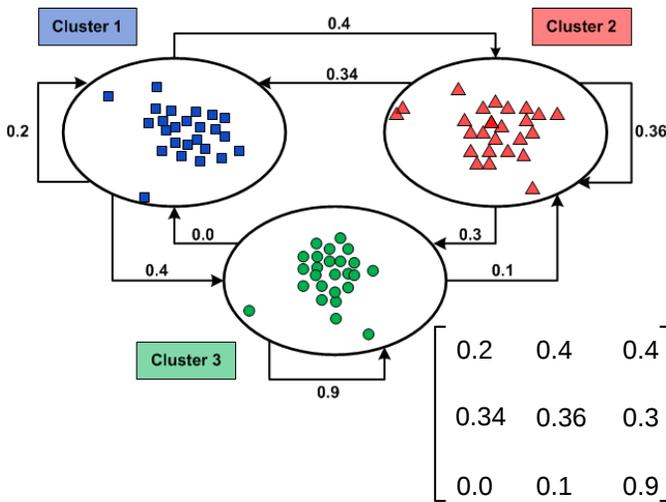
Figure 1: Matrix representation of Markov chain describing pattern of transitions between behaviors.

In Figure 1, we show an example of how we use a Markov chain to represent cluster transitions. Each one of the 3 clusters is a state (vertex) in the Markov chain. Therefore we generate a 3x3 matrix, where the **i, j** element of the matrix represents the probability that a memory behavior in cluster **i** will be followed by memory behavior in cluster **j**. In general, we first generate a Gaussian mixture model for all memory periods observed in the execution of a program A yielding **N** clusters. Then, we produce a $N \times N$ matrix **MC** consisting of the probabilities that any given memory pattern is followed by any other. We count all transitions between these clusters, by observing executions of program A with different inputs, and for each cluster i, such that $1 \leq i \geq N$, after the first cluster in temporal order, we increment MC[i-1][i] by one. We also increment a counter vector C, C[i-1], by one, indicating the number of transitions from cluster i-1 to any other cluster. After accumulating all the transitions from the execution, we divide all elements in row MC[j] by C[j], thus obtaining the probabilities of each cluster-to-cluster transition.

In this way, it is possible to obtain a general representation of a program's memory access behavior over the course of its execution.

3.3 Detecting Anomalous Behavior

Given a trained Markov chain for a specific program, if the program makes a transition from one pattern to another, and such a transition was never seen in the training process, it is reasonable to assume that this indicates anomalous behavior. Likewise, if a program commonly makes transitions from pattern X to pattern Y, but in a new execution it never performs such transitions, an anomaly might be happening. We have tested several metrics regarding the Markov chain matrix representation and selected 4 significant metrics to differentiate executions.

To confirm these intuitions and exemplify the mechanism at work, we trained a Gaussian mixture model using 5 *quicksort* executions with different inputs. More details of the methodology are provided in Section 4. In Figure 2, we show 4 metrics of the Markov chain being analyzed for regular *quicksort* executions and one execution with silent data corruption, generated by a bit flip.
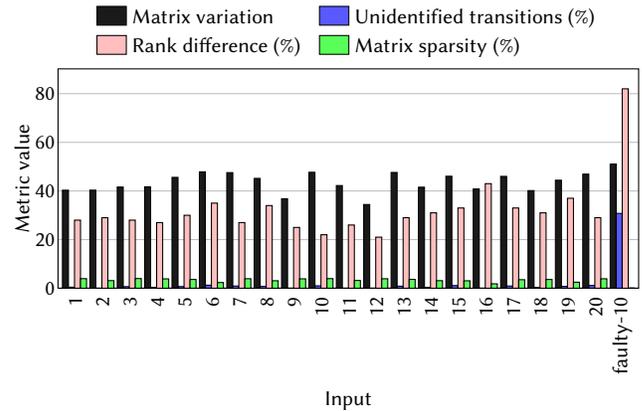
Figure 2: Four metrics to identify *quicksort* anomalous executions, with each observation composed of 16384 memory accesses and 20 different inputs.

The first metric is the total matrix variation, which is half the sum of the absolute values of the differences between the probability in each position of the trained Markov chain and the observed Markov chain. This metric tells us by how much transition probabilities differ from their expected values in the trained Markov chain. It ranges from 0 (identical matrices) to $2 \times N$, where the difference between the probabilities that compose each row sums to 2 for all N rows. As an example of maximum variation in a row, given a row with two positions where the trained Markov chain row shows values of 0 and 1, and the observed Markov chain row shows values of 1 and 0, the variation for that row will be $|0 - 1| + |1 - 0| = 2$. In Figure 2, we can observe that the high variance in *quicksort* makes it hard to tell it apart from the faulty execution.

The second metric is the ratio of unidentified transitions. We calculate this metric by keeping track of transitions that were seen in the trained Markov chain. Each transition in the observed Markov chain that had 0 probability for the trained Markov chain increments an "unidentified transition counter", and thus the unidentified

ratio is equal to the "unidentified transition counter" divided by the total number of observed transitions when generating the observed Markov chain. As with any ratio, this value can range from 0%, when no unidentified transitions were observed, to 100%, where all transitions of the observed execution had zero probability in the trained Markov chain. This metric can indicate when a program is exhibiting a behavior unseen by the trained Markov chain. In Figure 2, we observe low values for all executions of *quicksort*, whereas the faulty execution of input 10 shows a high value, which means it is exhibiting a different sequence of behaviors than those expected for *quicksort*.

The third metric is the matrix rank difference. We calculate the rank of the trained Markov chain and the rank of the observed Markov chain and subtract one from the other, then return the absolute value. The fourth metric is the matrix sparsity of the observed Markov chain. It is calculated as the total number of non-zero elements in the matrix divided by the total number of elements in the matrix. These two metrics are able to show when an observed Markov chain focuses on specific and general behaviors due to overfitting of the trained Markov chain's centroids to behaviors that are specific to the benchmark used for its training. A rank difference of 99% indicates that while the trained Markov chain is fully ranked, the observed Markov chain has 1% of its N rows as an independent vector, which means only one row has non-zero probabilities. As we have N rows and N columns, the rank difference ranges from 0 for two matrices with the same rank, to N-1, where one matrix has rank N and another matrix has rank 1, having a single transition with probability different from zero. This is confirmed by the matrix sparsity, indicating that the faulty execution with input 10 only exercised one general behavior, not the specific and detailed ones that overfit the *quicksort* benchmark. Matrix sparsity ranges from $1/(N \times N)$ % to 100%.

3.4 Mechanism Organization

Once we have the model fit to a composition of observations from multiple inputs to the same benchmark, we build a Markov chain matrix by observing the sequence of all component indices in which the memory periods in these training inputs were classified. We then construct another Markov chain matrix by observing the sequence of all component indices in which the memory periods of a new input were classified. This new input is the one for which we want to define whether the program is behaving normally. We do so by comparing the metrics previously mentioned and observing whether there are outliers in these metrics which could indicate anomalous behavior.

In Figure 3 we show the flowchart of our technique. The leftmost blocks describe the training stage; we first characterize the memory periods of an application for several of its inputs. We use multiple inputs here to account for the behavior variance between different inputs. Once we have the memory access patterns for all these memory periods, we generate a clustering of the memory periods using a Gaussian mixture model. Thus, we now have indices classifying memory periods into specific patterns. With these indices alone, we could already perform benchmark identification by observing the proportion of periods attributed to each index in an execution, although not quite accurately.
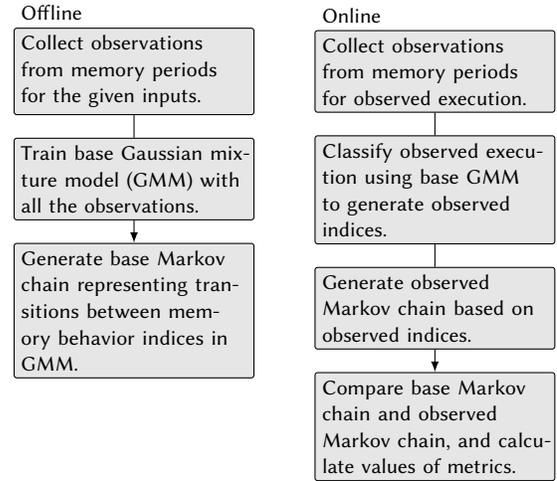
Figure 3: Flowchart of our proposed technique.

We then construct the base Markov chain, which represents the probabilities of one index (that is, a memory period behavior) being followed by another index, for all possible index pairs. We use a $N \times N$ matrix to represent the Markov chain, where N is the number of clusters in our model. Therefore, each row i of the Markov chain represents one behavior cluster, and each column k in this row corresponds to the cluster that was observed after i. The element i,k is the probability of behavior represented by index i being followed by behavior presented by k. Every time we re-execute the program later on (after the training has been completed) we build a second matrix, representing a Markov chain with the classifications obtained by running the execution to be tested for anomalies.

For illustrative purposes, in Figure 4, we compare the *quicksort* and *susan* execution metrics when each memory aggregation consists of 1024 memory accesses. The base benchmark which we assume to be executing is *quicksort*, that is, the Markov chain was trained using *quicksort* executions. We can observe that rank and matrix sparsity are distinctive metrics for different benchmarks, clearly indicating that the behavior of *susan* is anomalous when compared to the expected behavior from *quicksort*.

Since Gaussian mixture models are based on the expectation-maximization algorithm, an iterative method, whose complexity depends on convergence of the input and iteration threshold. Since we allowed at most 100 clusters, and a default threshold of 100 iterations, the time complexity of finding the components in the training phase is bound to $10000 \times n$, where n is the number of memory aggregations, i.e. the total number of memory accesses of the program divided by the granularity size. When creating the target application's Markov chain, each memory aggregation must be classified, resulting in additional n steps for each memory aggregation.

The amount of storage required by the matrices for 100 clusters is 160000 bytes when using double-precision floating point elements to represent probabilities. Since we used full covariance matrices,

the Gaussian mixture model requires *number of components × number of features × number of features*, thus requiring 28800 bytes, again assuming double-precision floating point. Therefore the total storage complexity of our technique, with the chosen parameters, is of 188000 bytes.
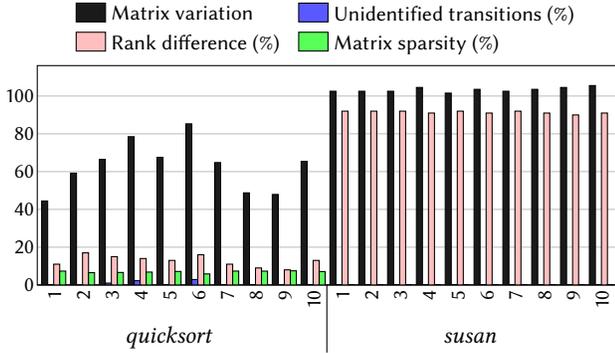
Figure 4: Four metrics that can distinguish between *quicksort* and *susan* executions, with 10 different inputs each and observations composed of 1024 memory accesses.

4 METHODOLOGY

The fundamental tool we used for this work is Pin [12] from Intel, which can trace memory accesses from applications. To perform machine learning, we used Sklearn's Gaussian mixture model implementation [18] and gradient boosting implementation [4].

We selected 7 benchmarks of *miBench* [7] for our experiments. The benchmarks selected were *quicksort*, *susan*, *patricia*, *dijkstra*, *jpeg compression*, *raw audio compression* and *gsm encoding*. We used Fursin et al.'s [5] datasets to ensure each benchmark had varied inputs which would express different behaviors. All programs were compiled using gcc, dynamic linking, and the optimization flag -O3.

The memory traces were then processed and the memory accesses were aggregated into observations constituting six features. The features are: 1) number of read accesses, 2) number of pages touched by reads accesses, 3) number of write-allocate accesses, 4) number of pages touched by write-allocate accesses, 5) number of instruction-read accesses and 6) number of pages touched by instruction accesses.

Once we had processed traces with these observations, we used Python and its numpy, math, and Sklearn libraries to construct a Gaussian mixture model [19] using a full covariance matrix. The ideal number of components should be different for every application; however, since the gaussian mixture model is resilient to overfitting, we empirically chose an arbitrary large number of components, 100, to classify the observations such that the library would not throw a memory crash exception due to large memory space used by the algorithm. In our experiments, 120 components would crash the model fitting process of the Sklearn mixture model with specific inputs.

Construction and processing of Markov chains for the metrics were performed using matrices. In the mechanism we assume a base Markov chain and an application Markov chain. The metric

values of the base Markov chain for a given application are obtained by training the gaussian mixture model with a number of correct executions with random inputs from the selected dataset. The difference between these values and the values obtained in the Markov chain generated by the application being tested are used to define whether the application's behavior is considered normal.

5 EXPERIMENTS

This section presents the experimental evaluation of our proposed mechanism. We start with the tests to identify a program, followed by the fault identification.

5.1 Program Identification

We first tested the ability of our mechanism to identify the executing program. This can allow the detection of unknown programs including malicious programs. Initially, we generated a Gaussian mixture using the concatenation of 5 different inputs for each of the 7 benchmarks. To distinguish among benchmarks, we executed this concatenation to train the GMM to generate the Markov chain metrics. The values obtained for the metrics in this Markov chain will be referenced as *base values*, and the Gaussian mixture will be referenced as *base GMM*. We then compared the same metrics when generating a Markov chain running a different benchmark under the base GMM. We were conservative in this test, only claiming a difference between the programs if any of the metrics was significantly different, using a threshold of 30% difference in the value of each metric when compared to the maximum or minimum value obtained for the training inputs in the original benchmarks. We reached a 100% coverage with 0% false positives (comparing the base metrics with executions using different inputs which were not used for training). Thereby, we can reliably identify benchmarks when we have access to the entire memory trace.

This is due to overfitting. There is a large probability that 100 centroids overfits the benchmark, and thus more centroids are being used to express small deviations in some of the parameters. However, these small deviations are characteristic of the benchmark, and deviations in other parameters will have their classification concentrated in a smaller number of centroids. Since most classifications of the memory periods of the different benchmark are being fit to few centroids of the base GMM, transitions are concentrated in few positions of the test benchmark Markov chain. The matrix sparsity (representing how many centroids were used) was able to differentiate most of the benchmarks; the remaining differences were covered by the other metrics.

5.2 Fault Identification

To observe how the memory access pattern changes for these cases, we performed a sweep over all possible bit flips in the binaries of each benchmark, generating all traces of silent errors for a single data input. Since some benchmarks had far too many bit flips that caused silent data corruption, we sampled one hundred different errors for each benchmark. In Figure 5, we show a conservative coverage of errors for each benchmark. For this coverage, we used 5 distinct metrics to differentiate faulty executions from normal executions, adding matrix trace i.e., the sum of the matrix elements along the diagonal, to the four metrics previously introduced. This

metric represents the tendency of a program to exhibit repetitive behavior. If any of the 5 metrics was above or below a certain threshold (in this case, 70% as is discussed below), i.e., if the value of the metric in the observed Markov chain was larger than *value of metric* $\times$ *1.7*, we classified the execution as faulty. Each of the metrics has a specific condition, e.g., it does not make sense to test if the rank difference metric is lower than for the training inputs, as it would mean the input being tested behaves even more closely to the expected behavior given by the base GMM.
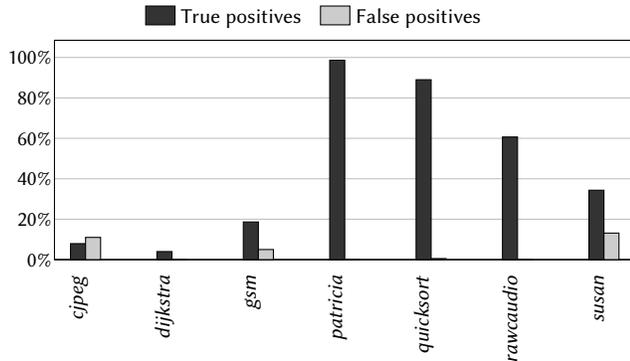
Figure 5: Error coverage and false positive rate obtained for each benchmark.

We can observe that the coverage of the technique is very dependent of the benchmark. Even with a conservative threshold we get false positives for some of *cjpeg*'s own inputs.

Since 70% is rather conservative, we also performed tests to obtain the receiver operating characteristics (ROC) [10] curve, which shows the trade-off between false positives and true positives for a range of threshold values. In Figure 6, we show that it is possible to obtain various trade-offs between true positives and false positives using different thresholds for each benchmark. In this figure, the 70% threshold is shown in the 0.3 coordinate. The chosen threshold of 0.53% means that if there is at least 47% difference for any metric, we classify the execution as anomalous. Here we obtain a better trade-off being more aggressive, as we allow a smaller deviation of any metric to classify the execution as anomalous.

Based on the results of Figure 6, we performed gradient boosting using the 5 metrics as features of an execution, each execution being one observation. We trained the gradient boosting with 70 correct executions and 70 incorrect executions, by choosing 10 correct and 10 incorrect executions of the 7 benchmarks. Our tests with the remaining inputs generate the coverage shown in Figure 7, which shows that with the correct machine learning model for the five metrics it is possible to achieve a higher fault coverage based on memory accesses. We empirically reached these results after using a gradient boosting with 200 estimators, 0.08 learning rate, 0.85 subsampling for each individual learner, and a max depth of 7 for the regression estimators. All other parameters' values were Sklearn's library defaults. Here we correctly classify 88.46% of all incorrect executions being tested as anomalous, while obtaining false positives in 9.87% of the correct executions being tested. Compared to the fixed thresholds previously observed, where the best trade-off
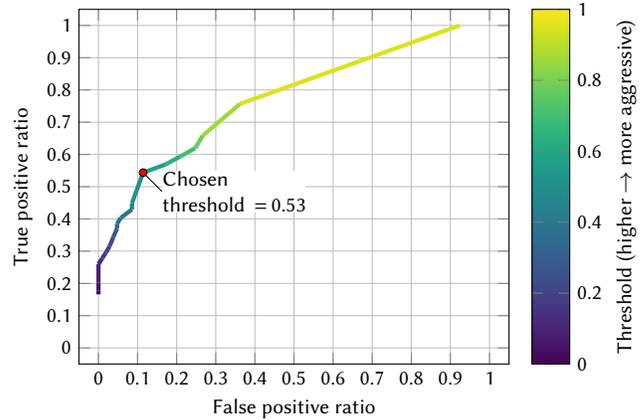
Figure 6: Receiver Operating Characteristics. The curve describes how our mechanism behaves based on the threshold used (from very conservative, 0, to very aggressive, 1).

offered 54% true positives and 11% false positives, we can see a significant improvement in the quality of the classifier. Naturally, it is possible to introduce bias in the gradient boosting to reduce the false positive ratio. This can be done either through selection of the training set or skeweing parameters. However, doing so resulted in significant loss in true positive detection ratio when trying to reduce false positive rates any further.
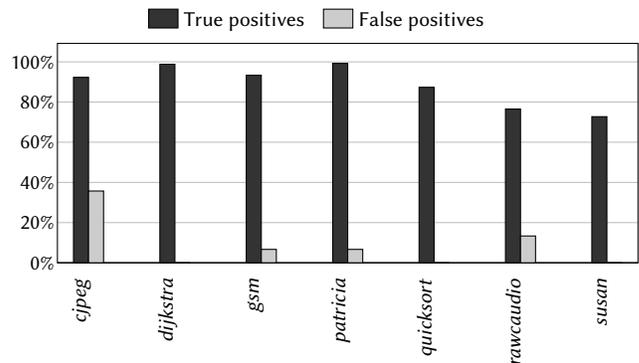
Figure 7: Error coverage and false positives obtained for each benchmark using a gradient boosting model.

6 RELATED WORK

Research in the area is mostly focused either on detecting faults [3, 15, 21] or detecting malicious behavior [16]. The former focuses on redundancy, whereas the latter commonly focuses on specific forms of malicious attacks. These techniques can be seen as complementary to our proposal, as our technique covers a broader spectrum of anomalies.

Work that is closer to our area includes Khanna et al. [9], where the authors construct a hidden Markov model (HMM) to detect system intrusions in ad hoc networks. They used given traces of system calls, and network protocol activity to construct usage and

activity profiles. They performed 2 weeks of training. In the first week, the system was profiled for normal activity. In the second week, DARPA attacks [11] were mounted on the system. They were able to obtain 90% true positive identification with less than 10% false positive identifications for a given set of parameters for the mechanism. In comparison, our work has a limited amount of input. We only look at memory references, and thus our capability of inference on benchmark behavior is constrained. Due to this, we chose a regular Markov chain process and comparison of Markov chains, although we could extend our research to use a hidden Markov model and obtain predictions during execution.

Balakrishnan et al. [1] describe a tool to analyze memory accesses in x86 static executables. This tool is able to create intermediate representations of any program, similar to those that can be created for programs written in a high-level language. They create a new technique, value-set analysis, to track the value of data objects at each point of the program. Using this analysis, it is possible to identify vulnerabilities like buffer overflow and self-modifying code. Compared to our work, their work shows the variability and relevance of memory accesses to the characteristic pattern of a program. By identifying anomalous write positions and value sets, they enable the capture of possible vulnerabilities before execution. Our work performs post-execution analysis on the memory reference stream, but is more comprehensive, as any anomalous activity (such as silent data corruption due to underfeeding voltage [20]) might be detected.

7 CONCLUSIONS

Our results show that the memory access stream can be used for program identification, which can allow detecting unknown or even malicious programs being executed. Furthermore, by applying our mechanism to different benchmarks, we were able to demonstrate, that under specific parameters' values, our mechanism can obtain good silent data corruption coverage for the considered benchmarks. We have achieved over 88% error coverage, while generating less than 10% false positives, in all of our tests. To the best of our knowledge, no technique can cover anomalies of multiple types in an unified mechanism as it is possible in our research.

With the addition of logic in main memories [17] or the usage of a companion core, an implementation of our mechanism would allow an external observer to act as a *watchdog* for the system, even if it was compromised due to silent error attacks. In future work, we would like to investigate specific benchmark characteristics that make faults hard to distinguish, and explore a physical implementation of our idea in the hybrid memory cube.

REFERENCES

[1] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. 5–23.

[2] Francieli Zanon Boito, Rodrigo Kassick, Philippe OA Navaux, and Yves Denneulin. 2015. Towards fast profiling of storage devices regarding access sequentiality. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM.

[3] E. Borin, Cheng Wang, Youfeng Wu, and G. Araujo. 2006. Software-based transparent and comprehensive control-flow error detection. In *International Symposium on Code Generation and Optimization (CGO'06)*. 13 pp.–.

[4] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[5] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier Temam. 2007. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *International Conference on High-Performance Embedded Architectures and Compilers*. 245–260.

[6] Crispin W Gardiner and others. 1985. *Handbook of stochastic methods*. Vol. 3. Springer Berlin.

[7] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization (WWC)*. 3–14.

[8] Tanay Karnik and Peter Hazucha. 2004. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing* 1, 2 (2004), 128–143.

[9] Rahul Khanna and Huaping Liu. 2006. System approach to intrusion detection using hidden markov model. In *Proceedings of the 2006 international conference on Wireless communications and mobile computing*. 349–354.

[10] Thomas A Lasko, Jui G Bhagwat, Kelly H Zou, and Lucila Ohno-Machado. 2005. The use of receiver operating characteristic curves in biomedical informatics. *Journal of biomedical informatics* 38, 5 (2005), 404–415.

[11] Richard P Lippmann, David J Fried, Isaac Graf, Joshua W Haines, Kristopher R Kendall, David McClung, Dan Weber, Seth E Webster, Dan Wyschogrod, Robert K Cunningham, and others. 2000. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, Vol. 2. 12–26.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

[13] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv* (2016).

[14] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *IEEE/ACM International Symposium Microarchitecture (Micro)*. 29–40.

[15] Nitin, I. Pomeranz, and T. N. Vijaykumar. 2015. FaultHound: Value-locality-based soft-fault tolerance. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 668–681.

[16] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security Symposium*. 447–462.

[17] J Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *Hot Chips*, Vol. 23.

[18] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and others. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[19] Carl Edward Rasmussen. 1999. The infinite Gaussian mixture model. In *NIPS*, Vol. 12. 554–560.

[20] Francesco Regazzoni, Thomas Eisenbarth, Luca Breveglieri, Paolo Ienne, and Israel Koren. 2008. Can knowledge regarding the presence of countermeasures against fault attacks simplify power attacks on cryptographic devices?. In *International Symposium on Defect and Fault Tolerance of VLSI Systems*. 202–210.

[21] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization (CGO)*. 243–254.

[22] Mark Russinovich. 2007. Inside the windows vista kernel: Part 3. *Microsoft TechNet Magazine* (2007).

[23] Ralph Gregory Taylor. 1998. Models of computation and formal languages. (1998).

[24] Doe Hyun Yoon and Mattan Erez. 2009. Memory mapped ECC: low-cost error protection for last level caches. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 116–127.