Data Mining the Memory Access Stream to detect Anomalous Behavior

Francis Birck Moreira, Matthias Diener, Israel Koren, Philippe Navaux

Outline

- Introduction & Background
- Motivation
- Methodology
- Experimental Results
- Conclusions

Introduction & Background

- Increasingly large computer systems with thousands of cores

- Low Mean-Time to Failure (MTTF) due to :
  - Crashes
  - Infinite loops
  - Wrong results (silent data corruption [1])

[1] Mukherjee et al. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor.

Introduction & Background

- Faults can happen due to:
  - Particle hits [2]
  - Device degradation[3]

- We are particularly interested in Silent Data Corruption (SDC)
  - An error occurs, data is changed, but the program executes normally and the OS does not notice anything unusual.

[2] Oliveira et al 2016. Evaluation and mitigation of radiation-induced soft errors in graphics processing units.

[3] Xu et al 2015. Improving processor lifespan and energy consumption using DVFS based on ILP monitoring.

Motivation

- Our proposal: detecting anomalous program behavior through memory access pattern.

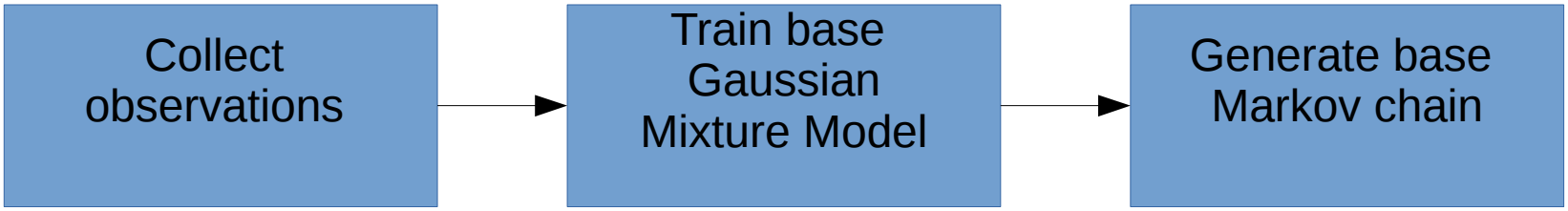
- Several advantages:
  - If processor is compromised, another component can act as external watchdog
  - Generality of memory accesses might be able to detect several types of anomalous executions, such as malicious attacks [4]

[4] Liu et al 2015. Last-level cache side-channel attacks are practical.

Motivation

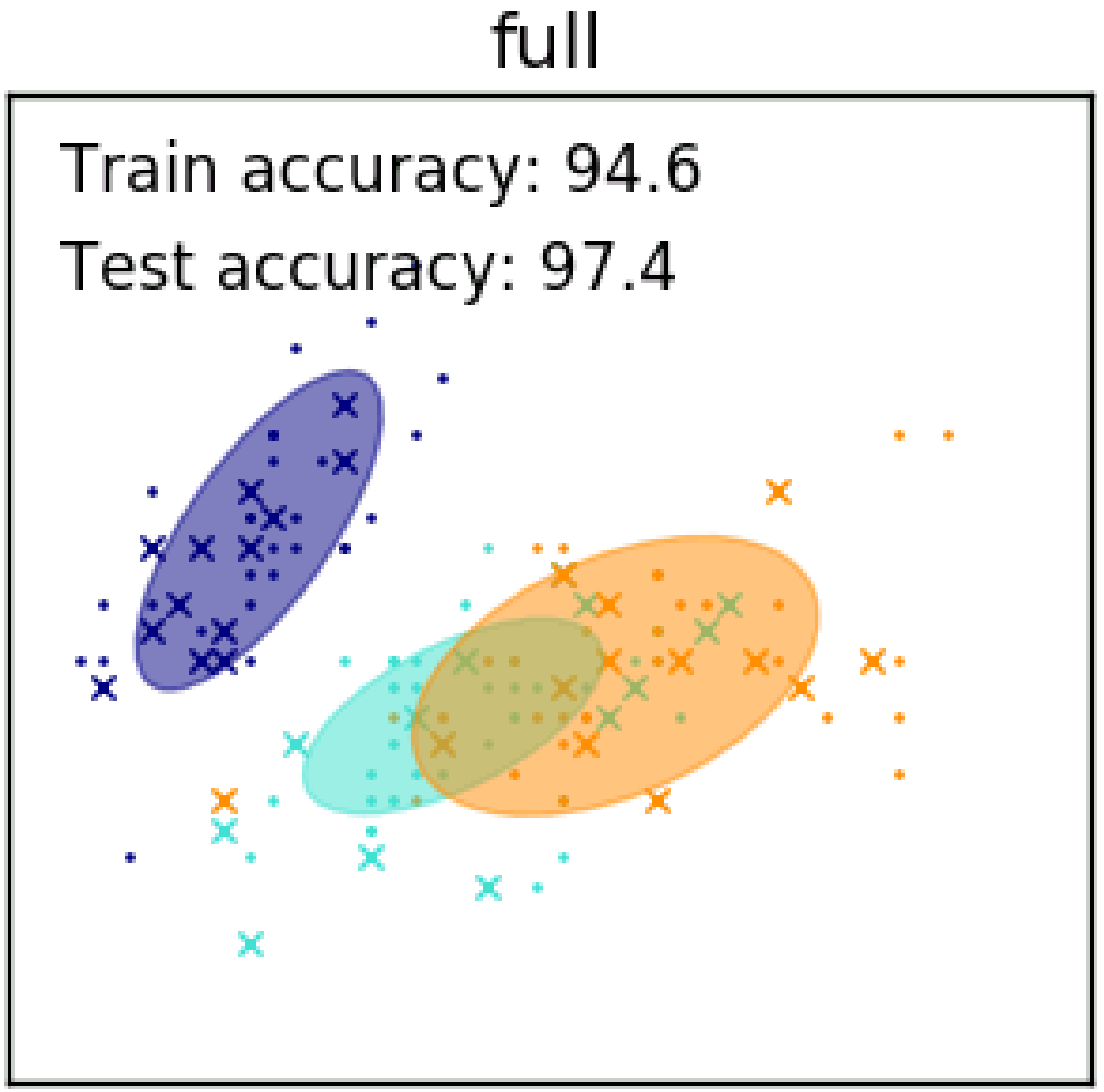
- Drawbacks:
  - Different inputs can generate different patterns
  - Operating system allocation makes it impossible to use physical addresses

Methodology - offline



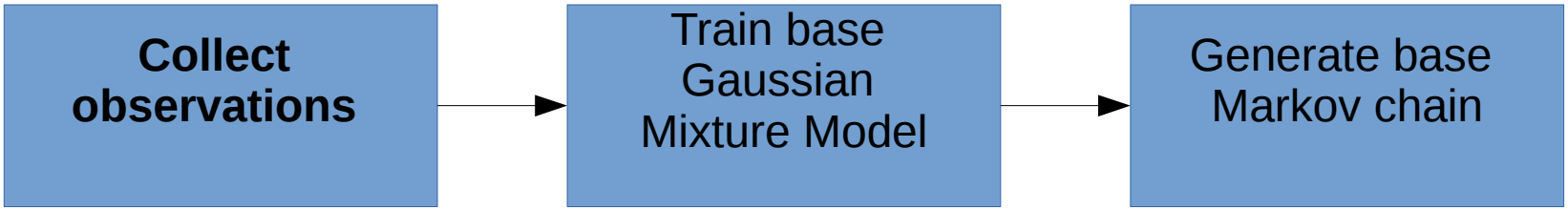
- Pattern detection through machine learning
- Unsupervised classifier, Gaussian Mixtures, clusters “phases” of memory access stream
- Markov process describes the temporal relationship between these phases

Gaussian Mixture Model Example



http://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_covariances.html

Methodology - offline

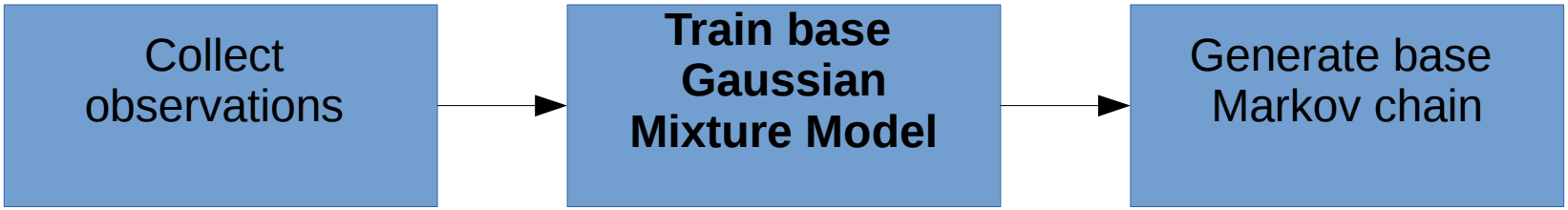


- Each observation is a fixed memory period of 1024/16384 memory accesses, with 6 features: #reads, #writes, #instructions, #pages-read, #pages-written, #instruction-pages

Results for Quicksort and Dijkstra

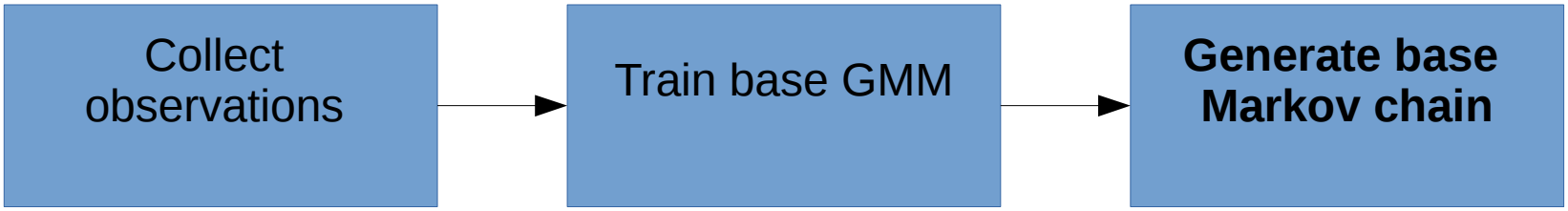
	quicksort	Quicksort (faulty)	dijkstra	Dijkstra (faulty
Reads	34,163,994	37,714,422	31,009,923	30,991,253
Writes	21,314,365	24,706,058	15,040,334	15,030,683
Instructions	172,865,49	174,279,168	162,436,143	162,349,776
Pages Read	180,995	158,208	167,082	179,613
Pages Written	67,880	58,277	73,456	85,977
Pages Instruction	121,890	125,854	64,364	64,303

Methodology - offline



- We use Gaussian Mixture Model to cluster memory periods
- Number of centroids is different for each program

Methodology - offline



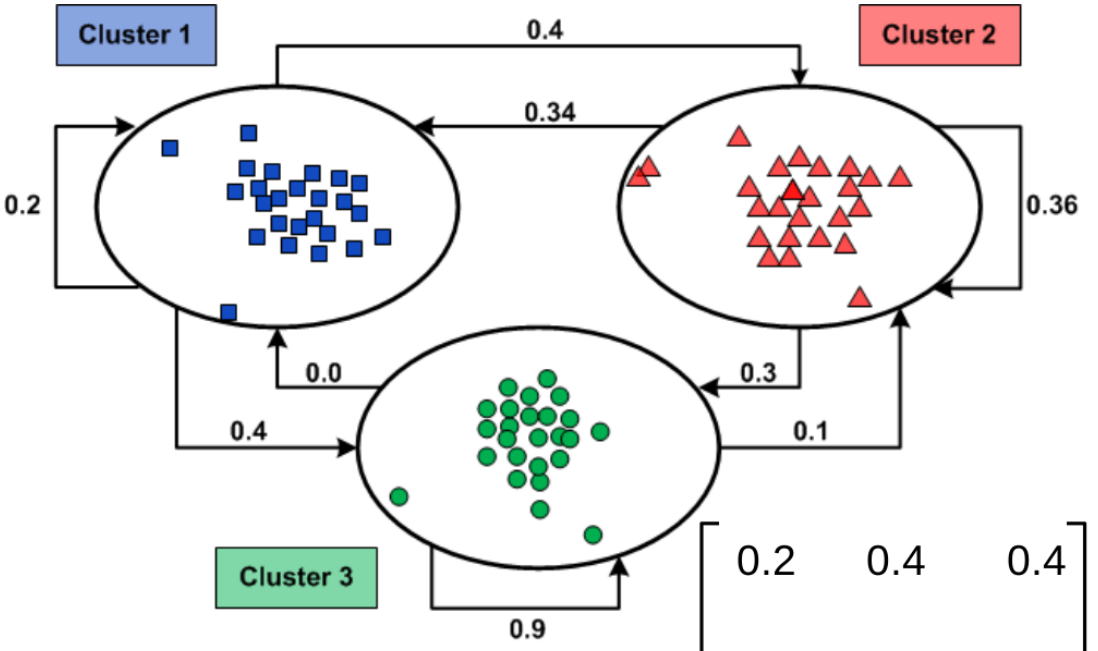
- We now have a GMM of normal executions (**base GMM**)
- We obtain centroids of memory phase types
- We construct base Markov chain based on the sequence of the centroid indices

Methodology - online

- Collect current execution observations
- Classify current execution periods with program's **base GMM**
- Generate **current execution Markov chain** with current execution period indices
- Compare **base Markov chain** and **current execution Markov chain**

Metrics for Comparing Markov Chains

- Total matrix variation
- Unidentified transition ratio
- Trace difference
- Rank difference
- Matrix sparsity



$$\begin{bmatrix} 0.2 & 0.4 & 0.4 \\ 0.34 & 0.36 & 0.3 \\ 0.0 & 0.1 & 0.9 \end{bmatrix}$$

Methodology - tools

- Set of benchmarks: miBench
- Set of inputs: Fursin et al [5]
  - 20 different inputs for each benchmark
- Tools:
  - Memory tracing - Pin 3.2
  - Machine learning and Markov chain comparison - Python libraries: numpy, math, sklearn

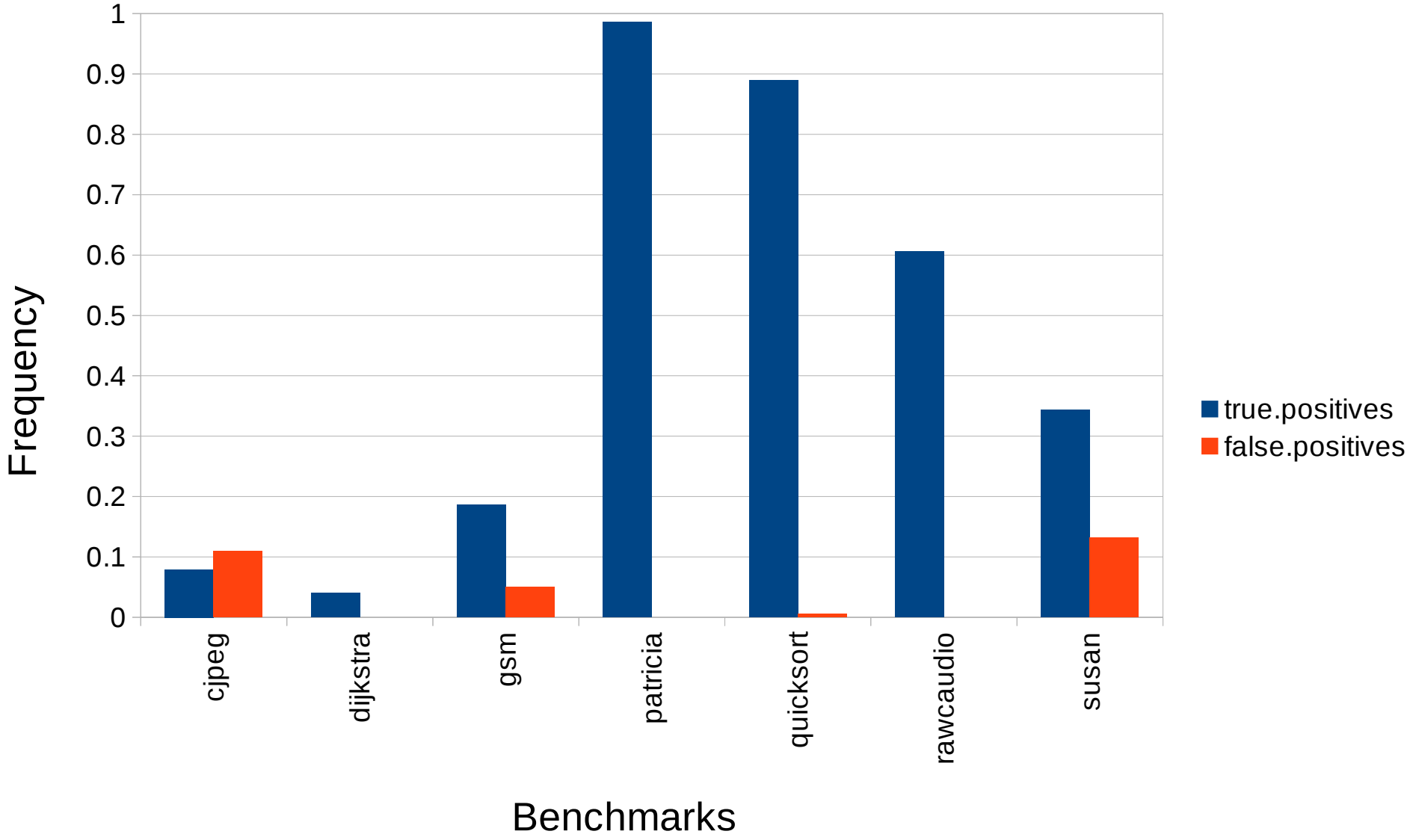
[5]Fursin et al 2007. Midatasets:Creating the conditions for a more realistic evaluation of iterative optimization.

Experimental Results

- Program identification can be accomplished

- For fault detection experiments, we used:
  - 5 inputs to train the GMM and the Markov chain
  - 15 other inputs to test false positive frequency
  - 100 different silent errors in different inputs to test fault coverage (aka true positive frequency)

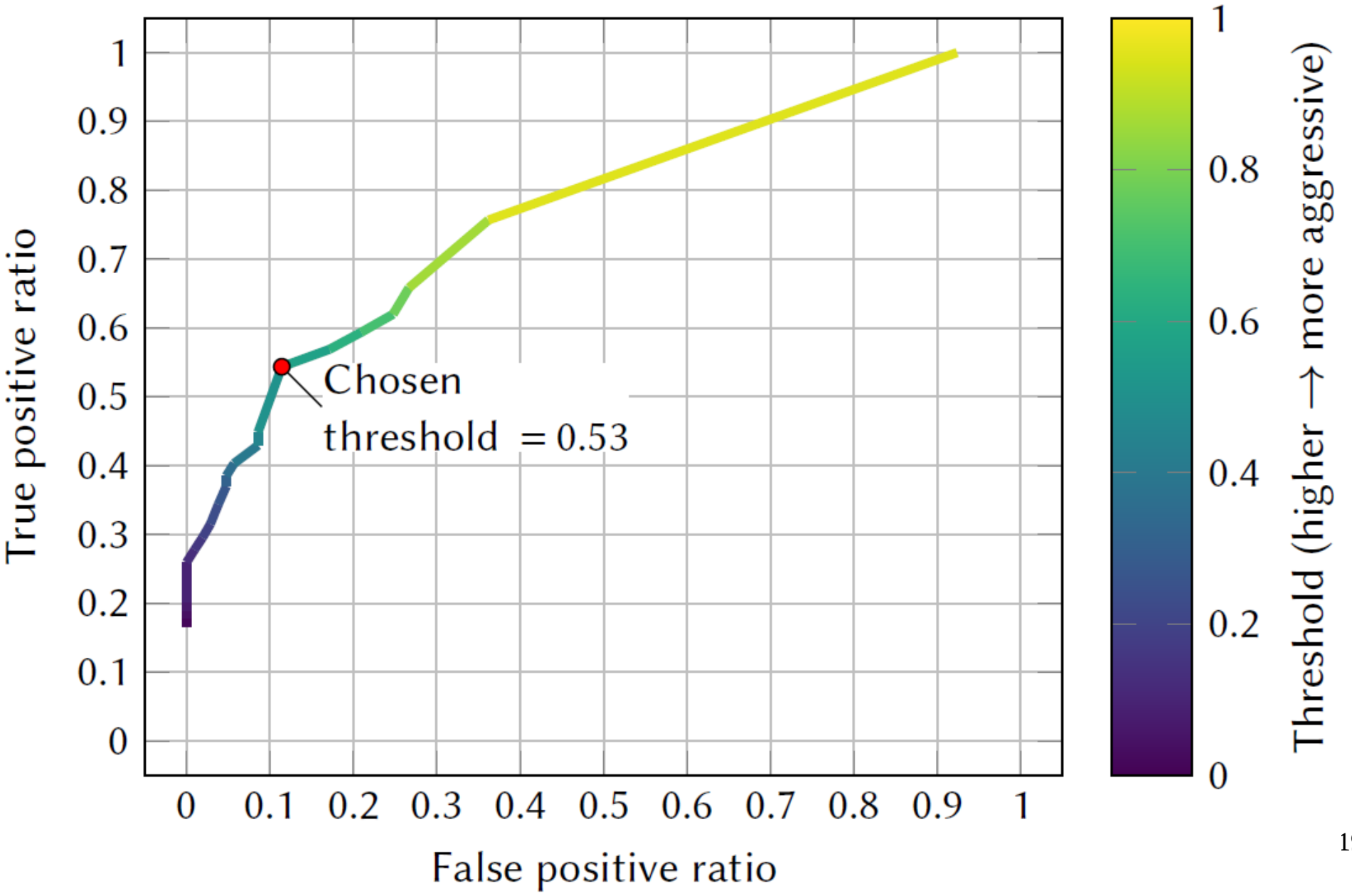
Experimental Results



Experimental Results

- Some programs' anomalous behavior is clearly easier to identify than others
- Very conservative threshold to ensure low false positive ratio
- So we decided to plot the curve for several threshold values:

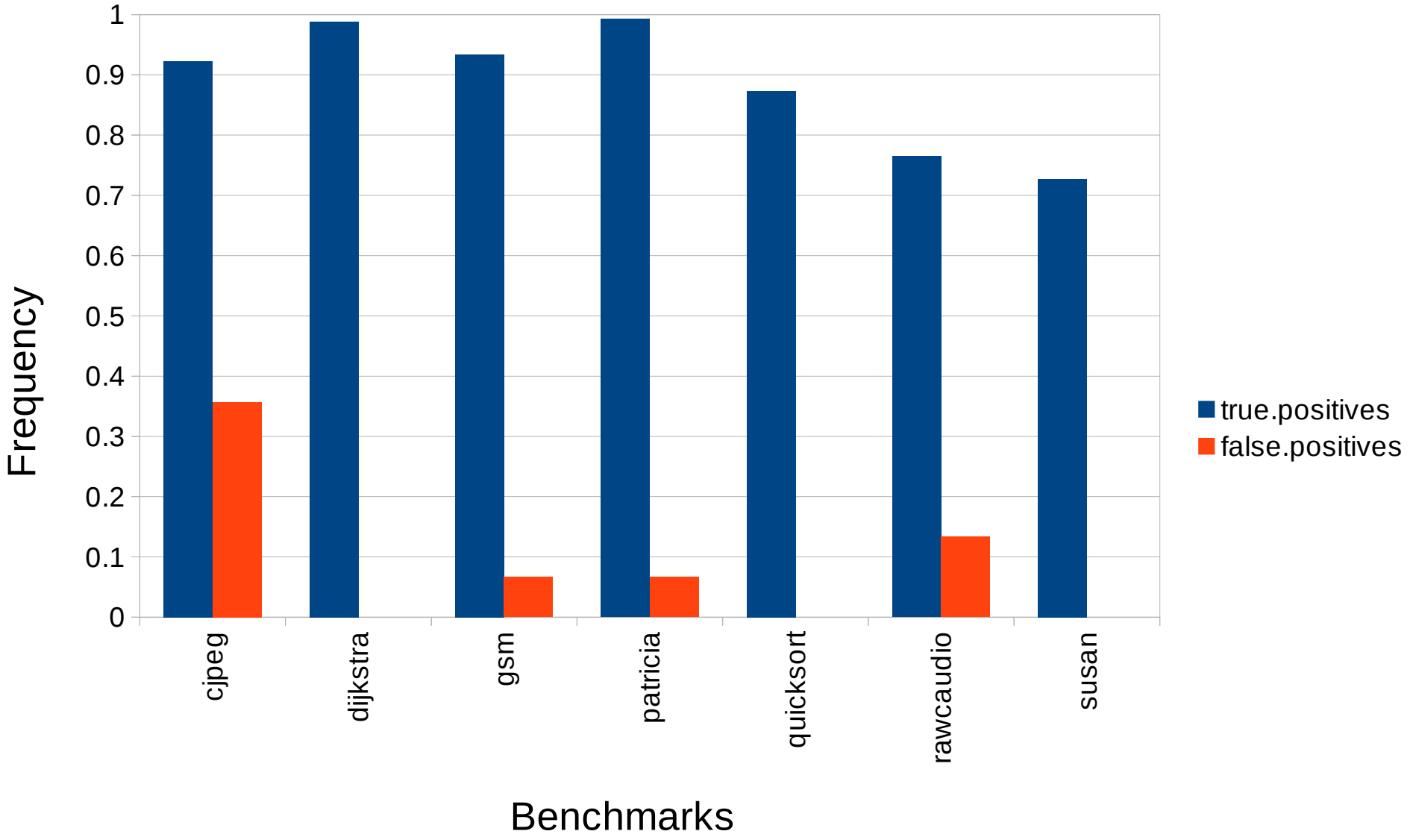
Experimental Results



Experimental Results

- Fixed thresholds do not correctly express deviations
- Some combination of the metrics with different thresholds is needed
- Which convinced us to turn to Gradient Boosting

Experimental Results with Gradient Boosting



Conclusions

- Gradient Boosting makes the technique quite effective: 88.5% true positive coverage, 8.9% false positive frequency
- False positive ratio can be smaller, but it will reduce coverage significantly
- Large false positive ratio in cjpeg, likely due to overfitting

Conclusions

- Future Work
  - Analyze and discern what makes cjpeg different from other benchmarks
  - If technique still can’t work for every benchmark, test specific interesting benchmarks, such as AES
  - To avoid post-mortem analysis, use Hidden Markov Models [6]

[6]Khanna et al 2006. System Approach to Intrusion Detection using Hidden Markov Models.

Data Mining the Memory Access Stream to detect Anomalous Behavior

Thank you!
This work received partial funding from CAPES/PVE, the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement no. 689772.

Francis Birck Moreira, Matthias Diener, Israel Koren, Philippe Navaux