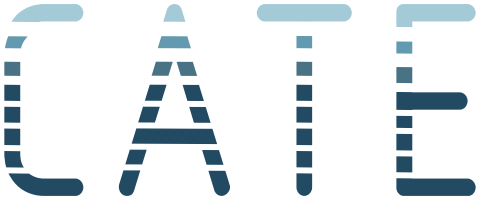


HUMBOLDT-UNIVERSITÄT ZU BERLIN

CATE: An Open and Highly Configurable Framework for Performance Evaluation of Packet Classification Algorithms

Wladislaw Gusew, Sven Hager, Björn Scheuermann

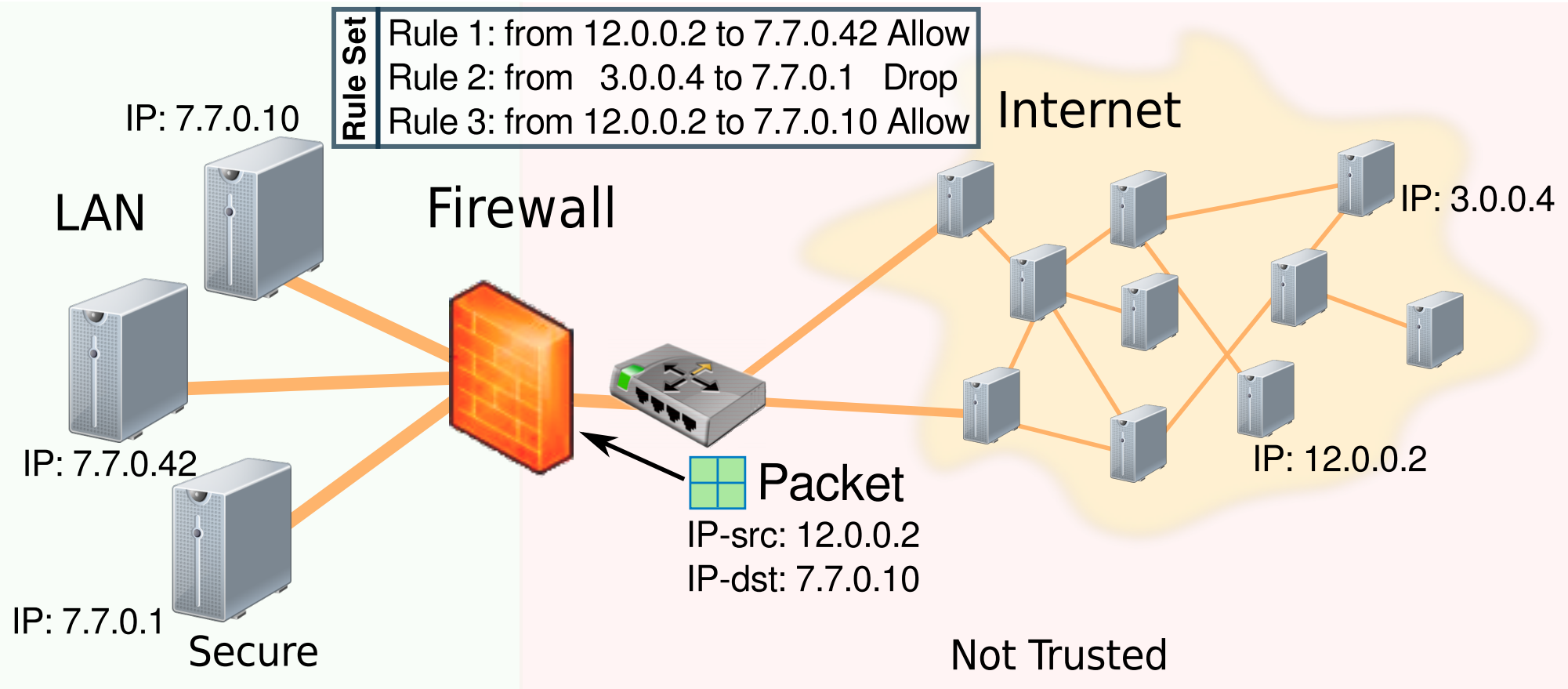
Computer Engineering Group, Humboldt University of Berlin

INTRODUCTION

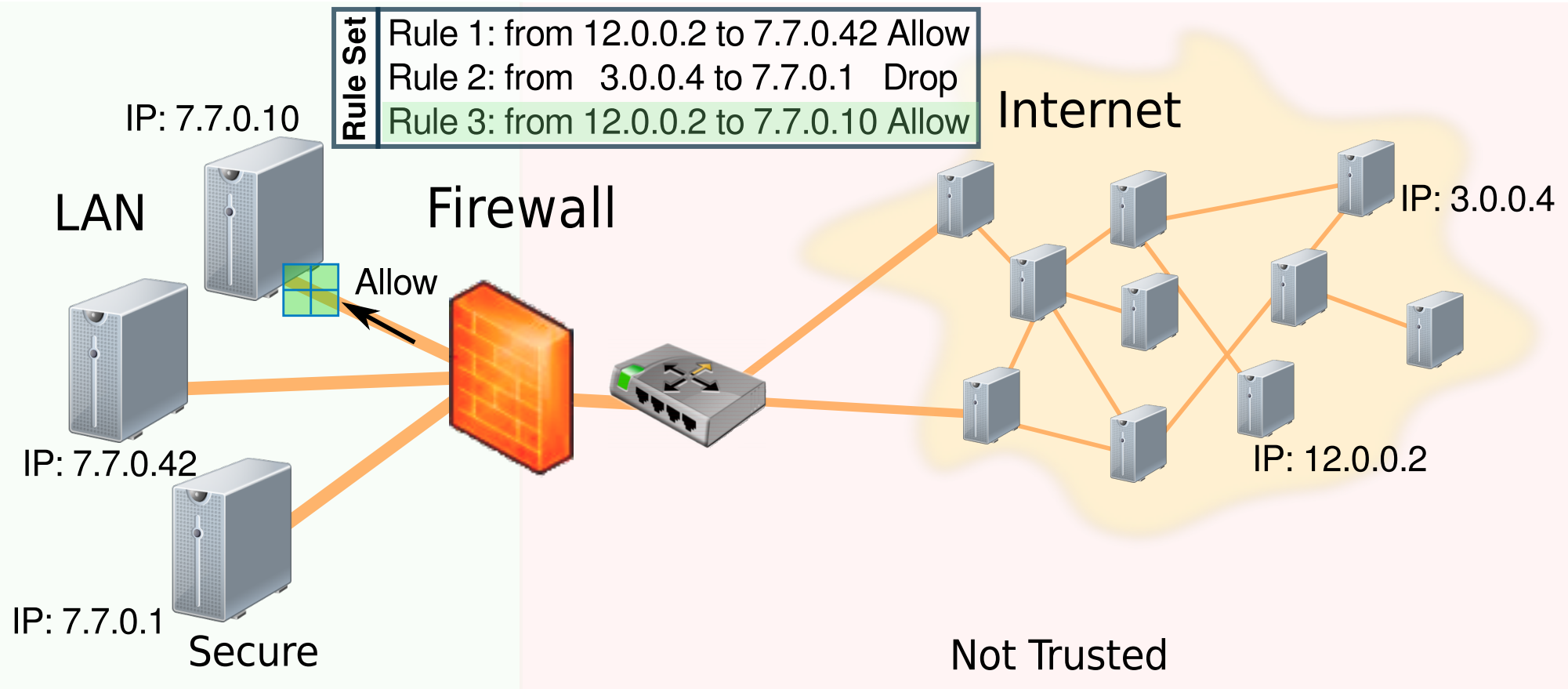
Network packet classification is a crucial task performed in several network applications, such as

- **Firewalls**
- Policy routing
- Traffic measurement and accounting
- Traffic rate limiting
- Intrusion Detection Systems

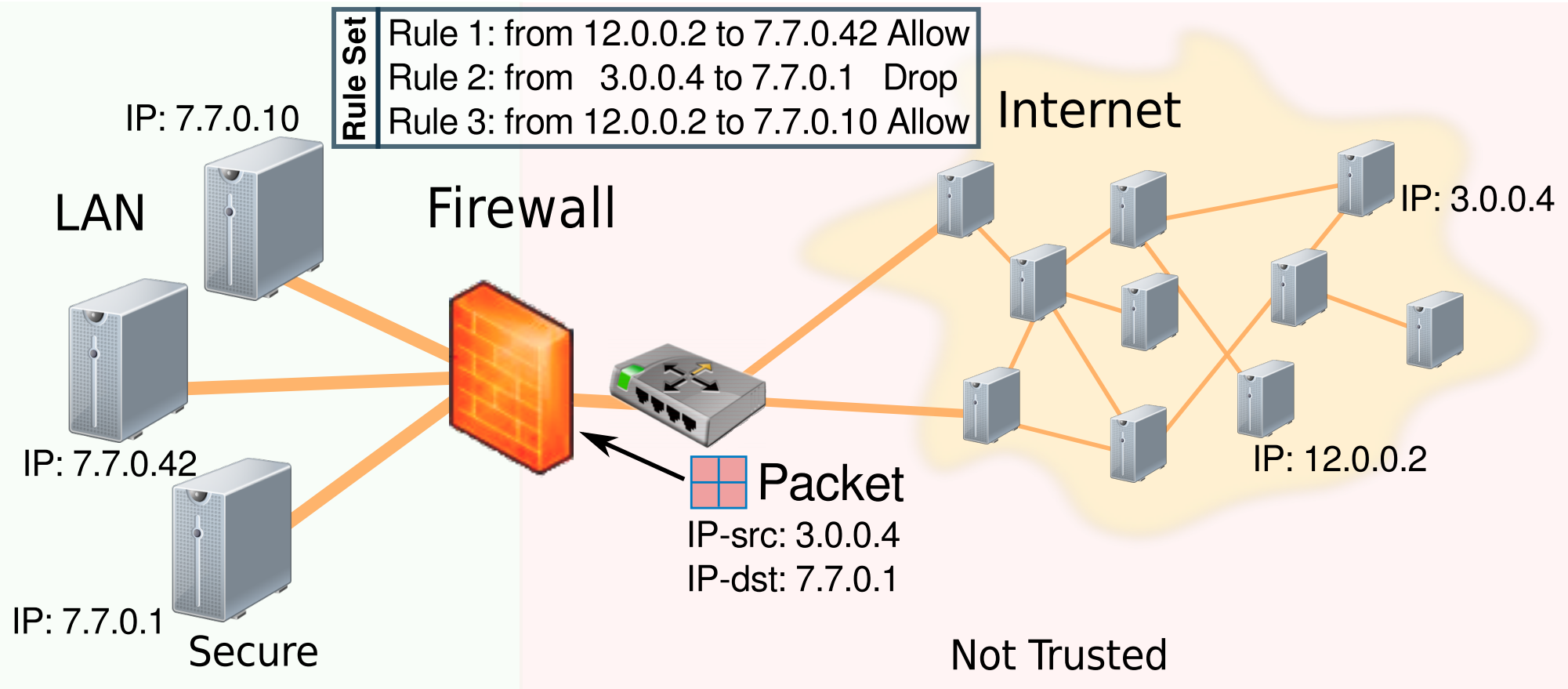
PACKET FILTERING IN FIREWALLS



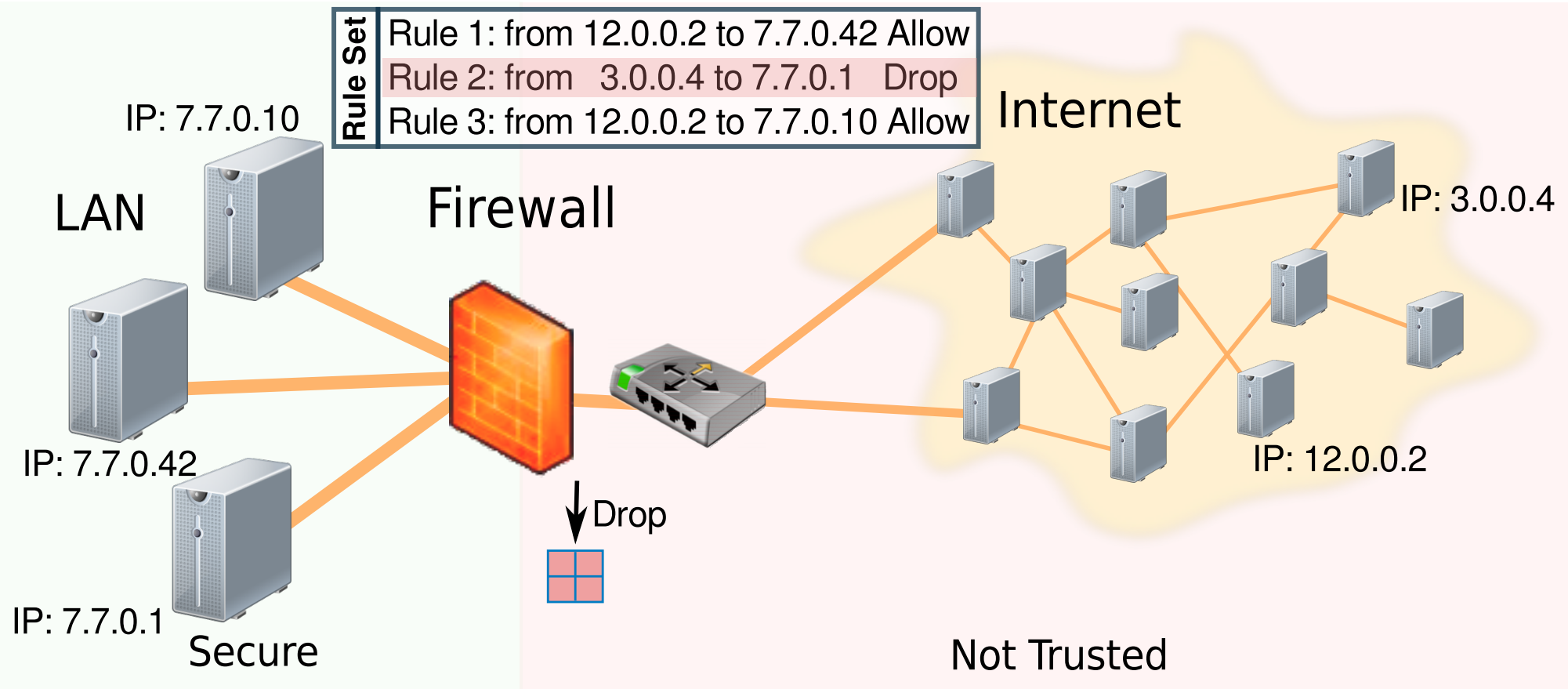
PACKET FILTERING IN FIREWALLS



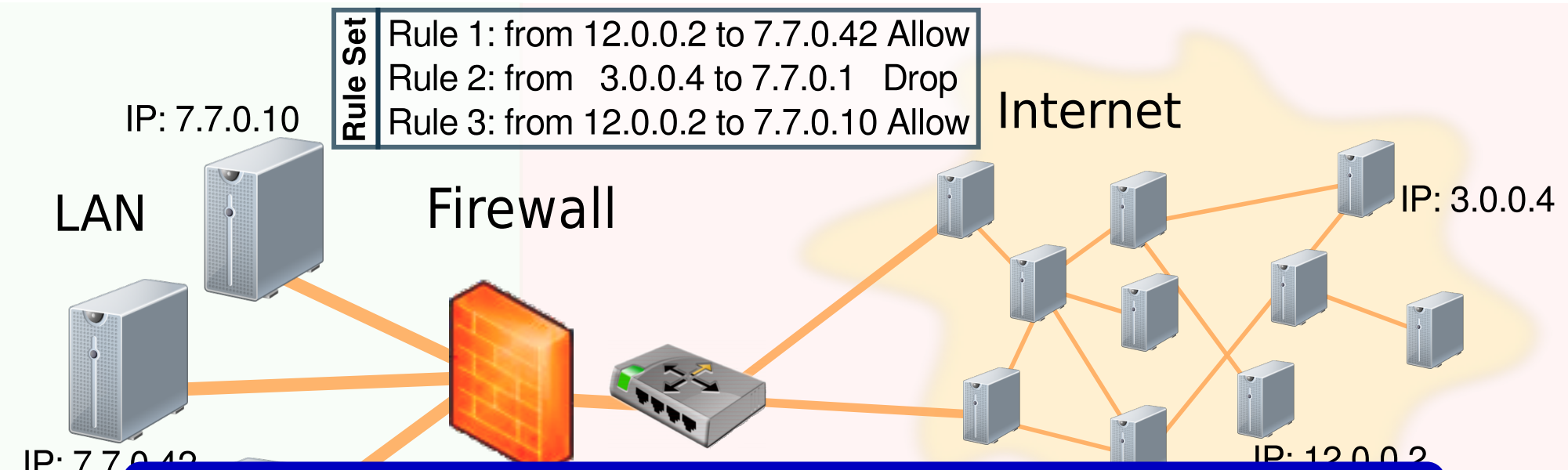
PACKET FILTERING IN FIREWALLS



PACKET FILTERING IN FIREWALLS



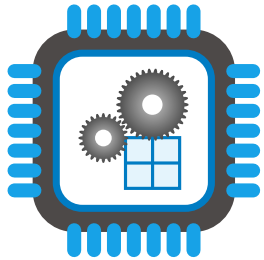
PACKET FILTERING IN FIREWALLS



Packet Classification

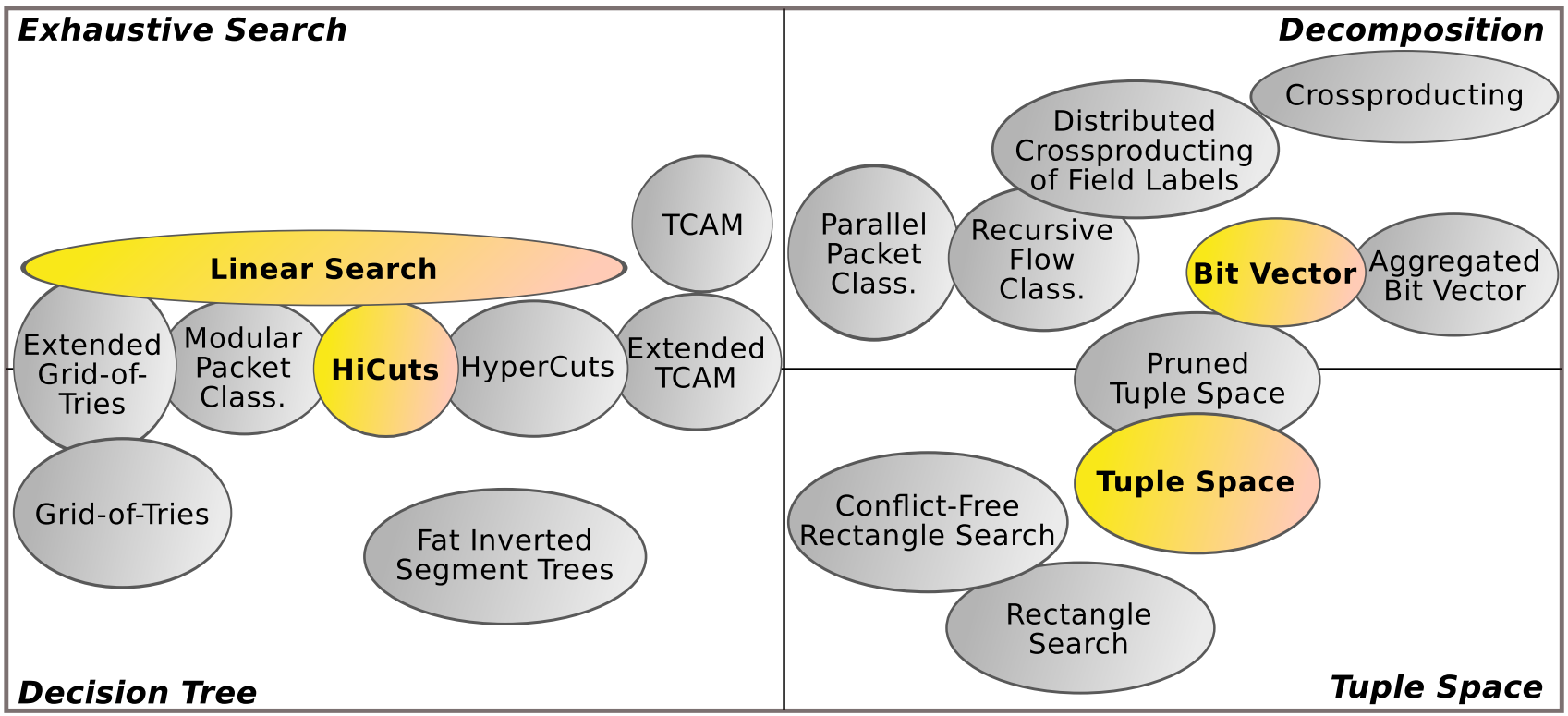
For each incoming packet find the highest priority matching rule in a rule set and apply its associated action.

CHALLENGING REQUIREMENTS



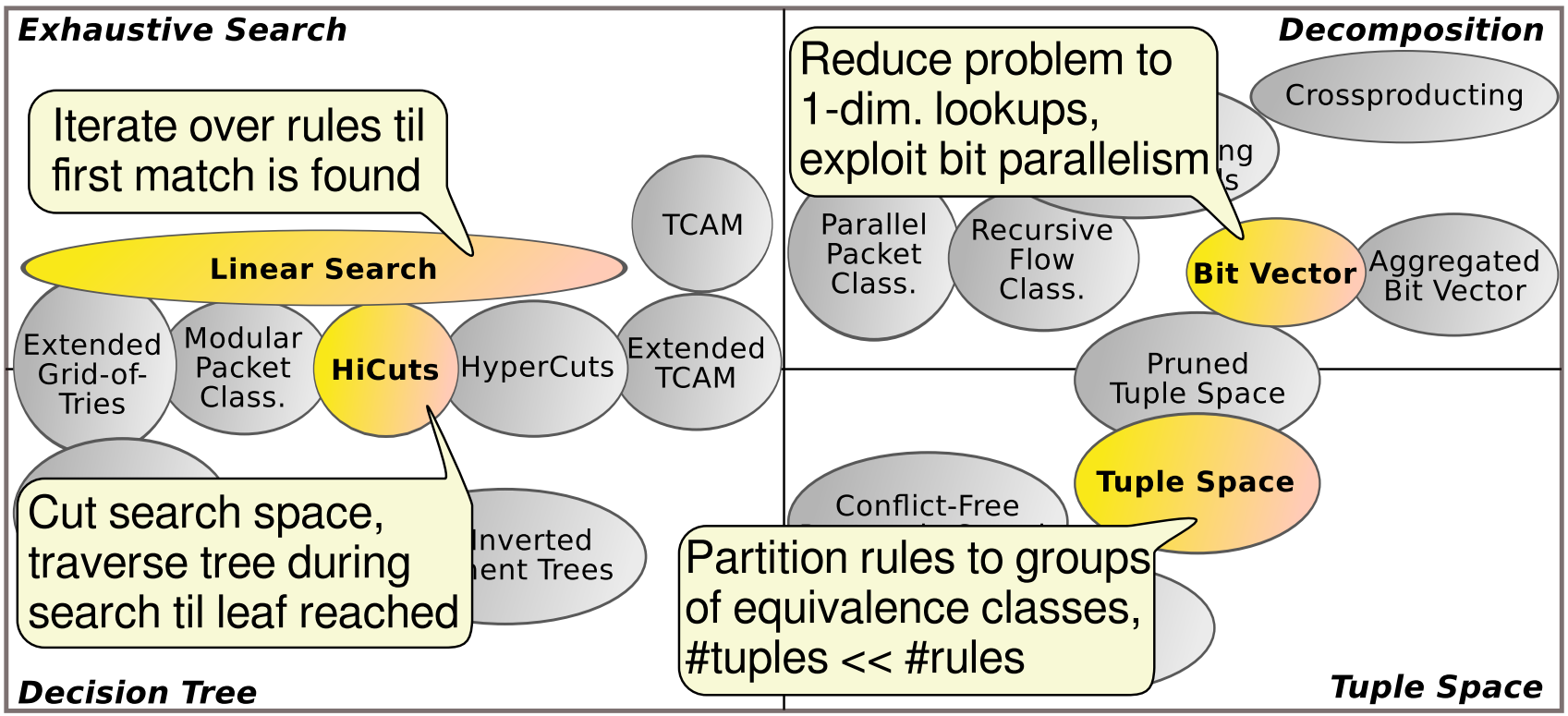
- Classifying at line speed:
  - For 40 Gbit/s: 125 millions IP-packets per second (with 40 Bytes each)
  - Time to classify and forward each packet is only **8 ns**
  - Main memory access time is **5 to 60 ns**
- Operate at the leading edge of today's hardware technology
- **Efficient packet classification is essential.**

CLASSIFICATION ALGORITHMS



Based on Taylor, David E. "Survey and taxonomy of packet classification techniques." ACM Computing Surveys (CSUR) 37.3 (2005): 238-275.

CLASSIFICATION ALGORITHMS



Based on Taylor, David E. "Survey and taxonomy of packet classification techniques." ACM Computing Surveys (CSUR) 37.3 (2005): 238-275.

PROBLEM STATEMENT

- Plenty of classification algorithms with different properties
- Differences in classification performance and memory footprint
- Comparisons problematic, as no integrative framework exists

Main Questions

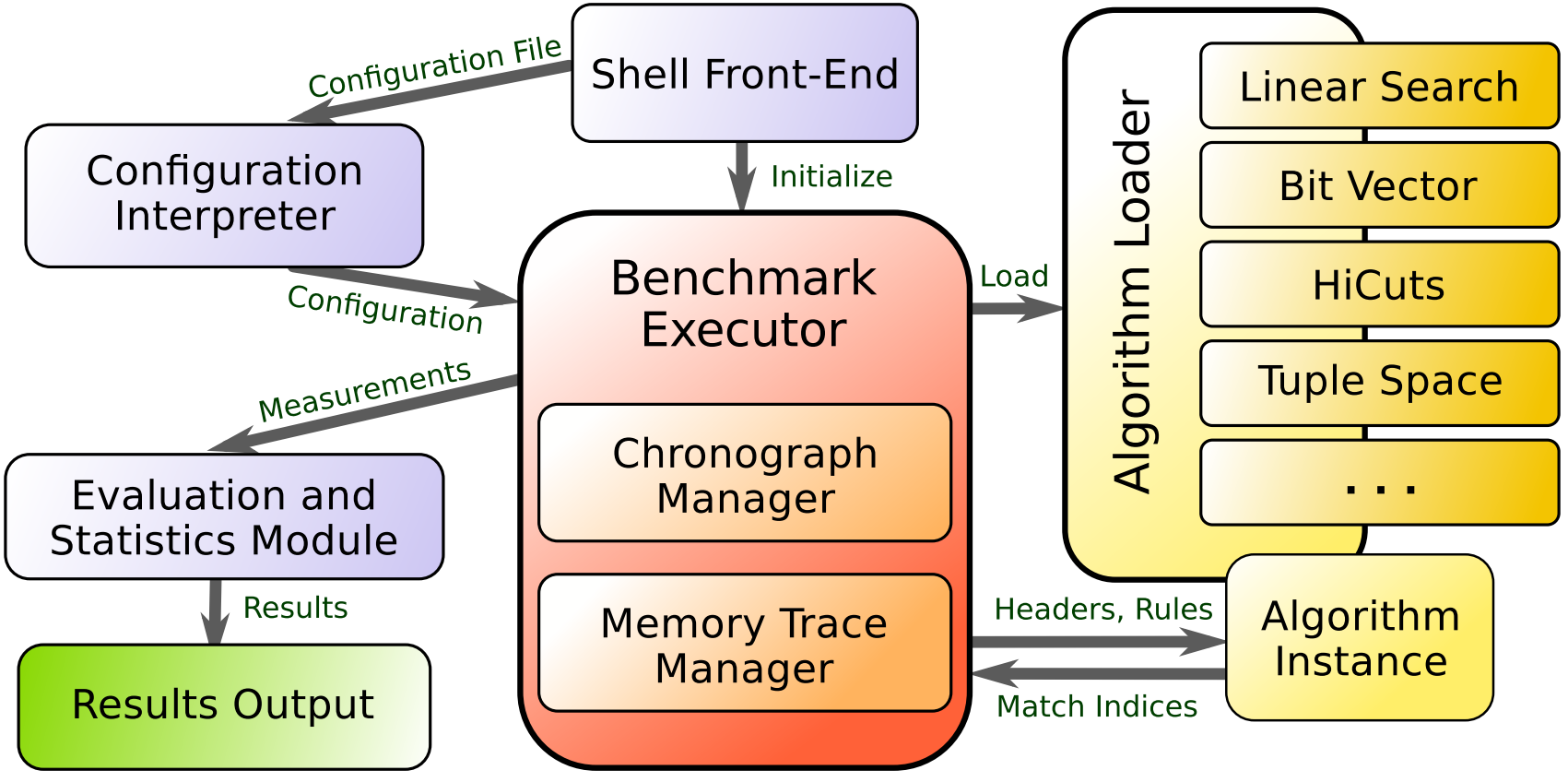
- **For developers**: Which classification algorithm is best suited for a certain application?
- **For researchers**: How to compare a new algorithm to already existing classification algorithms?

Classification
Algorithm
Testing
Environment

CLASSIFICATION ALGORITHM TESTING ENVIRONMENT

- Framework for benchmarking algorithms' software implementations
- Measured values: time durations and memory usage
- Implemented in C++, runnable on many targets
- Highly customizable with Lua as configuration language
- Uses white box classification algorithms

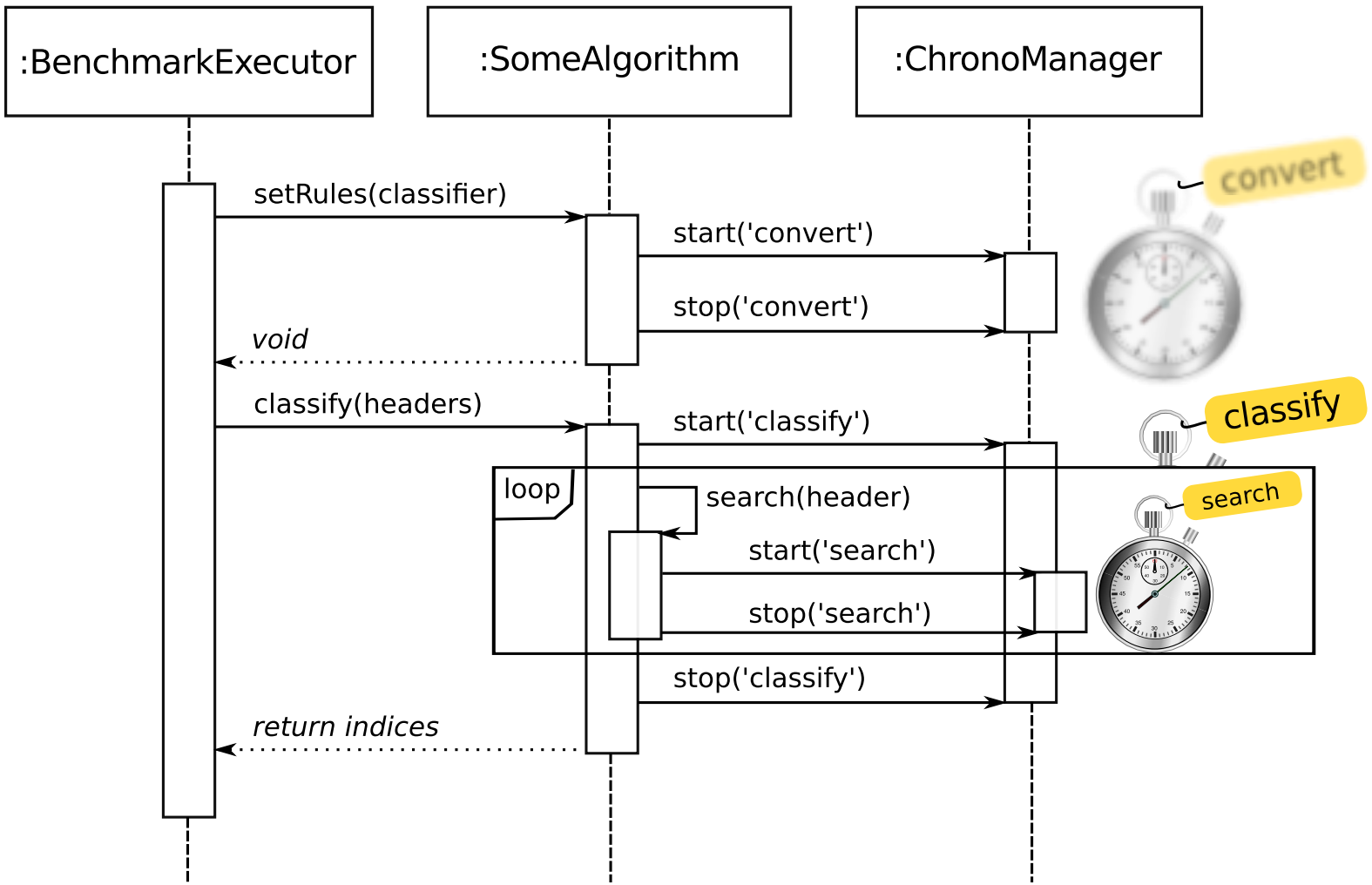
SOFTWARE ARCHITECTURE



MEASURING PERFORMANCE

- Take time duration(s)
- Compare benchmarks:
  equal workload – different time durations
- Important aspects: control and granularity

MEASURING PERFORMANCE



MEASURING MEMORY USAGE

- Goal: Precisely trace memory allocations and memory accesses (byte granularity)
- High-level programming languages provide an abstraction of memory management
- In C++, memory information can be acquired implicitly by the `sizeof`-operator

MEASURING MEMORY USAGE

- Realized by code-instrumentation
- Utilization of class inheritance and C++ template techniques
- Arbitrary data types can be regarded, e. g.
  - `int val;` ⇒ `MemTrace<int> val;`
  - `Stack st;` ⇒ `MemTrace<Stack> st;`
- Tracing memory usage imposes overhead on measured time durations

Evaluation

EVALUATION HIGHLIGHTS

1. Analyze probe effect (overhead by memory tracing)
2. Compare algorithms' performance for varying #rules
3. Effect of different rule set structures on classification performance
4. Vary field structure of packet headers

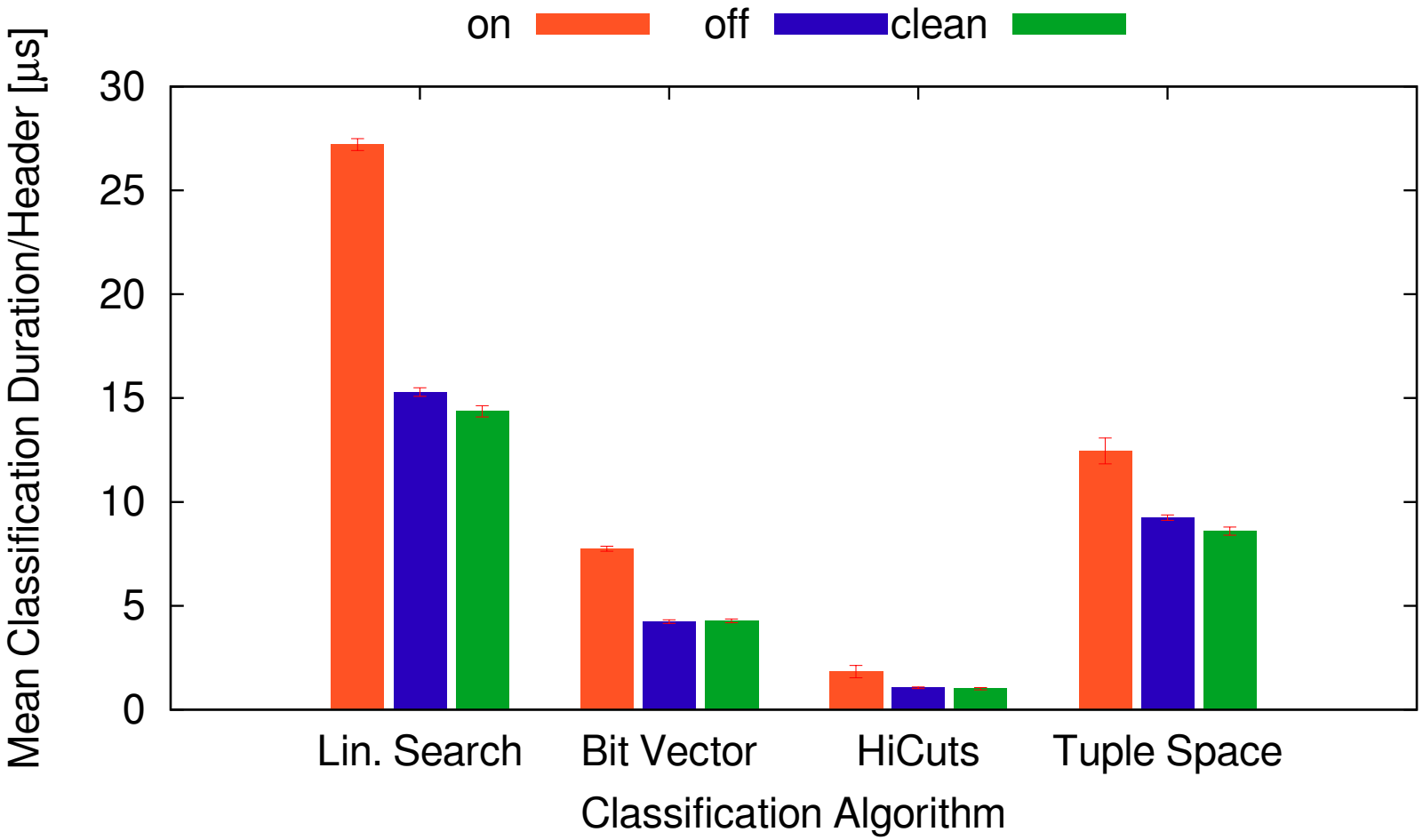
SYSTEM SETUP

- 2.27 GHz Intel i5 M430, single core, Linux 3.13
- Virtual Machine (32 bit) with 1024 MB RAM
- Synthetic *ClassBench* rules and 100K headers
- Measurements of time durations were repeated 8 times and arithmetic mean was taken.

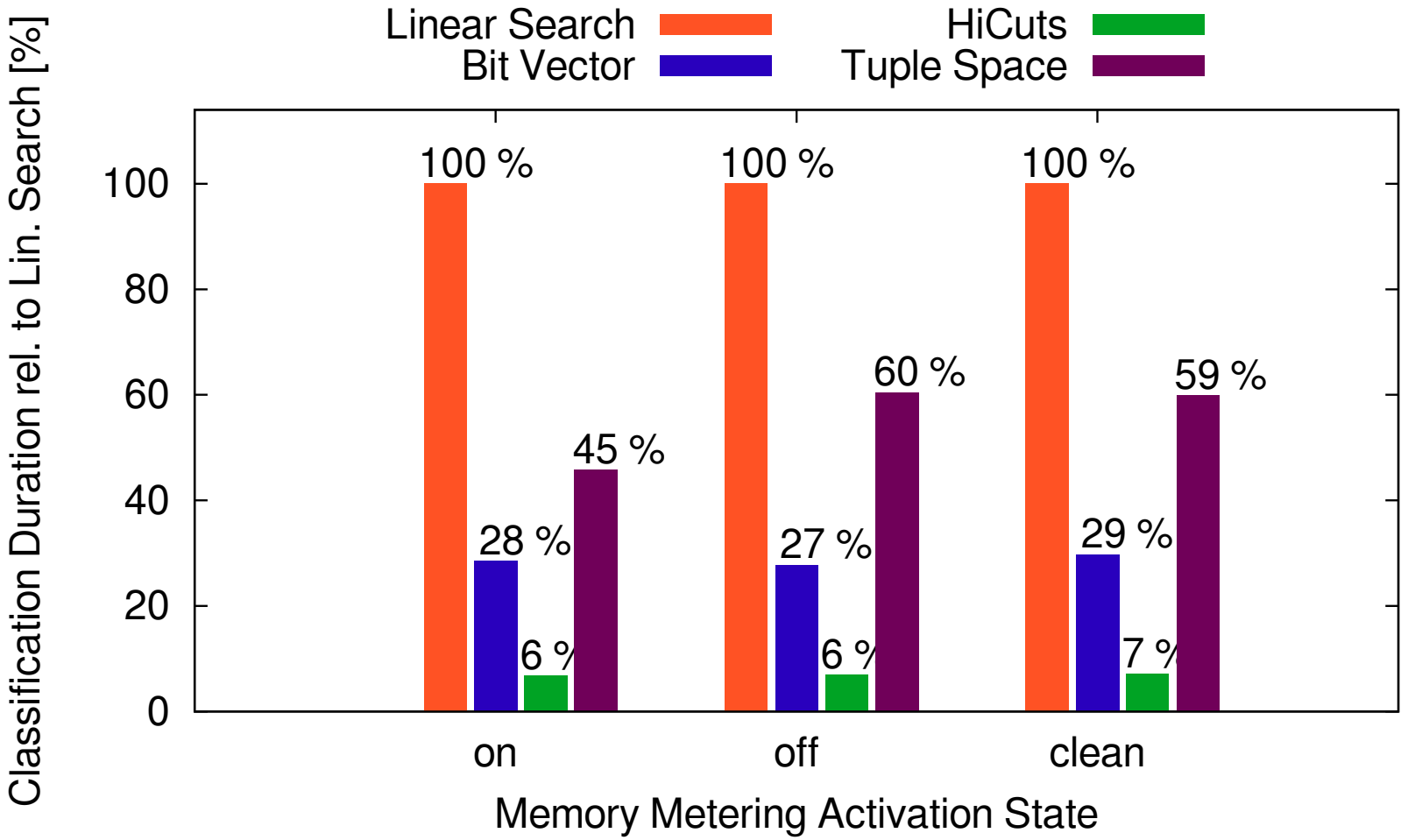
ANALYZING PROBE EFFECT

- Measure time durations in three operation modes:
  - enabled: memory allocations and accesses are traced during time measurements
  - disabled: memory usage tracing is disabled by preprocessor directive
  - cleaned: code-instrumentations are removed manually from code

ANALYZING PROBE EFFECT



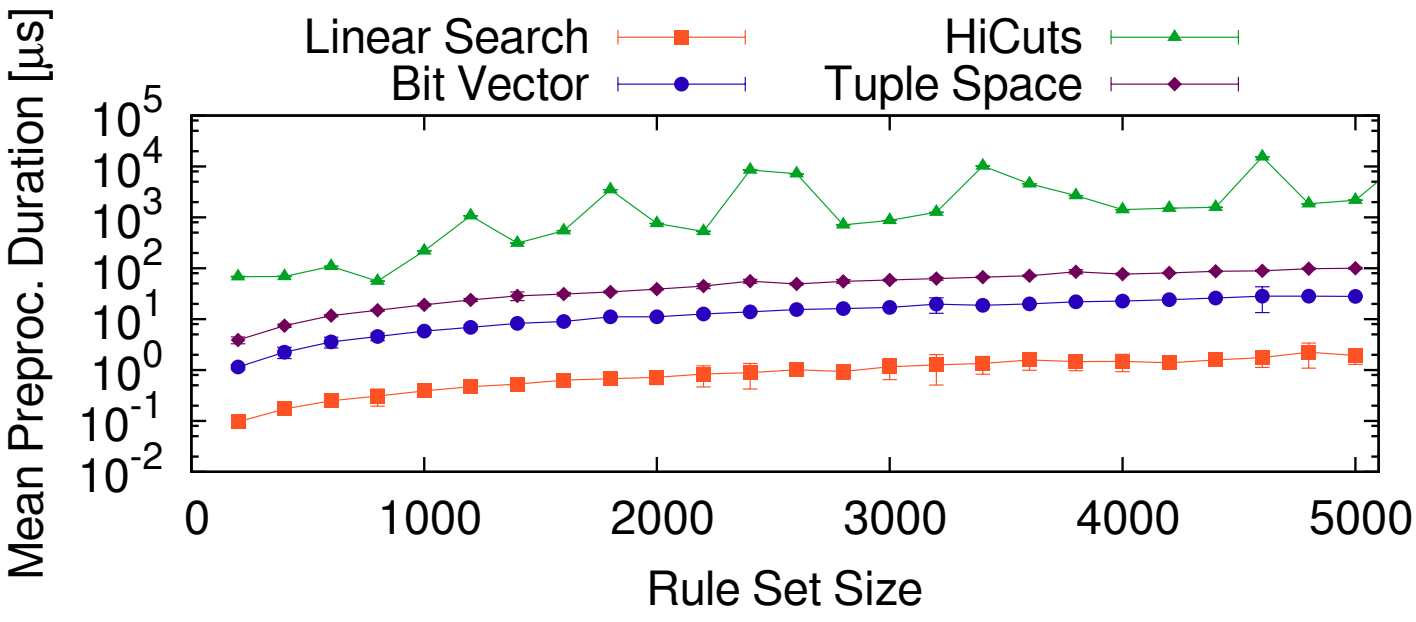
ANALYZING PROBE EFFECT

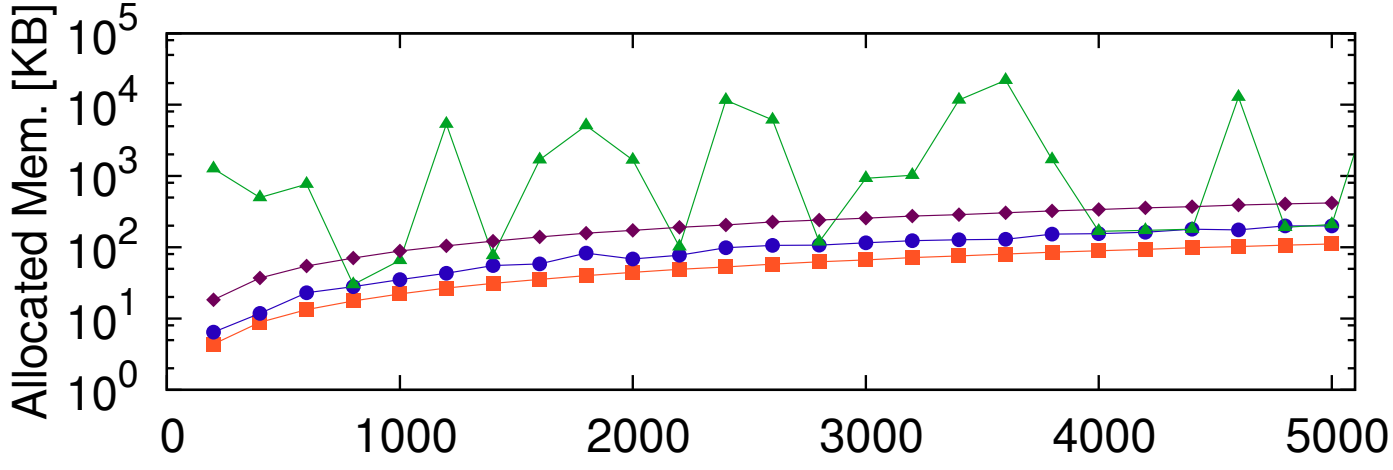


VARIATION OF RULES

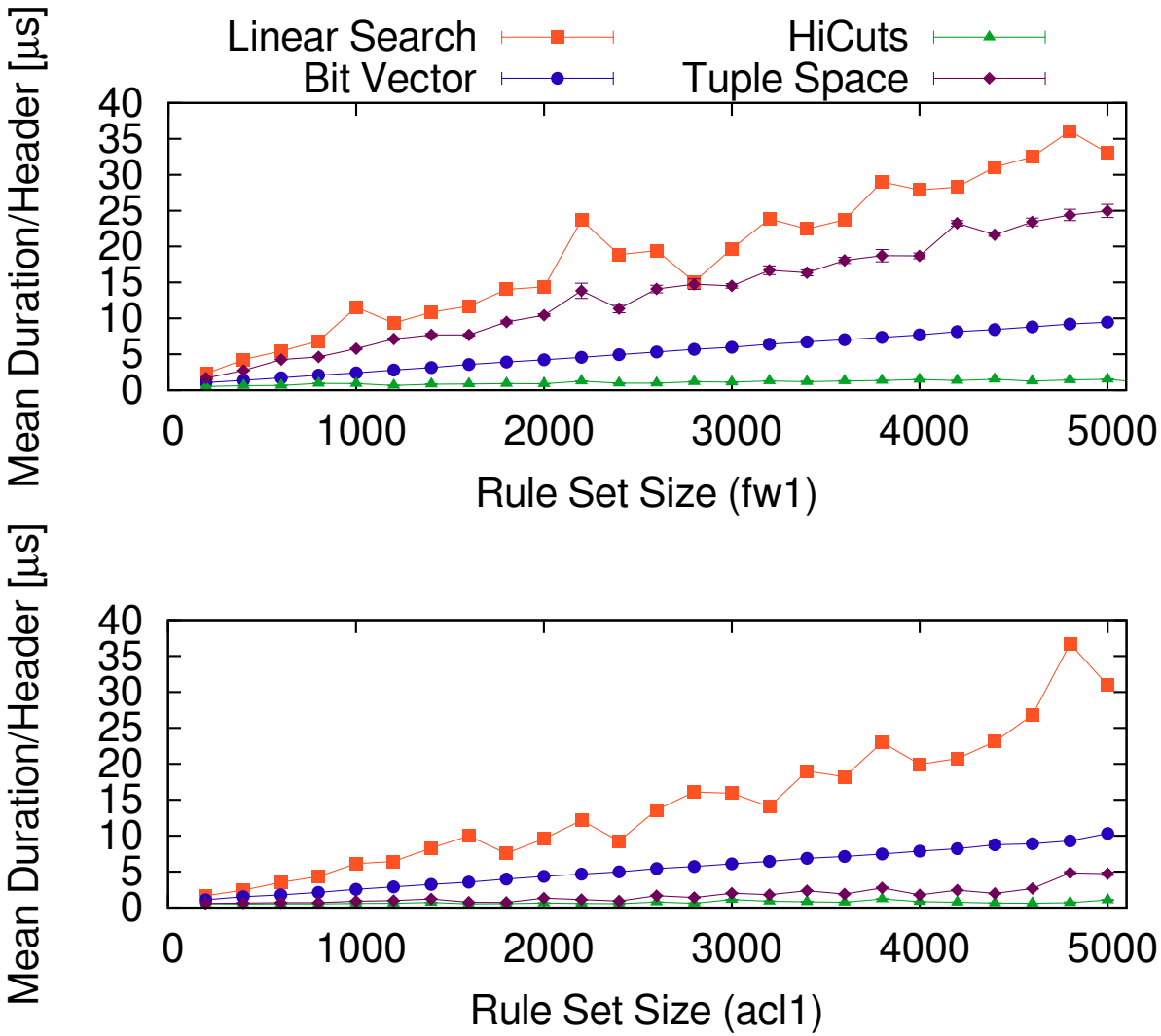
- Vary number of rules (200, 400, 600, ..., 5000)
- Use common 5-tuple: (32 bit, 32 bit, 8 bit, 16 bit, 16 bit)
- Observe performance during preprocessing and classification stages
- Compare performance for rule sets with different structure

VARIATION OF RULES - PREPROCESSING





VARIATION OF RULES - CLASSIFICATION

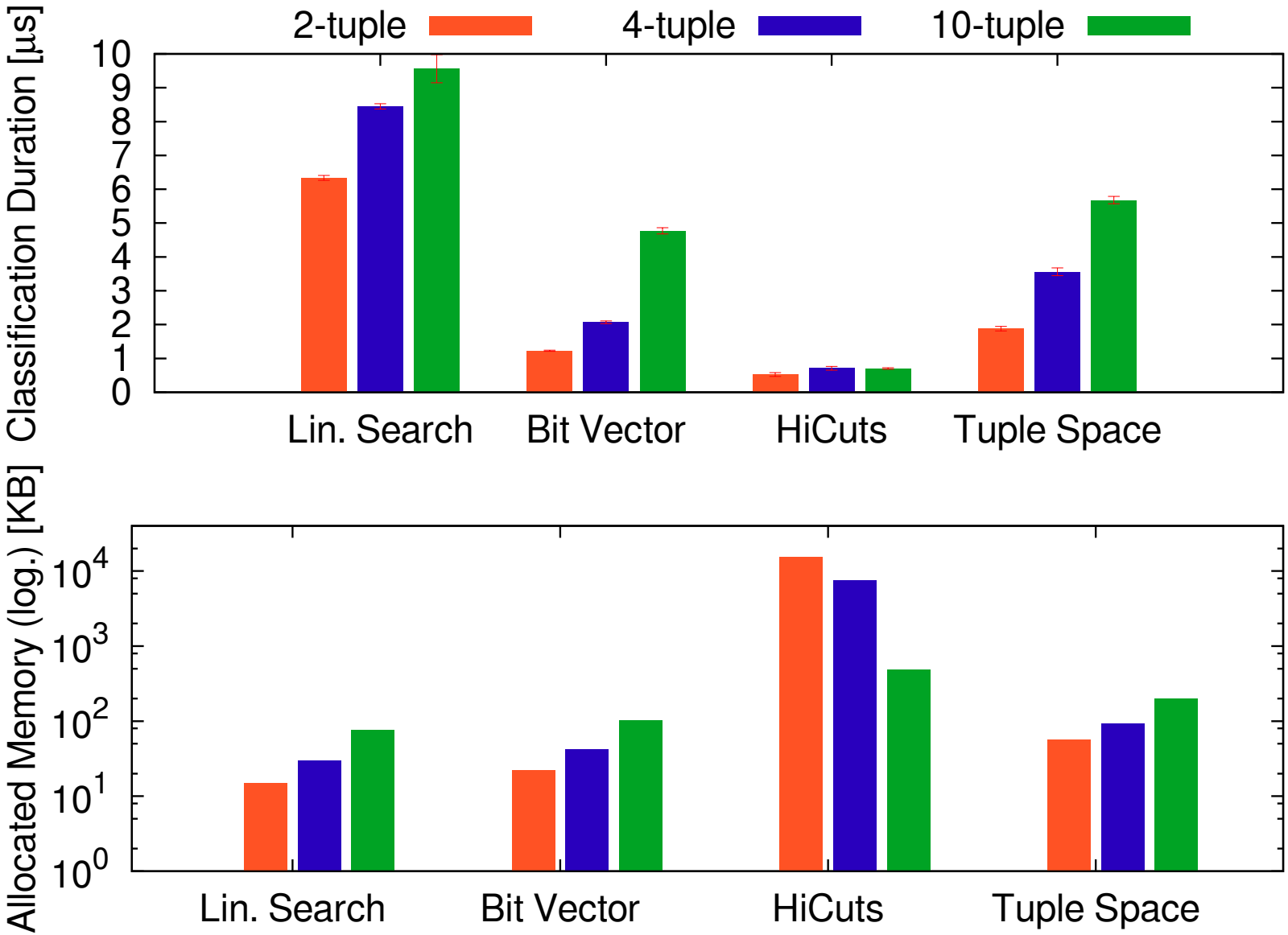


e. g., `fw1_4000`: 449 wildcard checks, `acl1_4000`: 22 wildcard checks

VARYING FIELD STRUCTURE

- Vary tuple size (2, 4, 10 fields), each field is 32 bit
- Expectations:
  - Increasing tuple size should not affect HiCuts' classification performance
  - Allocated memory should increase with increasing tuple size for all algorithms

VARYING FIELD STRUCTURE



Conclusions

RESULTS AND CONCLUSIONS

- CATE provides an integrative measurement environment for practical implementations.
- Probe effect does not change time duration relations between algorithms.
- Classification algorithms can show unexpected characteristics for specific applications.
  - Memory allocation for HiCuts' data structure is highly fluctuating.
  - Tuple Space Search performance depends on rule structure.
  - Increasing header fields can result in less memory allocation by HiCuts.

RESULTS AND CONCLUSIONS

- CATE provides an integrative measurement environment for practical implementations.

Open Access to CATE

Checkout CATE @github:

`http://gusew.github.io/cate`

structure.

- Increasing header fields can result in less memory allocation by HiCuts.