DFC: Accelerating String Pattern Matching for DPI-based NFV Applications

최병권, 채종욱, Muhammad Jamshed,
박경수, 한동수

KAIST

Trend : Popularity of Network Function Virtualization (NFV)

- NFV : Commodity hardware appliances → Software layer
  - Virtualizes entire class of network functions
  - E.g., IDS, Firewall, NAT, Load balancer, …

Home » Machine to Machine » Vodafone uses Affirmed Networks NFV Solutions to Deliver M2M Services

Vodafone uses Affirmed Networks NFV Solutions to Deliver M2M Services

dafone's

Telecommunications Mobile Network Wireless Market Research News

SDN, NFV & network virtualization market will grow at CAGR of 37% by 2020 according to market forecasts

ENTERPRISE IT/IT NETWORK

Cisco's new NFVI solution to speed up network services

IT Network | CBR Staff Writer | 11:47, February 25 2016

Pattern Matching for Deep Packet Inspection

- Looking for known patterns in packet payloads
  - String pattern matching (Fixed-length string) and Regex matching (PCRE)
  - 5K ~ 26K rules in public rule-sets for network applications

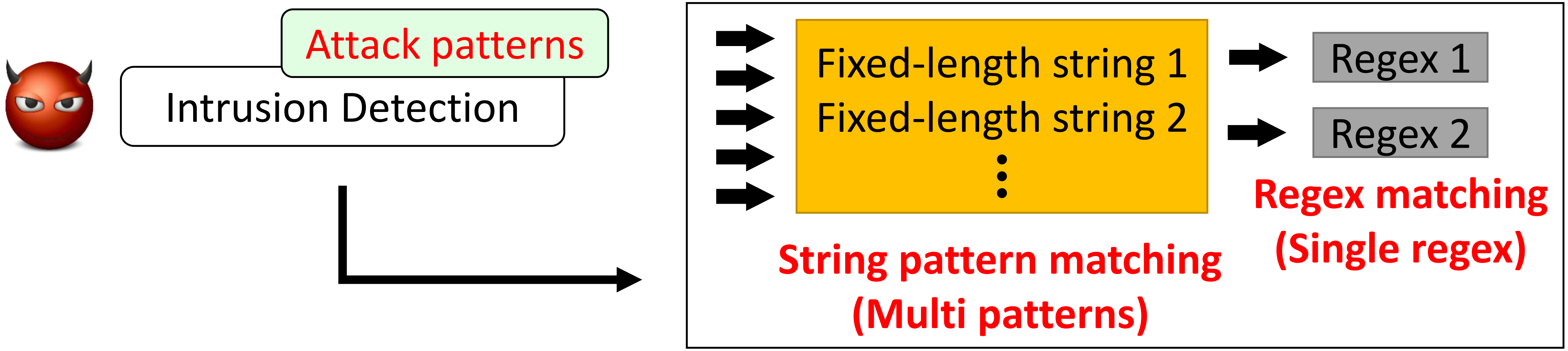
- Rule Examples
  - Rule 1 | Content: "Object" | PCRE: "/(ActiveX|Create)Object/i"
  - Rule 2 | Content: "Persits.XUpload" | PCRE: "\s*\([\x22\x27]Persits.XUpload/i"
  - Rule 3 | Content: "FieldListCtrl" | PCRE: "ACCWIZ\x2eFieldListCtrl\x2e1\x2e8/i"

String pattern matching | **Regular expression matching**

Pattern Matching for Deep Packet Inspection

- Looking for known patterns in packet payloads
  - String pattern matching (Fixed-length string) and Regex matching (PCRE)
  - 5K ~ 26K rules in public rule-sets for network applications

- Network applications using pattern matching



Pattern Matching for Deep Packet Inspection

- Looking for known patterns in packet payloads
  - String pattern matching (Fixed-length string) and Regex matching (PCRE)
  - 5K ~ 26K rules in public rule-sets for network applications

- Network applications using pattern matching

Attack patterns

Intrusion Detection

Attack patterns

Web Application Firewall

Banned words

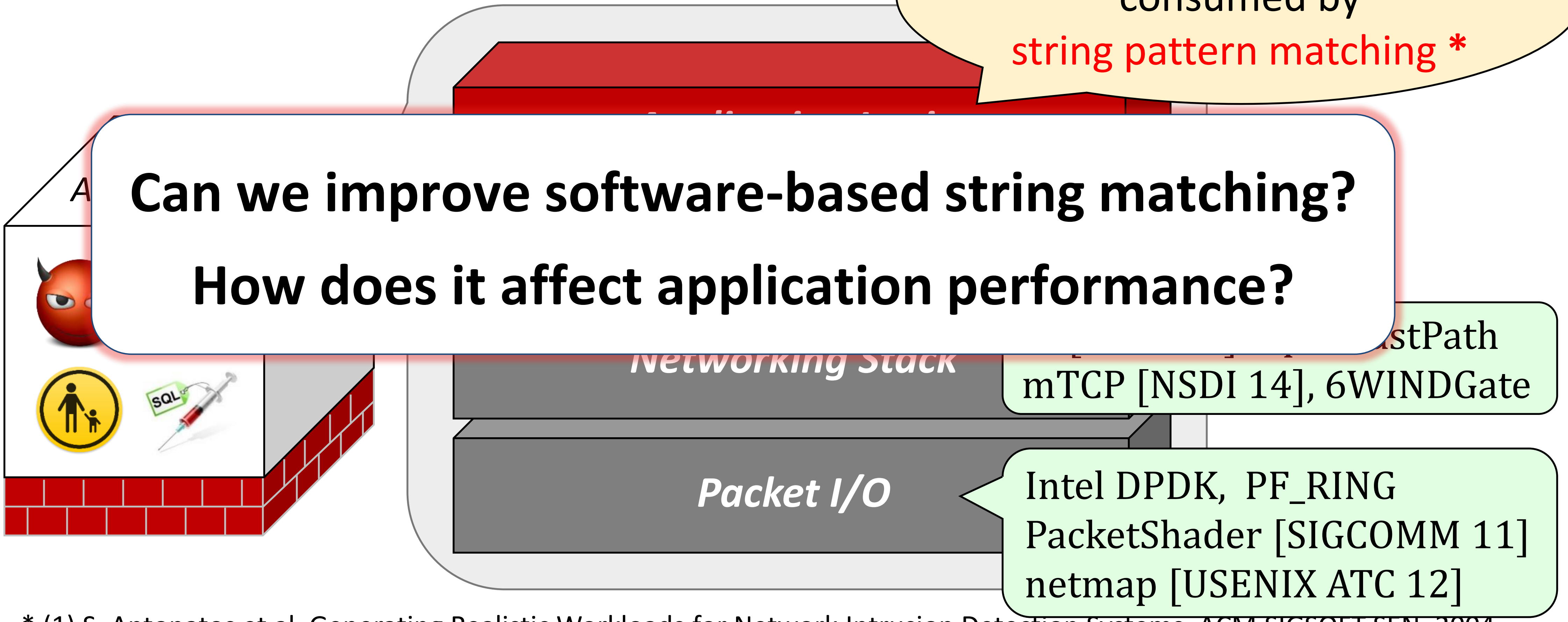
Parental Filtering

Watermark

Exfiltration Detection

However, String Pattern Matching is Performance Bott

70-80% of CPU cycles consumed by string pattern matching *



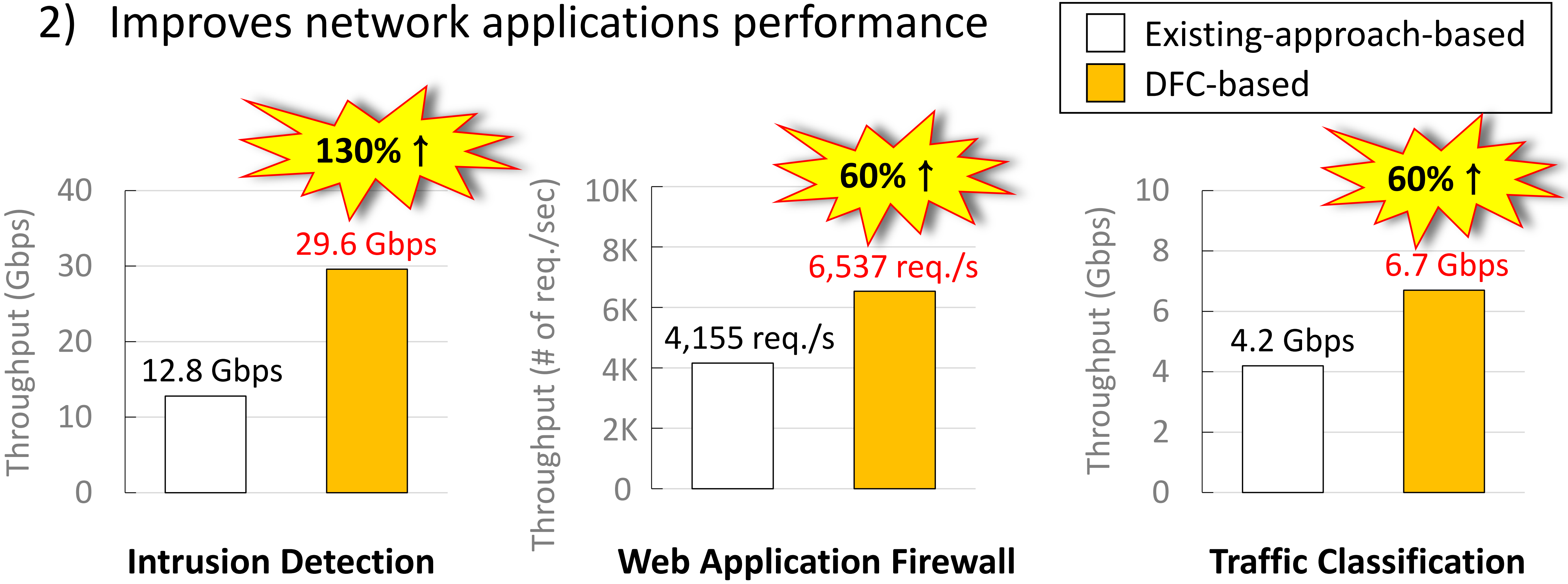
Can we improve software-based string matching?

How does it affect application performance?

* (1) S. Antonatos et al. Generating Realistic Workloads for Network Intrusion Detection Systems. ACM SIGSOFT SEN, 2004.
(2) M. A. Jamshed et al. Kargus: A Highly-scalable Software-based Intrusion Detection System. ACM CCS, 2012.
(3) Chris Ueland. Scaling CloudFlare's massive WAF. http://www.scalescale.com/scaling-cloudflaresmassive-waf/

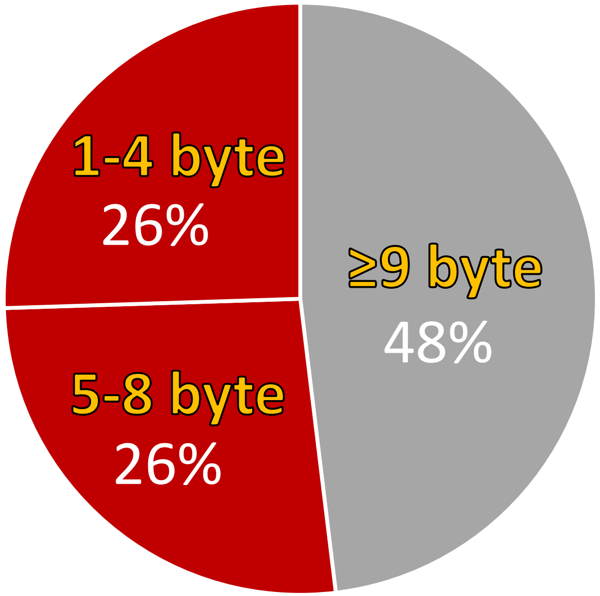
DFC: High-Speed String Matching

1) Outperforms state-of-the-art algorithm by a factor of up to 2.4
2) Improves network applications performance



Three Requirements of String Matching

- Support exact matching
  - As opposed to false positives
- Handle short and variable size patterns efficiently
  - 52% of patterns are short (< 9 byte).
- Provide efficient online lookup against a stream of data (e.g., network traffic)



< Pattern length distribution >

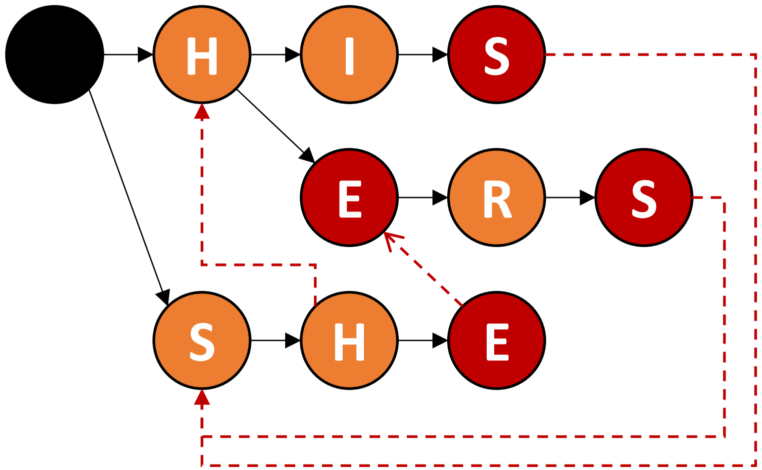
* Commercial pattern sets of IDS & Web Firewall (ET-Pro, Snort VRT, OWASP ModSecurity CRS)

Limitations of Existing Approaches

- Aho-Corasick (AC)
  - Widely used by Suricata, Snort, CloudFlare, …
  - Constructs a finite state machine from patterns
  - Locates all occurrences of any patterns using the state machine

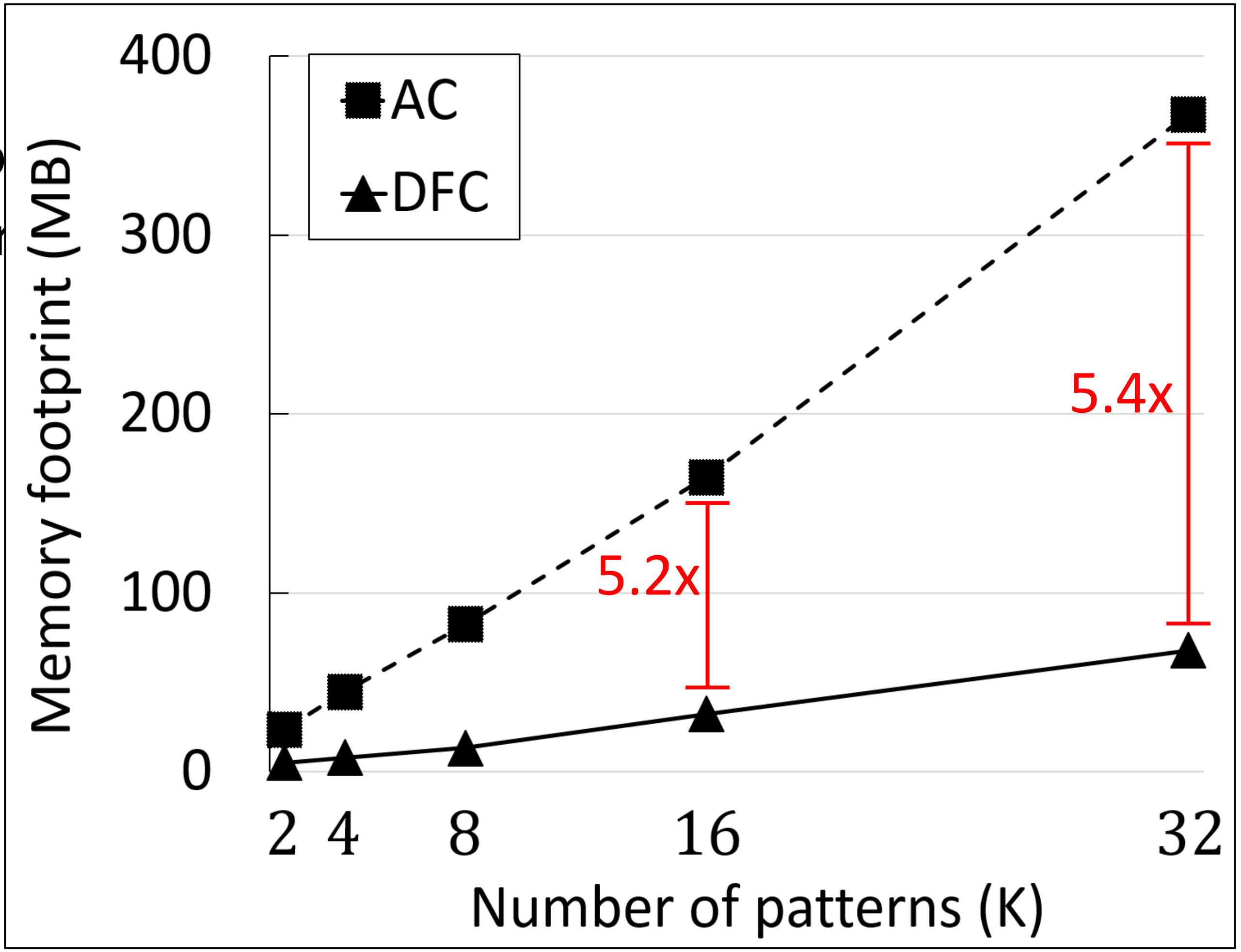
*** Example**

- Patterns: HIS | HERS | HE | SHE
- Input text : FINISHED
- Result: **SHE** **HE**



Limitations of Existing Approaches

- Aho-Corasick (AC)
  - Widely used by Suricata, Snort, Clou
  - Constructs a finite state machine fro
  - Locates all occurrences of any patter
- Limitations of AC
  - State machine is very large.
  - Working set ≫ CPU cache size
  - Instruction throughput is slow.



Limitations of Existing Approaches (Cont.)

- Heuristic-based approach (Boyer-Moore, Wu-Manber, …)
  - Advances window by multiple characters using “bad character” and “good suffix”
  - Not effective with short and variable size patterns
  - Hard to leverage instruction-level pipelining

- Hashing-based approach (Feed-forward Bloom filters (FFBF), …)
  - Compares hash of text block with hash of pattern
  - Requires expensive hash computations (2.5X more instructions than DFC)
  - Not effective with short and variable size patterns
  - Induces false positives

DFC: Design Goal

- Overcomes the limitations of existing approaches
  - Consumes small memory
  - Works efficiently with short and variable size patterns
  - Delivers high instruction-level parallelism

- Works efficiently even in worst case
  - Worst case where all packets contain attack patterns

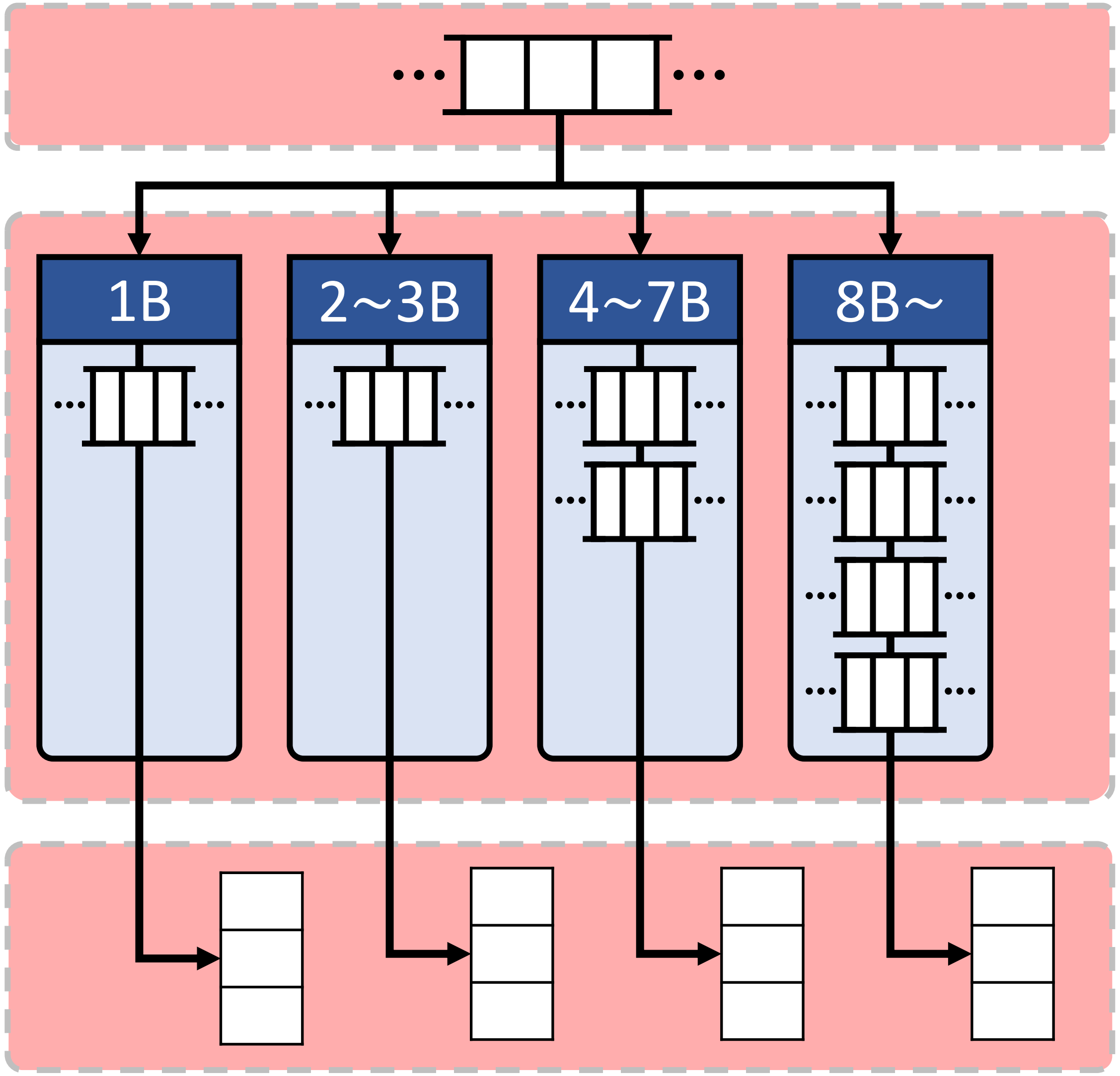
DFC: Overview

- Exploits a simple and efficient primitive
  - Used as a key building block of DFC
  - Requires small number of operations and memory lookups
  - Filters out innocent windows of input text

- Progressively eliminates false positives
  - Handles each pattern in a different way in terms of pattern length

- Verifies exact matching
  - Exploits hash tables

DFC: Component Overview



- Initial Filtering
  - Uses an efficient primitive “Direct filter”
  - Eliminates innocent windows of input text comparing few bytes (2∼3 byte)
- Progressive Filtering
  - Eliminates innocent windows further
  - Determines lengths of patterns that window might match
  - Applies additional filtering proportional to the lengths
- Verification
  - Verifies whether exact match is generated

DFC: Initial Filtering

- Uses a single Direct filter

1) No data dependency
(Instruction parallelism ↑)

Example pattern:

attack
athlete
author

01100100 01100101

Packet

py/attack/try-20

2) 2 SHIFTs and 1 AND
+
1 memory reference

No pa
beginning with

0 0 0 … 0 1

dc dd de as at au

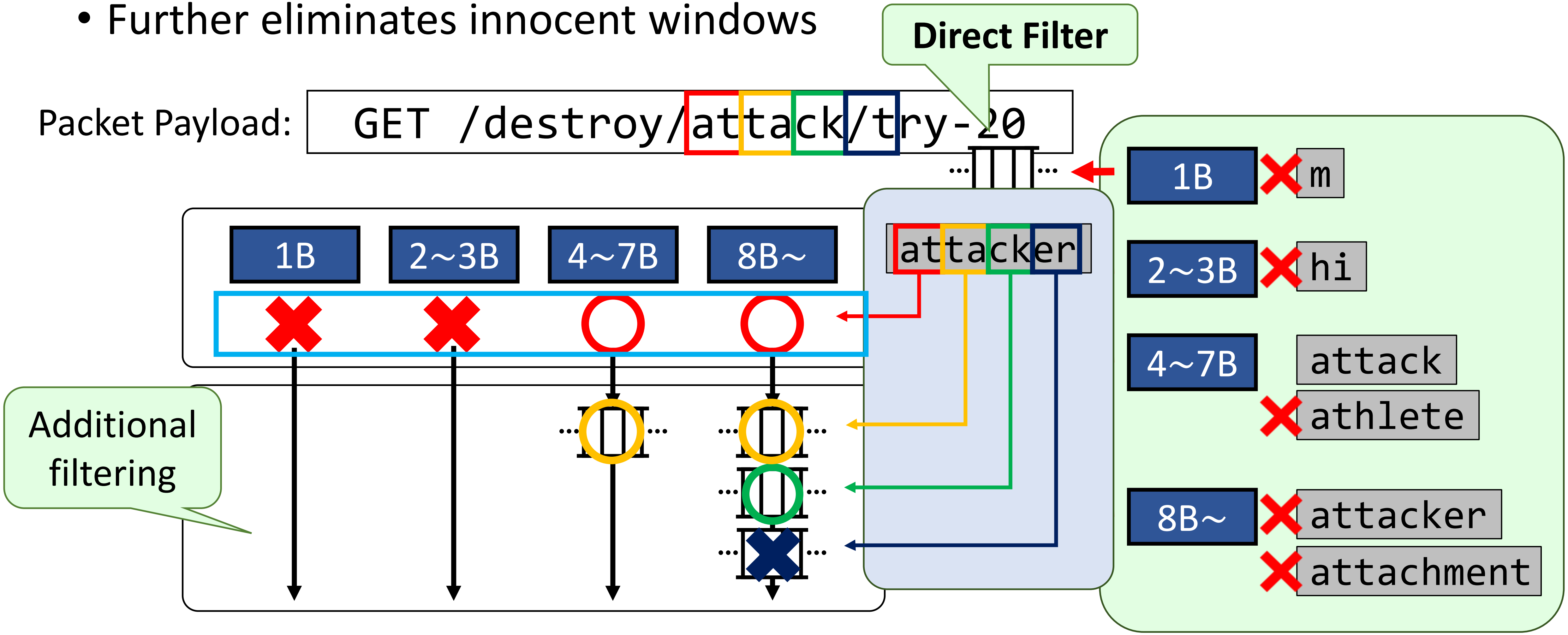
Further inspection

94% of windows are filtered out.

3) 2 byte → 2^{16} = 65536 = 8KB

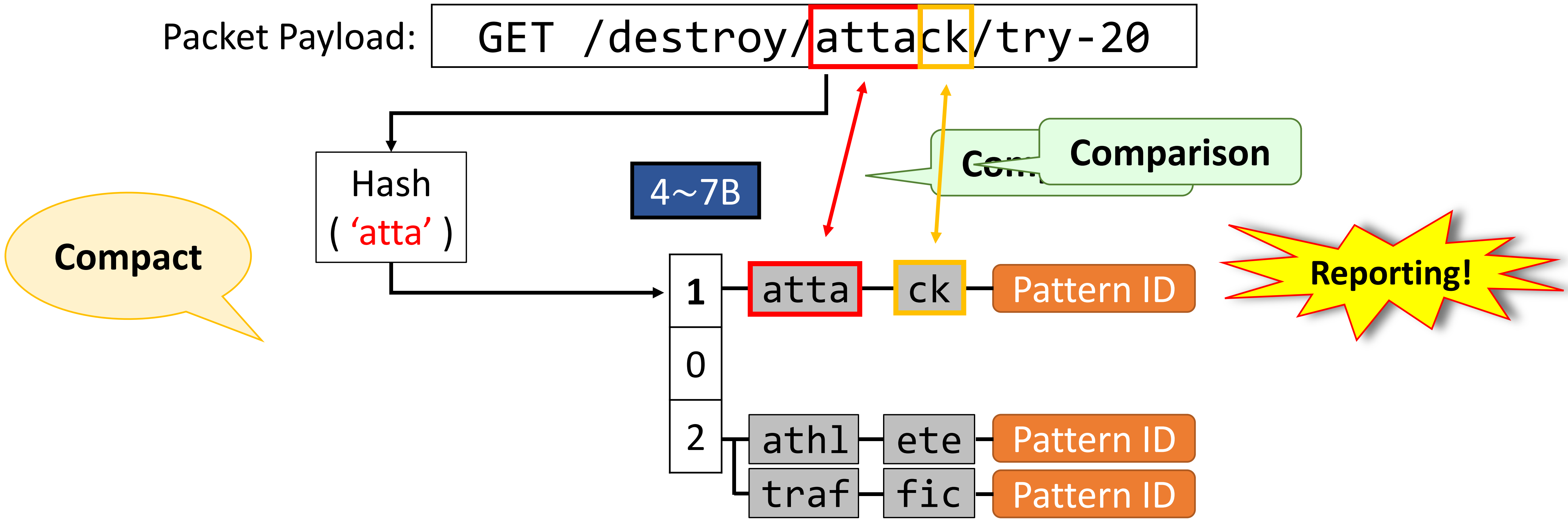
DFC: Progressive Filtering

- Further eliminates innocent windows

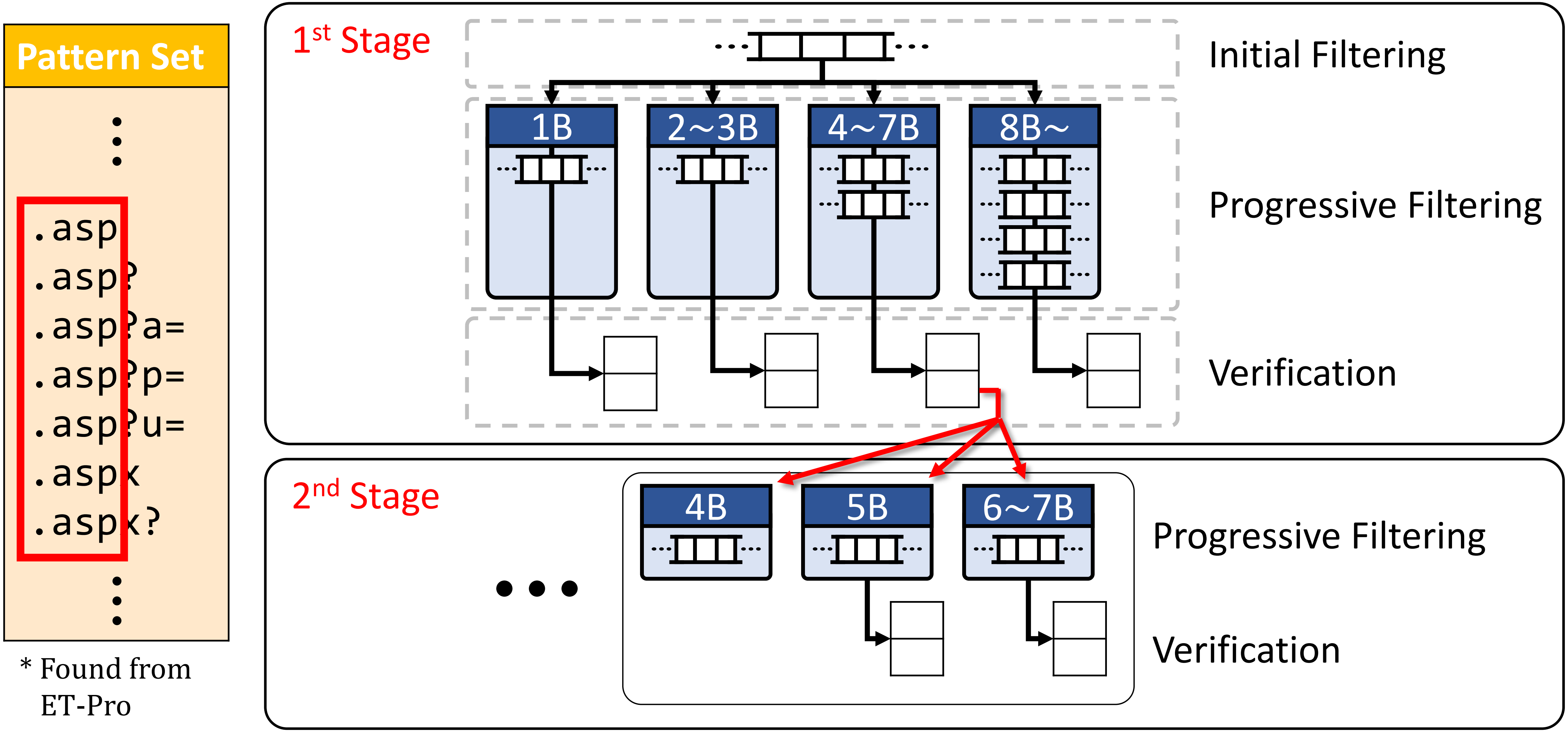


DFC: Verification

- Exact matching : (100 – 94%) * (100 – up to 84%) = only 4%!



DFC: Two-Stage Hierarchical Design

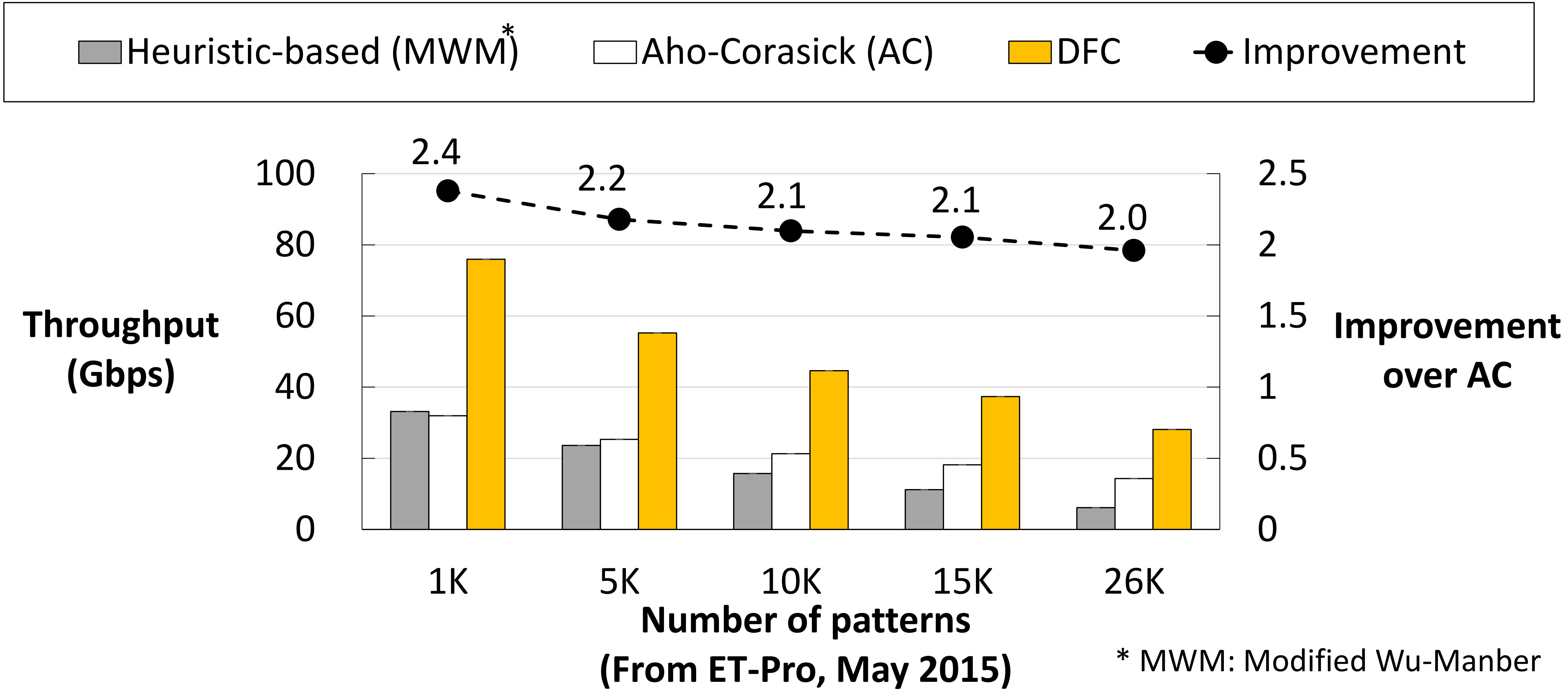


Evaluation

- Two questions
  1) Can we improve software-based string matching?
  2) How does it affect application performance?

- Machine Specification & Workload
  - Intel Xeon E5-2690 (16 cores, 20MB for L3 cache)
  - 128 GB of RAM
  - Intel®Compilers (icc)
  - Using real traffic trace from ISP in south Korea

Standalone Benchmark (1/2) – Average Case



* MWM: Modified Wu-Manber

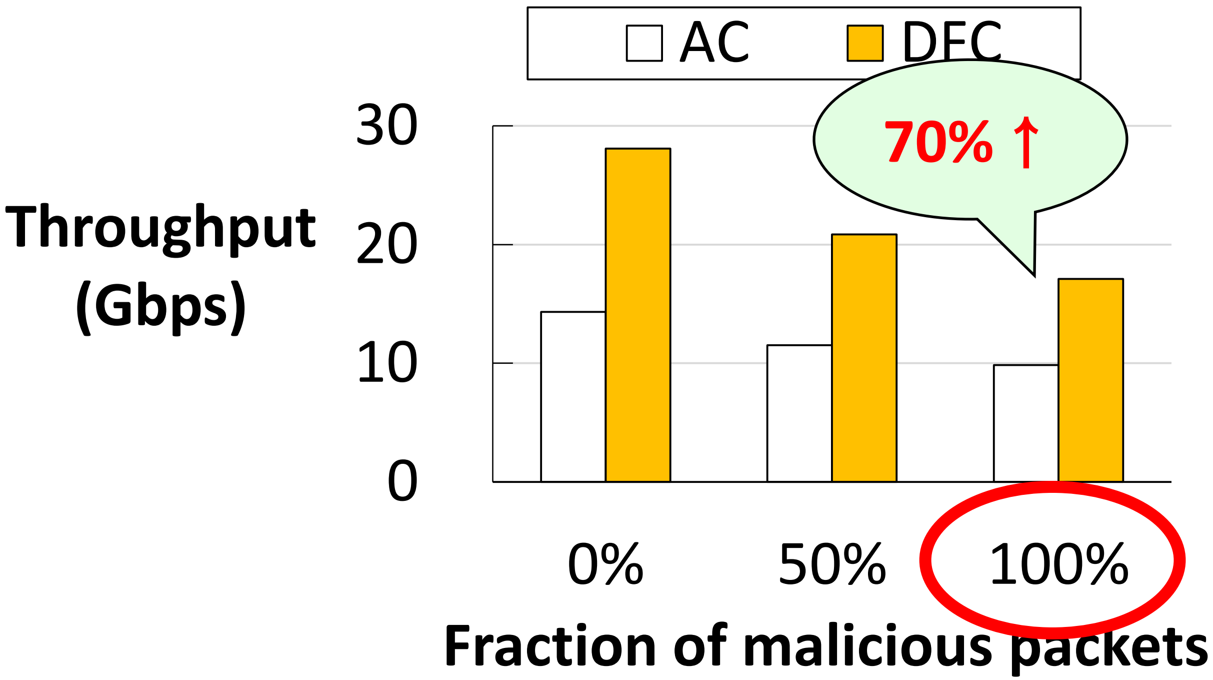
Standalone Benchmark (2/2) – Worst Case

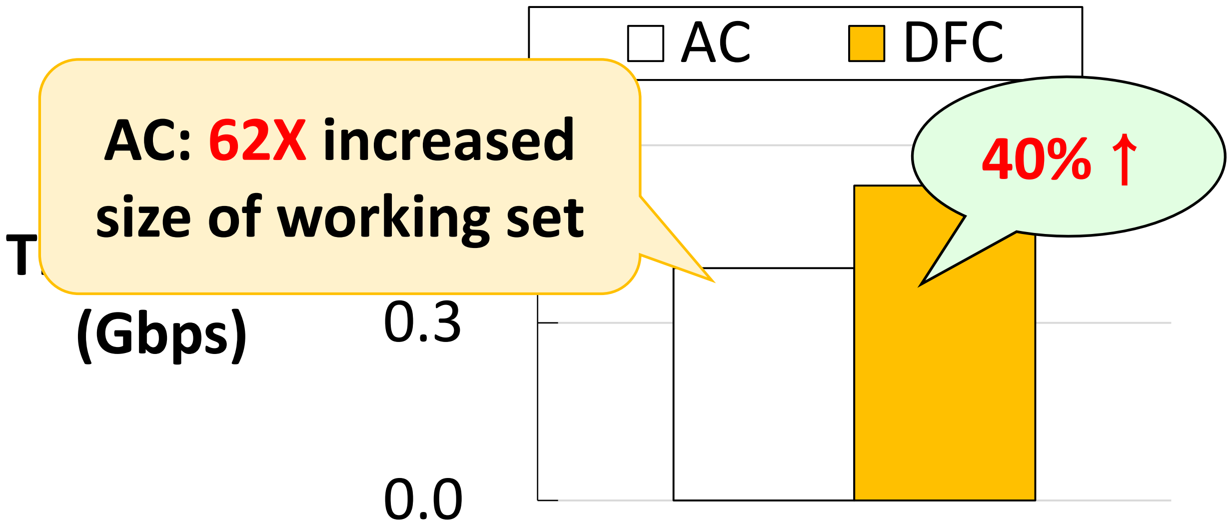
- Worst case 1 (Single pattern)

… innocent ATTACK innocent …

- Worst case 2 (Concatenated)

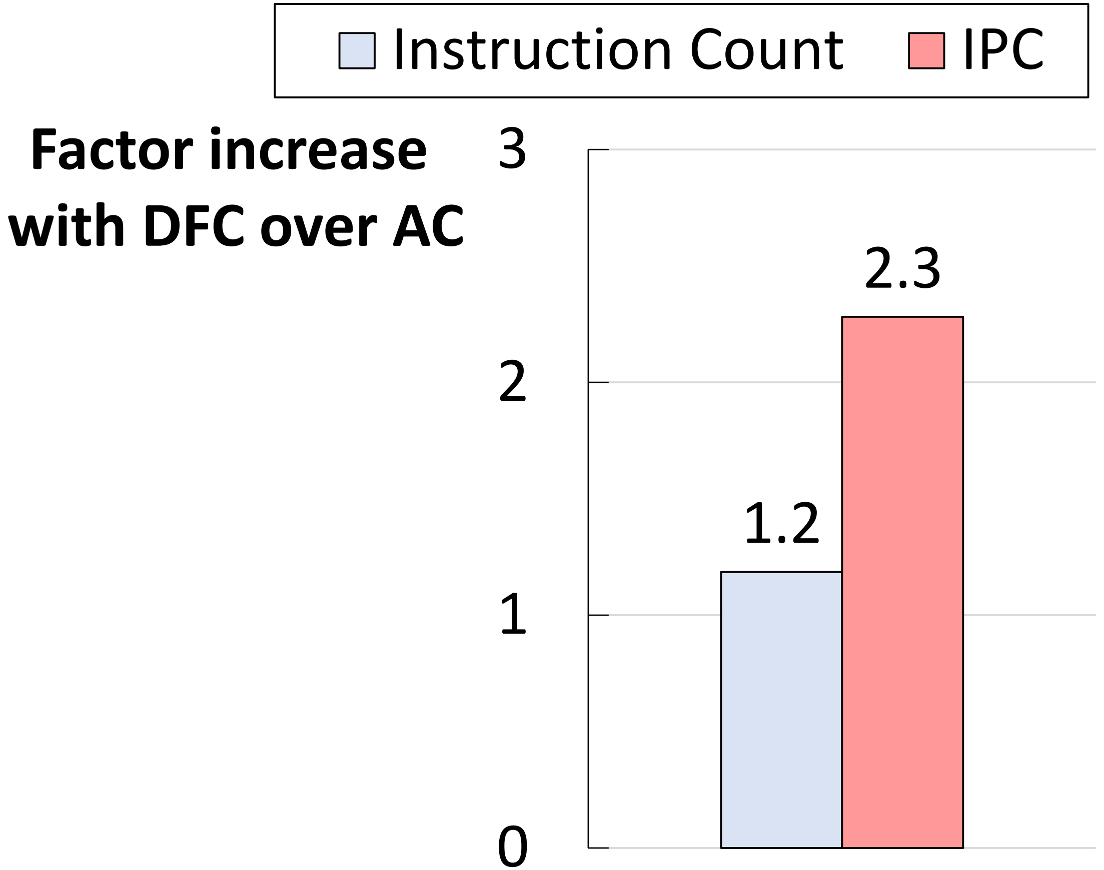
… ATTACK1 ATTACK2 ATTACK3 …

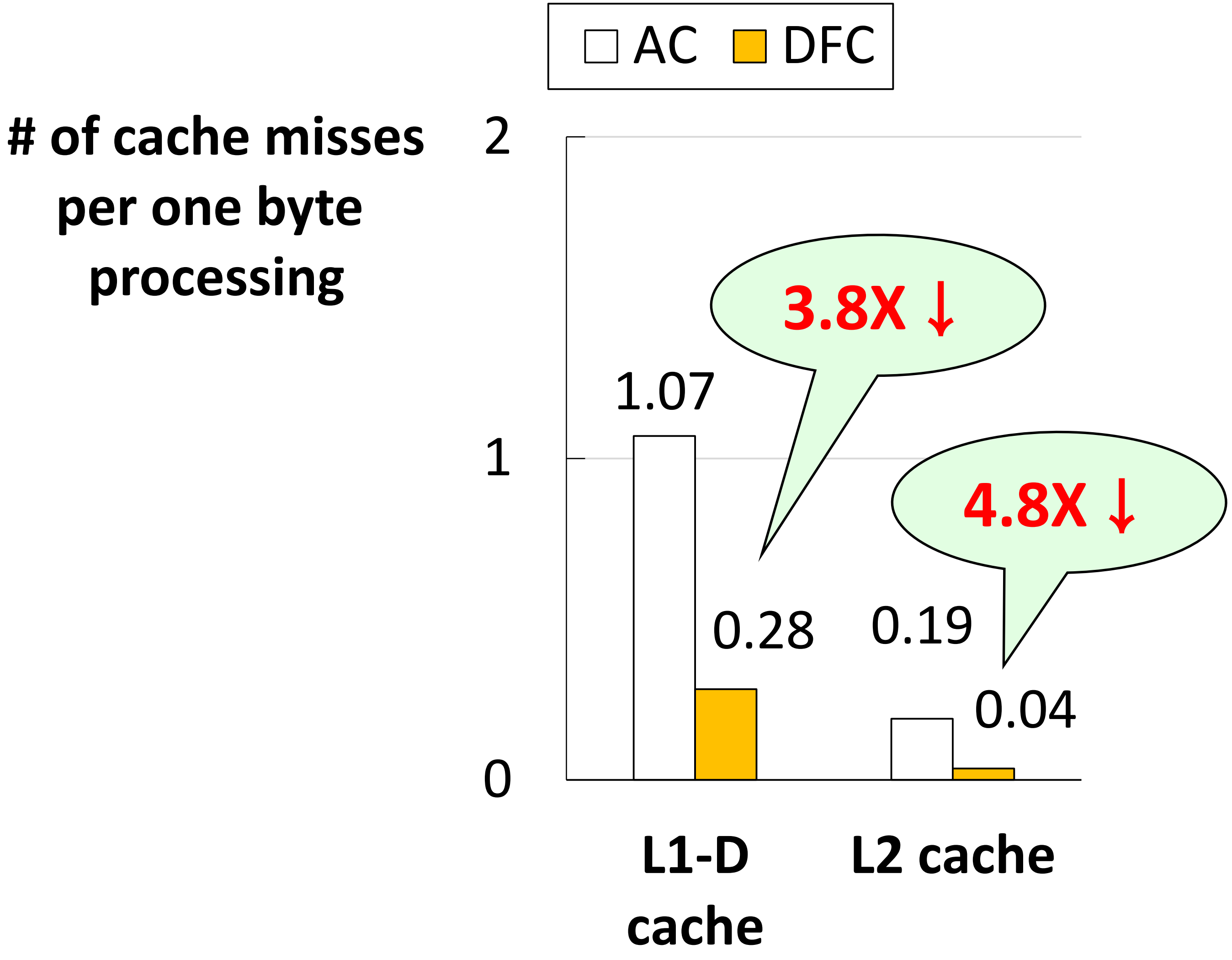




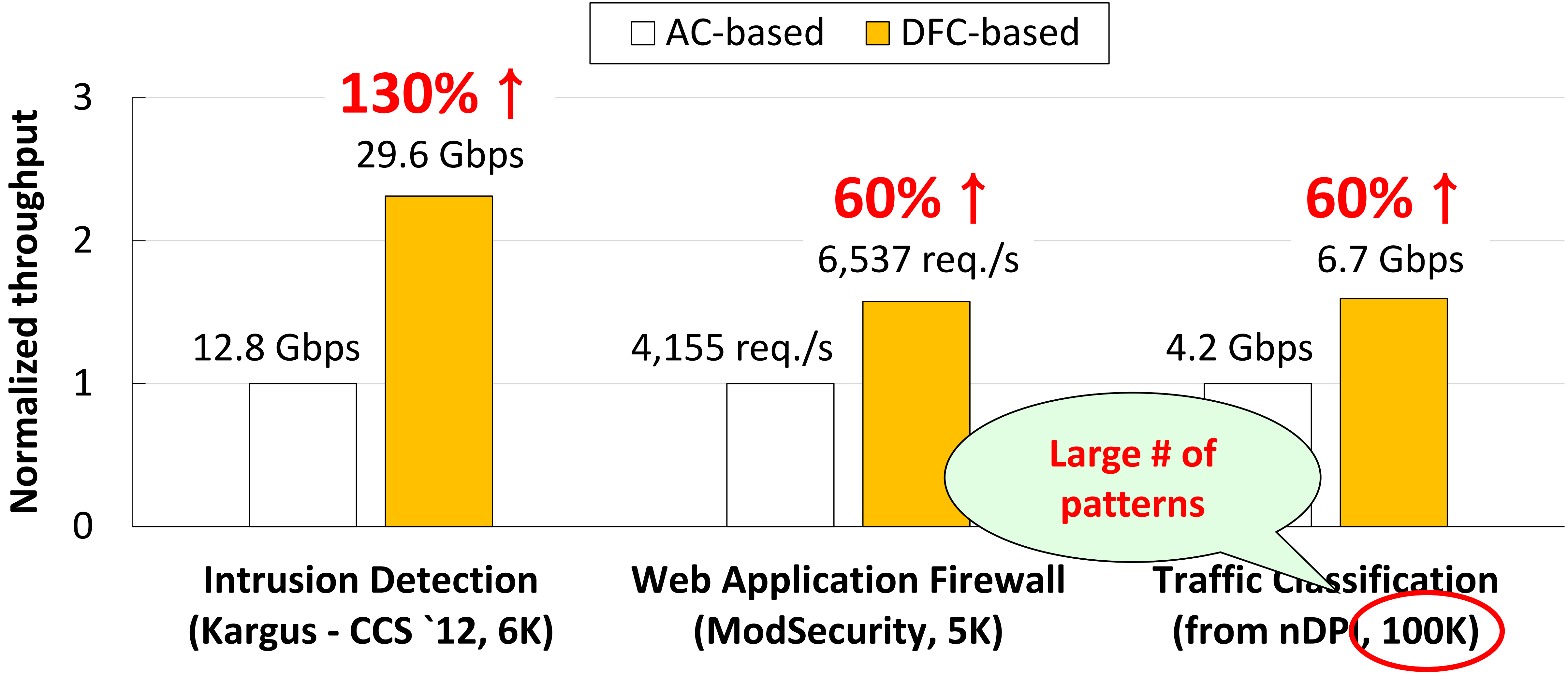
* Packet size : 1514B

Why does DFC work well?





Accelerating Network Applications using DFC



DFC: High-Speed String Pattern Matching

- **String pattern matching is a performance-critical task.**
- **DFC accelerates string pattern matching by**
  - Using small size of basic building block
  - Avoiding data dependency in critical path
- **DFC delivers 2.4X speedup compared to Aho-Corasick.**
  - 1.4X in the worst case
- **DFC improves application performance by up to 130%.**
- **Detailed information at ina.kaist.ac.kr/~dfc**