The Algorithm and Framework Designs and Optimizations for Scalable Automata Processing on HPC Platforms

Xiaodong Yu[1], Michela Becchi[2] (Master's advisor), and Danfeng (Daphne) Yao[1] (PhD advisor)

1. Dept. of Computer Science, Virginia Tech, USA {xdyu, danfeng}@vt.edu 2. Dept. of ECE, North Carolina State University, USA mbecchi@ncsu.edu

Introduction and Problem Statement

- Automata-based processing has been applied to diverse areas as the core of the searching engine:
  - network security, NLP, data mining, bioinformatics.
- There are two primary representations of automata:
  - NFA: has excellent spatial efficiency, but may require expensive per-character processing.
  - DFA: offers limited per-character processing at the cost of a large automaton (state explosion (SE) issue).
- Current SE issue solutions suffer from some limitations:
  - only work on small and simple datasets.
  - have prohibitive worst-case processing time.
  - can't guarantee fully functional equivalence.
- The searching engine of NIDS using conventional NFA/DFA can't directly handle the out-of-order packets.
  - it has to buffer the packets and then process flow only after reordering and reassembling the packets.
  - this is vulnerable to denial-of-service (DoS) attacks.
- Automata processing is inherently difficult to be parallelized especially on the many-core (e.g., GPU) devices due to:
  - tight dependencies between every two computation steps.
  - unstructured memory-access pattern.
- The newly proposed reconfigurable device – Micron's Automata Processor (AP) suffers from two problems:
  - Current APIs require the domain end-users to have insights of both AP architecture and automata processing.
  - Requires vast reconfiguring cost for a large dataset.
- It's unclear the advantages and disadvantages of different automata processing accelerators and the innovation space
  - CPU, GPU, FPGA, Automata Processor, etc

Contribution Summary

Algorithm Level:

A1. JFA: an FA variant that uses state variables to avoid SE, and is functionally equivalent to the corresponding DFA [2]

A2. O^{3}FA: an automata-based DPI engine for RegEx matching on out-of-order packets without requiring flow reassembly [3]

Implementation Level (frameworks):

F1. A GPU-based automata processing framework that supports different automata representations and provides various architecture-aware optimization techniques [1]

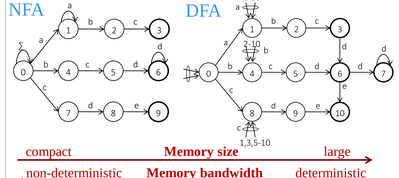
F2. Robotomata: a framework for auto APM-automata codes generation and optimization and fast reconfiguration on AP [4]

F3. A toolchain that allows the apple-to-apple comparison of NFA acceleration engines on different platforms [5]

Background Knowledge

Finite Automata

RegEx: (1) *a+bc* (2) *bcd+* (3) *cde*

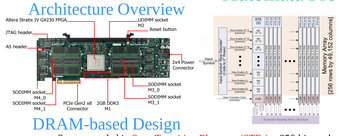


State Explosion

In practice, a desktop machine equipped with 2GB of memory will report an out-of-memory error after generating *1,000,000 DFA states*, which would be expected for a typical *100-state NFA*.

Automata Processor

Architecture Overview

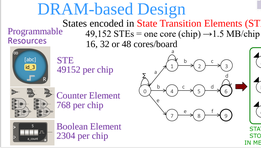


DRAM-based Design

States encoded in State Transition Elements (STEs) = 256-bit masks

49,152 STEs = one core (chip) →1.5 MB/chip

16, 32 or 48 cores/board



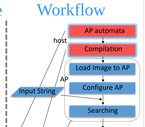
Programming Interface

ANML: Automata Network Markup Language

- has programmability issue
- very low-level and complex, requires programmers to manipulate STEs

Sample ANML codes

Workflow

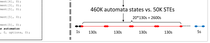


It has scalability issue.

For a large dataset, it requires multiple rounds of re-compilation, and each round takes a long time compared to the searching time.

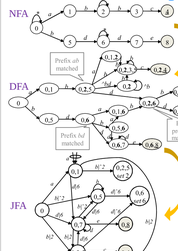
Example: *alu* dataset in BLAST

460K automata states vs. 50K STEs



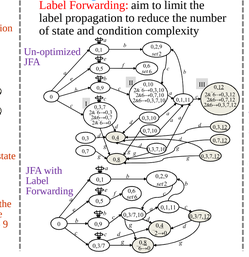
A1. JFA

Basic JFA Design



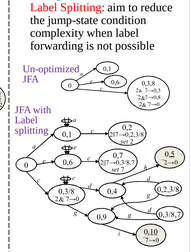
Optimization 1

Label Forwarding: aim to limit the label propagation to reduce the number of state and condition complexity



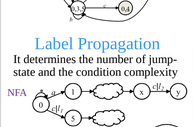
Optimization 2

Label Splitting: aim to reduce the jump-state condition complexity when label forwarding is not possible



Label Propagation

It determines the number of jump-state and the condition complexity



Experiment Results

The characteristics of JFA compared to NFA, DFA, Hybrid-FA (for Snort datasets)

Dataset	NFA		Small DFA (<100k states)		Large DFA		Hybrid-FA				JFA		
	# states	# trans.	# DFA	overall # states	overall # trans.	# DFA	overall # states	overall # trans.	# head-states	# tail-states	# tail-FAs	# border states	# states	# jump-states	# trans.
Backdoor	2,968	~41k	9	~148k	~1.354k	7	~172k	~1.527k	3,731	924	72	317	126,917	5,615	~2,008k
Spyware	4,690	~53k	11	~366k	~2.024k	7	3,880k	~15.200k	1,636	3,690	45	56	18,867	1,358	~71k
Dotstar.05	14,978	~70k	8	~268k	~1.596k	5	~604k	~3.421k	75,540	5,022	105	2116	36,537	162	~122k
Dotstar.1	14,688	~79k	10	~677k	~3.414k	7	2,346k	10,386k	65,536	6,187	116	2700	35,876	384	~121k
Dotstar.2	14,249	~88k	21	~736k	~3.498k	12	6,808k	31,471k	27,904	6,965	149	1995	34,762	1,616	~123k

A2. O^{3}FA

Out-of-Order Packets issue in NIDS

In Network Intrusion Detection Systems (NIDS), input streams have size at MB/GB level, whereas single packet payload is only 64KB

- divided to multiple substreams→ carried by different packets

Out-of-Order Packets issue: packets arrive in a random order, so match can escape from the NIDS due to cross-packet matching

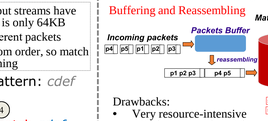
input stream: *abcdefgh* malicious pattern: *cdef*

Normal order: *abcdefgh* → *abcd* → *efgh* match *cdef*

Out of order: *abcdefgh* → *efgh* → *abcd* no match

Traditional Solution

Buffering and Reassembling



Drawbacks:

- Very resource-intensive
- Attackers can exhaust the packet buffer capacity to cause DoS

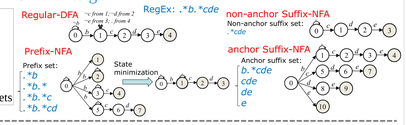
O^{3}FA Design

Regular DFA

- Constructed based on the given RegEx set
- Perform RegEx matching

Supporting-FAs

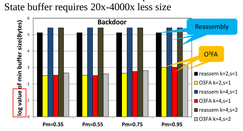
- Consist of both prefix-FA and suffix-FA
- Can be either in NFA or in DFA format
- Constructed based on prefix/suffix of RegEx
- Detect and record segments of incoming packets



Evaluations

Buffer size savings:

- O^{3}FA states buffer vs. traditional packets buffer
- State buffer requires 20x-4000x less size



Bandwidth overhead:

Ratio(%)= traversed NFA states # / input characters #

Dataset	k=2, s=1			
	P_{avg}=0.35	P_{avg}=0.55	P_{avg}=0.75	P_{avg}=0.95
Backdoor	0.0144	0.0202	0.0278	0.0349
Spyware	0.0590	0.1002	0.1163	0.1286
Dotstar0.05	0.0804	0.0838	0.1173	0.2595
Dotstar0.1	0.0526	0.0715	0.1054	0.2610
Dotstar0.2	0.0363	0.0611	0.1142	0.2977
Range0.5	0.0973	0.1015	0.2170	0.2238
Range1	0.0638	0.1180	0.2181	0.3027
E-M	0.0391	0.0627	0.1460	0.3140

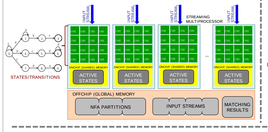
Traversal overhead:

Ratio(%)= O3FA traversal overhead / Traversed state # in reassembly methods

Dataset	k=2, s=1			
	P_{avg}=0.35	P_{avg}=0.55	P_{avg}=0.75	P_{avg}=0.95
Backdoor	0.0114	0.0102	0.0346	0.3277
Spyware	0.0059	0.0058	0.1333	2.4362
Dotstar0.05	0.0103	0.0076	0.2645	1.0132
Dotstar0.1	0.0041	0.0129	0.0116	2.2671
Dotstar0.2	0.0083	0.0092	0.0160	3.4655
Range0.5	0.0007	0.0011	0.0032	0.0137
Range1	0.0006	0.0011	0.0033	0.0123
E-M	0.0006	0.0011	0.0033	0.0168

F1. GPU-based Automata Processing

General GPU-based Design



GPU-based NFA

Baseline design

Optimization #1:

- Sort transition vector by source state
- Record max active state (atomic required)

Optimization #2:

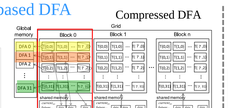
- Partition states into compibility groups

GPU-based DFA

RegEx Partitioning

- Avoid polynomial state explosion
- Increased memory bandwidth requirements

Compressed DFA



Bi-dimensional thread-blocks:

- x-dimension: DFA
- y-dimension: labeled transition (all must be queried)

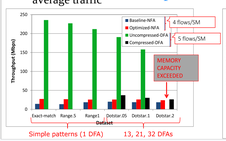
Max: 8 labeled transitions state for more regular execution

Software-managed Cache

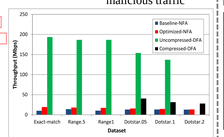
Using shared memory as cache to better leverage the temporal locality of state traversal

Experiment Results

average traffic



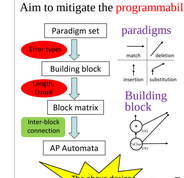
malicious traffic



F2. ROBOTOMATA (Framework for Automta Processor)

Paradigm-based Hierarchical Construction

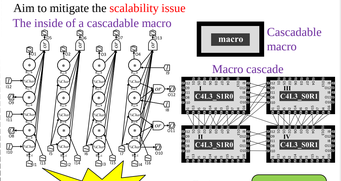
Aim to mitigate the programmability issue



Cascadable Macros

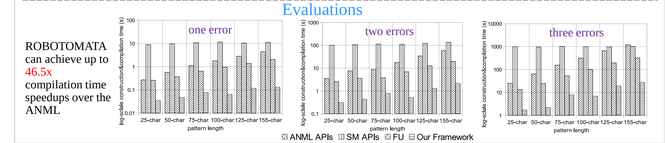
Aim to mitigate the scalability issue

The inside of a cascadable macro



Evaluations

ROBOTOMATA can achieve up to 46.5x compilation time speedups over the ANML



F3. Platform Comparison Toolchain

Goal of the toolchain

Apple-to-apple comparison between automata processing accelerators

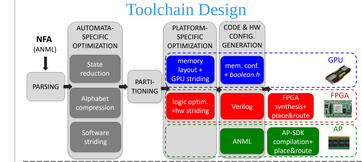
- 3 platforms: GPU, AP, FPGA
- non-deterministic finite automata (NFA)

Large datasets (million states)

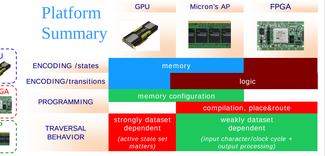
Dimensions of Comparison

- Resource requirements
- Traversal throughput
- Preprocessing time

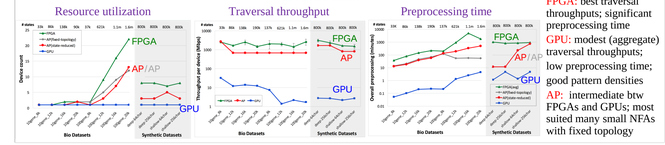
Toolchain Design



Platform Summary



Evaluations



FPGA: best traversal throughputs; significant preprocessing time

GPU: modest (aggregate) traversal throughputs; low preprocessing time; good pattern densities

AP: intermediate btw FPGAs and GPUs; most suited many small NFAs with fixed topology

References

1. **X. Yu** and M. Becchi, "*GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space*," In ACM CF '13
2. **X. Yu**, B. Lin and M. Becchi, "*Revisiting State Blow-up: Automatically Building Augmented-FA while Preserving Functional Equivalence*, " Selected Areas in Communications, IEEE Journal on (JSAC.2014)
3. **X. Yu**, W.-c. Feng, D. Yao and M. Becchi, "*O3FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection*, " In ACM/IEEE ANCS '16
4. **X. Yu**, H. Wang, K. Hou, W.-c. Feng, "*ROBOTOMATA: A Framework for Approximate Pattern Matching of Big Data on an Automata Processor*, " In IEEE BigData '17
5. M. Nourian, X. Wang, **X. Yu**, W.-c. Feng and M. Becchi, "*Demystifying automata processing: GPUs, FPGAs or Micron's AP?*, " In ACM ICS '17