Automated Synthesis of Adversarial Workloads for Network Functions

Luis Pedrosa, Rishabh Iyer,
Arseniy Zaostrovnykh, Jonas Fietz,
Katerina Argyraki

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Network
Architecture
Laboratory

Software NFs

The good:

- The flexibility of software
- The software development cycle

The bad:

- The reliability of software
- Inconsistent performance

The ugly:

- Adversarial traffic / DoS / Slowdowns

We need better tools…

Dynamic analysis: profiling

- Reasons about known inputs
- Helps find root cause / debug
- Only as good as the inputs used

We need better tools…

Static analysis

Reasons about potential inputs in abstract

Over-approximating: WCET

Under-approximating: adversarial inputs

Latency (not to scale)

CASTAN – Cycle Approximating Symbolic Timing Analysis for NFs

Statically analyze NF

- Analyze code
- Generate PCAP file with adversarial workload

Exploit

- The CPU cache hierarchy
- Algorithmic complexity

It works!

- Increased NF latency up to 3×

Outline

Introduction

SymbEx in a Nutshell

CASTAN

Evaluation

Conclusion

SymbEx in a Nutshell

Procedure

Interpret code with symbolic values

```
01: int var = input(); // α
02: return var++; // α+1
```

J. C. King, Symbolic Execution and Program Testing, 1976

SymbEx in a Nutshell

Procedure

Interpret code with symbolic values

```
int var = input(); // α
if (var >= 0) {
  return var;
} else {
  return -var;
}
```

SymbEx in a Nutshell

Procedure

Interpret code with symbolic values

Fork execution on symbolic conditions

Keep track of path constraints

```
int var = input(); // α
if (var >= 0) {
  return var;  // α if α>=0
} else {
  return -var; // -α if α<0
}
```

SymbEx in a Nutshell

Procedure

Interpret code with symbolic values

Fork execution on symbolic conditions

Keep track of path constraints

SMT solver finds concrete inputs

```
int var = input(); // α
if (var >= 0) {
  return var;  // α if α>=0, e.g. α=0
} else {
  return -var; // -α if α<0, e.g. α=-1
}
```

SymbEx in a Nutshell

Challenges

Path Explosion!

Typically exponential **# of paths / branch**

Unbounded **with loops**

Impractical to SymbEx ***exhaustively***

SymbEx in a Nutshell

Mitigation

Can’t do everything: prioritize!

Directed Symbolic Execution

- Prioritize executing relevant paths over others
- Graph search with heuristic
- Try to reach a bug / increase coverage / etc.
- Stop SEE when satisfied (or impatient)

CASTAN

Overview

Generate adversarial NF workloads

- Packet sequence ⇒ more CPU cycles / packet

Under-approximate: not WCET

Largely automated

CASTAN

Approach

Exploits performance variation

1. **CPU cache: +DRAM accesses**

2. **Algorithmic complexity: +instructions**

3. **Hashing: reverse to expose internals**

CASTAN

Attacking the CPU Cache

Symbolic Pointers

Index into memory with packet:

`array[packet.dst_addr]`

Find packets ⇒ memory addresses ⇒ DRAM access

CPU Cache Model

Simple 1-tier model of the LLC

Models contention, associativity, write-back

Empirical contention set model

CASTAN

Attacking Algorithmic Complexity

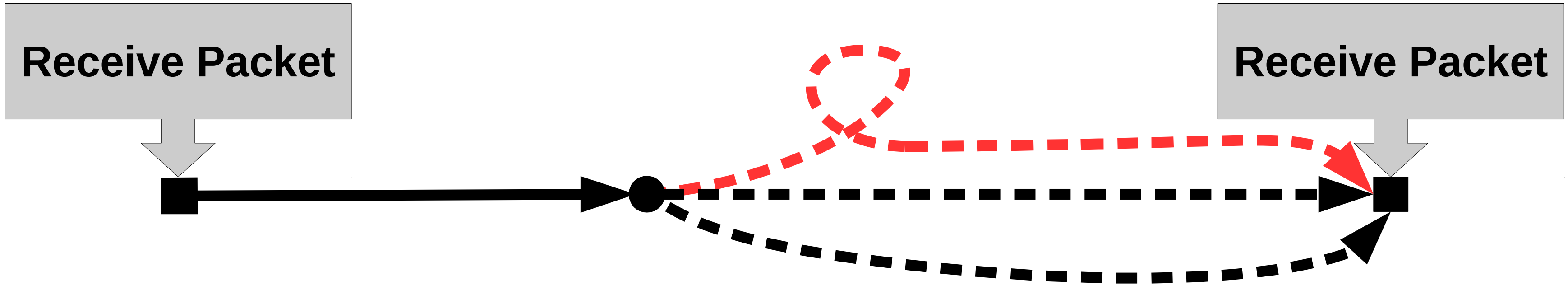
Maximize Instructions / Packet

Find packets ⇒ longer code paths

Guide SymbEx with a Heuristic

Maximize cycles w/o inducing breadth-first-search

Estimate cycles / packet



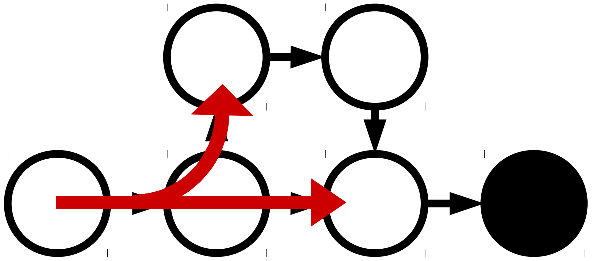
CASTAN

Attacking Algorithmic Complexity

CFG Distance Heuristic

max(successors)+cost<current>

cost = cycles conservatively assuming an L1 hit



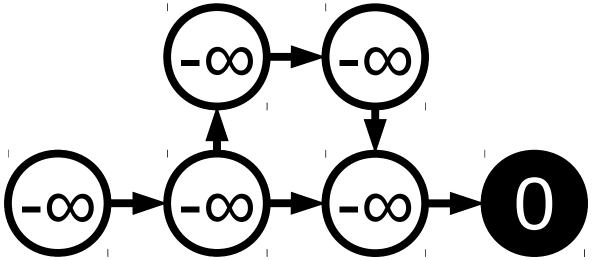
CASTAN

Attacking Algorithmic Complexity

CFG Distance Heuristic

max(successors)+cost<current>

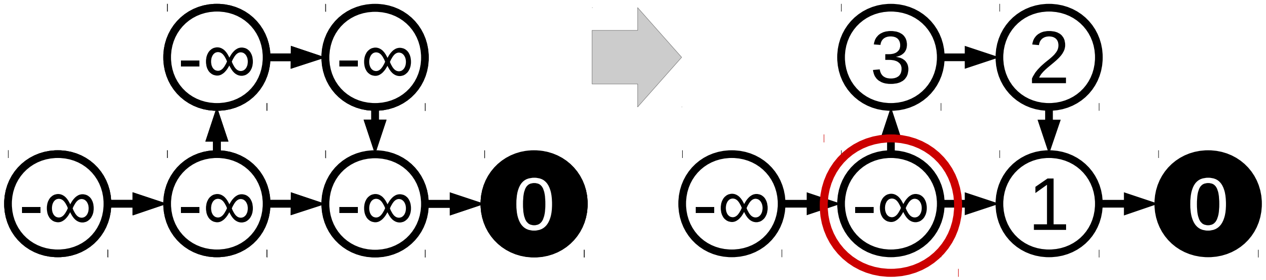
cost = cycles conservatively assuming an L1 hit



CFG Distance Heuristic

max(successors)+cost<current>

cost = cycles conservatively assuming an L1 hit



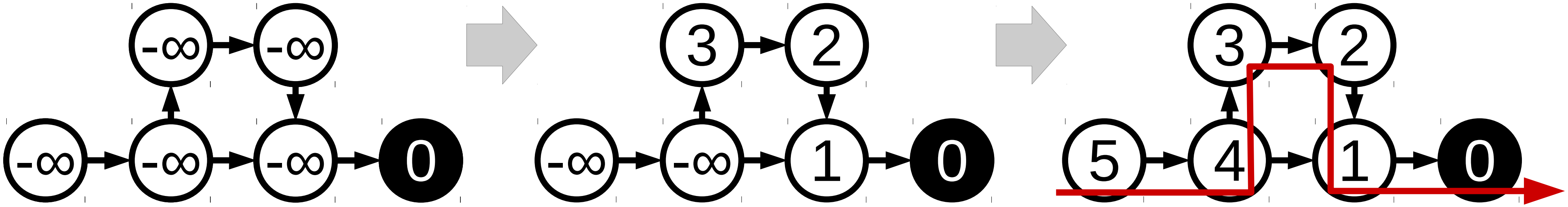
CASTAN

Attacking Algorithmic Complexity

CFG Distance Heuristic

max(successors)+cost<current>

cost = cycles conservatively assuming an L1 hit



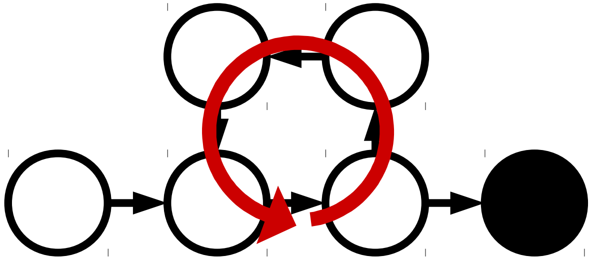
CASTAN

Attacking Algorithmic Complexity

Handling Loops

Distance vector algorithm

Limit repeats to 2 (unrolls loops once)



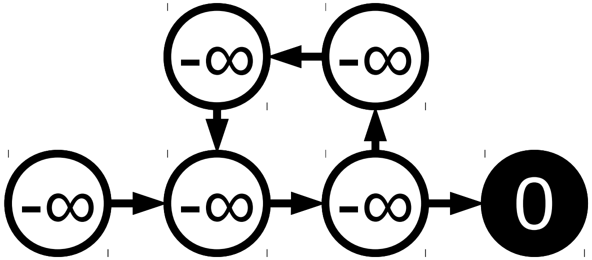
CASTAN

Attacking Algorithmic Complexity

Handling Loops

Distance vector algorithm

Limit repeats to 2 (unrolls loops once)



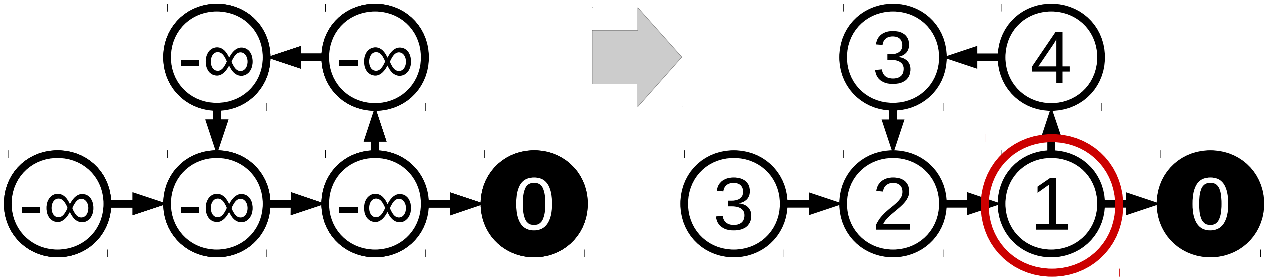
CASTAN

Attacking Algorithmic Complexity

Handling Loops

Distance vector algorithm

Limit repeats to 2 (unrolls loops once)



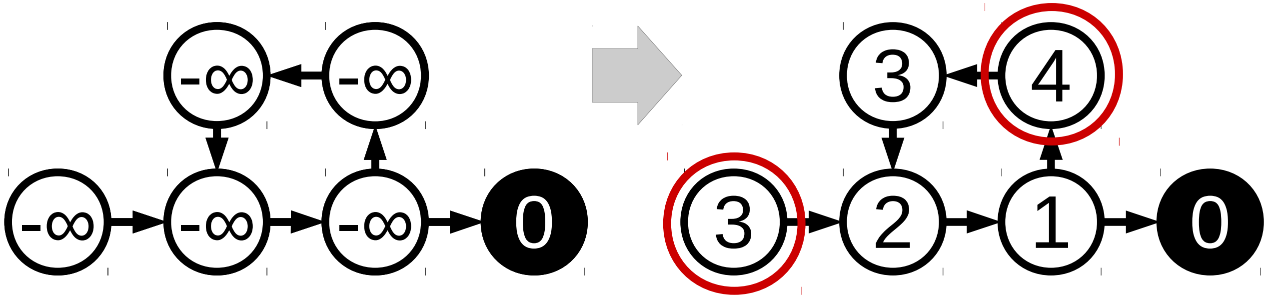
CASTAN

Attacking Algorithmic Complexity

Handling Loops

Distance vector algorithm

Limit repeats to 2 (unrolls loops once)



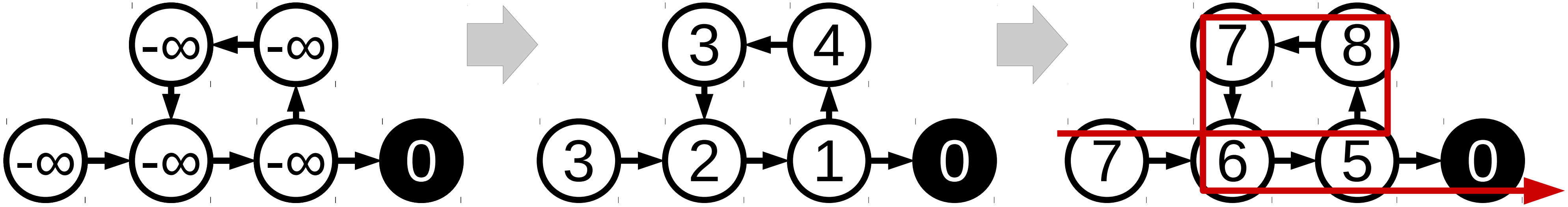
CASTAN

Attacking Algorithmic Complexity

Handling Loops

Distance vector algorithm

Limit repeats to 2 (unrolls loops once)



CASTAN

Handling Hash Functions

SymbExing hash functions is hard

- Complex expression / Path explosion
- Reason about hash value, without computing it?

CASTAN

Handling Hash Functions

SymbExing hash functions is hard

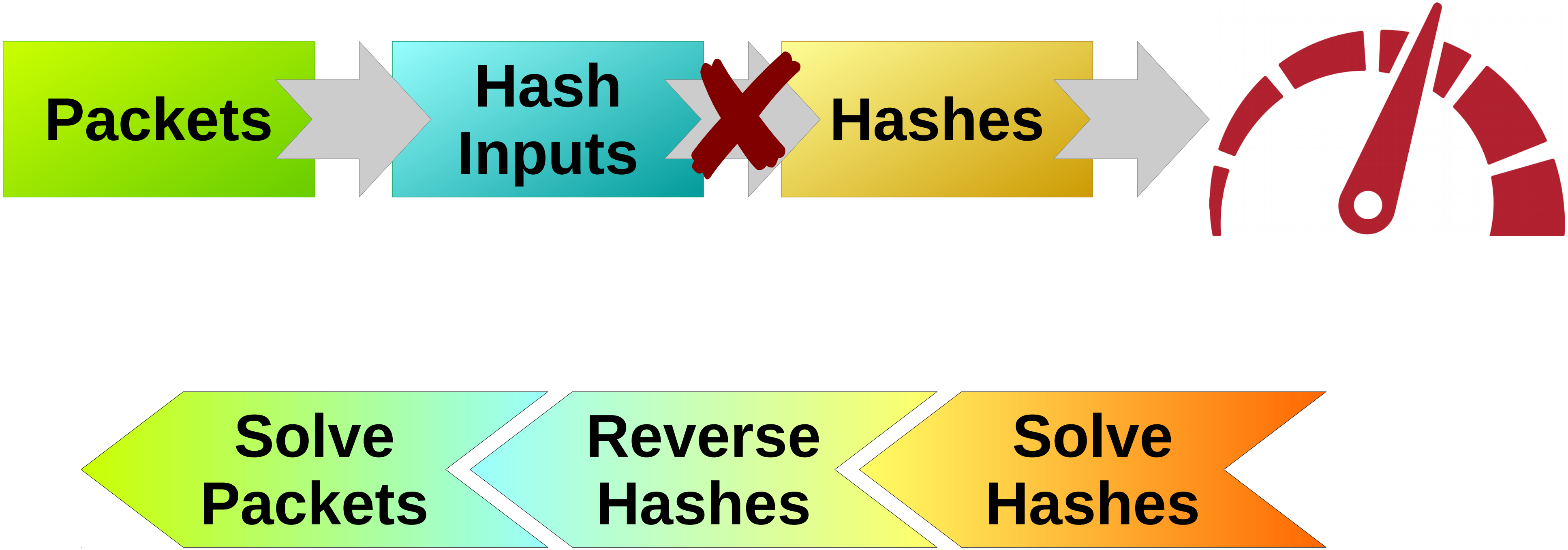
- Complex expression / Path explosion
- Reason about hash value, without computing it?

Havocing

- Annotate and disable hash function
- Assign hash value a new symbol
- Analyze data structure internals unencumbered
- Find packet ⇒ hash value ⇒ expected behavior

CASTAN

Handling Hash Functions



Packets

Hash Inputs

Hashes

Solve Packets

Reverse Hashes

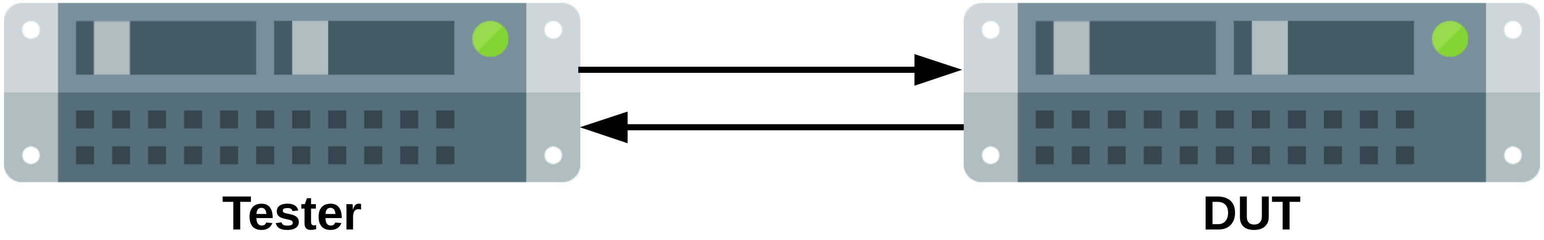
Solve Hashes

Evaluation

Setup

Network Measurement Campaign

- E2E Latency / Throughput
- Intel Xeon E5-2667v2 3.3GHz
  - 25.6MB LLC / 32GB RAM
- Intel 82599ES 10Gb NICs



Evaluation

NFs

11 NF Implementations

3 types, different data structures

	NAT	LB	LPM
Unbalanced Tree	●	●	
Red-Black Tree	●	●	
Hash Ring	●	●	
Hash Table	●	●	
Hierarchical Lookup (DPDK)			●
Single Lookup			●
Patricia Trie			●

Evaluation

NFs

11 NF Implementations

3 types, different data structures

Algorithmic Complexity

	NAT	LB	LPM
Unbalanced Tree	●	●	
Red-Black Tree	●	●	
Hash Ring	●	●	
Hash Table	●	●	
Hierarchical Lookup (DPDK)			●
Single Lookup			●
Patricia Trie			●

Cache

Evaluation

Workloads

Baseline

NOP

Adversarial

CASTAN (~50 flows), Manual (~50 flows)

Random

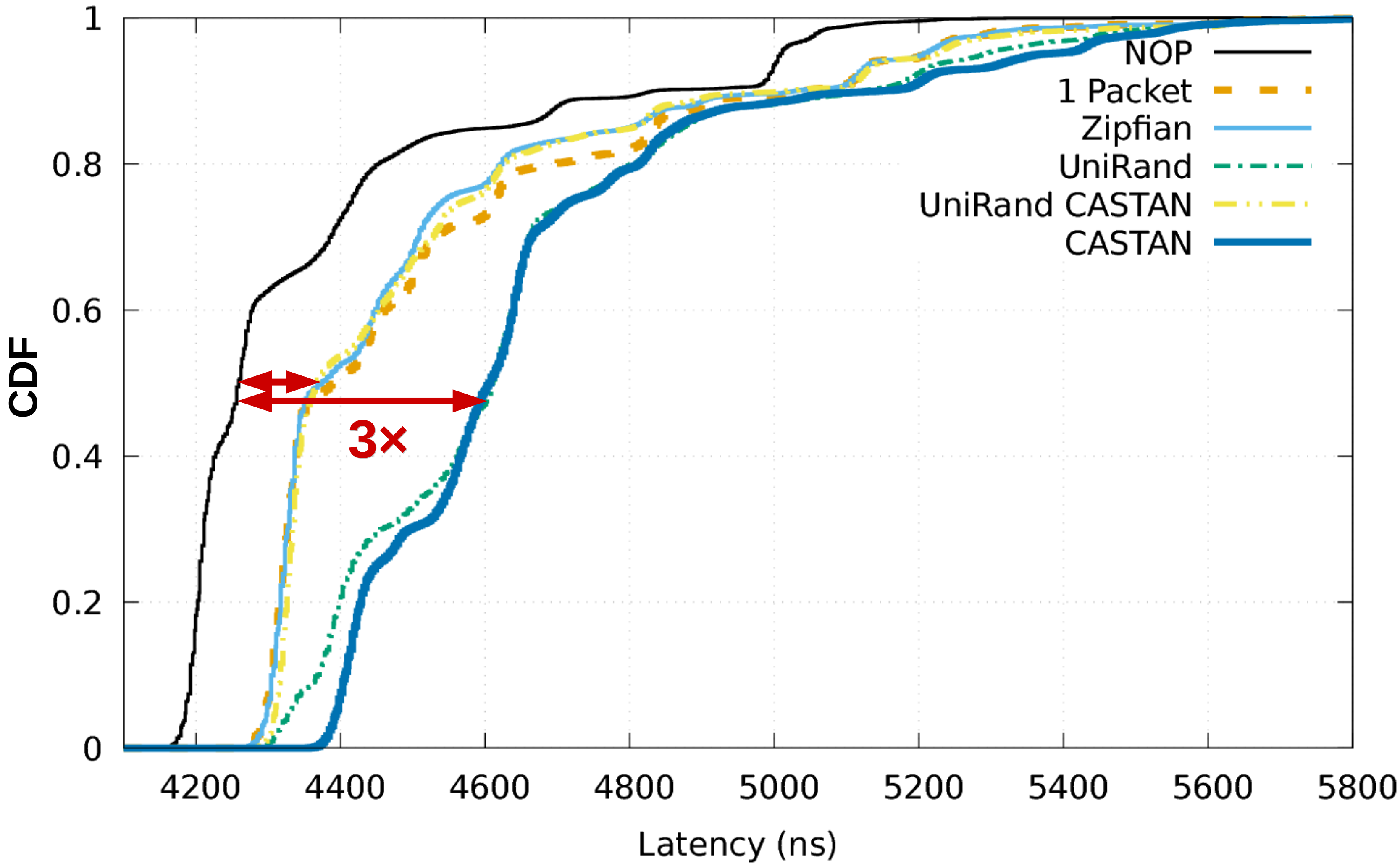
UniRand (1Mflows)

Zipf (100kpkts, 6.7kflows)

UniRand CASTAN (# flows = CASTAN)

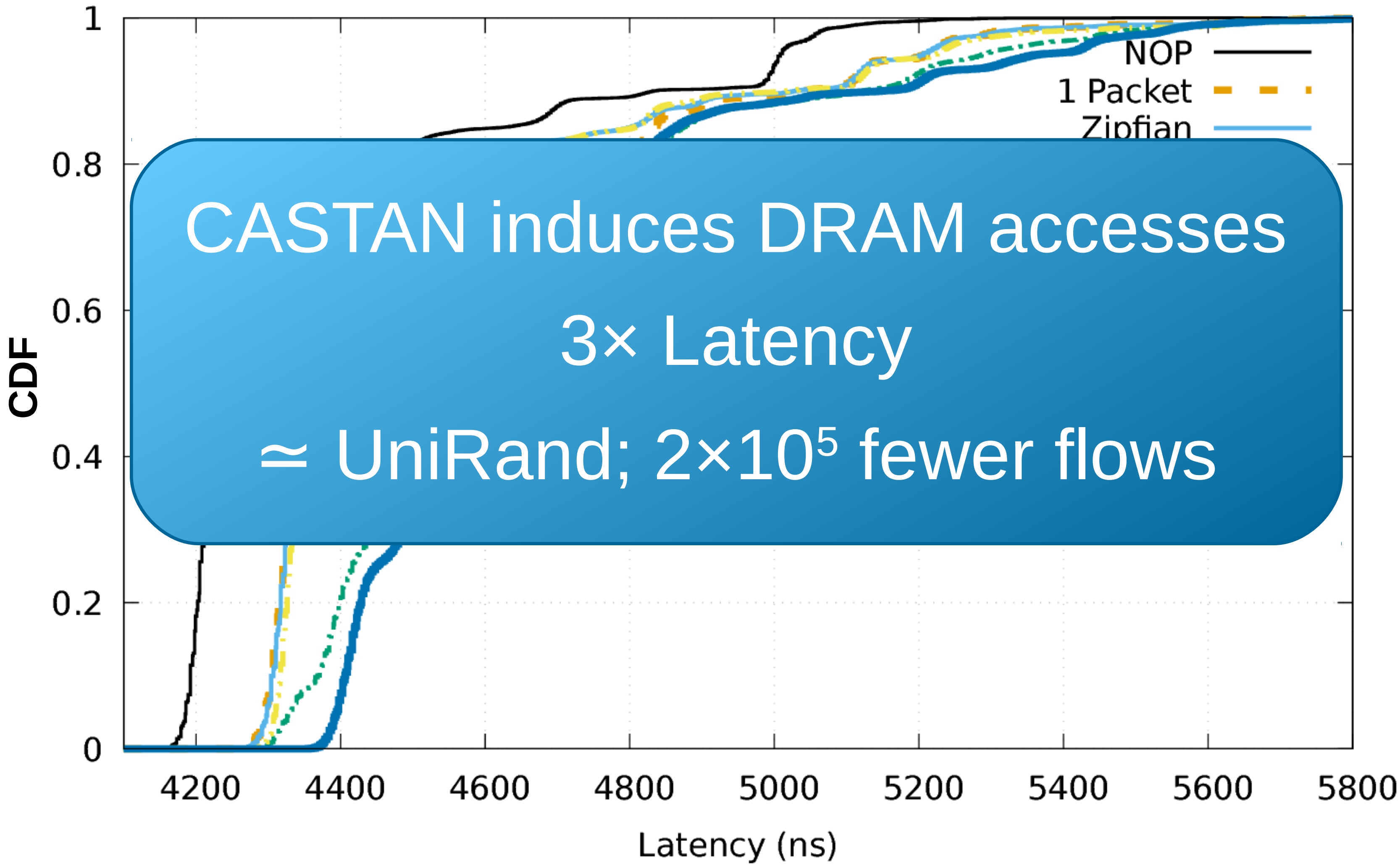
Evaluation

LPM / Single Lookup Table



Evaluation

LPM / Single Lookup Table



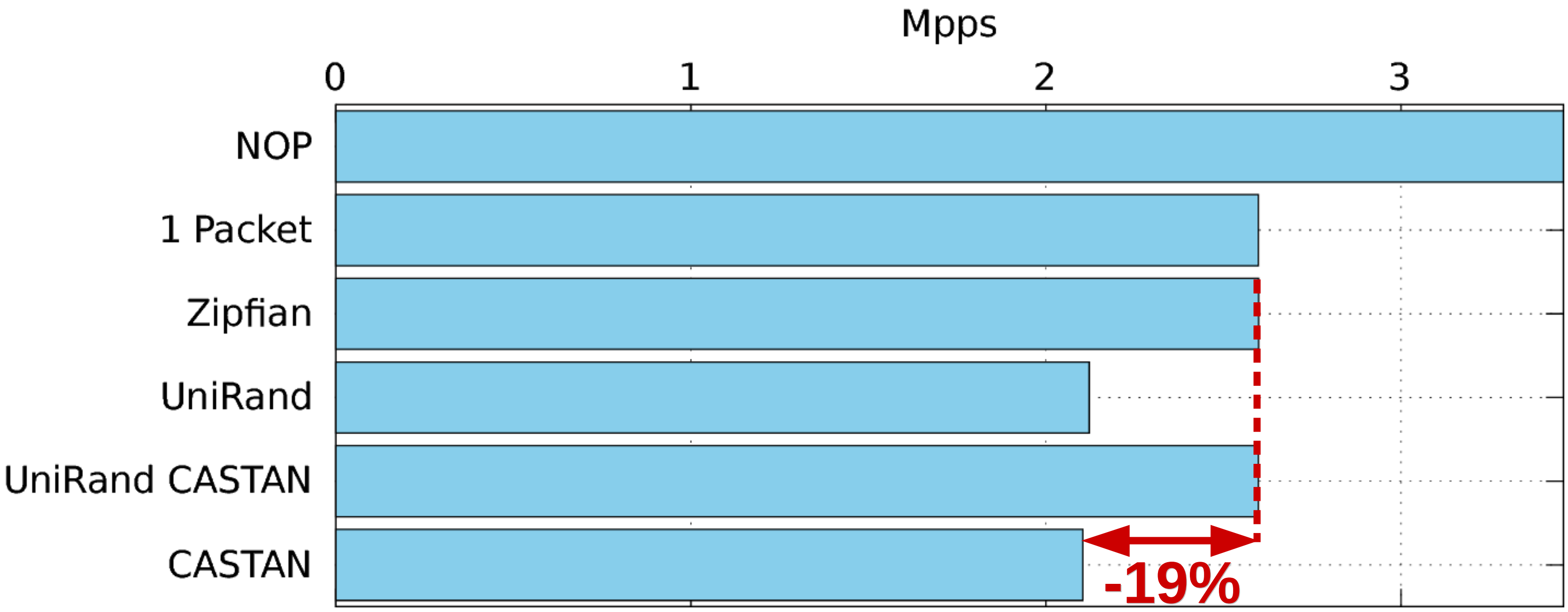
CASTAN induces DRAM accesses

3× Latency

≃ UniRand; 2×10^5 fewer flows

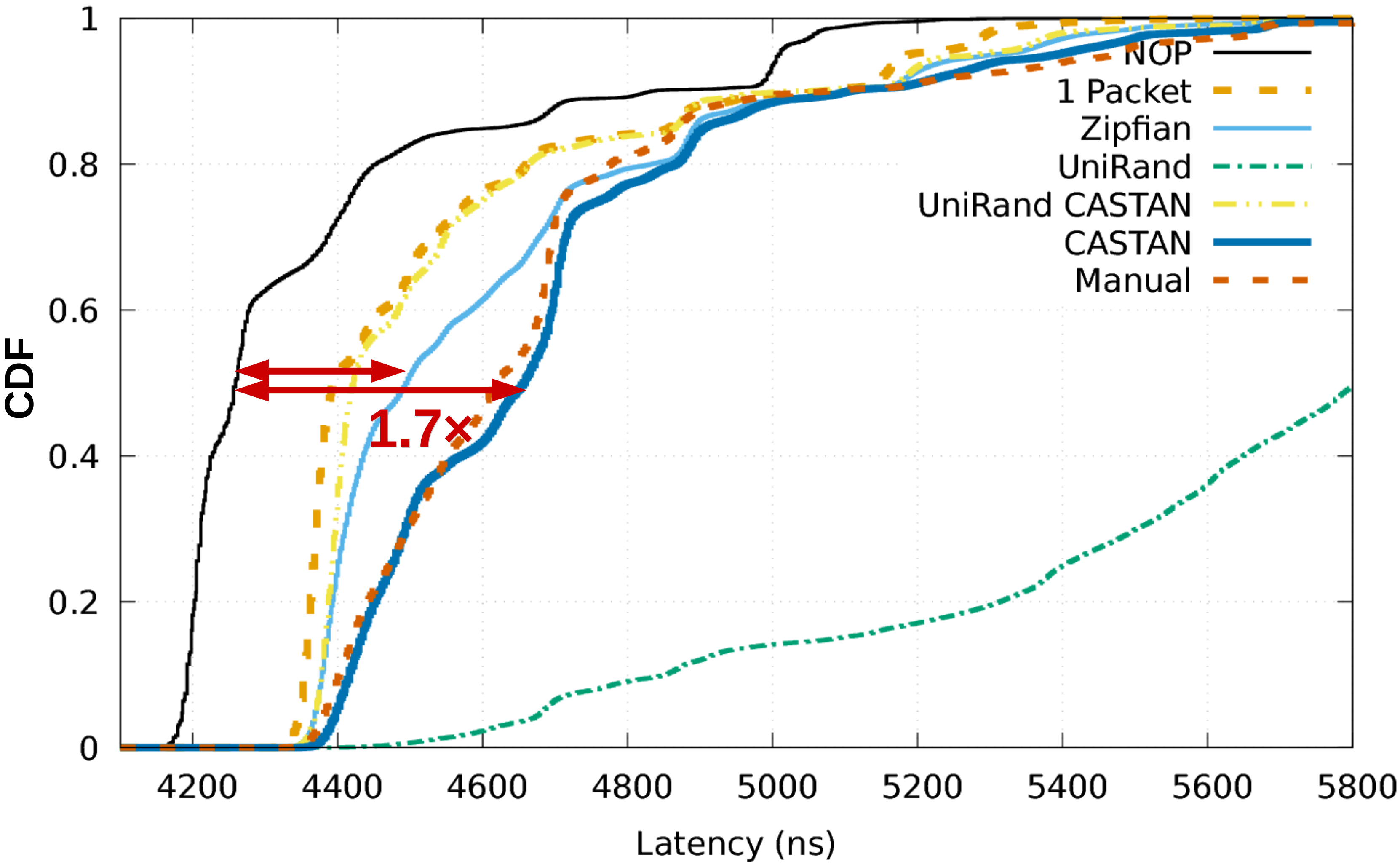
Evaluation

LPM / Single Lookup Table



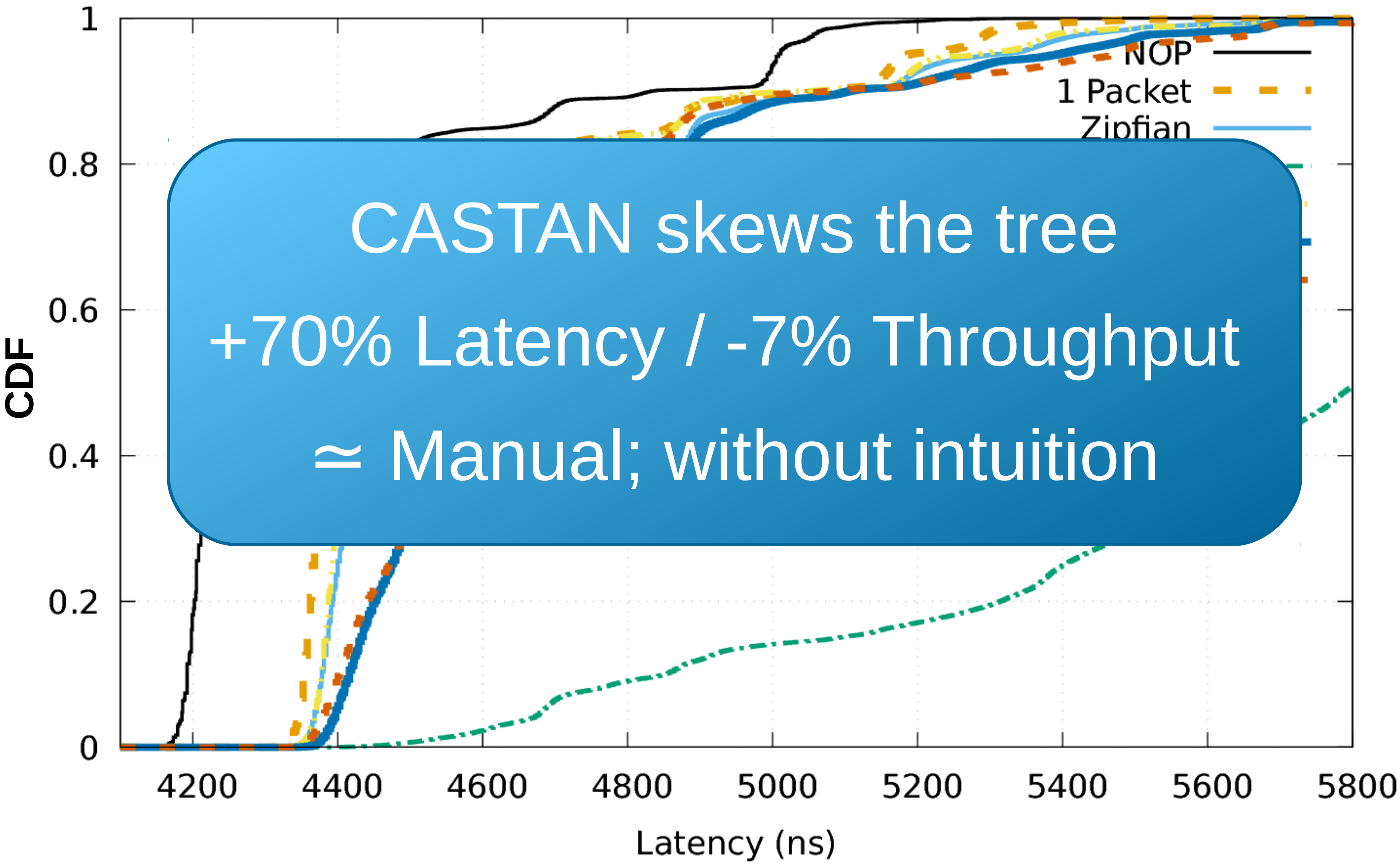
Evaluation

NAT / Unbalanced Tree



Evaluation

NAT / Unbalanced Tree



CASTAN skews the tree

+70% Latency / -7% Throughput

≃ Manual; without intuition

Conclusion

CASTAN

Attacks complexity, CPU cache, hash functions

Little developer input

Adversarial Workloads

≃ Manual when available

> Uniform random for same number of flows

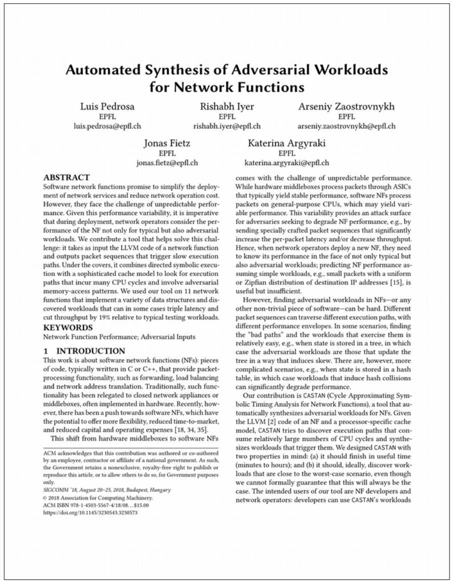
Up to <u>+201% latency</u> / <u>-19% throughput</u>

Find out more!

Look for our poster!

Get the source and more:

https://pedrosa.2y.net/Projects/CASTAN



Automated Synthesis of Adversarial Workloads for Network Functions

Luis Pedrosa
EPFL
luis.pedrosa@epfl.ch

Rishabh Iyer
EPFL
rishabh.iyer@epfl.ch

Arseniy Zaostrovnykh
EPFL
arseniy.zaostrovnykh@epfl.ch

Jonas Fietz
EPFL
jonas.fietz@epfl.ch

Katerina Argyraki
EPFL
katerina.argyraki@epfl.ch

ABSTRACT

Software network functions promise to simplify the deployment of network services and reduce network operation cost. However, they face the challenge of unpredictable performance. Given this performance variability, it is imperative that during deployment, network operators consider the performance of the NF not only for typical but also adversarial workloads. We contribute a tool that helps solve this challenge: it takes as input the LLVM code of a network function and outputs packet sequences that trigger slow execution paths. Under the covers, it combines directed symbolic execution with a sophisticated cache model to look for execution paths that incur many CPU cycles and involve adversarial memory-access patterns. We used our tool on 11 network functions that implement a variety of data structures and discovered workloads that can in some cases triple latency and cut throughput by 19% relative to typical testing workloads.

KEYWORDS

Network Function Performance; Adversarial Inputs

1 INTRODUCTION

This work is about software network functions (NFs): pieces of code, typically written in C or C++, that provide packet-processing functionality, such as forwarding, load balancing and network address translation. Traditionally, such functionality has been relegated to closed network appliances or middleboxes, often implemented in hardware. Recently, however, there has been a push towards software NFs, which have the potential to offer more flexibility, reduced time-to-market, and reduced capital and operating expenses [18, 34, 35].

This shift from hardware middleboxes to software NFs comes with the challenge of unpredictable performance. While hardware middleboxes process packets through ASICs that typically yield stable performance, software NFs process packets on general-purpose CPUs, which may yield variable performance. This variability provides an attack surface for adversaries seeking to degrade NF performance, e.g., by sending specially crafted packet sequences that significantly increase the per-packet latency and/or decrease throughput. Hence, when network operators deploy a new NF, they need to know its performance in the face of not only typical but also adversarial workloads; predicting NF performance assuming simple workloads, e.g., small packets with a uniform or Zipfian distribution of destination IP addresses [15], is useful but insufficient.

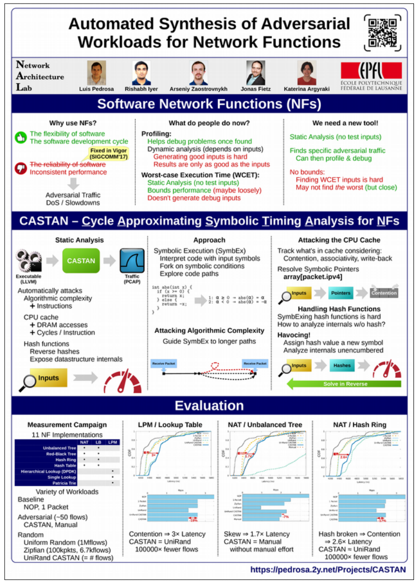
However, finding adversarial workloads in NFs—or any other non-trivial piece of software—can be hard. Different packet sequences can traverse different execution paths, with different performance envelopes. In some scenarios, finding the "bad paths" and the workloads that exercise them is relatively easy, e.g., when state is stored in a tree, in which case the adversarial workloads are those that update the tree in a way that induces skew. There are, however, more complicated scenarios, e.g., when state is stored in a hash table, in which case workloads that induce hash collisions can significantly degrade performance.

Our contribution is CASTAN (Cycle Approximating Symbolic Timing Analysis for Network Functions), a tool that automatically synthesizes adversarial workloads for NFs. Given the LLVM [2] code of an NF and a processor-specific cache model, CASTAN tries to discover execution paths that consume relatively large numbers of CPU cycles and synthesizes workloads that trigger them. We designed CASTAN with two properties in mind: (a) it should finish in useful time (minutes to hours); and (b) it should, ideally, discover workloads that are close to the worst-case scenario, even though we cannot formally guarantee that this will always be the case. The intended users of our tool are NF developers and network operators: developers can use CASTAN's workloads

---

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230573



Automated Synthesis of Adversarial Workloads for Network Functions

Network
Architecture
Lab

Luis Pedrosa
Rishabh Iyer
Arseniy Zaostrovnykh
Jonas Fietz
Katerina Argyraki

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Software Network Functions (NFs)

Why use NFs?

The flexibility of software
The software development cycle

Fixed in Vigor (SIGCOMM'17)

~~The reliability of software~~
Inconsistent performance

Adversarial Traffic
DoS / Slowdowns

What do people do now?

Profiling:
Helps debug problems once found
Dynamic analysis (depends on inputs)
Generating good inputs is hard
Results are only as good as the inputs

Worst-case Execution Time (WCET):
Static Analysis (no test inputs)
Bounds performance (maybe loosely)
Doesn't generate debug inputs

We need a new tool!

Static Analysis (no test inputs)

Finds specific adversarial traffic
Can then profile & debug

No bounds:
Finding WCET inputs is hard
May not find *the* worst (but close)

CASTAN – Cycle Approximating Symbolic Timing Analysis for NFs

Static Analysis

Executable (LLVM)
CASTAN
Traffic (PCAP)

Automatically attacks
Algorithmic complexity
+ Instructions
CPU cache
+ DRAM accesses
+ Cycles / Instruction
Hash functions
Reverse hashes
Expose datastructure internals

Inputs

Approach

Symbolic Execution (SymbEx)
Interpret code with input symbols
Fork on symbolic conditions
Explore code paths

```
int abs(int x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

1: α ≥ 0 → abs(α) = α
2: α < 0 → abs(α) = -α

Attacking Algorithmic Complexity

Guide SymbEx to longer paths

Receive Packet
Receive Packet

Attacking the CPU Cache

Track what's in cache considering:
Contention, associativity, write-back

Resolve Symbolic Pointers
array[packet.ipv4]

Inputs → Pointers → Contention

Handling Hash Functions

SymbExing hash functions is hard
How to analyze internals w/o hash?

Havocing!
Assign hash value a new symbol
Analyze internals unencumbered

Inputs → Hashes

Solve in Reverse

Evaluation

Measurement Campaign

11 NF Implementations

	NAT	LB	LPM
Unbalanced Tree	•	•	
Red-Black Tree	•	•	
Hash Ring	•	•	
Hash Table	•	•	
Hierarchical Lookup (DPDK)			•
Single Lookup			•
Patricia Trie			•

Variety of Workloads
Baseline
NOP, 1 Packet
Adversarial (~50 flows)
CASTAN, Manual
Random
Uniform Random (1Mflows)
Zipfian (100kpkts, 6.7kflows)
UniRand CASTAN (= # flows)

LPM / Lookup Table

Contention ⇒ 3× Latency
CASTAN = UniRand
100000× fewer flows

NAT / Unbalanced Tree

Skew ⇒ 1.7× Latency
CASTAN = Manual
without manual effort

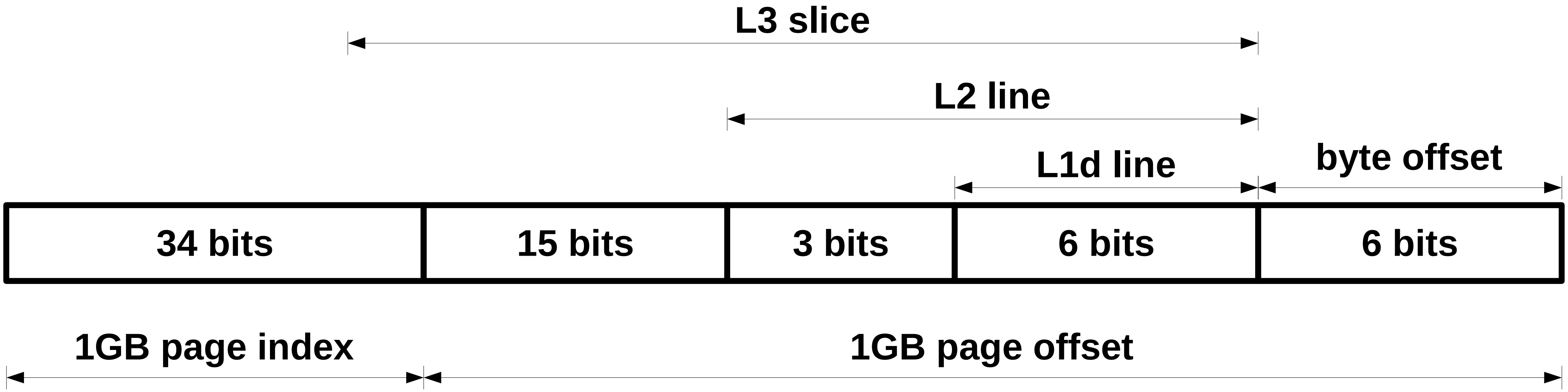
NAT / Hash Ring

Hash broken ⇒ Contention
⇒ 2.6× Latency
CASTAN ≈ UniRand
100000× fewer flows

https://pedrosa.2y.net/Projects/CASTAN

Backup Slides

Cache Structure



L3 slice

L2 line

L1d line

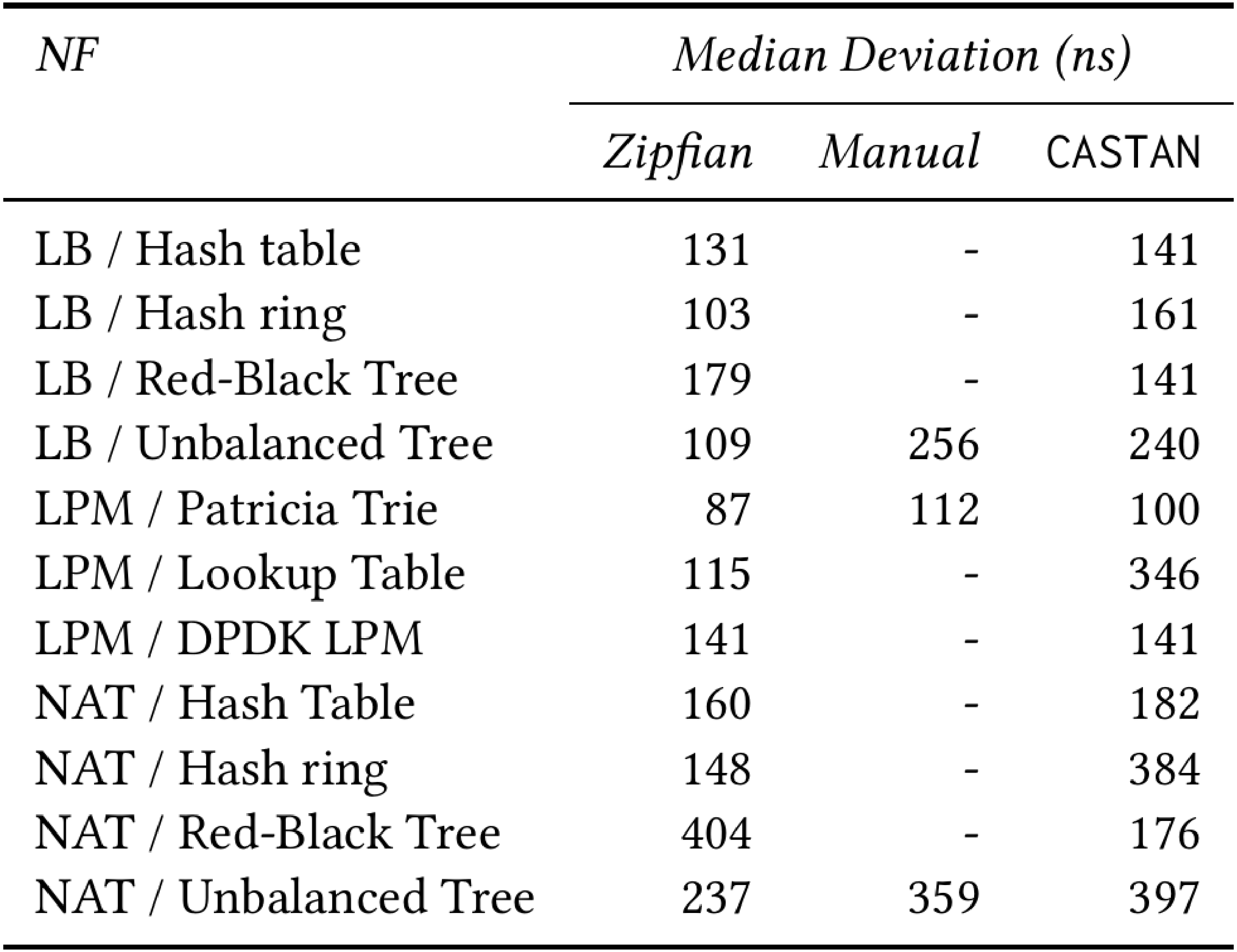
byte offset

34 bits	15 bits	3 bits	6 bits	6 bits

1GB page index

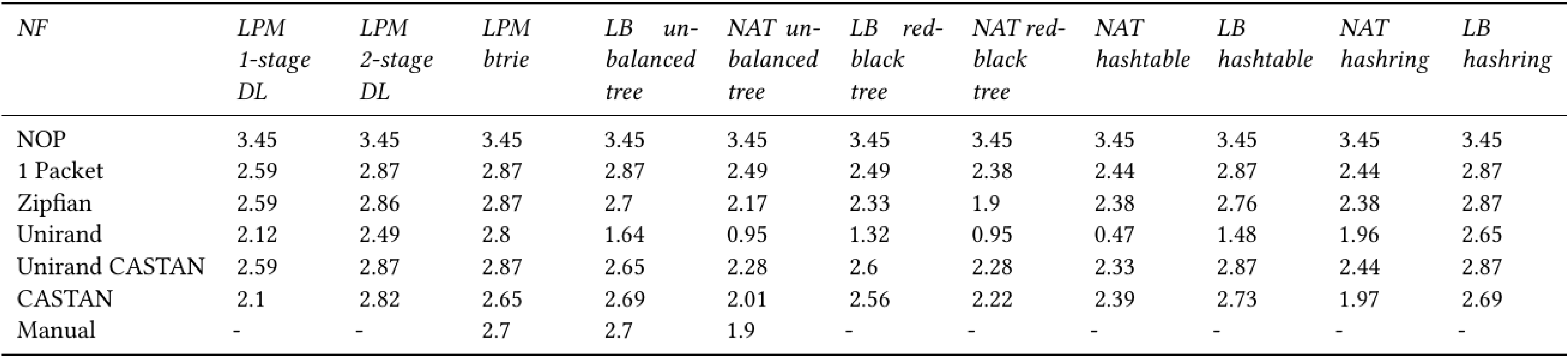
1GB page offset

Latency Deviation from NOP



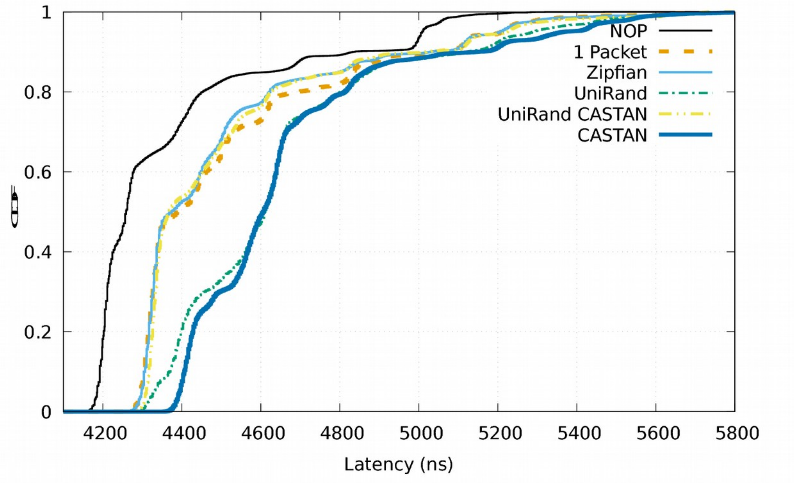
NF	*Median Deviation (ns)*		
	Zipfian	*Manual*	CASTAN
LB / Hash table	131	-	141
LB / Hash ring	103	-	161
LB / Red-Black Tree	179	-	141
LB / Unbalanced Tree	109	256	240
LPM / Patricia Trie	87	112	100
LPM / Lookup Table	115	-	346
LPM / DPDK LPM	141	-	141
NAT / Hash Table	160	-	182
NAT / Hash ring	148	-	384
NAT / Red-Black Tree	404	-	176
NAT / Unbalanced Tree	237	359	397

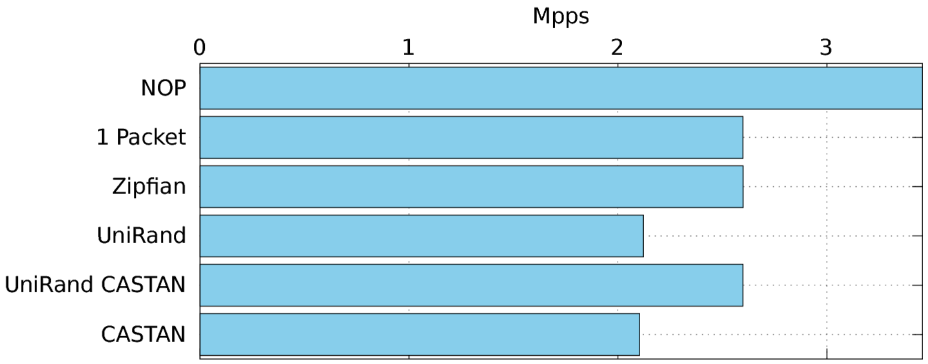
Throughput

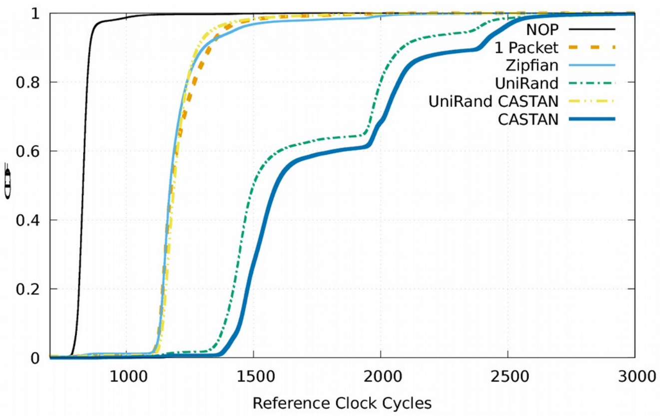


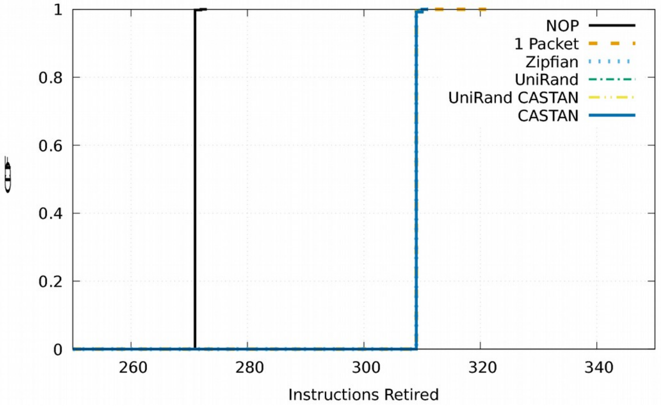
NF	*LPM 1-stage DL*	*LPM 2-stage DL*	*LPM btrie*	*LB unbalanced tree*	*NAT unbalanced tree*	*LB red-black tree*	*NAT red-black tree*	*NAT hashtable*	*LB hashtable*	*NAT hashring*	*LB hashring*
NOP	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45
1 Packet	2.59	2.87	2.87	2.87	2.49	2.49	2.38	2.44	2.87	2.44	2.87
Zipfian	2.59	2.86	2.87	2.7	2.17	2.33	1.9	2.38	2.76	2.38	2.87
Unirand	2.12	2.49	2.8	1.64	0.95	1.32	0.95	0.47	1.48	1.96	2.65
Unirand CASTAN	2.59	2.87	2.87	2.65	2.28	2.6	2.28	2.33	2.87	2.44	2.87
CASTAN	2.1	2.82	2.65	2.69	2.01	2.56	2.22	2.39	2.73	1.97	2.69
Manual	-	-	2.7	2.7	1.9	-	-	-	-	-	-

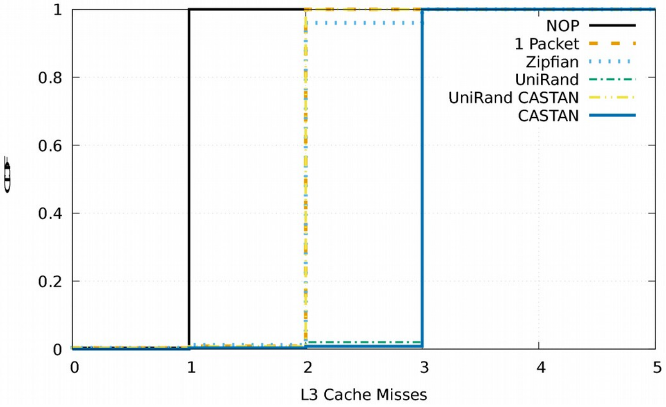
LPM / Single Lookup Table



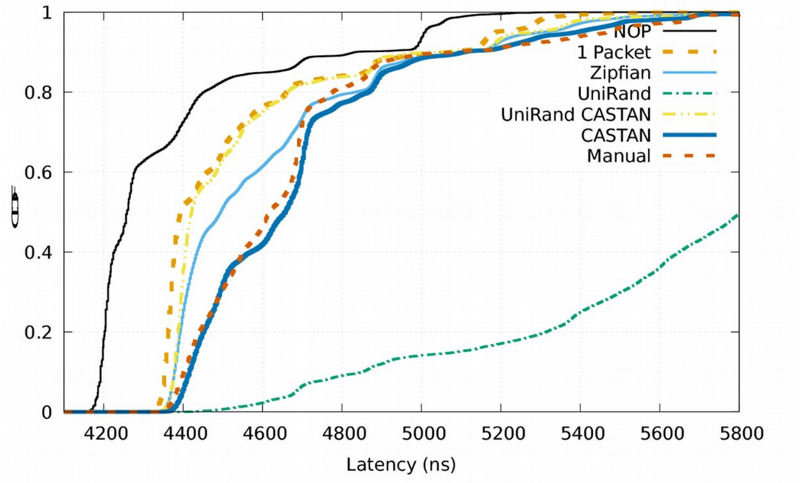


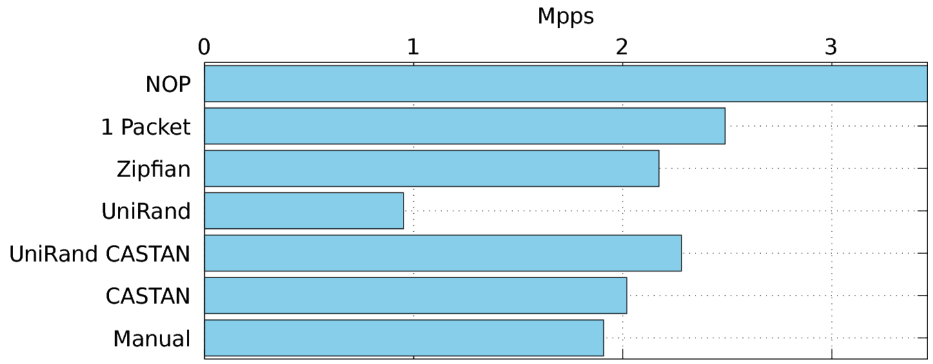


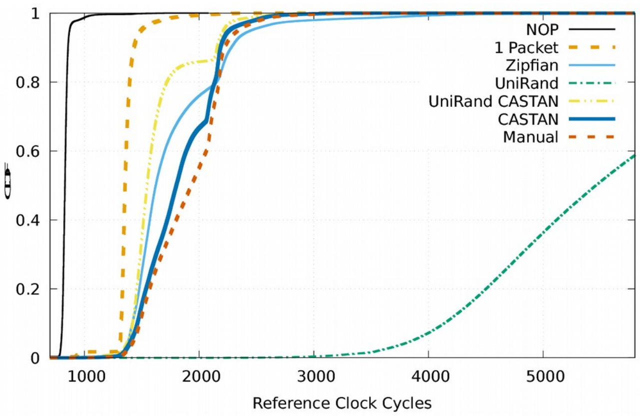


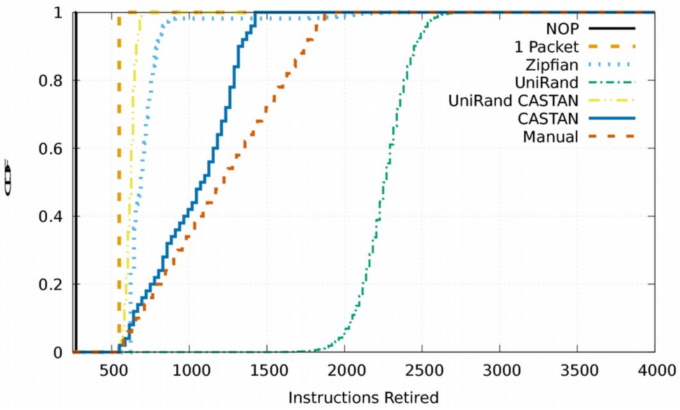


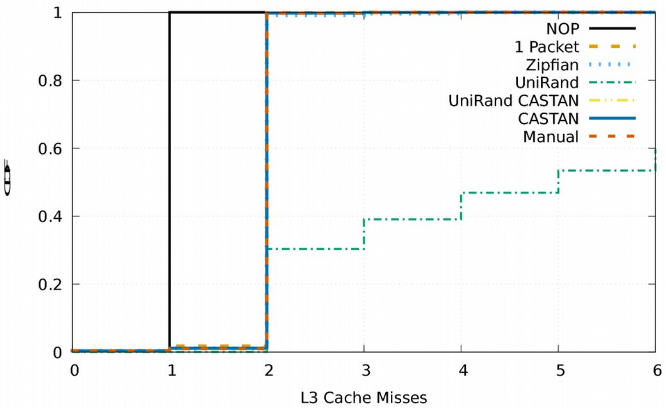
NAT / Unbalanced Tree



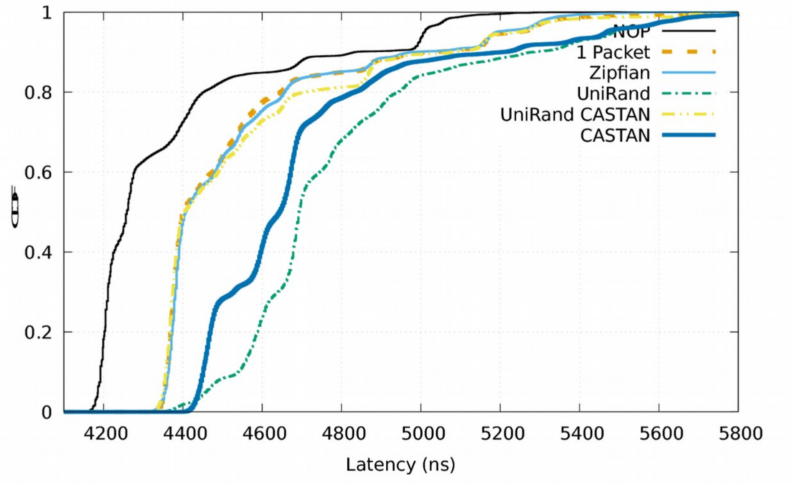


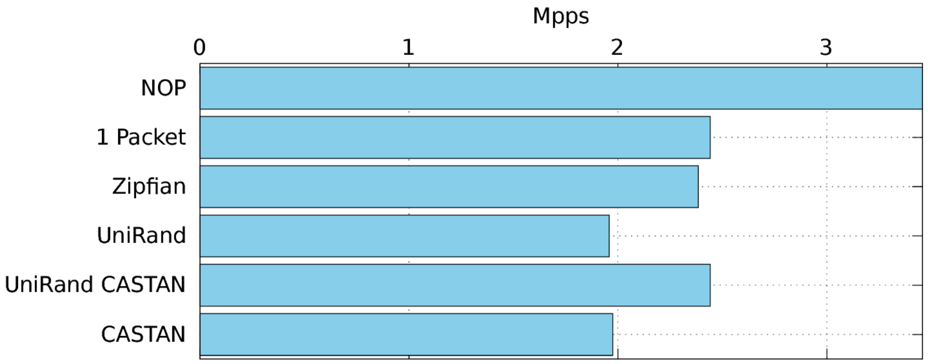


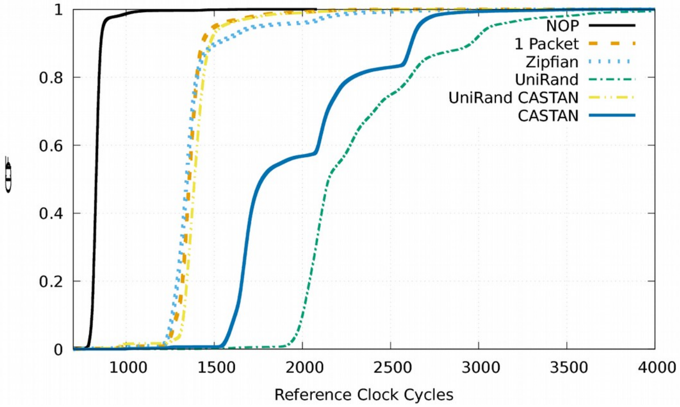


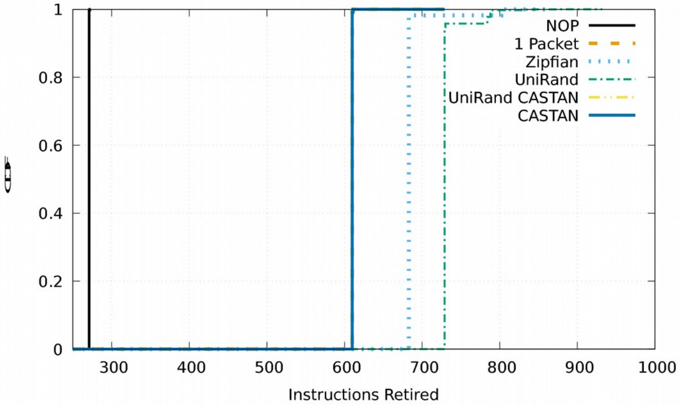


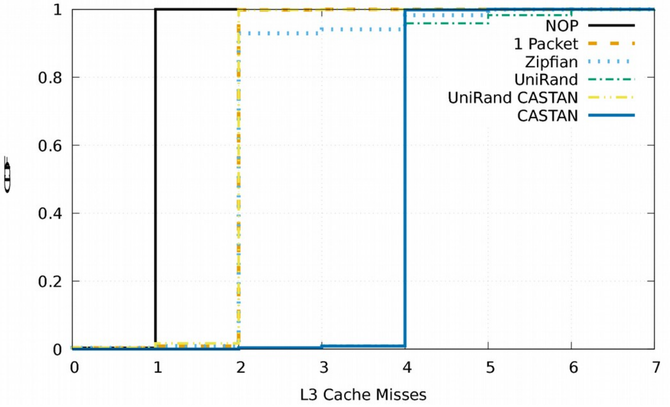
NAT / Hash Ring



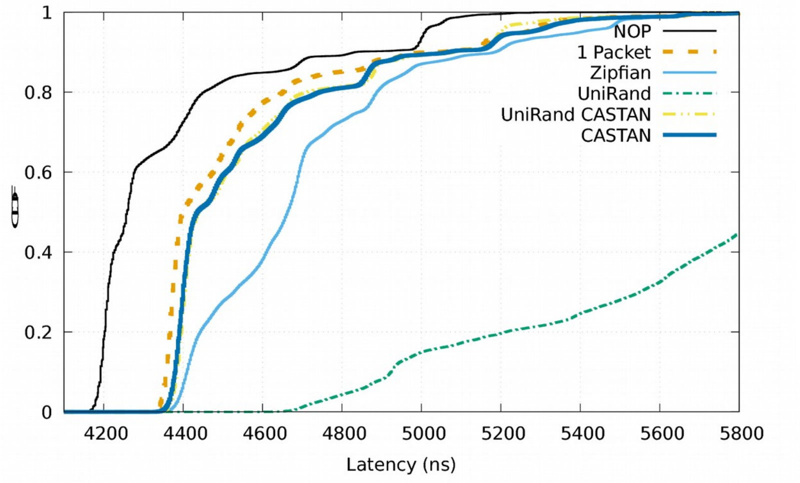


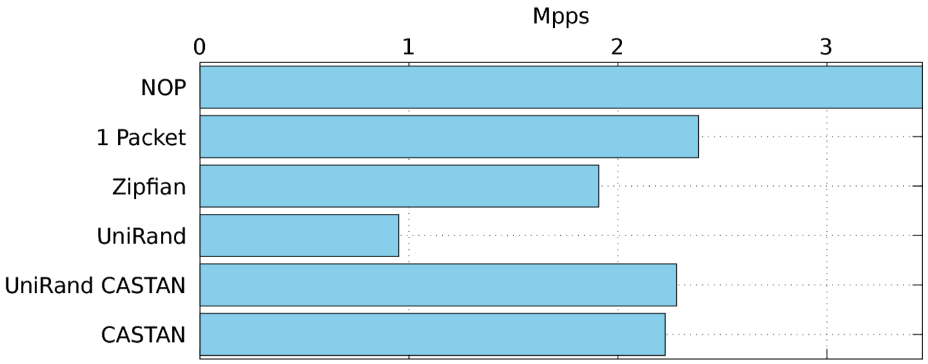


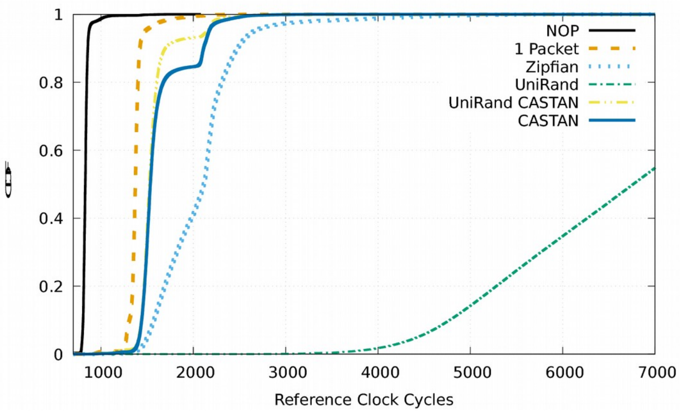


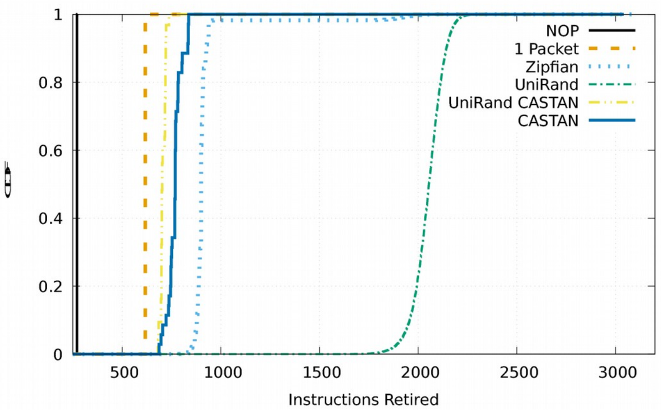


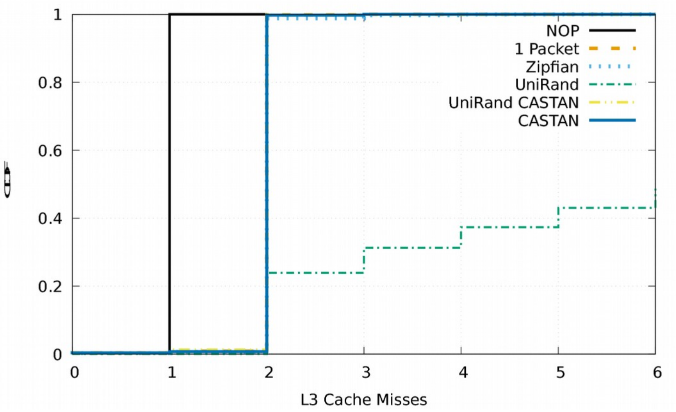
NAT / Red-Black Tree



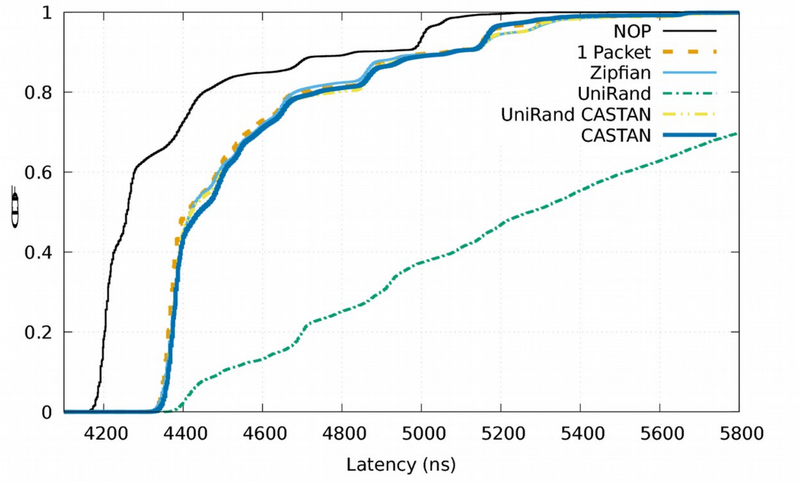


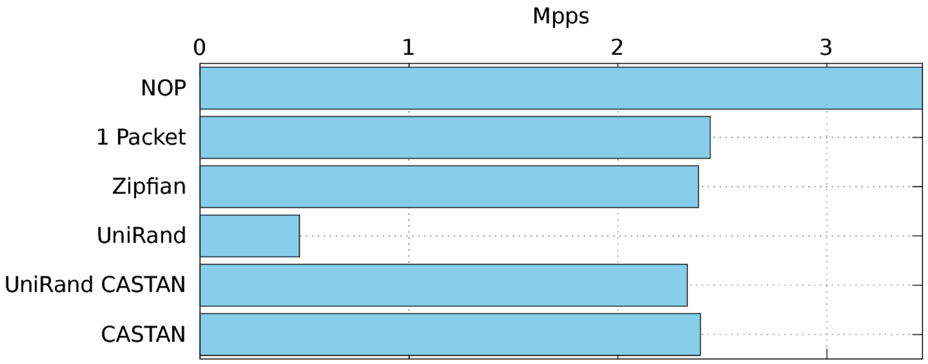


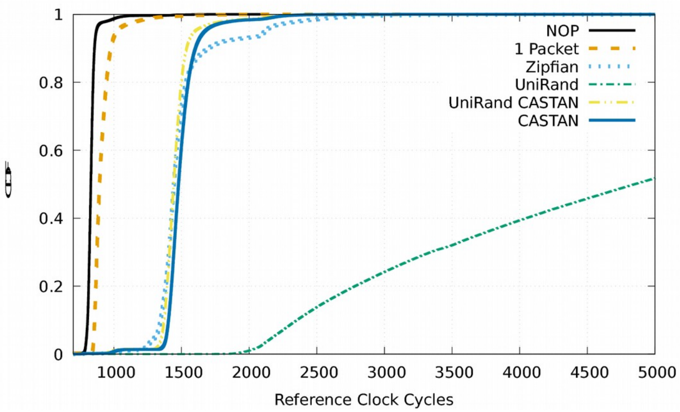


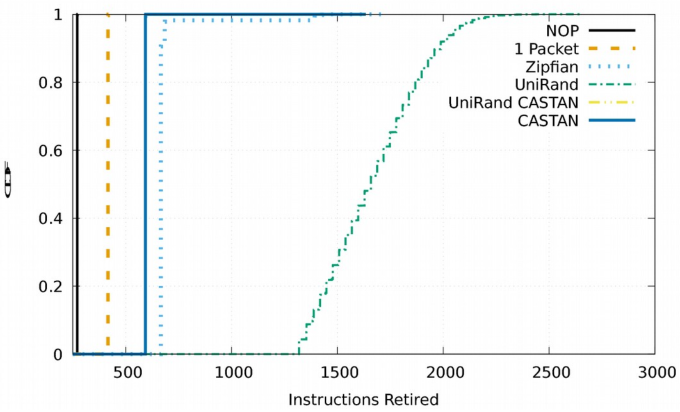


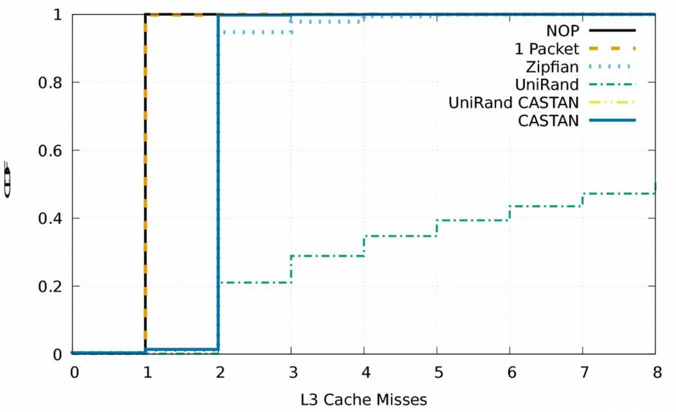
NAT / Hash Table



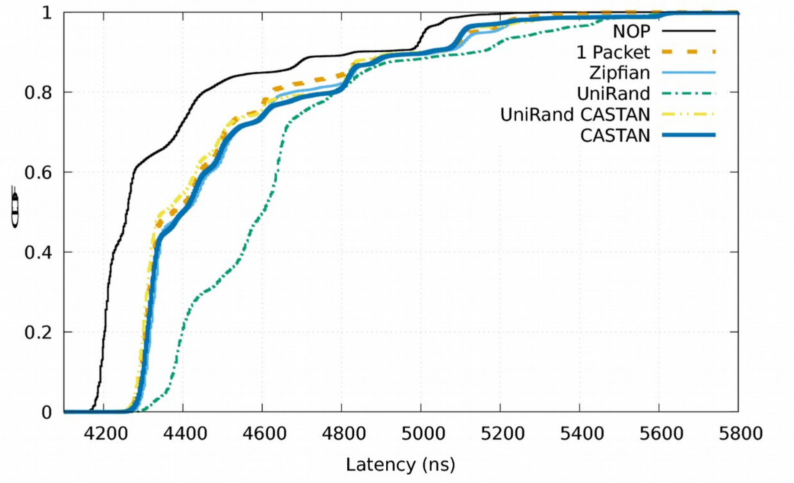


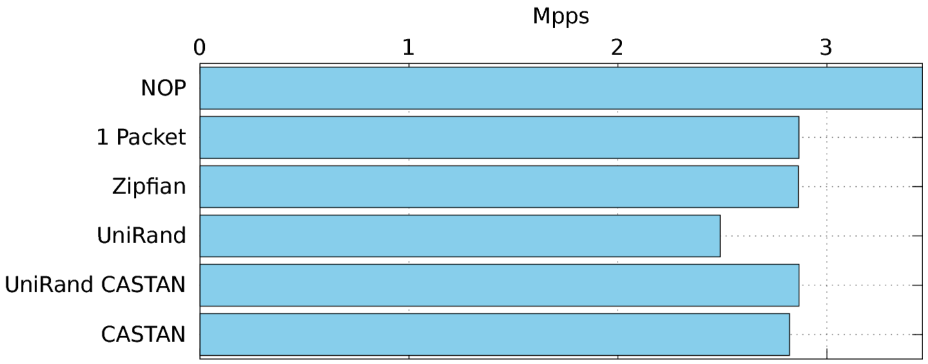


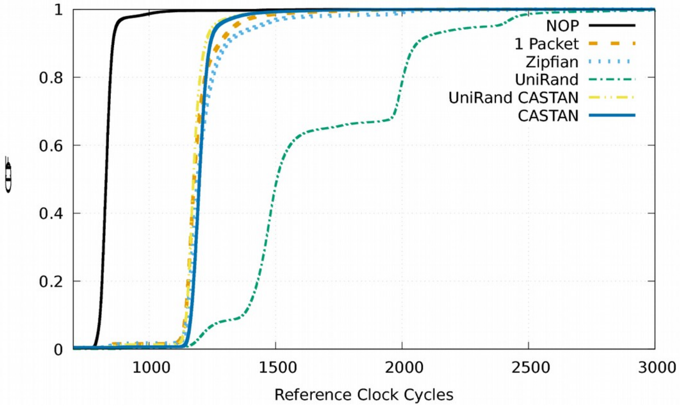


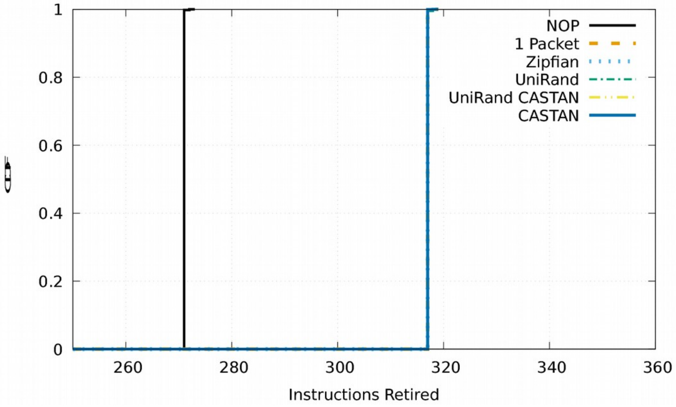


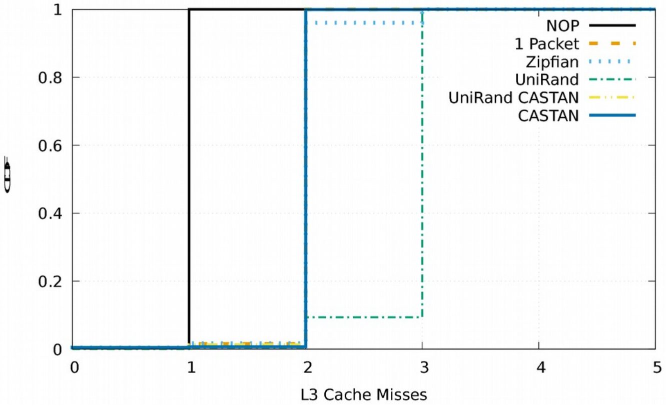
LPM / Hierarchical Lookup (DPDK)



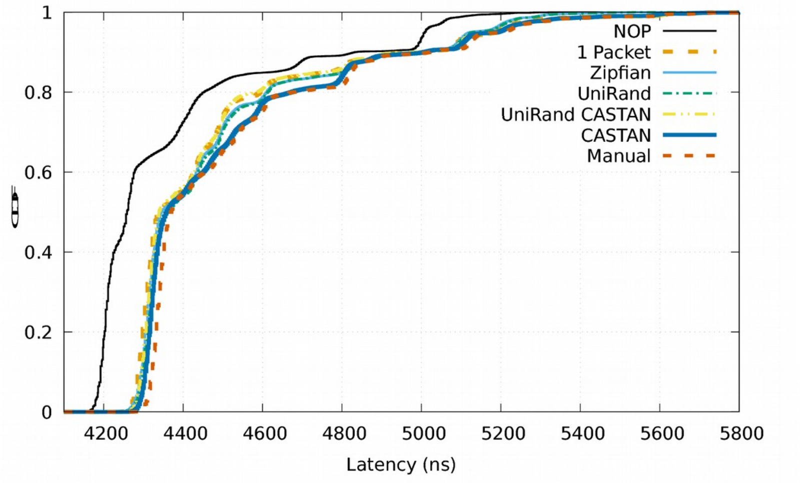


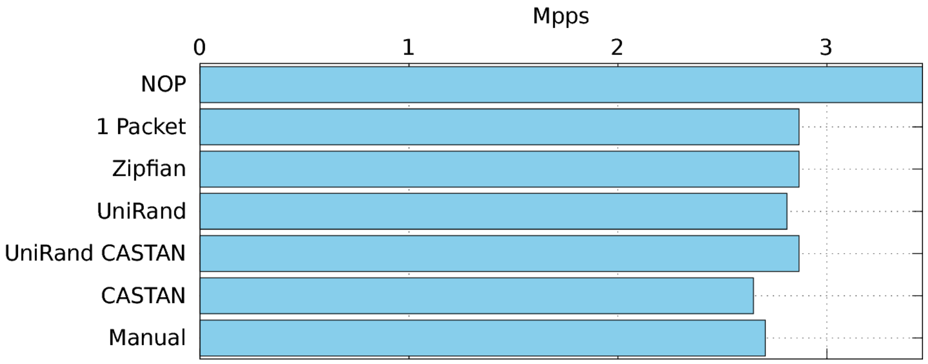


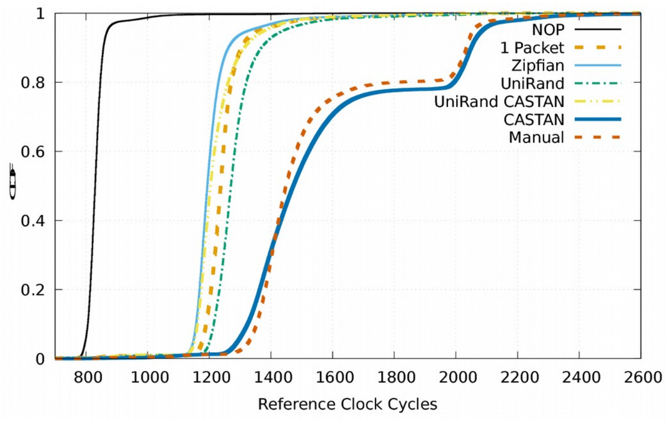


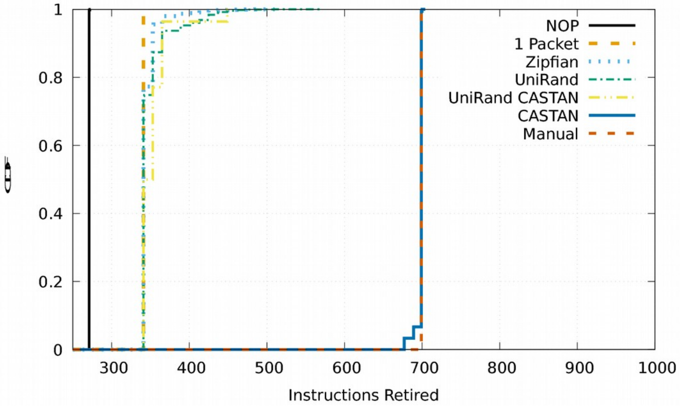


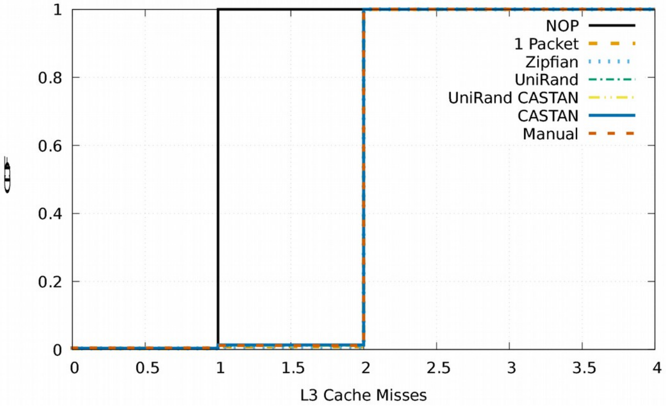
LPM / Patricia Trie



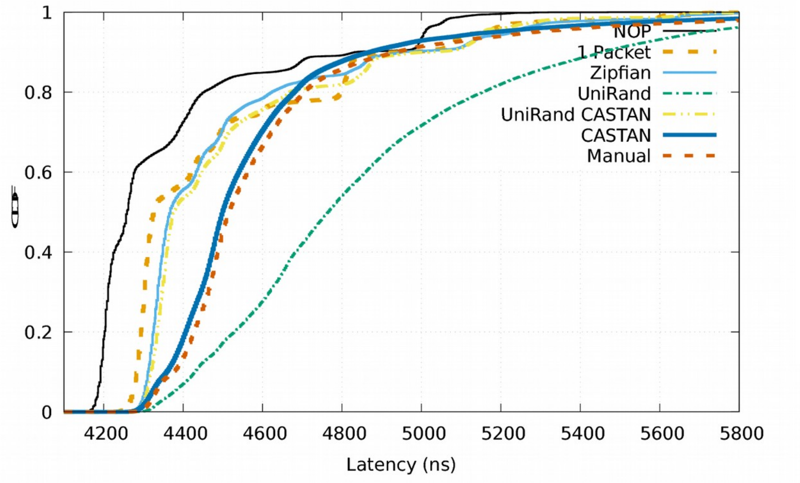


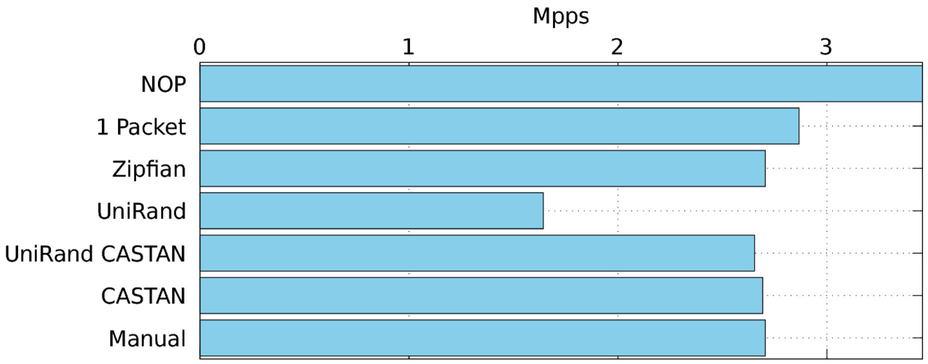


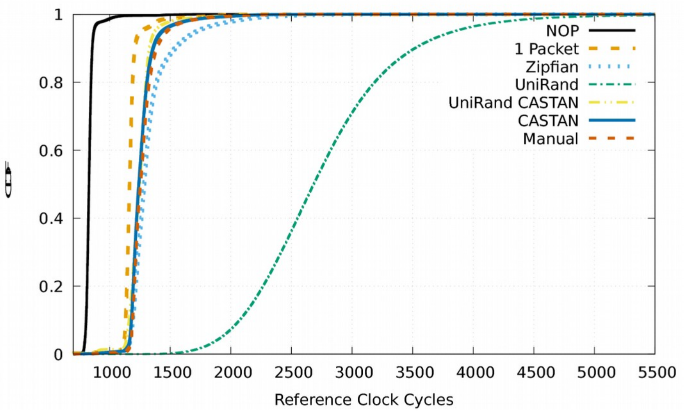


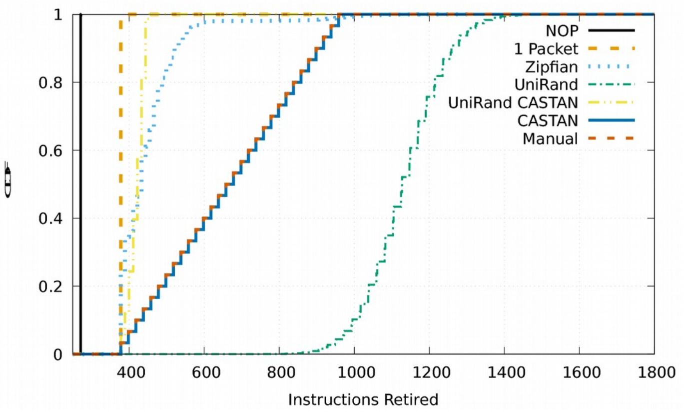


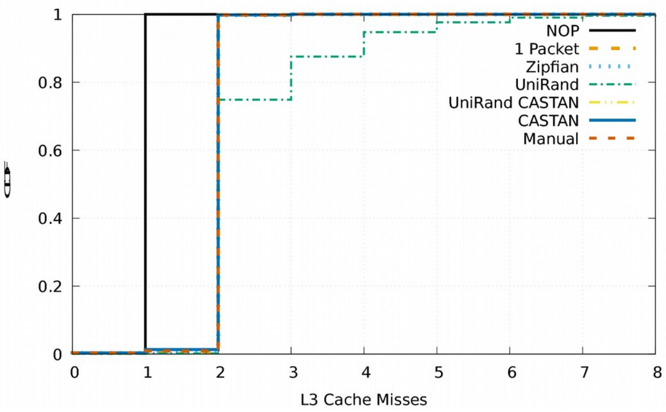
LB / Unbalanced Tree



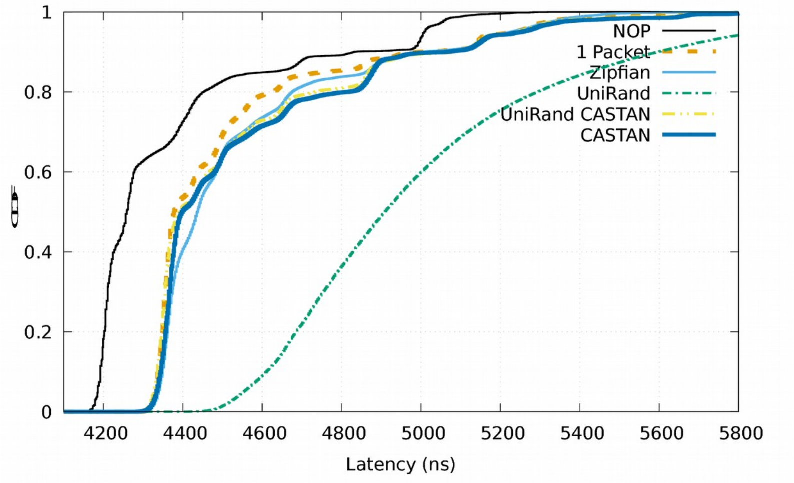


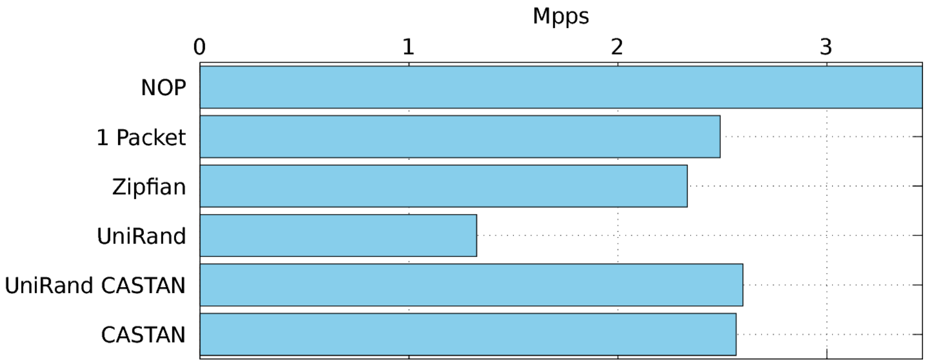


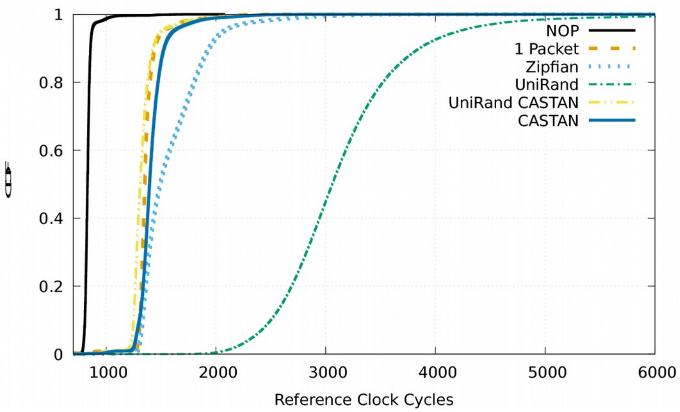


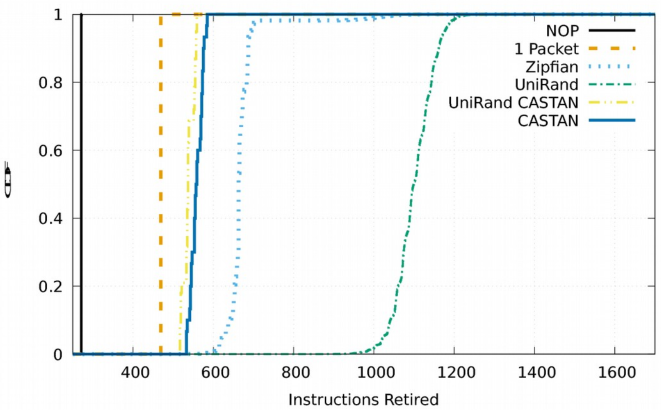


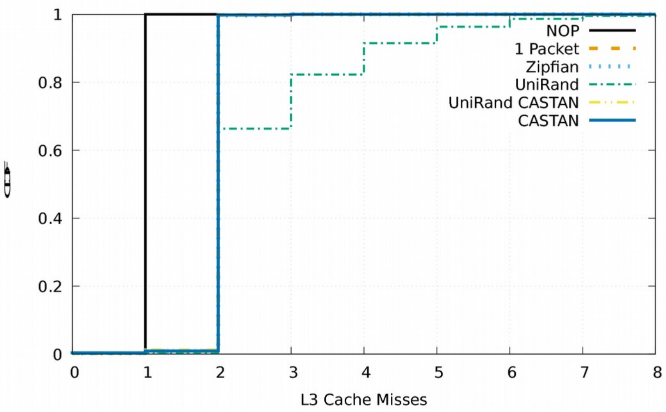
LB / Red-Black Tree



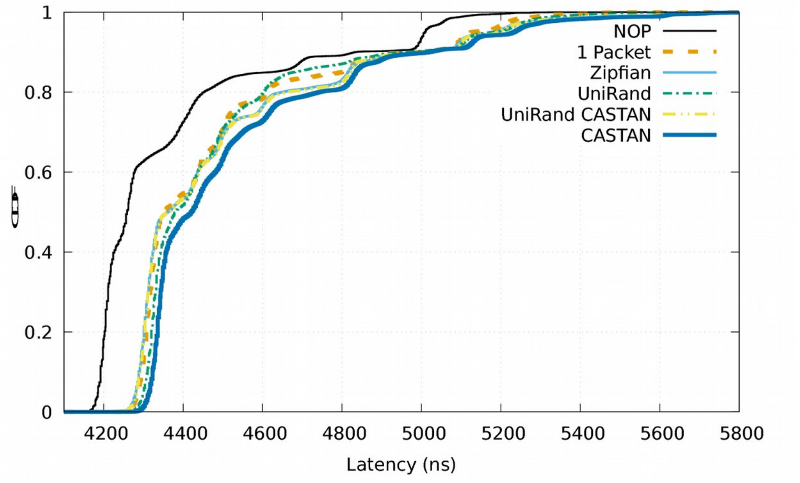


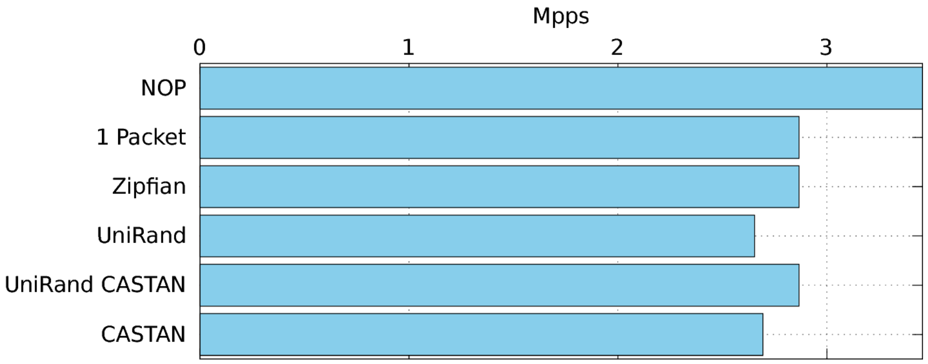


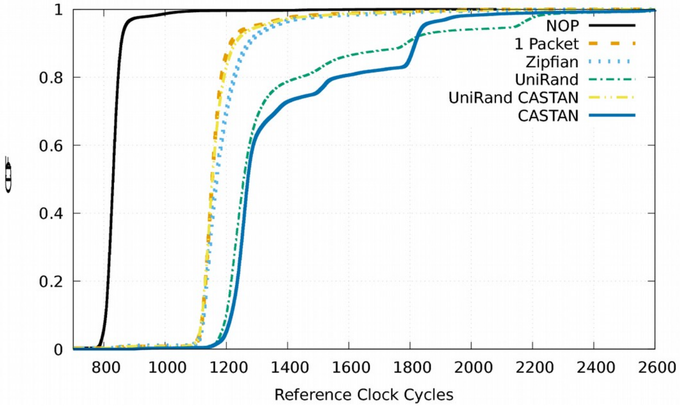


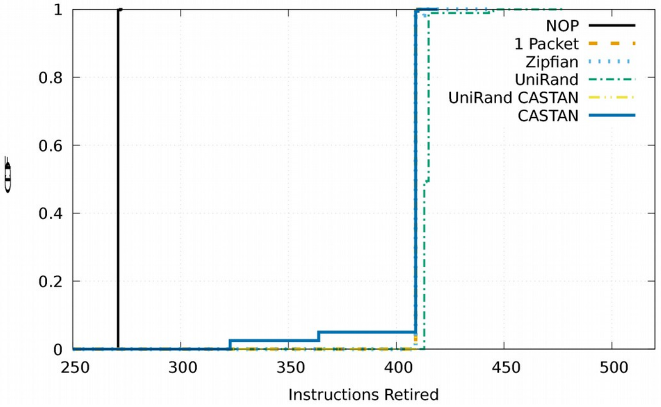


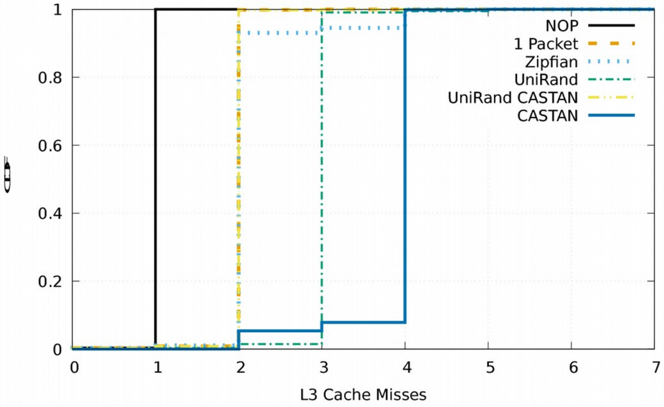
LB / Hash Ring


1
0.8
0.6
0.4
0.2
0
4200
4400
4600
4800
5000
5200
5400
5600
5800
Latency (ns)
NOP
1 Packet
Zipfian
UniRand
UniRand CASTAN
CASTAN


Mpps
0
1
2
3
NOP
1 Packet
Zipfian
UniRand
UniRand CASTAN
CASTAN


1
0.8
0.6
0.4
0.2
0
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
Reference Clock Cycles
NOP
1 Packet
Zipfian
UniRand
UniRand CASTAN
CASTAN


1
0.8
0.6
0.4
0.2
0
250
300
350
400
450
500
Instructions Retired
NOP
1 Packet
Zipfian
UniRand
UniRand CASTAN
CASTAN


1
0.8
0.6
0.4
0.2
0
0
1
2
3
4
5
6
7
L3 Cache Misses
NOP
1 Packet
Zipfian
UniRand
UniRand CASTAN
CASTAN

LB / Hash Table