

# Introduction to OpenResty XRay

Deep monitoring, analyses and diagnostics for your online applications



Try OpenResty XRay for free

<https://openresty.com/en/xray/>

info@openresty.com

# Challenges in the World of Software

Rapid business iteration

Multi-level  
team members

Inadequate testing

Increasing business  
complexity

High CPU usage

Excessive memory  
usage (including  
memory leaks)

Insufficient hard disk IO  
resources

Long latency response

Exceptions and errors  
that are difficult to  
reproduce offline  
(including process  
crashes)

# Challenges in the age of K8s/Docker Containers

---

Lots of containers, lots of applications, lots of distributions, lots of technology stacks

---

Minimized containers lack the most basic debugging tools

---

Minimized set of container permissions

---

Automatically discard and restart containers when something goes wrong. Software bugs are easily swept under the carpet

---

Virtualized containers, Microservices - further increase the software complexity

# Disadvantages of Traditional Methods

Invasive - need to modify applications

Slow response

Require big data storage and analysis

Superficial indicators only

Observations without causes

Lack of in-depth analysis and diagnostics of the entire technology stack

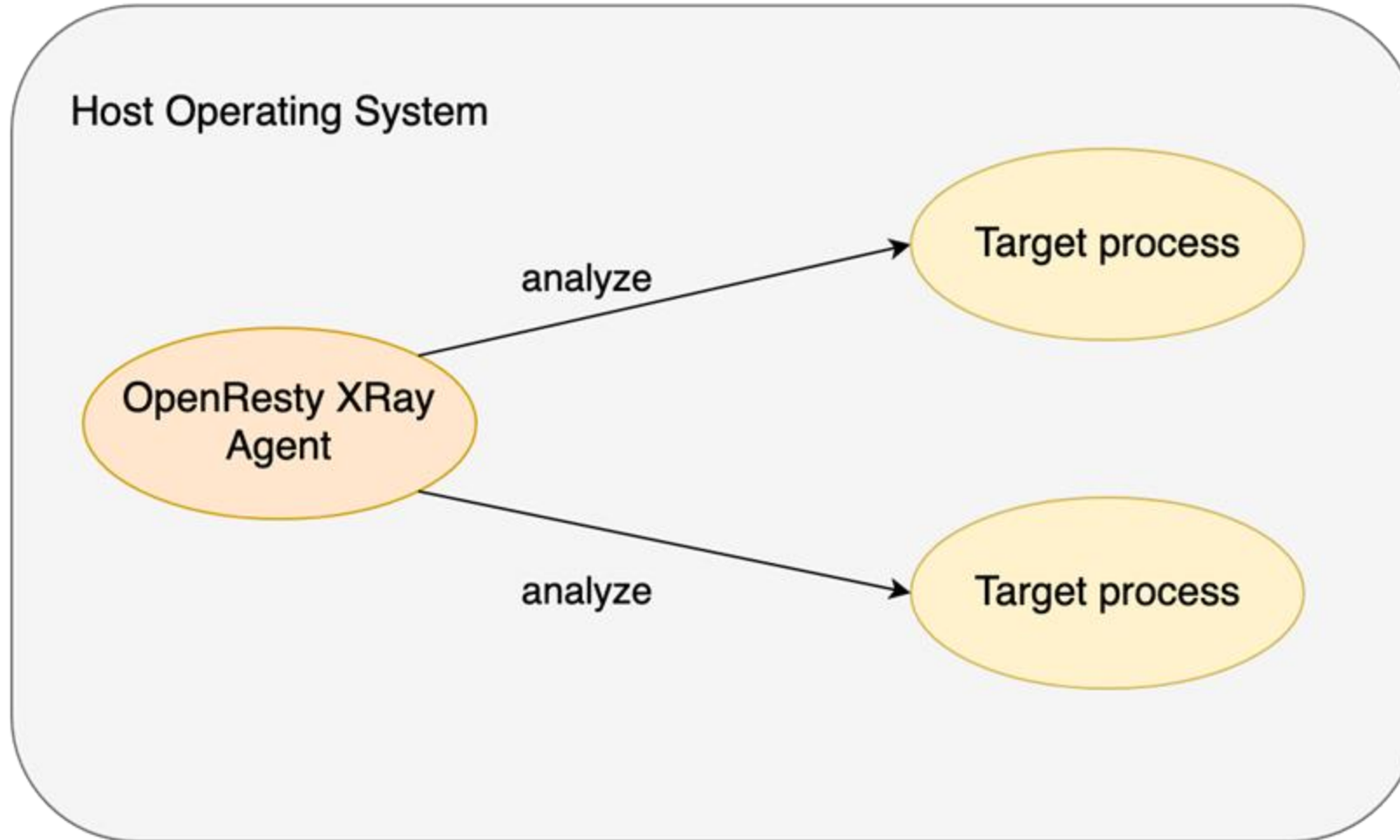
Complex, highly overloaded and error-prone data collection and processing process

# OpenResty XRay

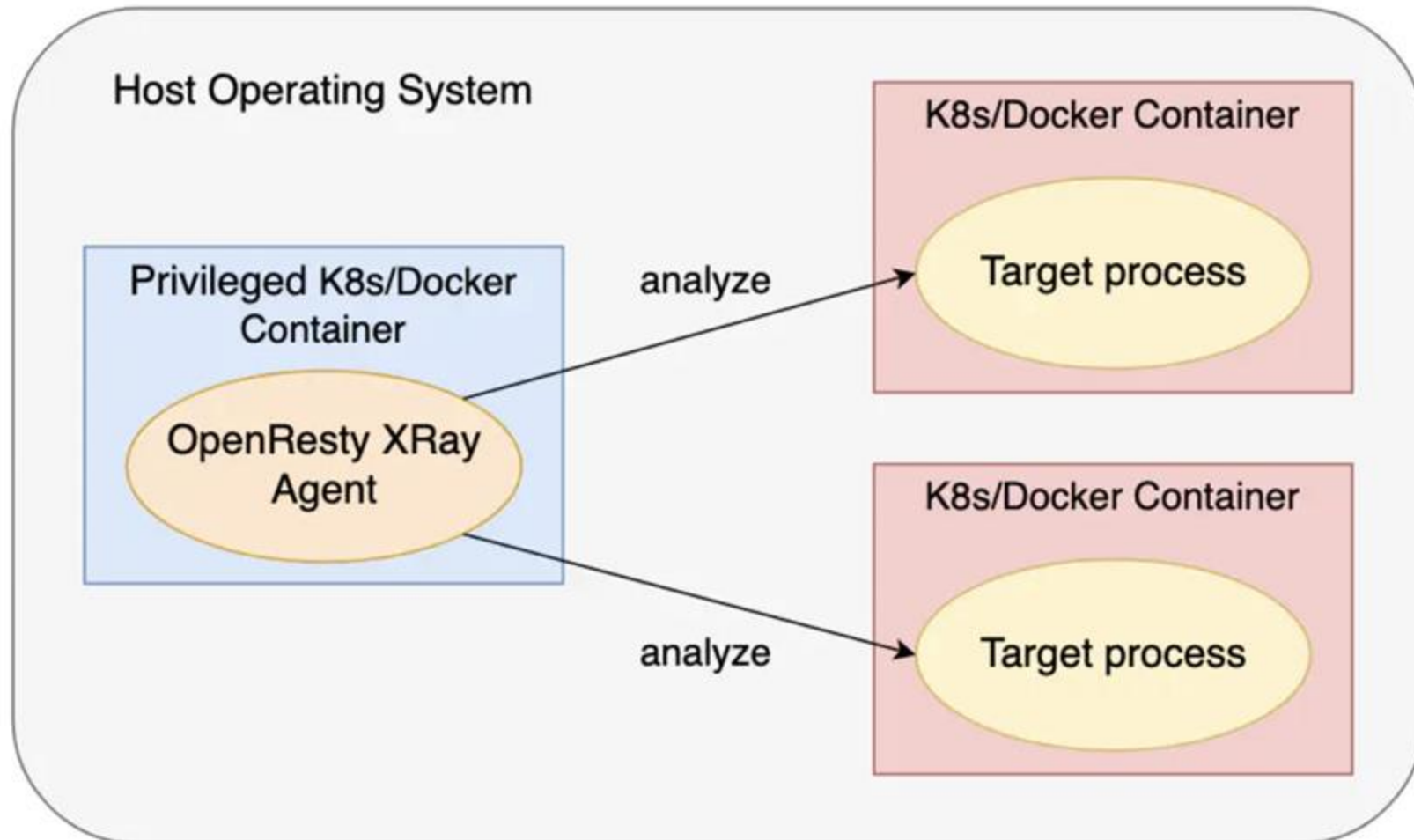


- OpenResty XRay is a dynamic tracing product.
  - Enables real-time analysis of various cloud and server applications.
  - Treats running processes and containers as read-only databases and extracts the necessary information to resolve performance issues, exceptions, errors, and security vulnerabilities.
  - Features a knowledge base, inference engine, and hundreds of advanced analyzers.
  - Can diagnose and narrow down the root cause of deep problems without changing or affecting the target application.
-

# OpenResty XRay Analyzes Non-container Application Processes Directly



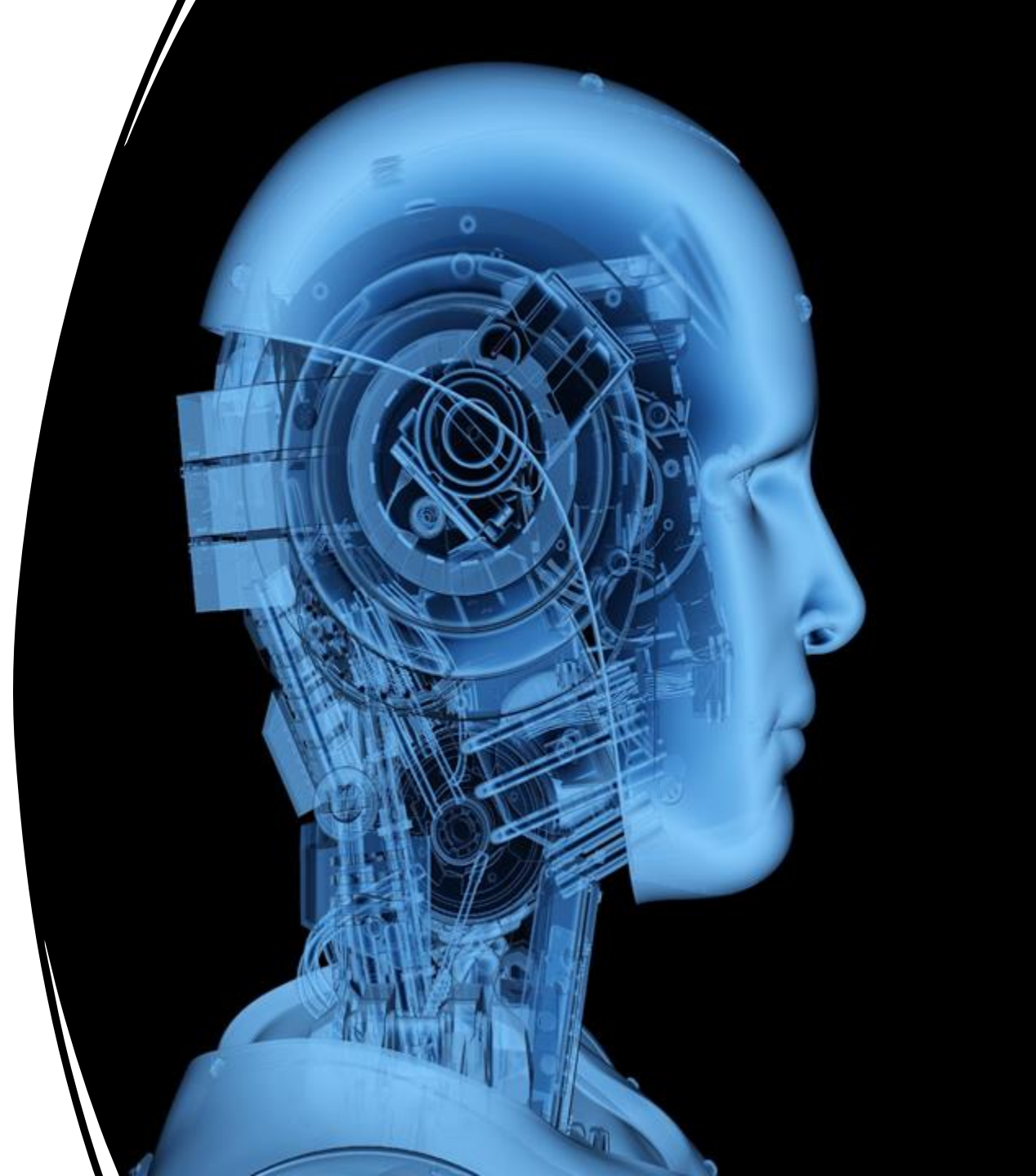
# OpenResty XRay Penetrates Containers and Analyzes Applications



# 100% Non-Invasive

---

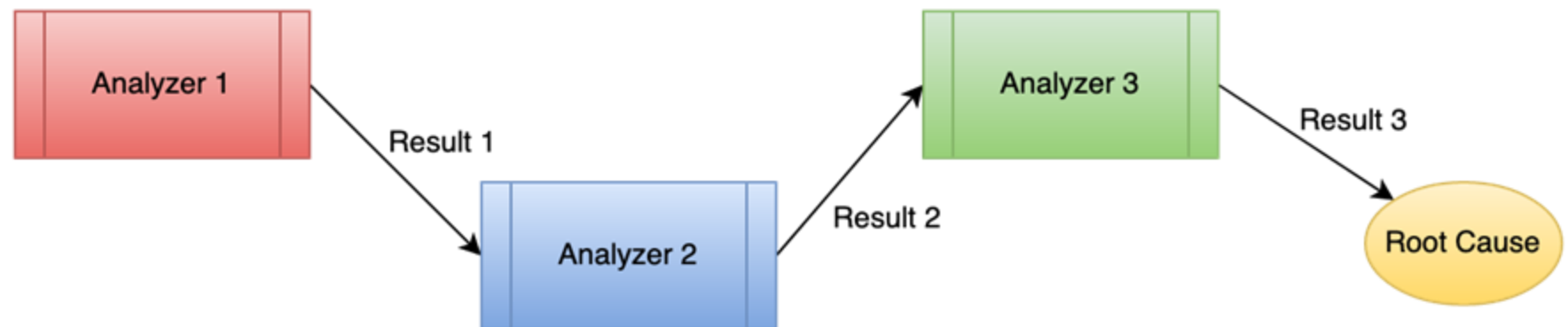
- No need to modify your application
- No need to add new plug-ins, modules, or patches to your application
- No need to inject any code into your applications
- No need to restart your application processes
- No need to use special startup or compilation options in your application
- No need to rebuild your existing application containers or packages





# OpenResty XRay Fully Automatic Sampling, Unattended Usage Mode

- Time sampling
- Event-driven sampling (CPU changes, memory changes, IO changes, exceptions and errors)
- Reasoning chains driven



# Extremely Low Performance Overhead

---

- Performance overhead is strictly 0 when not sampling
- Performance overhead is usually not noticeable when sampling



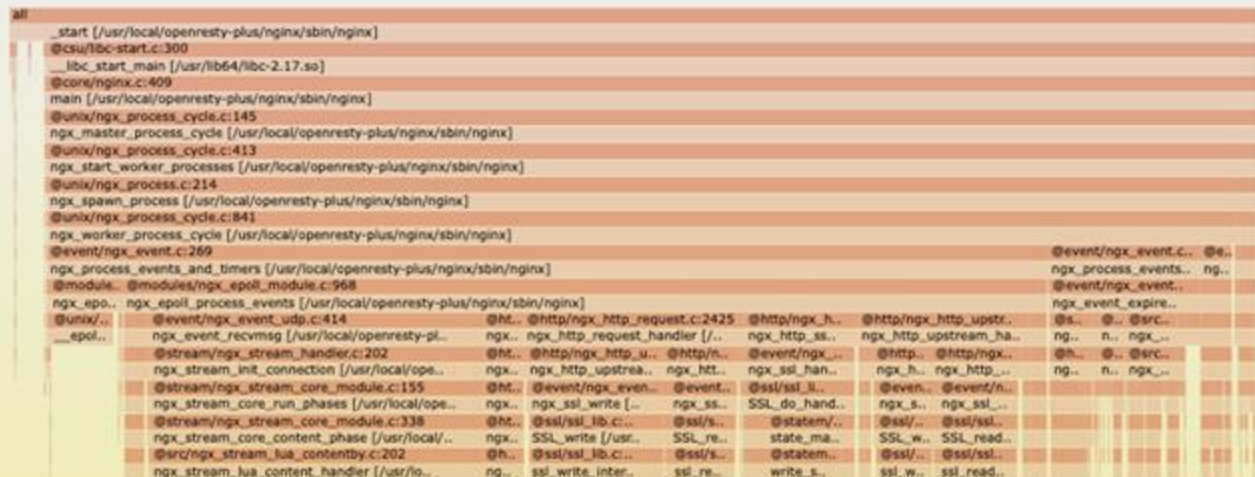
# OpenResty XRay CPU Performance Analysis

- High CPU usage can reduce the system stability and quality of service, and even make services unavailable.
- How CPU time is distributed over different code paths in different scenarios (flame graphs, automatic flame graph interpreter).
- Covers code paths of different software levels: business programming language level (Lua/Python/PHP/Perl/Go/etc.), system programming language level (C/C++/Rust), OS kernel level (network protocol stack/process scheduler/memory management/system calls).
- Examples of common CPU bottlenecks: duplicate computations (lack of cache), SSL handshake related, garbage collection (GC) overhead, dynamic memory allocation overhead, serialization and deserialization, unexpectedly frequent system calls, infinite loops, wrong regular expression matching, inefficiently implemented (third-party) software libraries, spinlock contention.

### C-Land CPU Flame Graph for LuaJIT

Search

whole application



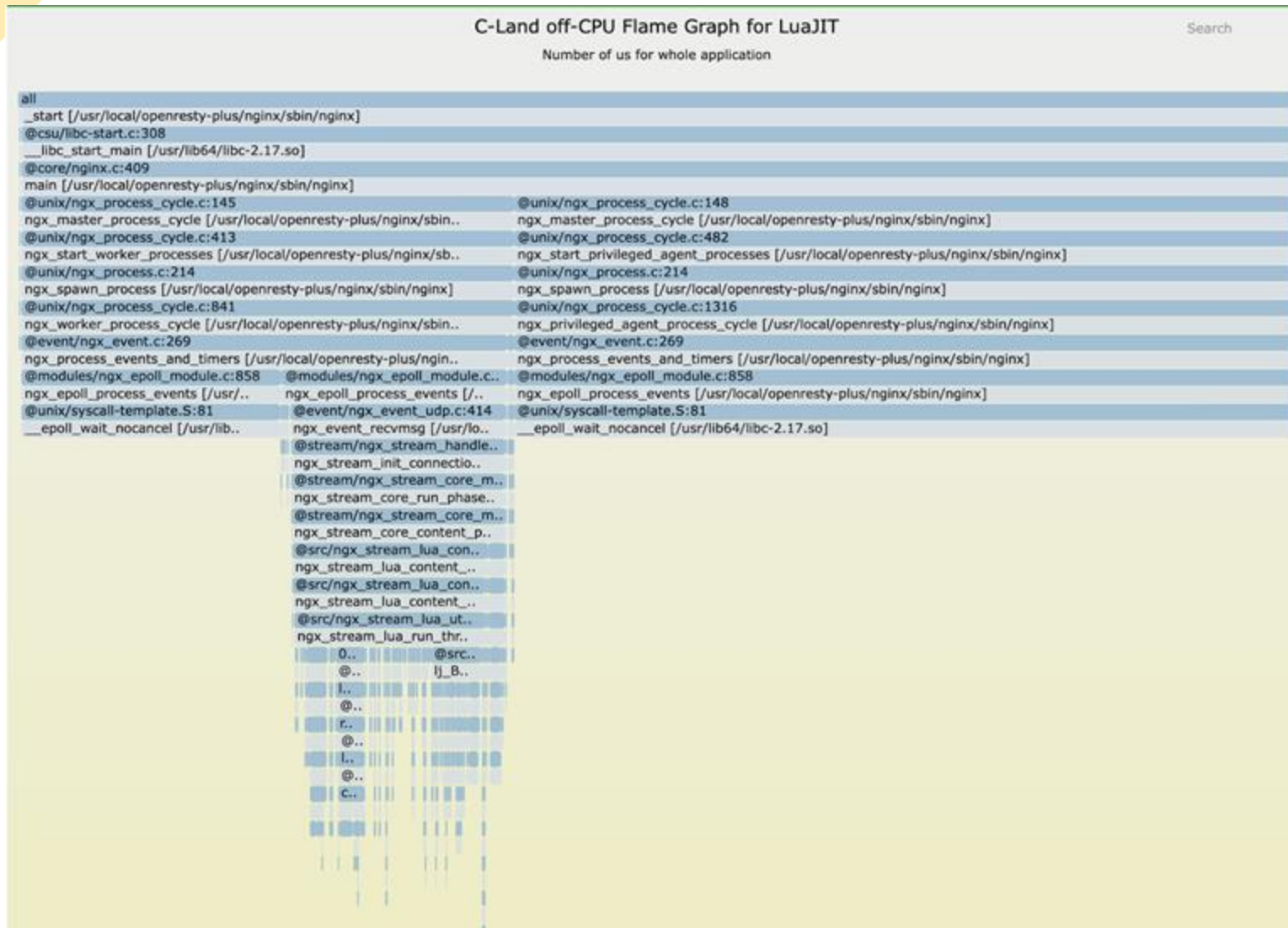
### Lua-Land CPU Flame Graph

Search

whole application



# OpenResty XRay CPU Blocking (off-CPU) Analysis

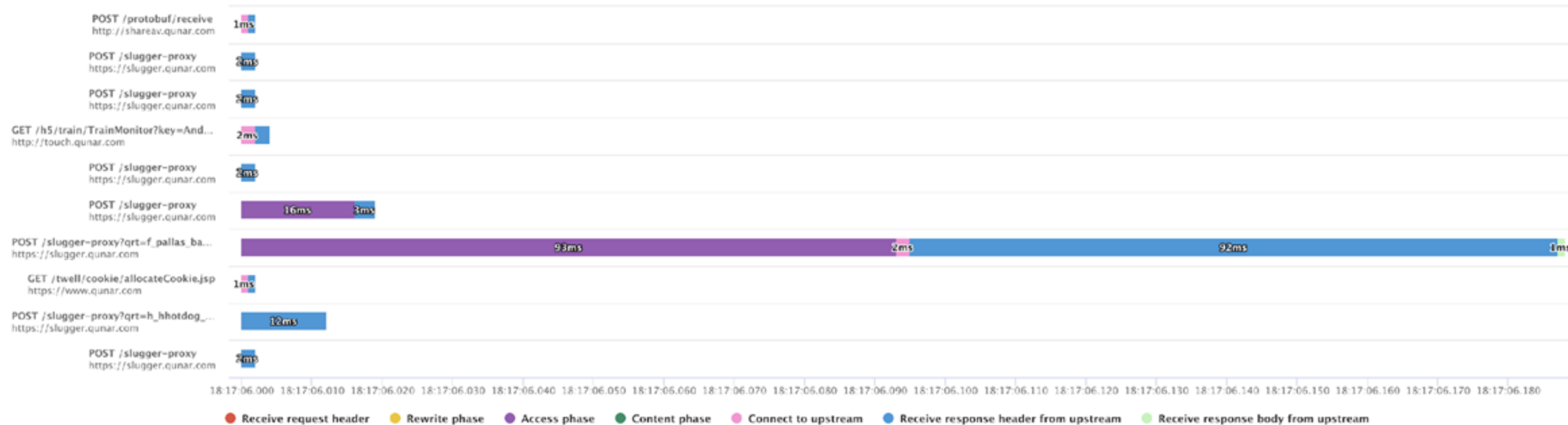


# OpenResty XRay Analysis of Request Latency

- Breakdowns the latency to different operation and processing phases of applications
- Precise packet capture, only captures network packets on problematic connections (including high latency, timeouts, connection errors, when upper layer applications report errors, etc.)
- Latency statistics of asynchronous non-blocking IO (e.g. the distribution of Lua concurrent yield time over Lua code paths)

## Nginx Request Latency

In 0 seconds, total 10 requests, matched 10 requests

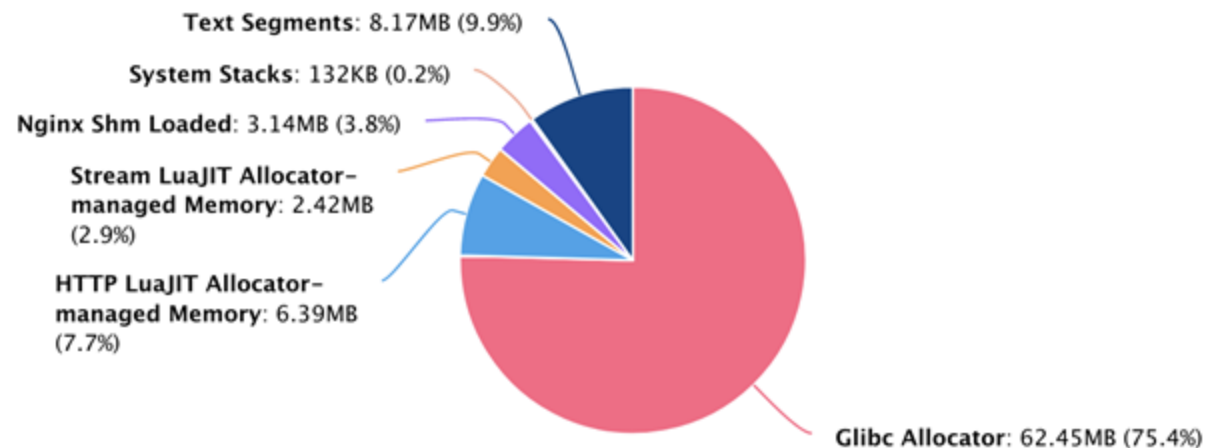


# OpenResty XRay Memory Usage Analysis

- Memory usage of C memory allocators such as Glibc/Jemalloc (including Glibc memory fragmentation).
- How memory is distributed quantitatively over all GC objects (e.g. Lua objects, Python objects, PHP objects, etc.), by reference relationships between GC objects.
- Memory leak, memory fragmentation, or delayed release?

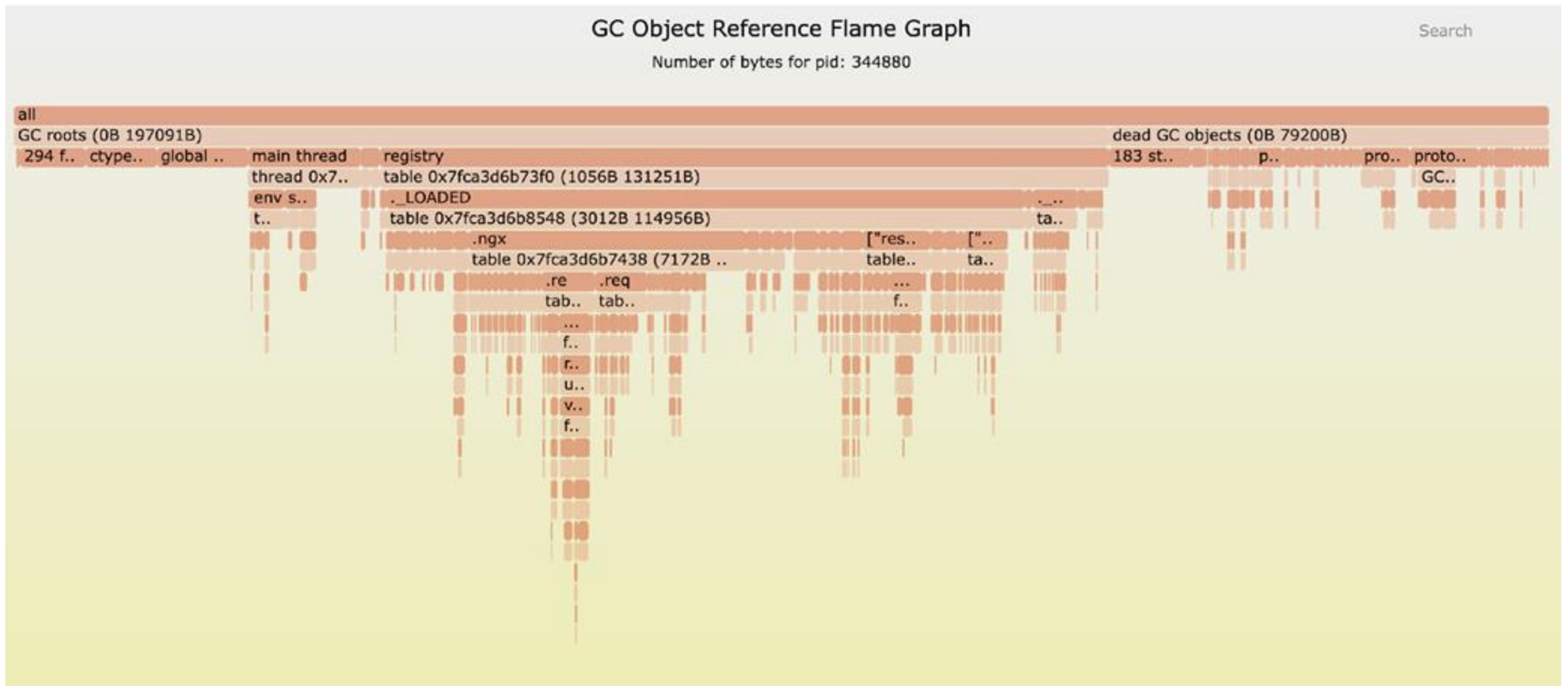
## Application-Level Memory Usage Breakdown

02/04/2023 20:45, Total: 82.78MB



# GC Object Reference Relationship Flame Graph

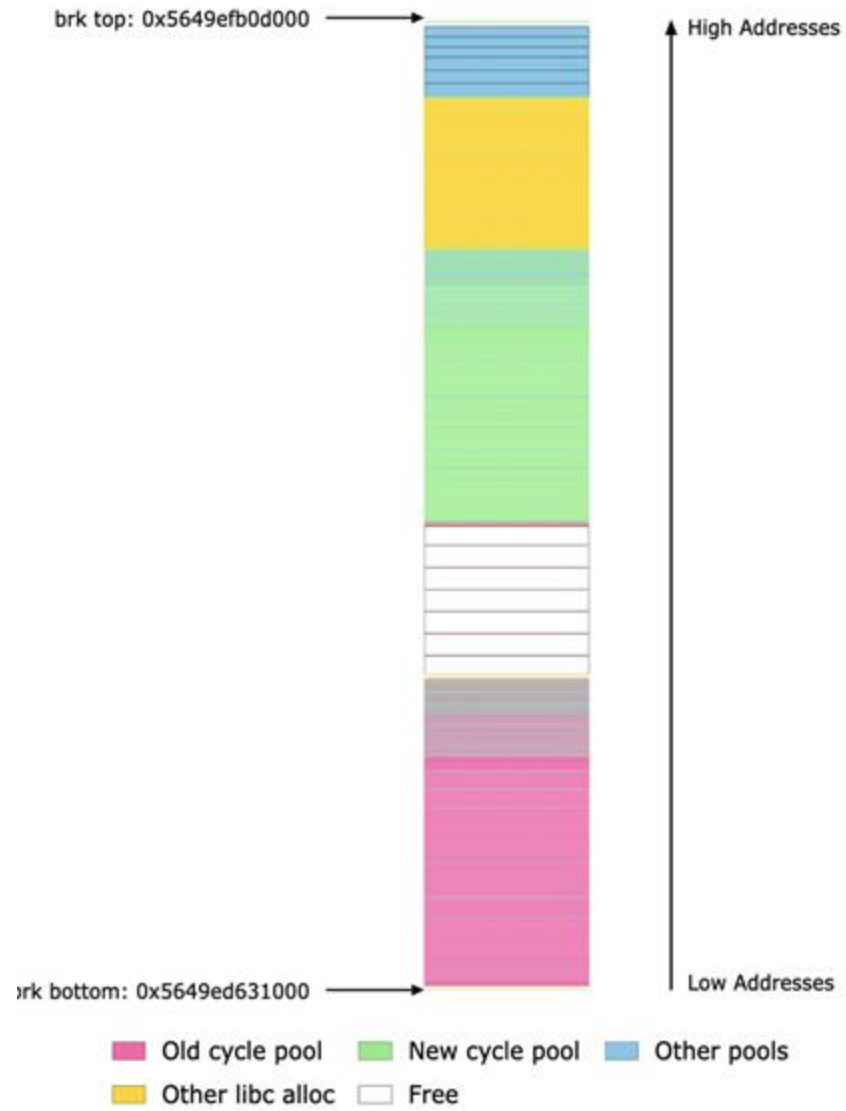
Quantitative visualization of how memory is distributed over all object reference paths





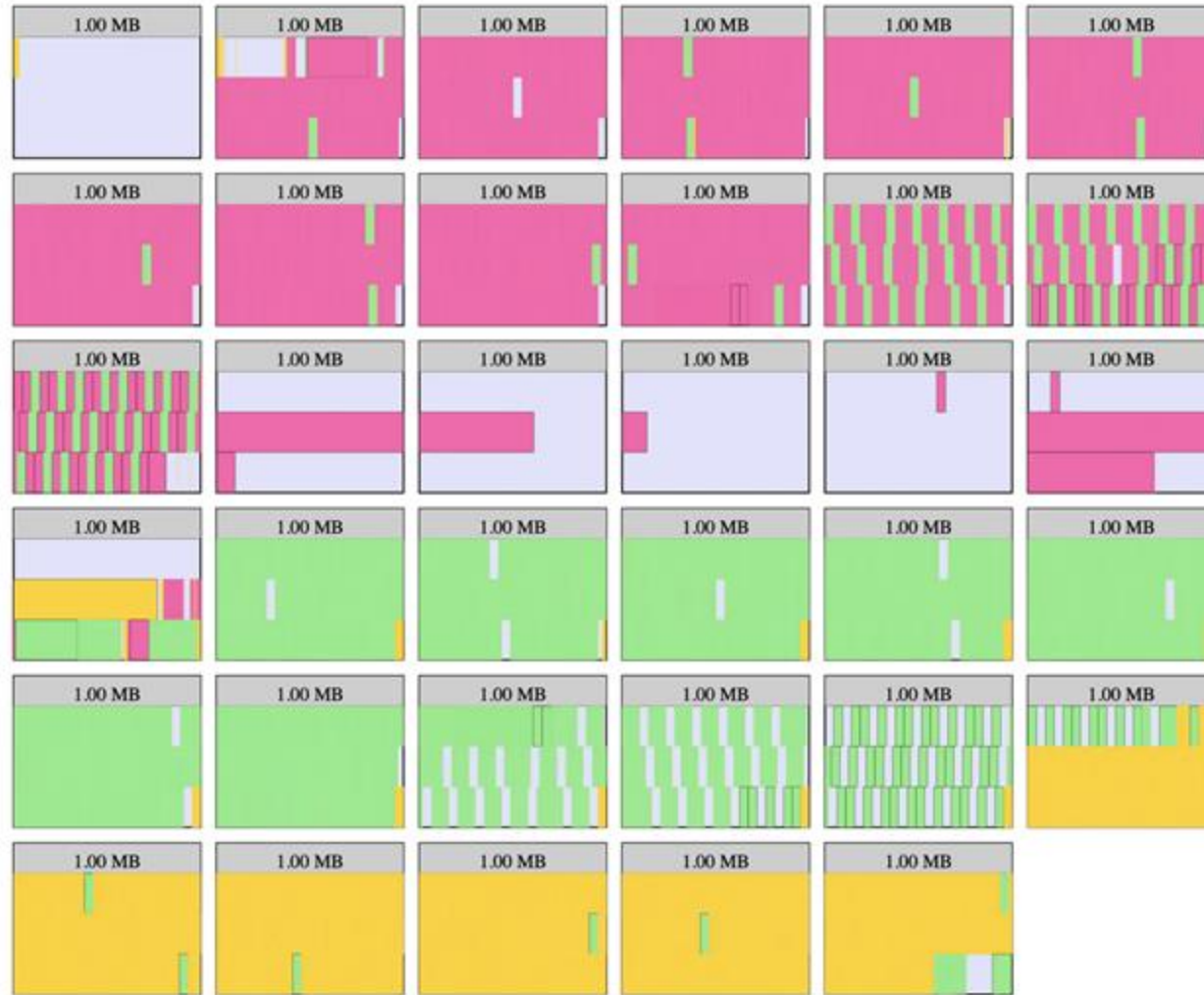
# Nginx memory pool allocations via the brk syscall

When the old and new nginx cycles coexist



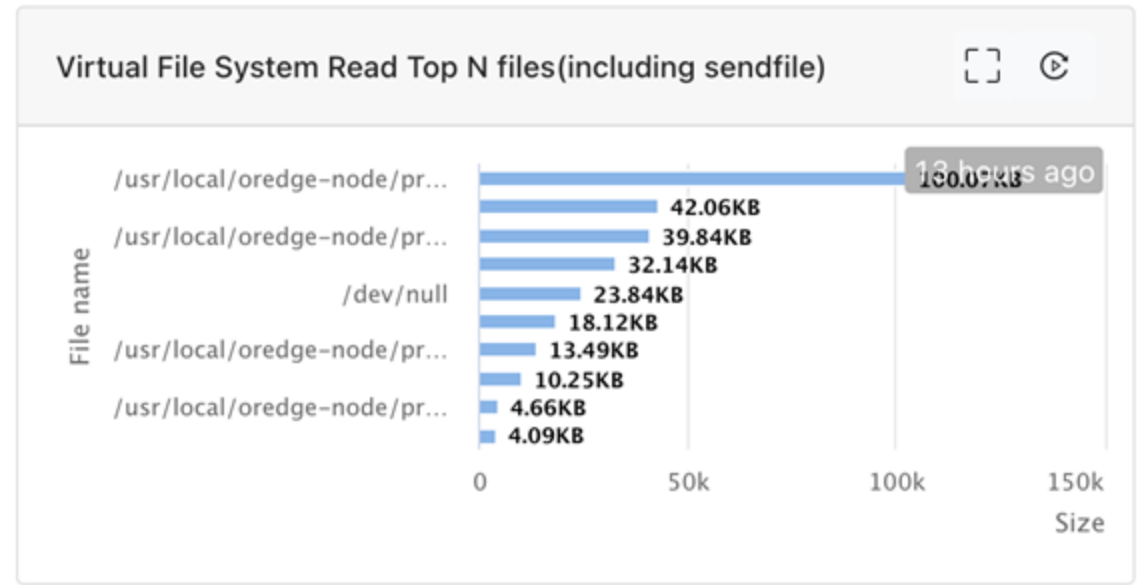
## Nginx memory pool allocations via the mmap syscall

For total 35.00 MB in 35 memory mappings with 49,464 chunks



free      other libc alloc  
old cycle pool      new cycle pool

# OpenResty XRay File IO Performance Analysis



# Intelligent online network packet capture

Captures packets only on abnormal network connections



# OpenResty XRay Automated Analysis Diagnostic Reports

Daily report Weekly report 2023-01-12 [← Previous day](#)

OpenResty XRay Analysis Report for Demo [See the full report →](#) Agent(s): 1 Online time: 23 hours Watch time: 1 day

2023-01-04 16:00:00 ~ 2023-01-05 15:40:59

OpenResty **CPU 25** off-CPU 1 Errors & Exceptions 4 Memory 16

CPU Usage: min: 0%, avg: 0.78%, max: 102%

Command line: nginx: master process /usr/local/openresty/nginx/sbin/nginx  
Exe path: /usr/local/openresty/nginx/sbin/nginx

**CPU 25**

- Lua code execution takes up to **100.00%** of the CPU time of the target processes. [More ↓](#)  
Suggestions: To see how the CPU time is distributed across all the hot Lua code paths, we can analyze the *Lua-land CPU ...* [More ↓](#)
- NEW** [10.1%] #1 of the hottest Lua code paths for CPU time: `lj_err_argt` ← `lj_lib_checkstr[10]` ← `lj_fff_fallback` ← `[builtin#strin` [More ↓](#)
- NEW** [11.9%] #2 of the hottest Lua code paths for CPU time: `gc_onestep` ← `lj_gc_step` ← `lj_gc_step_jit` [More ↓](#)
- NEW** [45.5%] #3 of the hottest Lua code paths for CPU time: `lj_BC_FUNCC` ← `print` ← `C:ngx_http_lua_ngx_print` [More ↓](#)
- The `string.gmatch()` Lua function calls takes up to **89.80%** of the CPU time of the target processes. [More ↓](#)  
Suggestions: Try optimizing the CPU time usage on the code paths highlighted.
- NEW** [29.2%] #1 of the hottest C-land code paths for CPU time: `__GI__writev` ← `ngx_writev` ← `ngx_linux_sendfile_chain` ← `ni` [More ↓](#)

Read blog posts on automated analysis and diagnostic reports

# Automatic Memory Problem Diagnostic Reports

## Memory 15

- **NEW** Glibc allocator takes up to **72.68 MB** in a target process. [More ↓](#)
- **NEW** Glibc arena takes up to **72.68 MB** in a target process. [More ↓](#)
- **NEW** In-use total memory in glibc arena takes up to **67.06 MB** in a target process. [More ↓](#)
- **NEW** Reserved free memory in glibc arena takes up to **5.67 MB** in a target process. [More ↓](#)
- **NEW** Free memory reserved by the LuaJIT allocator takes up to **12.19 MB** in a target process. [More ↓](#)
- **NEW** In-use total memory by the LuaJIT allocator takes up to **4.26 MB** in a target process. [More ↓](#)
- **NEW** Lua GC size of all types takes up to **1.45 MB** in a target process. [More ↓](#)
- **NEW** [10.7%] #1 of the hottest reference paths for LuaJIT GC object: `table 0x7f950c0a0828 (584B 524.40KB)` ← `[light userdata]` [More ↓](#)
- **NEW** [12.4%] #2 of the hottest reference paths for LuaJIT GC object: `trace 0x7f950a93a8e8 (2.36KB 161.13KB)` ← `next side` ← [More ↓](#)

# Automatic Latency Analysis and Diagnosis

## Latency 5

▼ ↑ 22.2% [100%] #1 of the hottest Lua code paths for Newly Created CoSocket: C:ngx\_http\_lua\_socket\_tcp\_connect ← connect ←

C:ngx\_http\_lua\_socket\_tcp\_connect ← check\_peer ← spawn\_checker ← check\_peers ← pcall ← [builtin#pcall]

See [Job 4418885005](#) for more details.

[Collapse ↑](#)

▶ [100%] #2 of the hottest Lua code paths for Newly Created CoSocket: C:ngx\_http\_lua\_socket\_tcp\_connect ← connect ← C:ngx\_http\_lua\_socket\_tcp\_connect ← \_request ← request\_admin ← pcall ← [builtin#pcall] [More ↓](#)

▶ ↑ 105.67 ms [106.97 ms] #1 of the hottest Lua code paths for Request Yield Latency: lua\_yield ← lj\_BC\_FUNCC ← ngx\_sleep ← C:ngx\_http\_lua\_ngx\_sleep ← limit\_req\_rate ← helper\_1 ← xpcall ← [builtin#xpcall] ← run\_rewrite\_phase ← access ← access\_by\_lua(nginx.conf:586) [More ↓](#)

▶ ↑ 4.00 ms [16.00 ms] #2 of the hottest Lua code paths for Request Yield Latency: lua\_yield ← ngx\_stream\_lua\_socket\_tcp\_receive ← lj\_BC\_FUNCC ← receive ← C:ngx\_stream\_lua\_socket\_tcp\_receive ← go ← content\_by\_lua(nginx.conf:140) [More ↓](#)

▶ [1.58 ms] #2 of the hottest Lua code paths for Request Yield Latency: lua\_yield ← lj\_BC\_FUNCC ← ngx\_sleep ← C:ngx\_stream\_lua\_ngx\_sleep ← limit\_req ← process\_req ← go ← content\_by\_lua(nginx.conf:132) [More ↓](#)

# off-CPU Automatic Diagnostic Report

## off-CPU 8

- ▶ NEW Lua code execution takes up to **99.99%** of the off-CPU time of the target processes. [More ↓](#)

Suggestions: Try optimizing the off-CPU time usage on the code paths highlighted.

- ▶ NEW [21.5%] #1 of the hottest Lua code paths for off-CPU time: `__read_nocancel` ← `_IO_file_underflow@@GLIBC_2.2.5[2]` ← `_IO_default_xsgetn[2]` ← `lj_BC_FUNCC` ← `read` ← `[builtin#` [More ↓](#)
- ▶ NEW [31.7%] #2 of the hottest Lua code paths for off-CPU time: `__read_nocancel` ← `fread[2]` ← `lj_BC_FUNCC` ← `read` ← `[builtin#io.method.read]` ← `_getAccessLog` ← `_getIfNumber` [More ↓](#)
- ▶ NEW [32.6%] #3 of the hottest Lua code paths for off-CPU time: `__read_nocancel` ← `lj_BC_FUNCC` ← `read` ← `[builtin#io.method.read]` ← `_getProxyIgnoreHeaderInfo` ← `_getIfNumber` [More ↓](#)



# Automatic Diagnostic Reports on Errors and Exceptions

## Errors & Exceptions 2

- ▶ **NEW** [100%] #1 of the hottest Lua code paths throwing out Lua exceptions: 71) ← no field package.preload['test'] ← no file ' / [More ↓](#)
- ▶ **NEW** [100%] #2 of the hottest Lua code paths throwing out Lua exceptions: 166) ← no field package.preload['resty.http'] ← r [More ↓](#)

# Automatic Analysis of Security Issues

---

Automatic checking and reporting of connections without TLS encryption

Dynamic scanning of TLS connections without certificate source verification

Check the usage of a non-secure version of the SSL protocol

Scan remote shell command execution events and code contexts



# Core Dump Process Remains Analysis (Process Crashes)

All

Core Dump Analysis

Application Type \*

OpenResty

Analyzer \*

openresty-core-dump-analysis (OpenResty Core Dump Analysis)

Core File \*

/tmp/core.3859164

Executable File Path \*

/usr/local/openresty/nginx/sbin/nginx

► Advanced Settings

Core File Meta Data

File Name: /tmp/core.3859164

Executable File Path: /usr/local/openresty/nginx/sbin/nginx

Size: 14.72MB

Analyze

# Extract Deep Information from Core Dump Files

Analysis 7173425 [🔗](#)

## (gdb) lbt

```
C:ngx_md5_body
trace#1:access.lua:4
check_token
@/usr/local/openresty/lualib/access.lua:3
auth
@/usr/local/openresty/lualib/access.lua:21
@access_by_lua(nginx.conf:51):2
```

## (gdb) full\_lbt

```
C:ngx_md5_body
trace#1:access.lua:4
check_token
@/usr/local/openresty/lualib/access.lua:3
auth
@/usr/local/openresty/lualib/access.lua:21
headers = "access"
token = "hello"
@access_by_lua(nginx.conf:51):2
```

Compiler Output

Analyzer Output

Graphs

## (gdb) ngx\_process\_info

```
parent: 3859163
process: worker 0
```

## (gdb) cur\_http\_req

```
current phase: access
schema: http, req_size: 52, resp_size: 0GET / HTTP/1.1
Host: localhost:80
TOKEN: hello
```

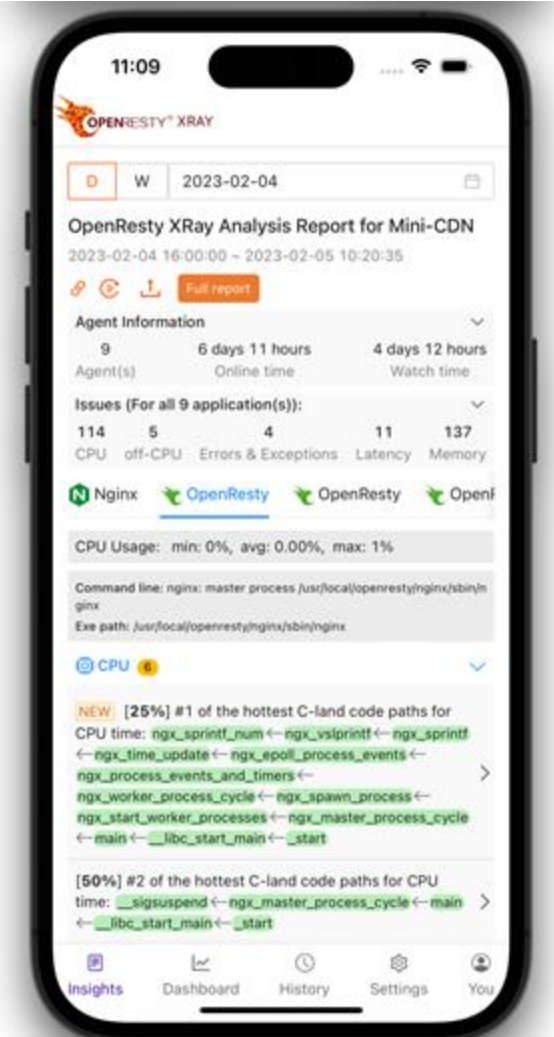
## (gdb) ubt

```
0x42e958 ngx_md5_body [/usr/src/debug/openresty-1.21.4.1/build/nginx-1.21.4/src/core/ngx_md5.c:199]
0x42f1be ngx_md5_final [/usr/src/debug/openresty-1.21.4.1/build/nginx-1.21.4/src/core/ngx_md5.c:91]
0x4ea997 ngx_http_lua_ffi_md5 [/usr/src/debug/openresty-1.21.4.1/build/nginx-1.21.4/./ngx_lua-0.10.21/s
7f763447ffd3: []
0x4f86a1 ngx_http_lua_run_thread [/usr/src/debug/openresty-1.21.4.1/build/nginx-1.21.4/./ngx_lua-0.10.2
```

# OpenResty XRay Mobile App

Watch your applications from  
anywhere, any time

- Android (Google Play)
- iOS (Apple Store)



## OpenResty XRay is Not just for OpenResty Applications

- Nginx, LuaJIT, OpenResty
- Preliminary Support: Python, PHP, Perl, Redis, PostgreSQL
- Coming soon: Go, Ruby, NodeJS, Java



# Supports Most Mainstream Linux Distributions and Container Deployment



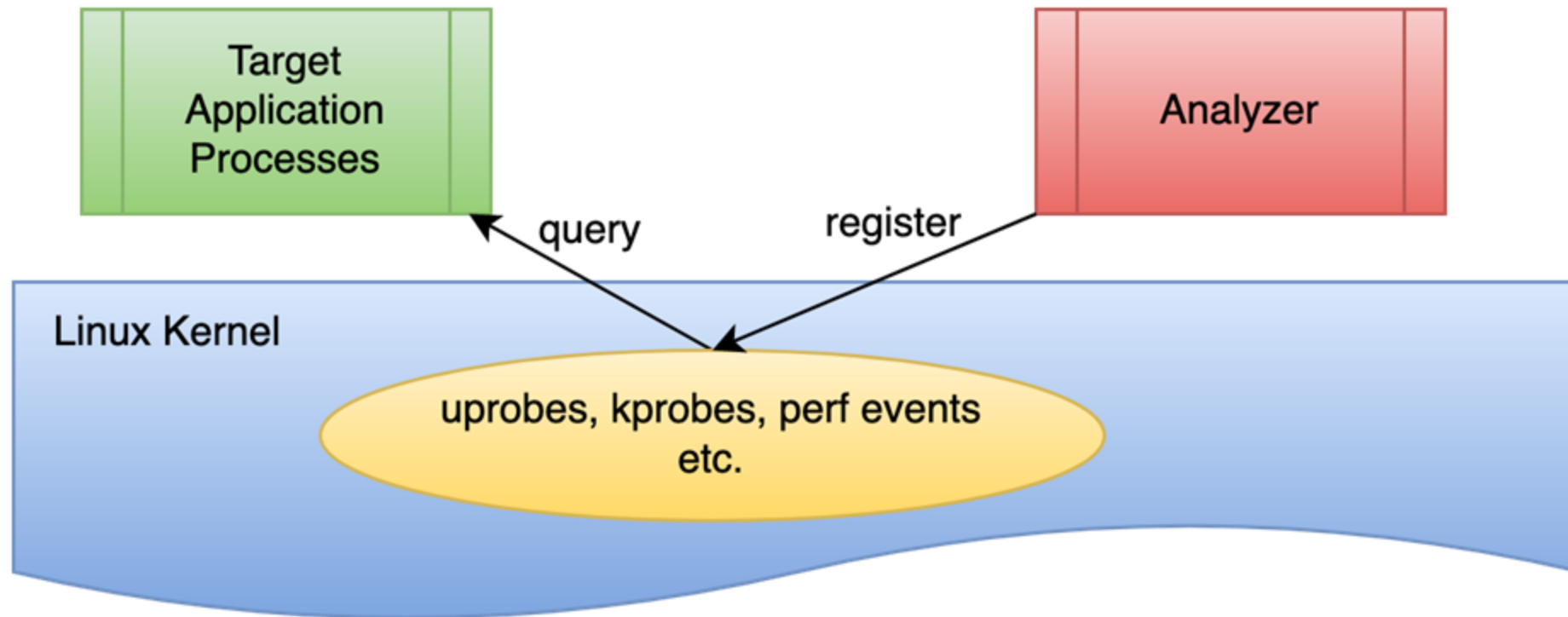
Ubuntu



Debian



# OpenResty XRay is based On the Advanced Dynamic Tracing Technology





# Advantages of Dynamic Tracing

- Non-invasive, no need to modify applications
- Hot-plugging, usually does not need application cooperation (many open-source dynamic tracing tools still need application cooperation)
- Overhead is normally low and aggregation can be done at the data source
- Online real-time debug capability in a postmortem manner
- Full technology stack analysis from all angles
- On-demand sampling
- Strictly 0 loss when not sampling

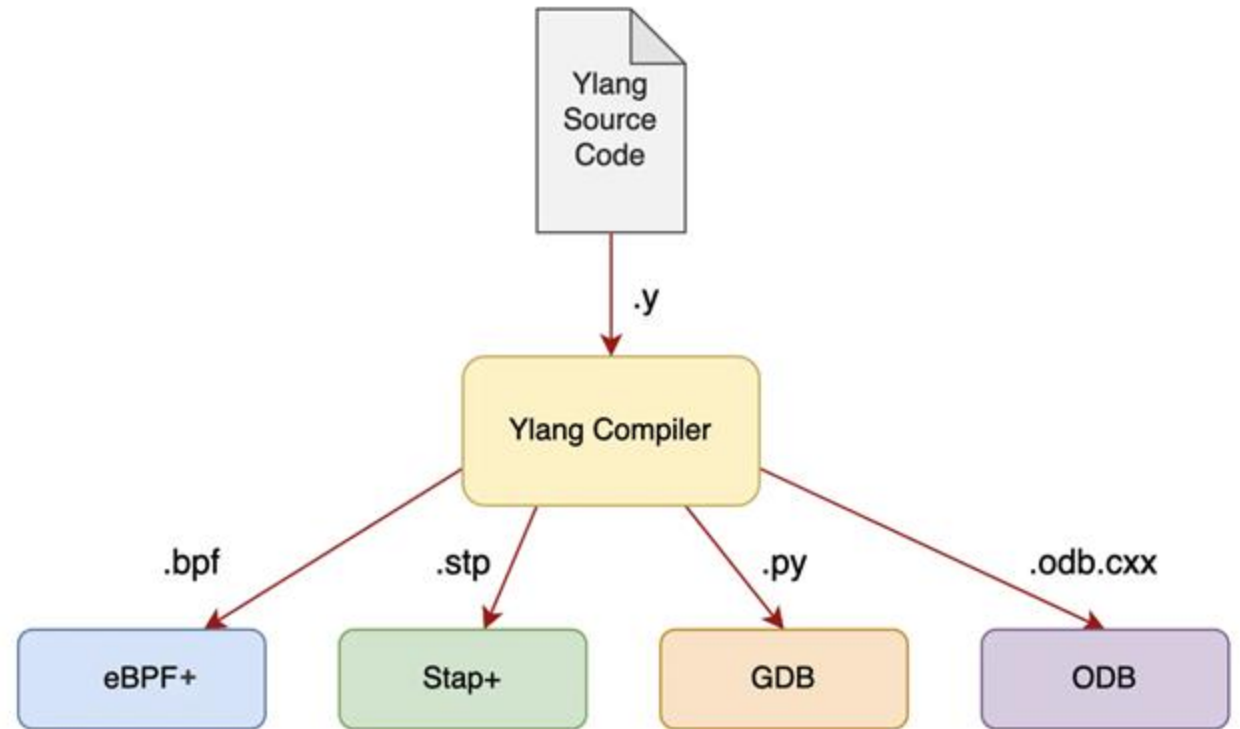
# OpenResty XRay New generation of Dynamic Tracing Technology

---

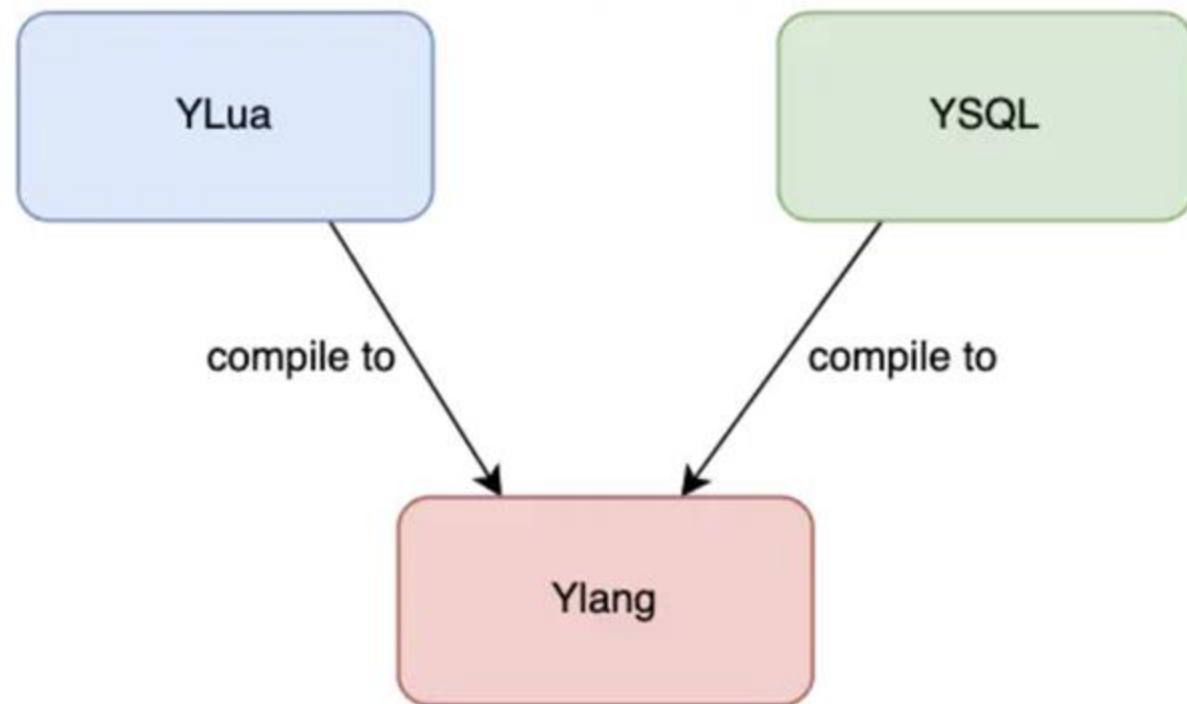
- Y-language (Ylang) compiler (supports most of the syntax of GNU C and standard C)
- Ylua Language
- YSQL Language
- Stap+ has significantly improved SystemTap
- eBPF+ significantly improves eBPF (while LLVM+ also significantly improves open-source LLVM)
- ODB is an ultra-lightweight version of GDB
- The Ylang compiler can also generate highly optimized Python extension codes of GDB
- Stringent performance loss control aimed at online production environments

“Write once,  
run  
everywhere”

From Y-language code to  
various runtime codes



# More Abstract Languages Based on Y Language



# Write and Debug Analysis Tools Written in Languages like Ylang/YLua/YSQL on the OpenResty Xray Console's Web UI

OPENRESTY® XRAY Version: 844 prd.openresty.com (3... yichun@openresty.com English

All analyzers / Edit analyzer [Clone this analyzer](#) [+ Add a new analyzer](#)

Analyzer name: process-exit Analyzer description: C-land CPU Flame Graph

YSQL YLua YLang [Learn YLang](#) [Run](#) [Save](#)

```
1 // c-cpu: C-land CPU flame graph sampling tool.
2 // Copyright (C) OpenResty Inc.
3 // All rights reserved.
4
5 _target sig_atomic_t ngx_quit;
6 _target sig_atomic_t ngx_debug_quit;
7 _target ngx_uint_t ngx_exiting;
8 _target sig_atomic_t ngx_reconfigure;
9 _target sig_atomic_t ngx_reopen;
10
11 _probe _process.begin
12 {
13     printf("ngx_quit %ld\n", ngx_quit);
14     printf("ngx_exiting %ld\n", ngx_exiting);
15     printf("ngx_reconfigure %ld\n", ngx_reconfigure);
16     _exit();
17 }
18
19
```

[Run](#) [Clear the editor](#)

[Analysis history for this analyzer](#)

▼ Try this analyzer against:

Target

By Applications	By Processes	By Executables	Whole System
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Application: openresty (PGID 1471, master proc) [Update](#)

Target Processes: PID 2782 (worker process) CPU: 1% [Update](#)

► Advanced Settings

▼ YLang Settings

Dependent debug data to compile the analyzer

**i** We have a huge package archive database for public software and we will not install any debuginfo packages on the target machine.

Kernel debug info:  Not required  Required  Good to have

Unwind data:  NO NEED

**i** Need unwind data when you need the backtrace.

# OpenResty XRay Hundreds of Standard Analyzers

## ▼ OpenResty XRay Standard Analyzers

- c-alloc-fgraph
- c-count-alloc-free
- c-memory
- c-memory-leak-fgraph
- c-off-cpu
- c-on-cpu
- collect-luajit-ffnames
- cpu-hogs
- epoll-loop-blocking-distr
- epoll-sched-latency-distr
- epoll-wait-ret-distr
- epoll-wait-timers
- epoll-wait-timers-fgraph
- file-system-fgraph
- func-latency-distr
- glibc-chunks
- jemalloc-bins

- kernel-on-cpu
- lj-add-timer-lua-fgraph
- lj-alloc-stats
- lj-c-memory-leak-fgraph
- lj-c-off-cpu
- lj-c-on-cpu
- lj-config
- lj-dump-loaded-mods
- lj-err-mem
- lj-excep-lua-fgraph
- lj-free-stats
- lj-gc-step-calls
- lj-lua-exception
- lj-gco-ref
- lj-gco-stat
- lj-lua-err-msg
- lj-lua-new-timer-errors
- lj-lua-newcdata
- lj-lua-newfunc
- lj-lua-newgco

- ngx-add-timer-event-fgraph
- lj-trace-stats
- ngx-add-timer-event-timer-distr
- lj-vm-states
- mmap-leaks
- musl-libc-chunks
- ngx-access-log-buffer-size
- ngx-config
- ngx-config-servers
- ngx-cpu-hottest-hosts
- ngx-cpu-hottest-uris
- ngx-downstream-keepalive-stats
- ngx-dump-req
- ngx-dump-timers
- ngx-epoll-wait-timers
- ngx-err-log-lvl-distr

# Debug symbols

- OpenResty XRay has a central package database indexing hundred TB of debug symbols for public packages. This database is still growing rapidly.
- The target machine does not need to install or store debug symbols, as long as they have been indexed by the OpenResty XRay Central Package Database.
- For applications where debug symbols cannot be found or were not generated at compile time, OpenResty XRay will be able to automatically rebuild debug symbols (a prototype of a working machine learning algorithm already exists).



# Trusted By Lots of Enterprise Customers



**Qunar.Com**

zoom



# Learn More

[OpenResty XRay Frequently Asked Questions](#)

[Visit OpenResty XRay official blog](#)

[Visit OpenResty XRay official website](#)

[Try OpenResty XRay for free](#)