OFF ONE 2023

ISP RAS

Fuzzing for SDL: Select, Cover, Reveal

Speaker: Vartan Padaryan, Ph.D.

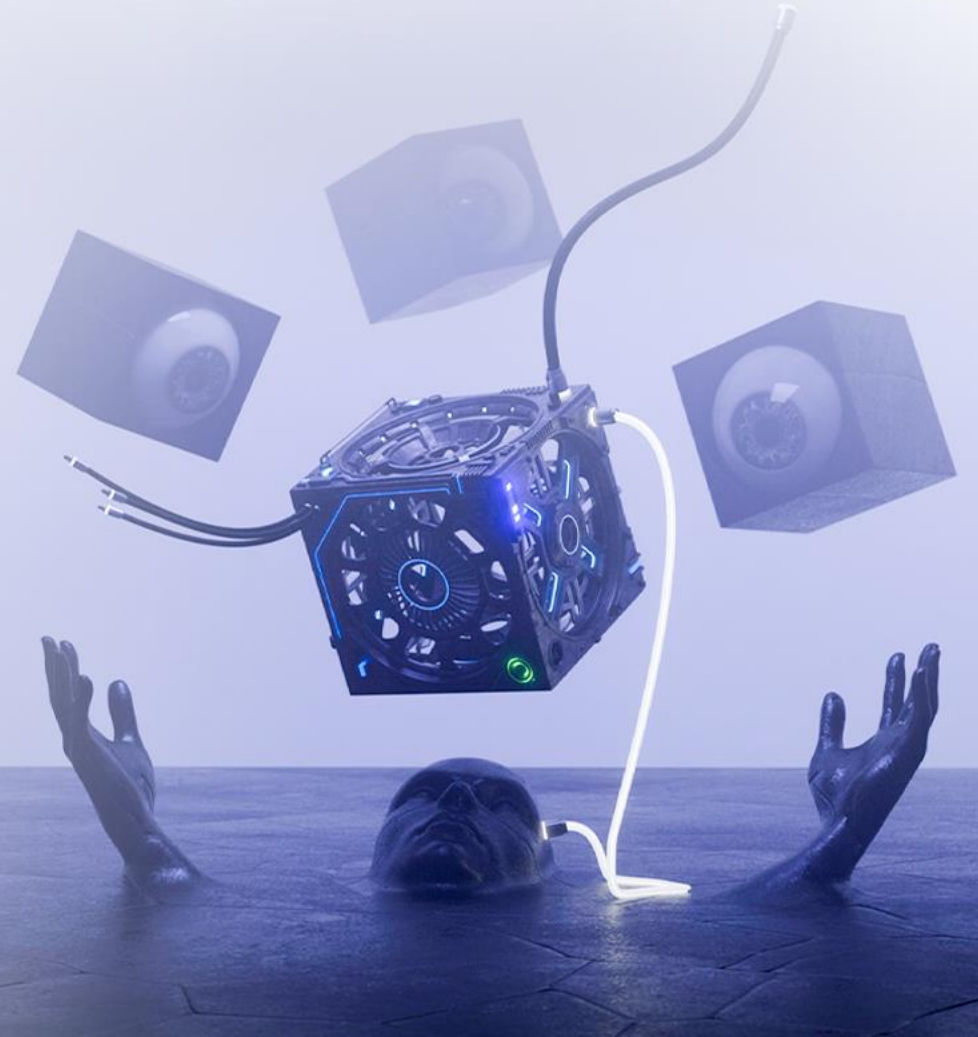
Head of Binary Code Reverse Engineering Laboratory, ISP RAS

Speaker: Vlad Stepanov

ISP RAS

Speaker: Alexey Vishnyakov, Ph.D.

Infosec.exchange/@VishnyaSweet



About ISP RAS

25+ success years of research and development based on foundational scientific school

700+ researchers and engineers

30+ work directions, including: program analysis and cybersecurity, big data analysis, artificial intelligence, operating systems, mathematical modeling, nD-modeling

Long-term contracts (10+ years) and joint R&D labs with Samsung and Huawei

Industrial digital cardiology projects jointly with Sechenov University (Moscow) and others

Cooperation with international open source communities (e.g. finding and fixing errors in the Linux operating system and PyTorch and TensorFlow, popular machine learning frameworks)

Three system programming chairs in leading Russian universities: Moscow State University, Moscow Institute for Physics and Technology, Higher School of Economics

Our products are used by 100+ companies in Russia and abroad

We:

- **ensure security of world famous products**
- **create analytical systems that simplify work in many application areas**
- **improve the world famous open source software**

Cybersecurity Technology Stack

We are the only organization in Russia who has created and implemented a full stack of technologies to ensure the life cycle of secure software development

Trusted safe compiler

SAFEC

No open source competitors

Static source code analysis tool

SVACE

World level analysis quality and usability (like Synopsis Coverity Static Analysis, Perforce Klocwork Static Code Analysis, Fortify Static Code Analyzer) that can be tailored out to create a unique in-house tool

Complex dynamic analysis system

CRUSHER

No open source competitors, similar tools are closed and US owned

Automatic analysis of attack surface

NATCH

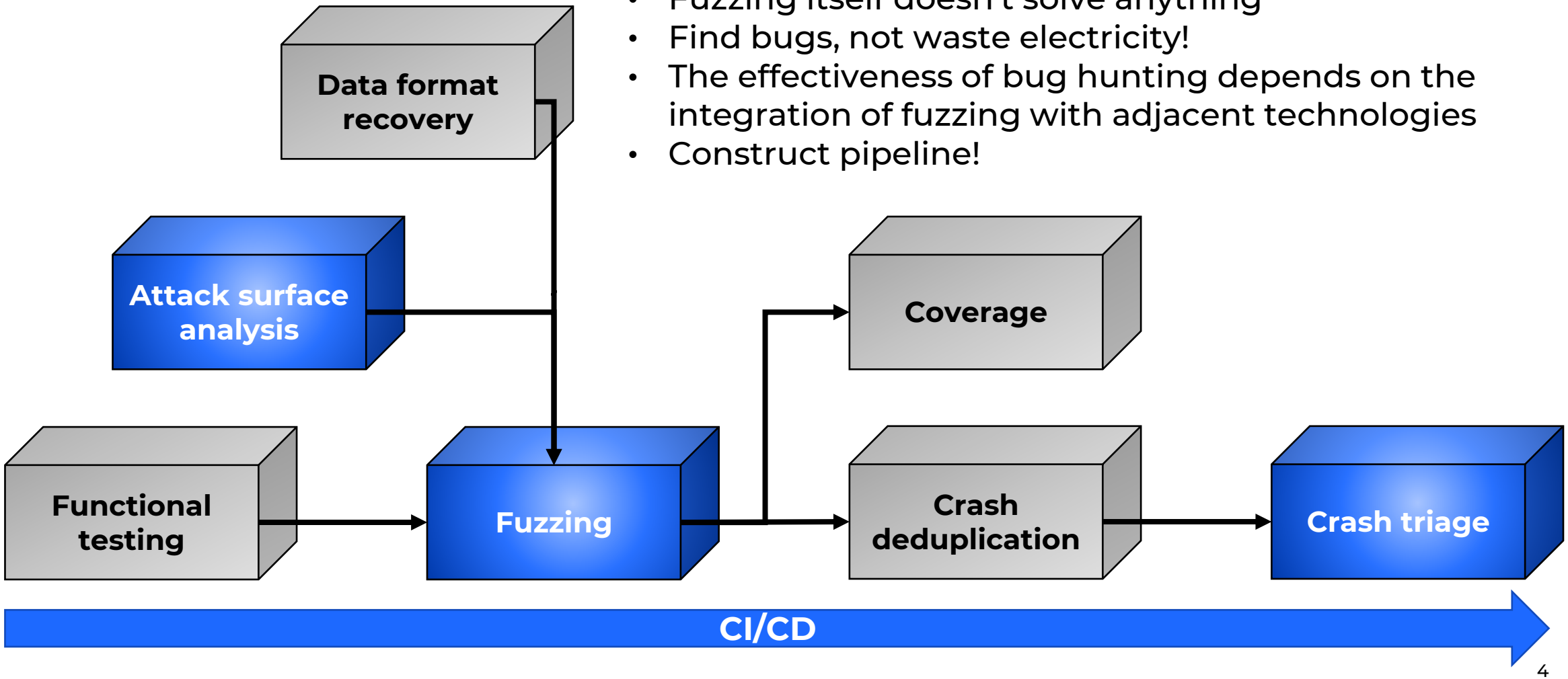
No open source competitors

Main static source code analysis tool in Samsung since 2015, also used in Huawei, Kaspersky Lab and 100+ other companies world wide

- checks all Samsung mobile software based on Android and Tizen (own Samsung operating system used in TVs, entertainment systems, appliances, smartphones):
  - ✓ is used by 10 000 developers
  - ✓ has analyzed 300 billion lines of code
- finds 50+ critical error types in program source code
- unites 6 programming languages, 20+ compilers, 10+ architectures

Every End is a New Beginning

- Fuzzing itself doesn't solve anything
- Find bugs, not waste electricity!
- The effectiveness of bug hunting depends on the integration of fuzzing with adjacent technologies
- Construct pipeline!



Natch: Detecting Attack Surface with Dynamic Taint Analysis

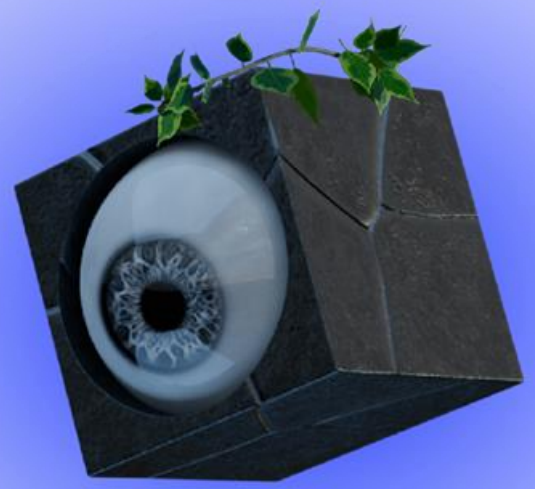
Motivation

The main concept that sets the priority of choosing targets for fuzzing is the attack surface.

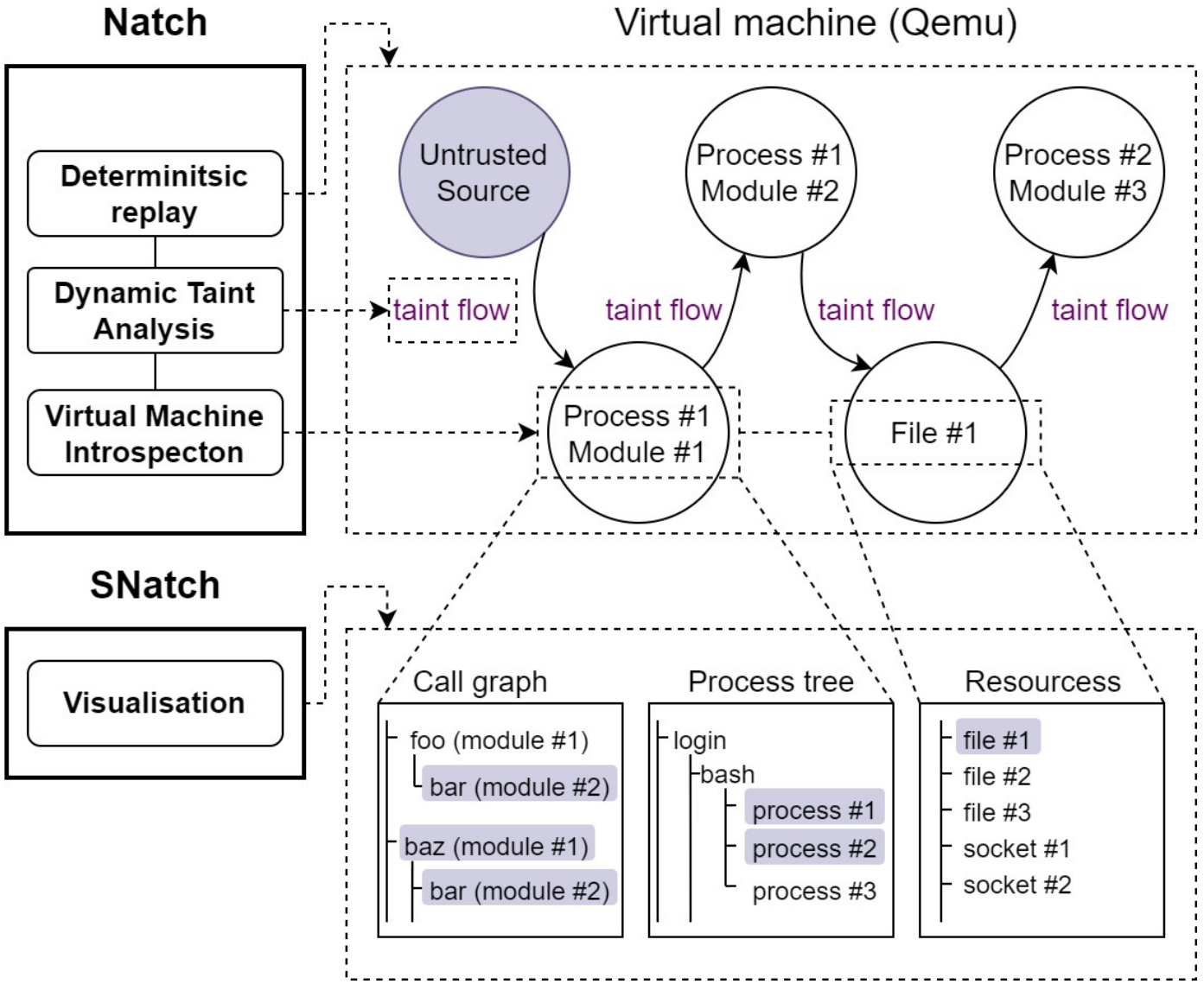
Attack surface is a set of software system interfaces directly or indirectly available for external influence. Determining the attack surface now is a manual job done by an expert.

Attack Surface

- Files
- Processes
- Sockets
- Scripts
- Loadable modules
- Tainted data handlers



Automatic Attack Surface Detecting



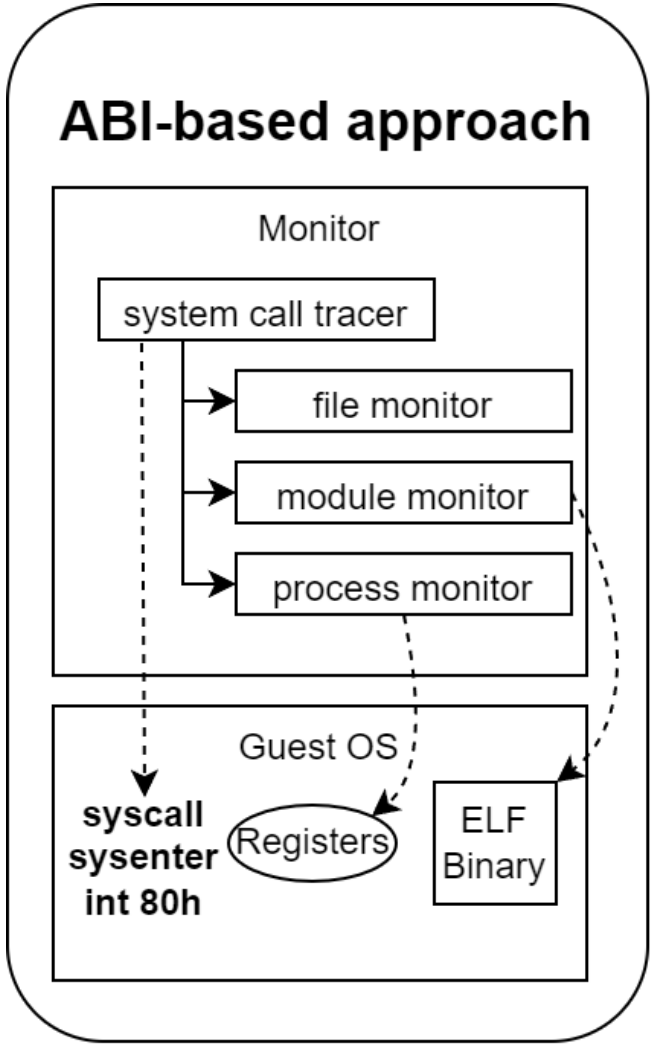
Our goals

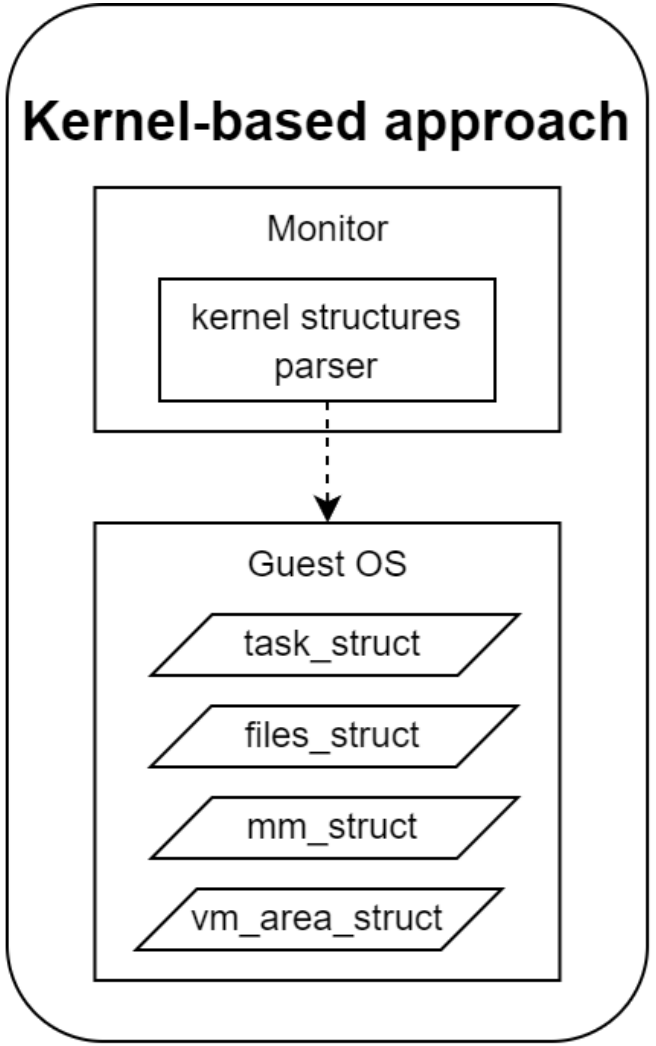
- Simplify and reduce the cost of developing and certifying complex software
- Increase the security of software products by eliminating human error in determining the attack surface

Virtual Machine Introspection

Our approach:

- There is no need to inject agents into the guest code or have access to the source code of the system
- It is based on the system calls hooking, parsing the parameters of system functions and dumps of loaded modules
- It also parses Linux kernel structures that store information about running processes





VMI Profile Generation

Existing solutions

- Debugger-assisted methods (Volatility)
- Compiler-assisted methods (SigGraph)
- Guest-assisted methods (Panda, Decaf)
- Binary analysis-assisted methods (Origen, AutoProfiler, Katana)

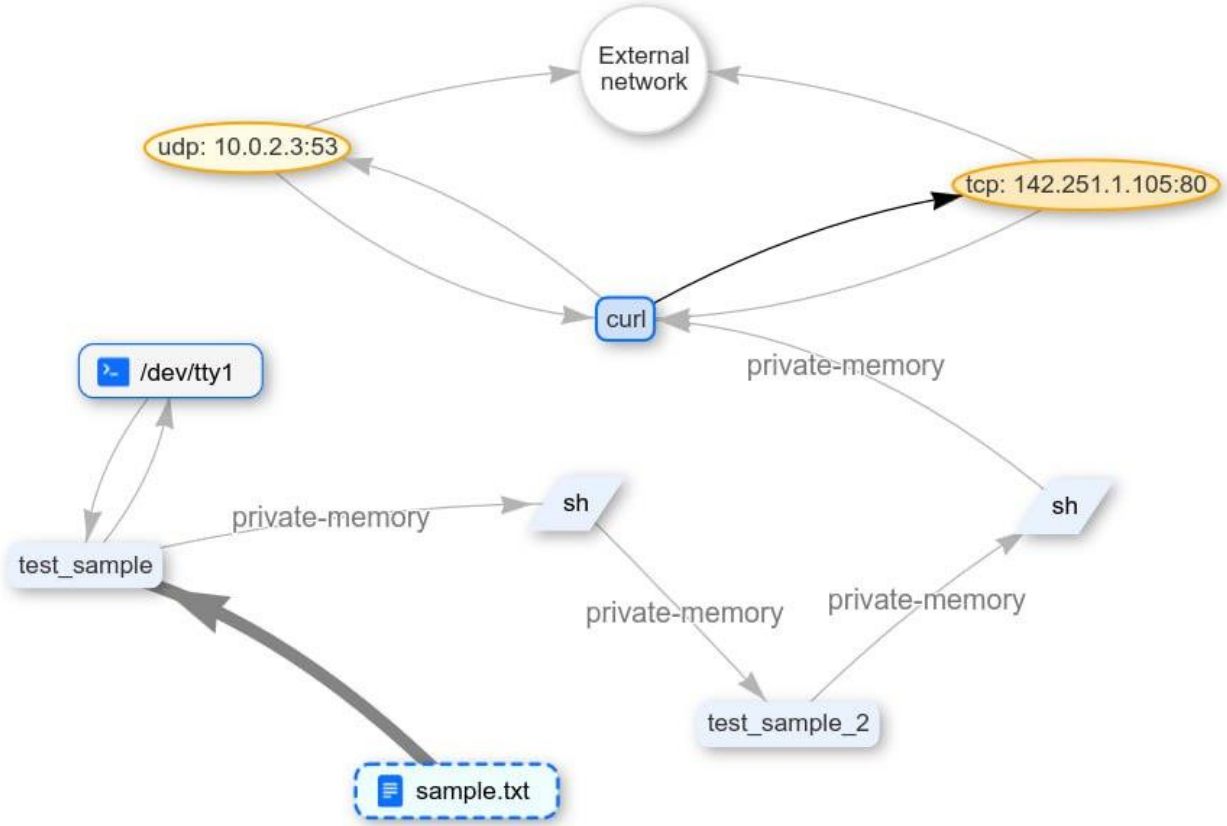
Our approach is based on heuristics!

```
Tuning started. Please wait a little...
Generating config file: task_config.ini
Trying to find 19 kernel-specific parameters
[01/19] Parameter - task_struct->pid                : Found
[02/19] Parameter - task_struct->comm               : Found
[03/19] Parameter - task_struct->group_leader       : Found
[04/19] Parameter - task_struct->parent             : Found
[05/19] Parameter - mount fields                    : Found
[06/19] Parameter - files_struct fields             : Found
[07/19] Parameter - vm_area_struct size             : Found
[08/19] Parameter - vm_area_struct->vm_start        : Found
[09/19] Parameter - vm_area_struct->vm_end          : Found
[10/19] Parameter - vm_area_struct->vm_flags        : Found
[11/19] Parameter - mm->map_count                   : Found
[12/19] Parameter - mm_struct fields                : Found
[13/19] Parameter - task_struct->mm                 : Found
[14/19] Parameter - mm->arg_start                   : Found
[15/19] Parameter - socket struct fields            : Found
[16/19] Parameter - task_struct->state              : Found
[17/19] Parameter - task_struct->exit_state         : Found
[18/19] Parameter - cred->uid                       : Found
[19/19] Parameter - task_struct->cred               : Found
Detected 49032 system events
Detected 19 of 19 kernel-specific parameters. Creating config file...

Tuning completed successfully!
```

Dynamic Information Flow Tracking

Limitations

- Transfers over files, sockets, and shared memory are tracked by hooking system calls and parsing kernel structures
- For other data transfers tracking, we additionally allocate 2 bytes of shadow memory for each byte of guest memory



Natch Usage

1.

Prepare the virtual machine image and the target software

2.

Record the target software execution in virtual machine

3.

Choose input data for tracking (files, network connections)

4.

Replay the execution and save attack surface

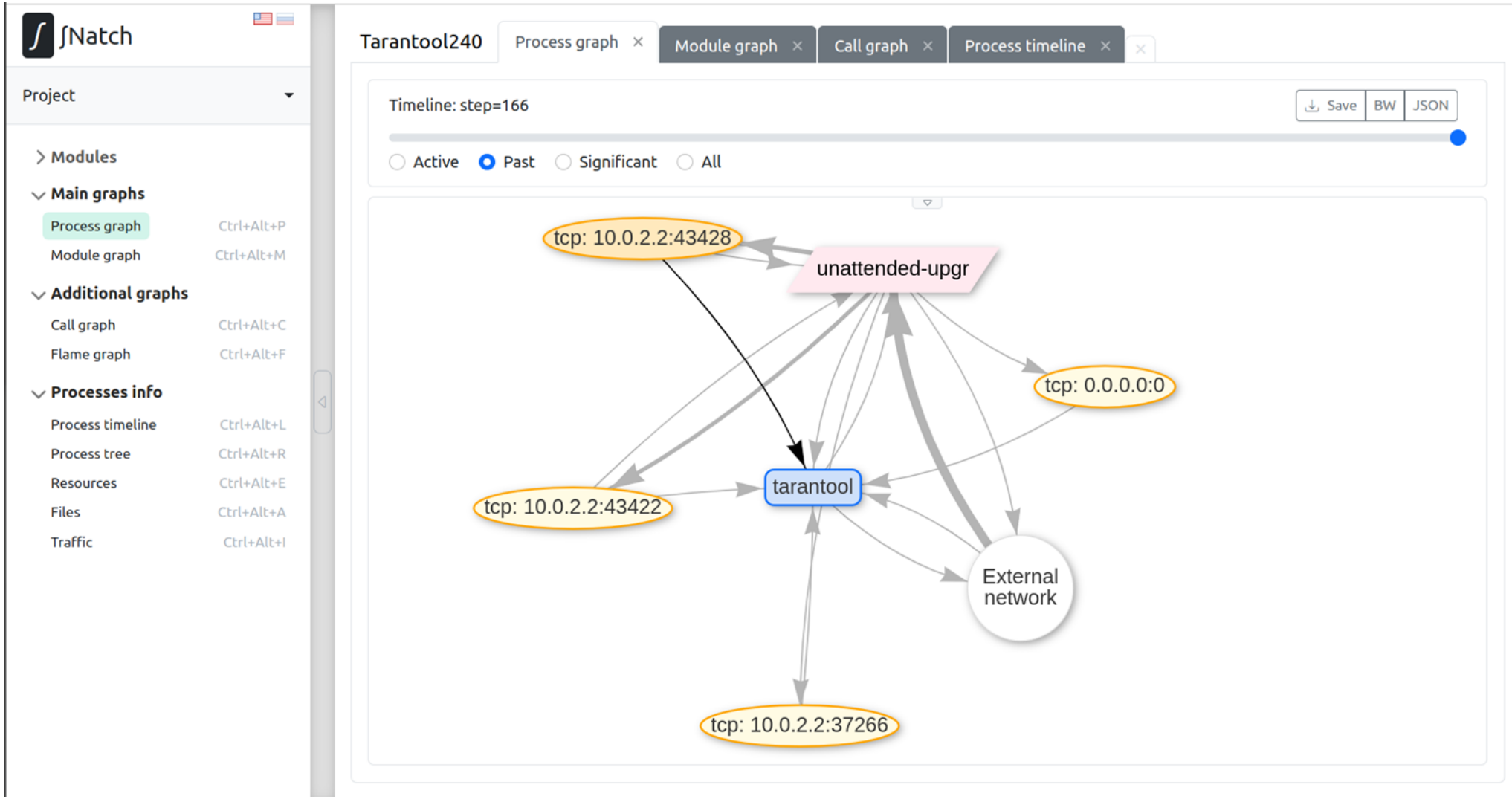
5.

Load the resulting attack surface into SNatch

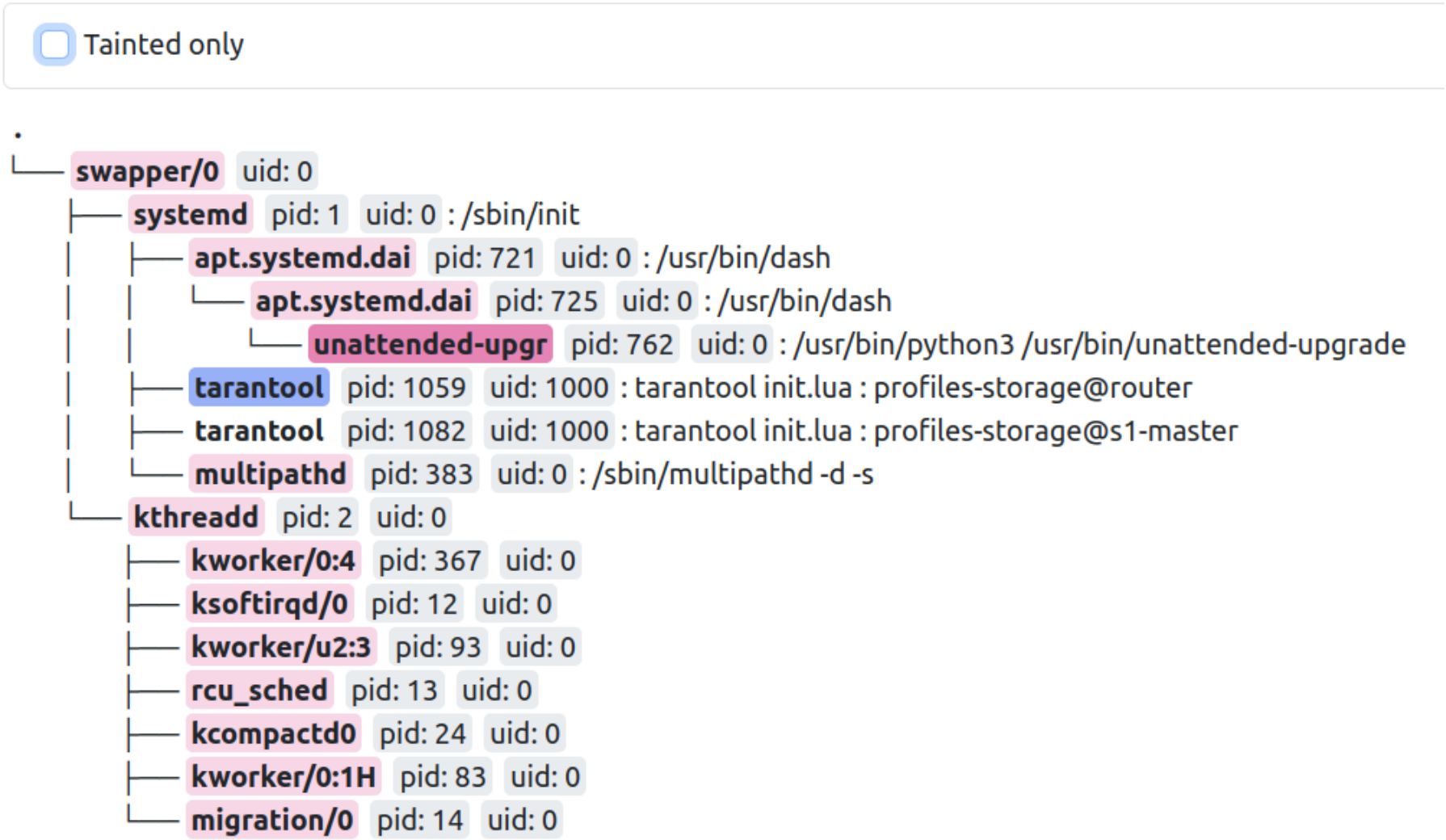
6.

Explore the attack surface with interactive reports in the browser

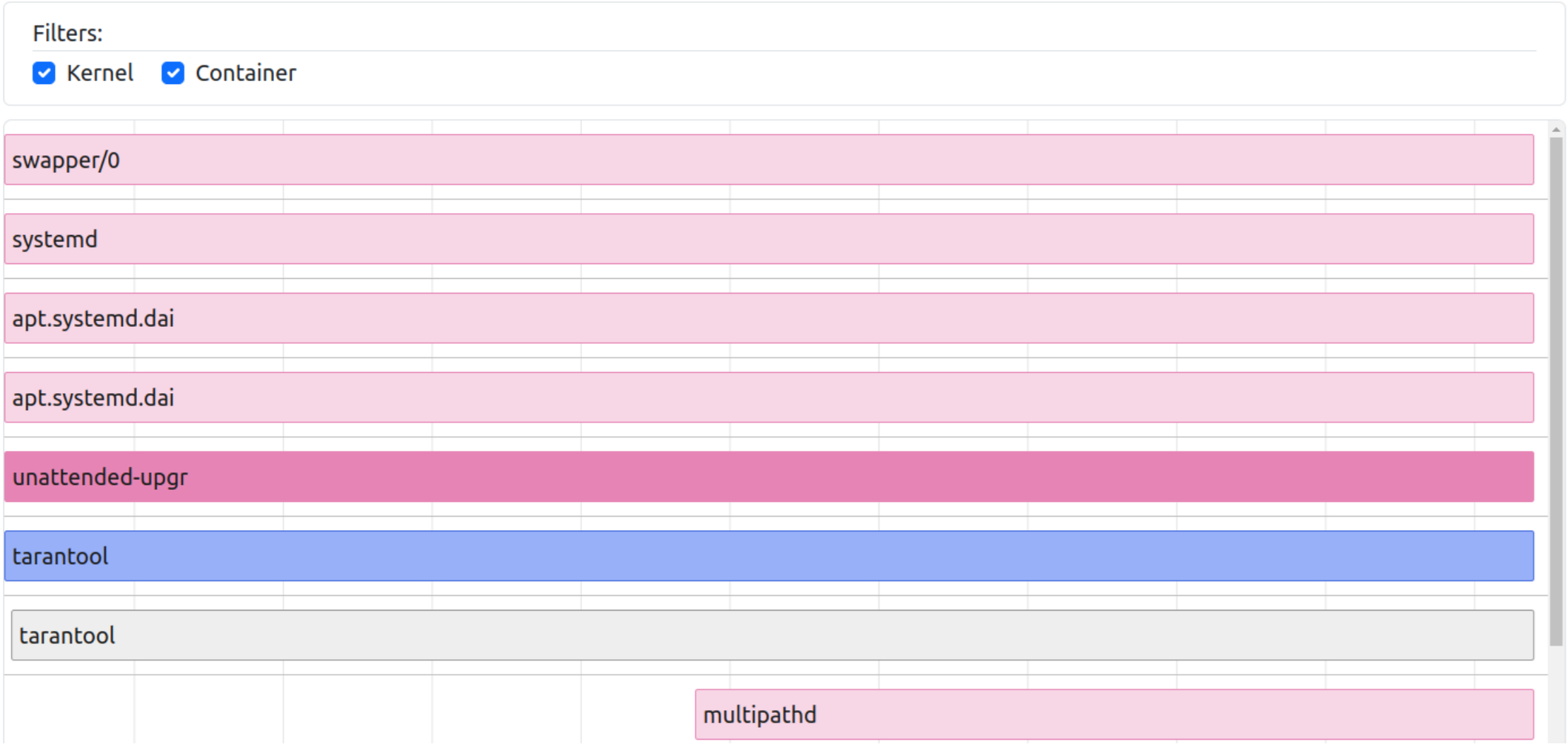
SNatch: Attack Surface Visualization



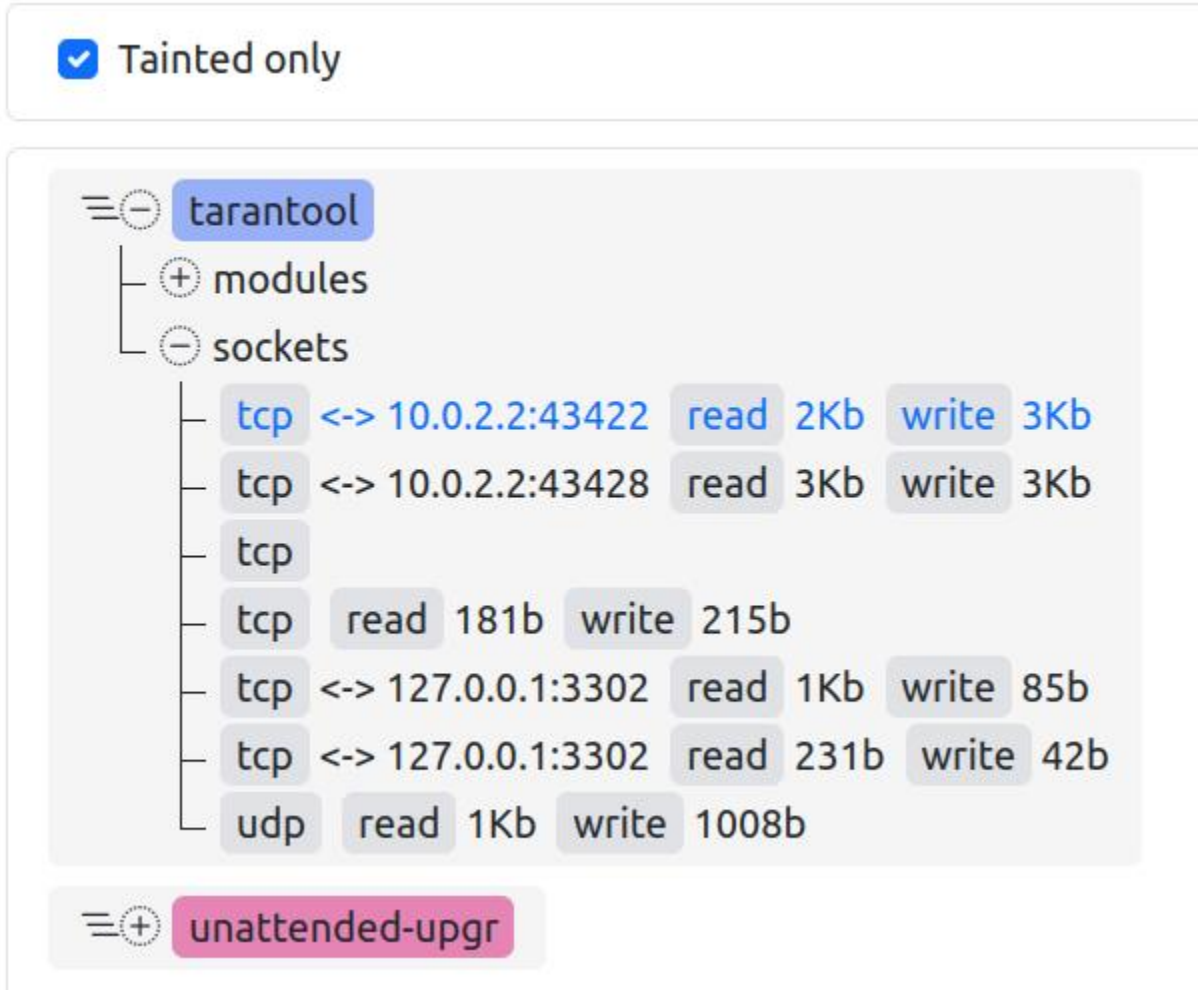
Process Tree

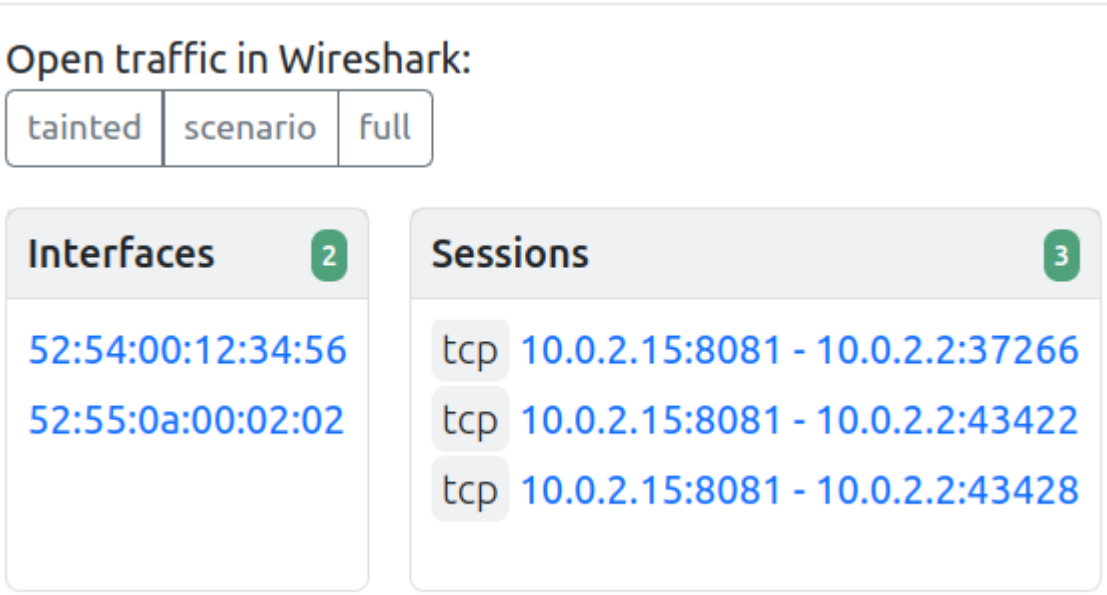


Process Timeline

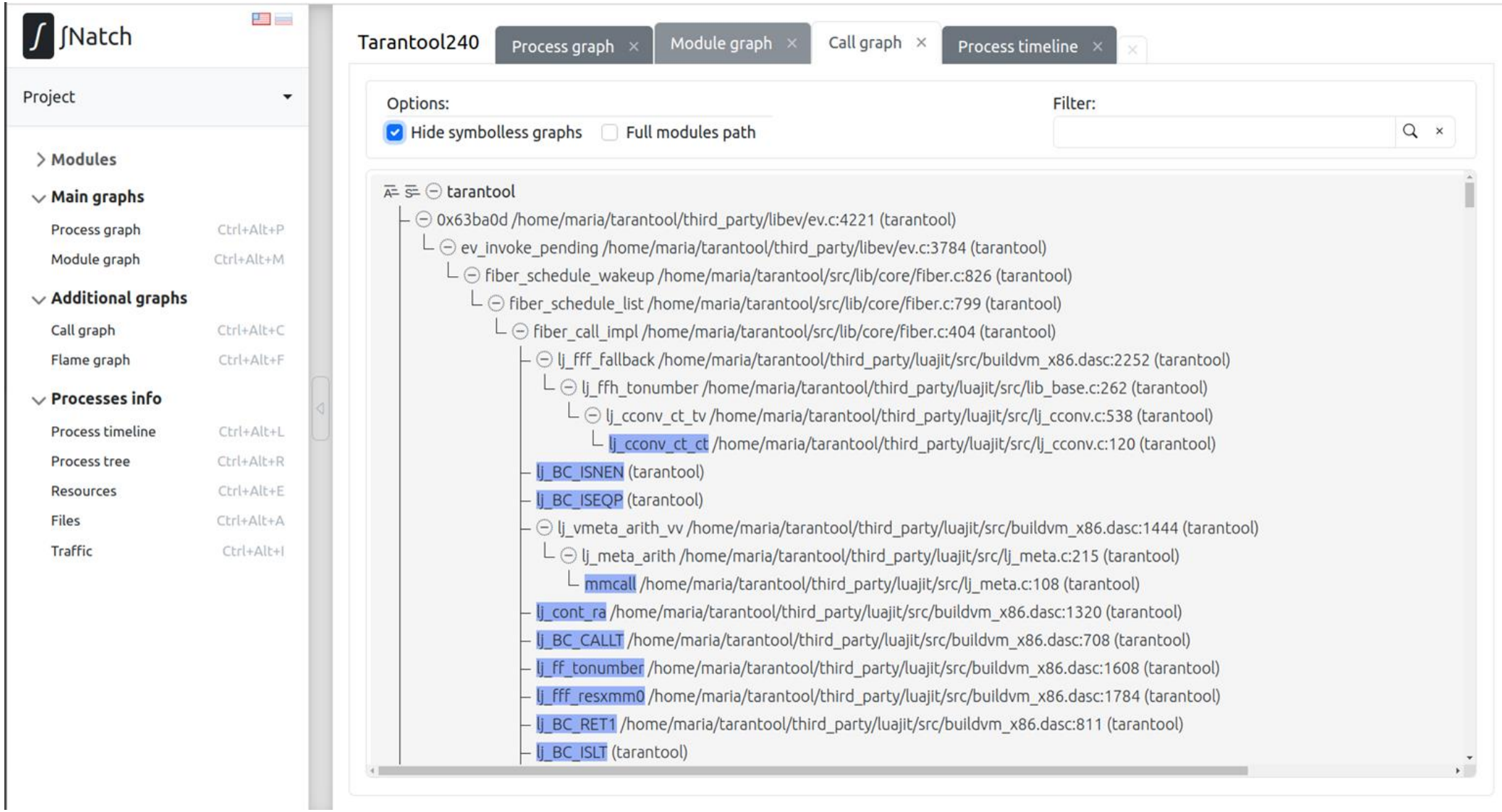


Resources And Traffic

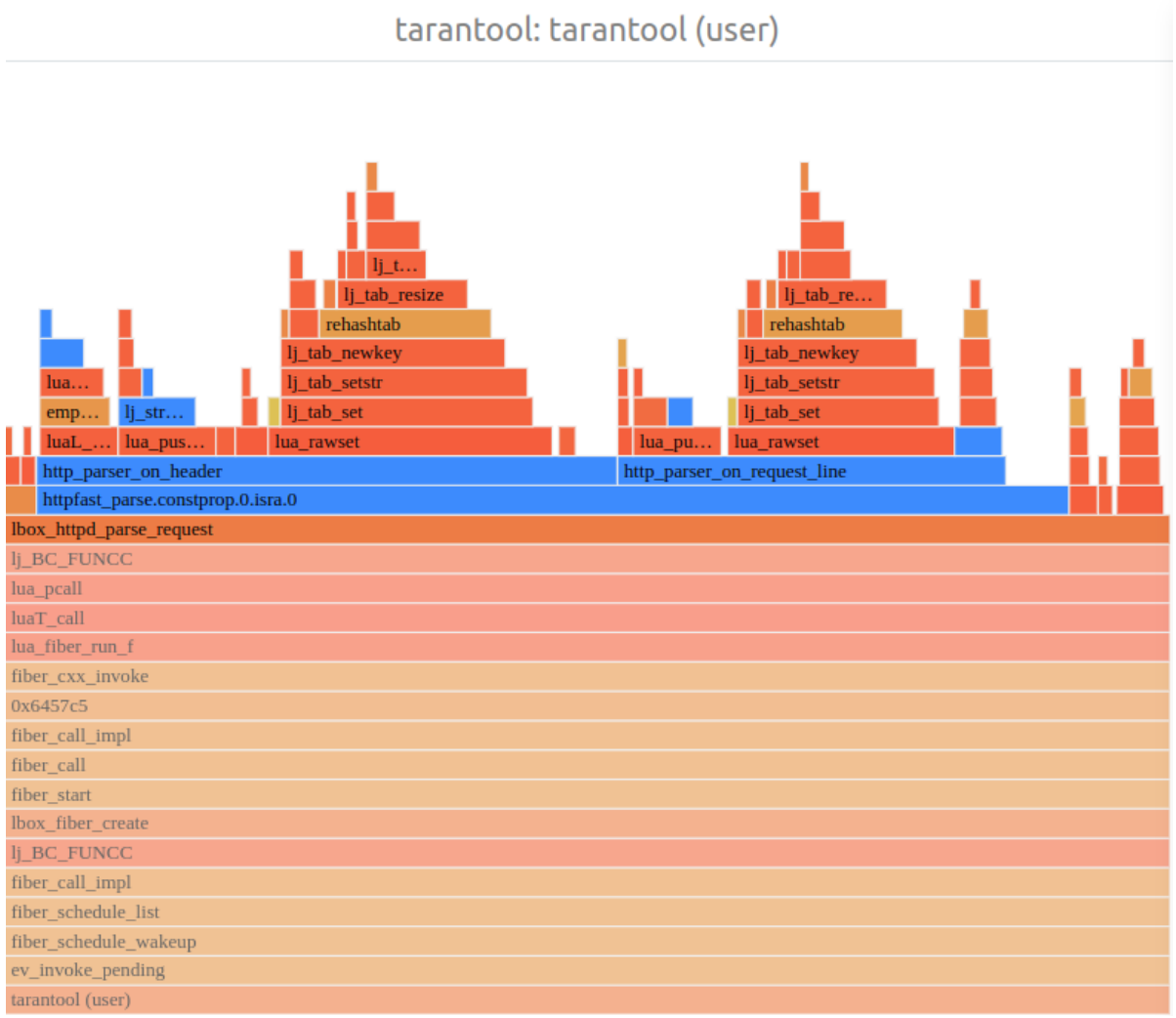




Call Graph


JNatch
Project
Modules
Main graphs
Process graph Ctrl+Alt+P
Module graph Ctrl+Alt+M
Additional graphs
Call graph Ctrl+Alt+C
Flame graph Ctrl+Alt+F
Processes info
Process timeline Ctrl+Alt+L
Process tree Ctrl+Alt+R
Resources Ctrl+Alt+E
Files Ctrl+Alt+A
Traffic Ctrl+Alt+I
Tarantool240
Process graph
Module graph
Call graph
Process timeline
Options:
Hide symbolless graphs
Full modules path
Filter:
tarantool
0x63ba0d /home/maria/tarantool/third_party/libev/ev.c:4221 (tarantool)
ev_invoke_pending /home/maria/tarantool/third_party/libev/ev.c:3784 (tarantool)
fiber_schedule_wakeup /home/maria/tarantool/src/lib/core/fiber.c:826 (tarantool)
fiber_schedule_list /home/maria/tarantool/src/lib/core/fiber.c:799 (tarantool)
fiber_call_impl /home/maria/tarantool/src/lib/core/fiber.c:404 (tarantool)
lj_fff_fallback /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:2252 (tarantool)
lj_ffh_tonumber /home/maria/tarantool/third_party/luajit/src/lib_base.c:262 (tarantool)
lj_cconv_ct_tv /home/maria/tarantool/third_party/luajit/src/lj_cconv.c:538 (tarantool)
lj_cconv_ct_ct /home/maria/tarantool/third_party/luajit/src/lj_cconv.c:120 (tarantool)
lj_BC_ISNEN (tarantool)
lj_BC_ISEQP (tarantool)
lj_vmeta_arith_vv /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:1444 (tarantool)
lj_meta_arith /home/maria/tarantool/third_party/luajit/src/lj_meta.c:215 (tarantool)
mmcall /home/maria/tarantool/third_party/luajit/src/lj_meta.c:108 (tarantool)
lj_cont_ra /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:1320 (tarantool)
lj_BC_CALLT /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:708 (tarantool)
lj_ff_tonumber /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:1608 (tarantool)
lj_fff_resxmm0 /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:1784 (tarantool)
lj_BC_RET1 /home/maria/tarantool/third_party/luajit/src/buildvm_x86.dasc:811 (tarantool)
lj_BC_ISLT (tarantool)

Flame Graph



name lbox_httpd_parse_request

module /home/vlad/Work/Images/TarantoolOriginal/bins/profile-storage/.rocks/lib/tarantool/http/lib.so

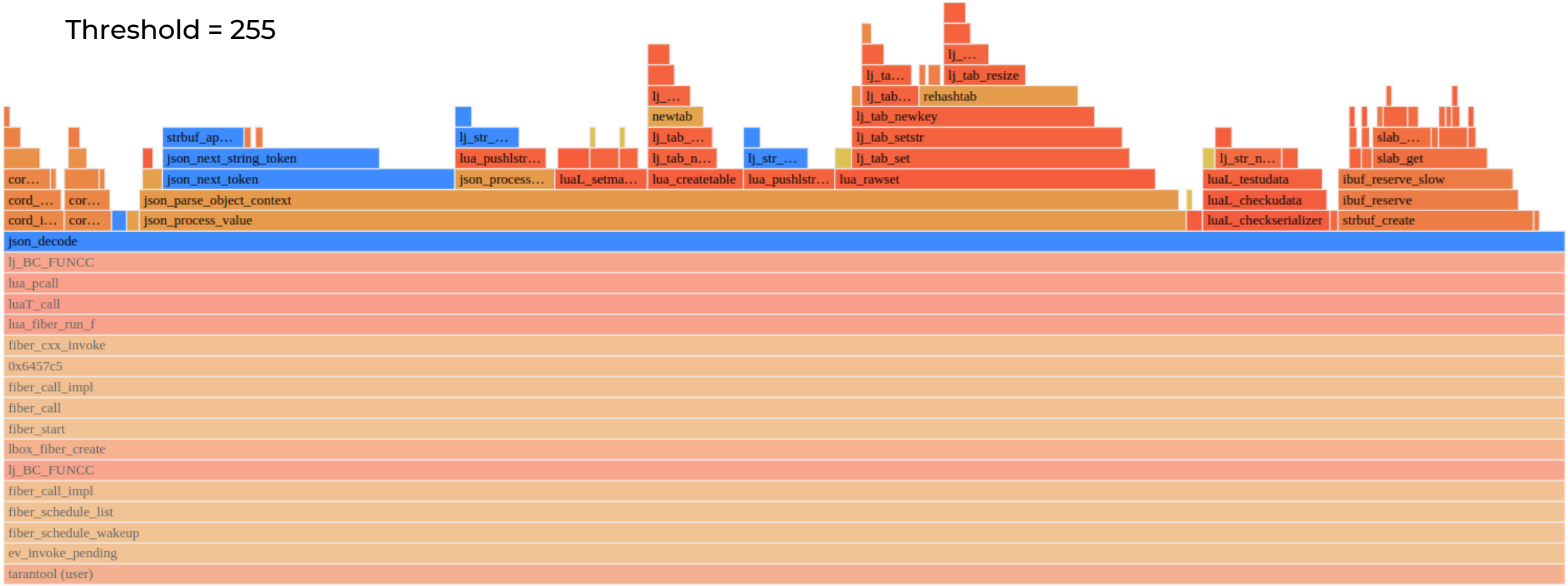
duration 29390 icounts

children:

http_parser_on_body	810
httpfast_parse.constprop.0.isra.0	26036
lua_createtable	718
lua_gettop	12
lua_pushstring	364
lua_pushvalue	98
lua_rawset	1195
lua_settop	32
lua_tolstring	62

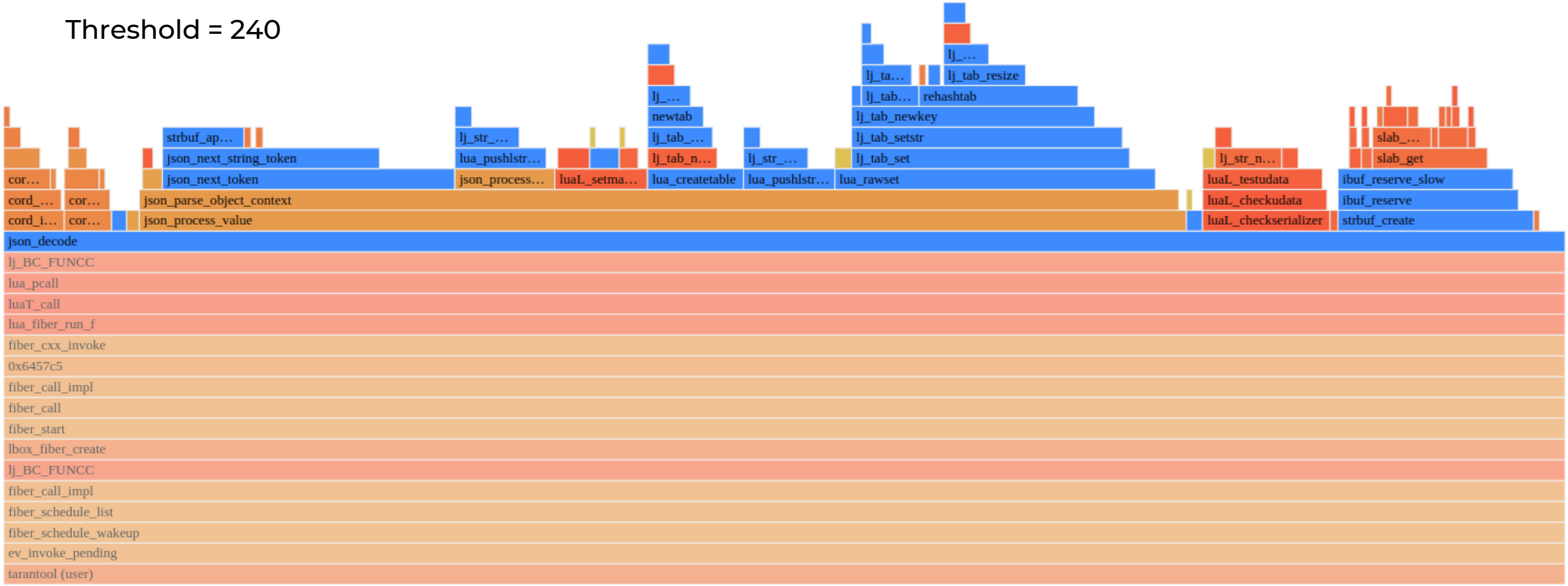
Taint Analysis

Threshold = 255



Taint Analysis

Threshold = 240



Other Possibilities

- Determination of processes running in docker containers
- Getting a list of Python scripts running in each process
- Building call trees for Python functions that process tainted data

NO FF ONE 2023

Sydr: Continuous Hybrid Fuzzing and Dynamic Analysis for SDL

What is Sydr?

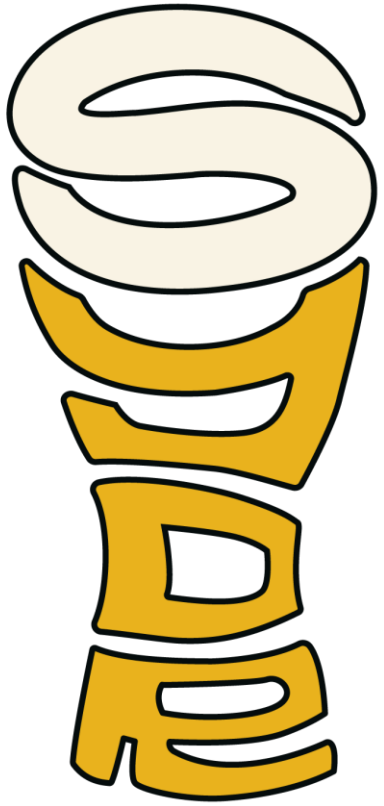
Sydr is a dynamic symbolic execution tool that explores new paths and enables error detection. Sydr uses DynamoRIO for concrete execution and Triton for symbolic execution.

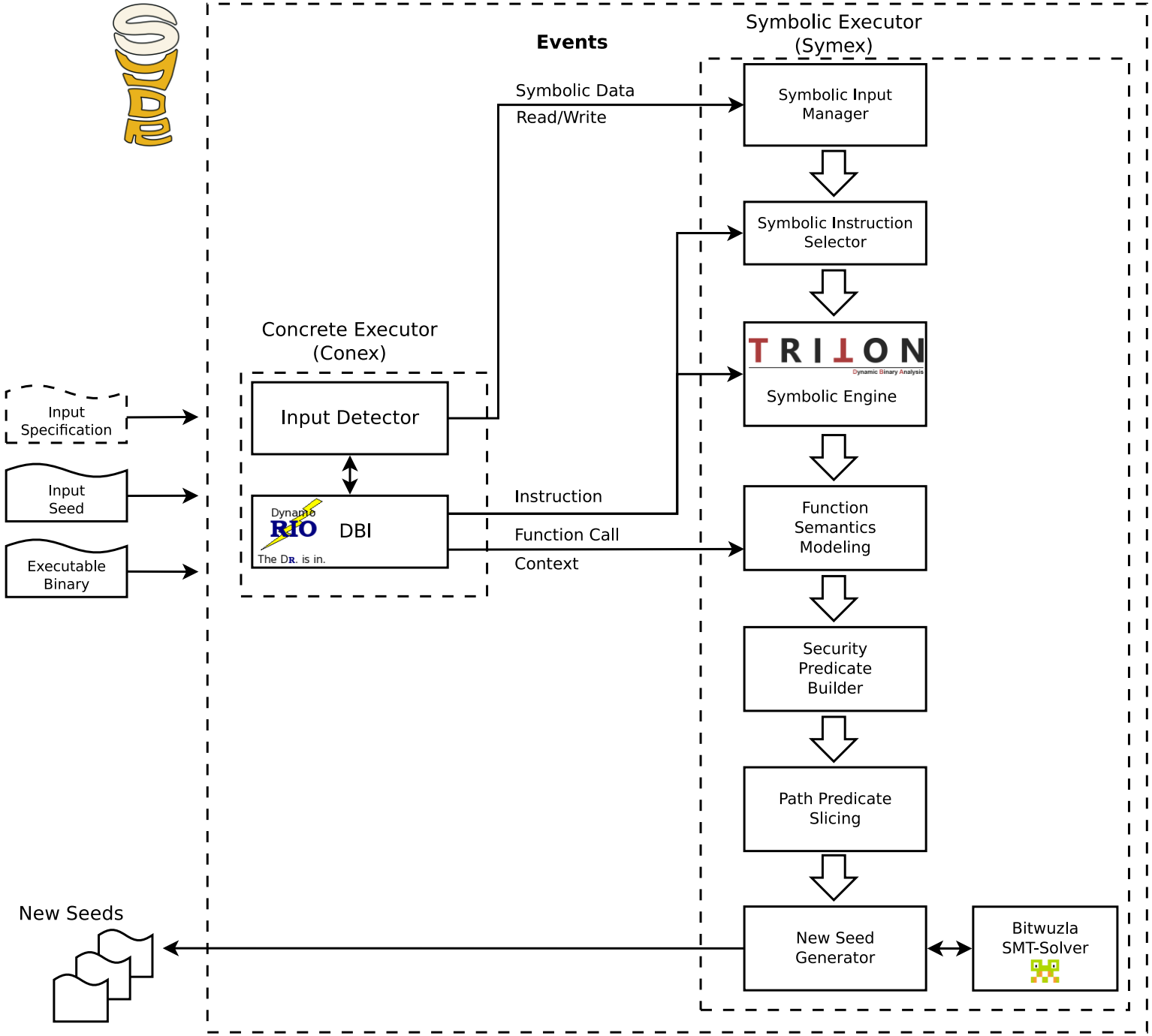
Sydr-Fuzz is a dynamic analysis tool for security development lifecycle. It combines fuzzing (libFuzzer, AFL++) with the power of dynamic symbolic execution (Sydr).

Sydr-Fuzz supports multiple programming languages including C/C++ (libFuzzer/AFL++), Rust (cargo-fuzz/afl.rs), Go (go-fuzz), Python (Atheris), and Java (Jazzer). All languages except Python and Java support symbolic execution with Sydr.

Dynamic Symbolic Execution with Sydr

- Each input byte is modeled by a free **symbolic variable**
- Instructions interpretation produces SMT formulas
- **Symbolic state** maps registers and memory to SMT formulas
- **Path predicate** contains taken branch constraints
- Sydr inverts branch conditions to explore new paths and solves security predicates to detect errors (out of bounds, integer overflow, etc.)




Events
Symbolic Executor
(Symex)
Symbolic Data
Read/Write
Symbolic Input
Manager
Symbolic Instruction
Selector
TRITON
Dynamic Binary Analysis
Symbolic Engine
Concrete Executor
(Conex)
Input Detector
Input
Specification
Input
Seed
Executable
Binary
Dynamo
RIO
The DR. is in.
DBI
Instruction
Function Call
Context
Function
Semantics
Modeling
Security
Predicate
Builder
Path Predicate
Slicing
New Seeds
New Seed
Generator
Bitwuzla
SMT-Solver

Sydr-Fuzz Usage

Sydr-Fuzz project:

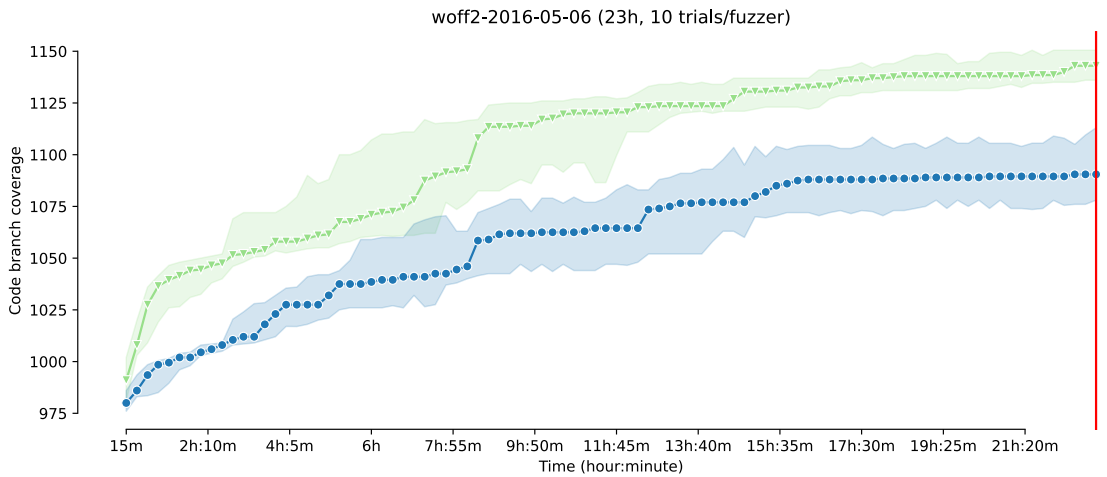
- corpus
- crashes
- libfuzzer/aflplusplus/sydr/atheris/jazzer (work directories)
- casr (crash clusters and ubsan reports)
- security (symbolic checkers)
- coverage
- sydr-fuzz*.log (logs)

TOML-config:

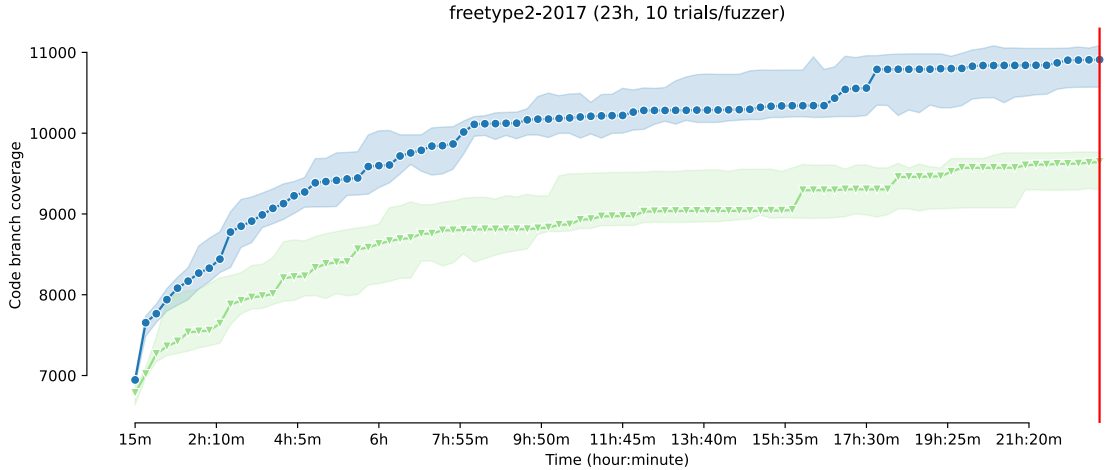
```toml
[sydr]
target = "/decode_wav_sydr @@"
jobs = 2

[aflplusplus]
target = "/decode_wav_fuzz"
args = "-x wav.dict -i /corpus"
jobs = 2

[cov]
target = "/decode_wav_cov @@"
```

Run: `sydr-fuzz -c config.toml run|cmin|security|cov-html|casr`

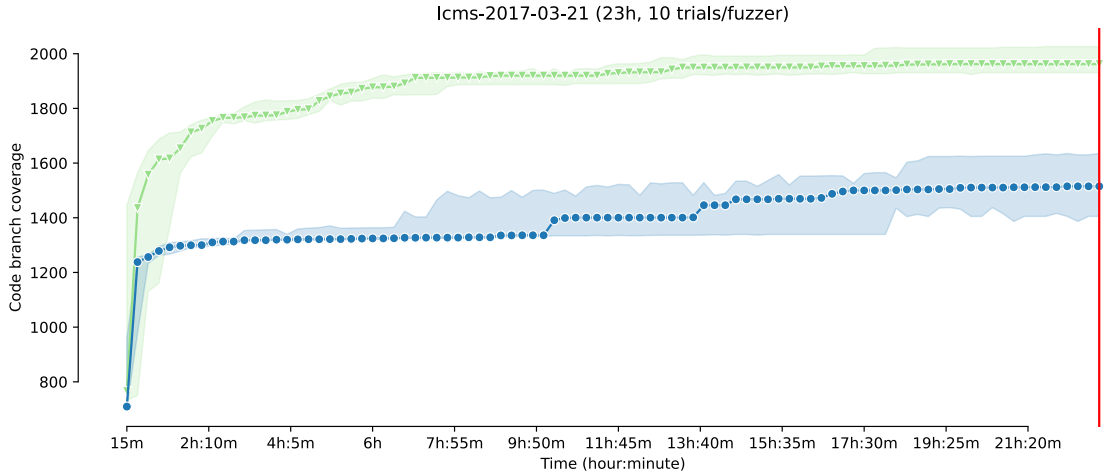
1. Sydr-Fuzz achieved higher coverage than other fuzzers
2. Sydr-Fuzz outperformed existing fuzzers on most benchmarks



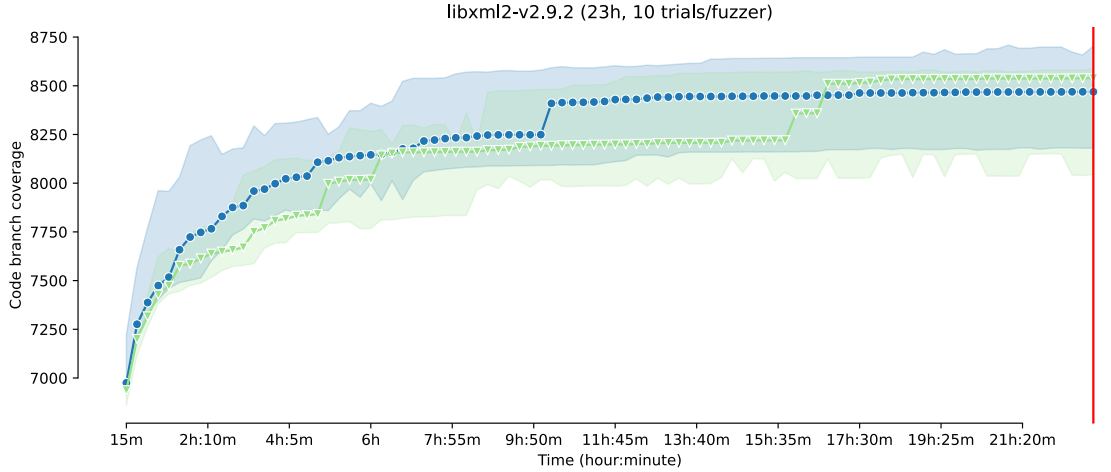
Sydr+libFizzer vs 2xlibFuzzer



Sydr+AFL++ vs SymQEMU+AFL++



Sydr+AFL++ vs 2xAFL++



Sydr+AFL++ vs FUZZOLIC+AFL++

sydr-fuzz.github.io/fuzzbench

OSS-Sydr-Fuzz: Hybrid Fuzzing for Open Source

github.com/ispras/oss-sydr-fuzz – fork of OSS-Fuzz for hybrid fuzzing with Sydr-Fuzz

- **65+** projects and **500** fuzz targets
- Sydr-Fuzz discovered **135+** new bugs in 25+ projects: TensorFlow, PyTorch, Cairo (GTK), OpenJPEG, Poppler, ICU, Tarantool, Torchvision, etc. All trophies on GitHub
- 20+ issues were found by Sydr symbolic security predicates

Sydr-Fuzz: Dynamic Analysis Pipeline

1.

Hybrid fuzzing with Sydr and libFuzzer/AFL++; coverage-guided Python (Atheris) and Java (Jazzer) fuzzing:
`sydr-fuzz run`

2.

Corpus minimization:
`sydr-fuzz cmin`

3.

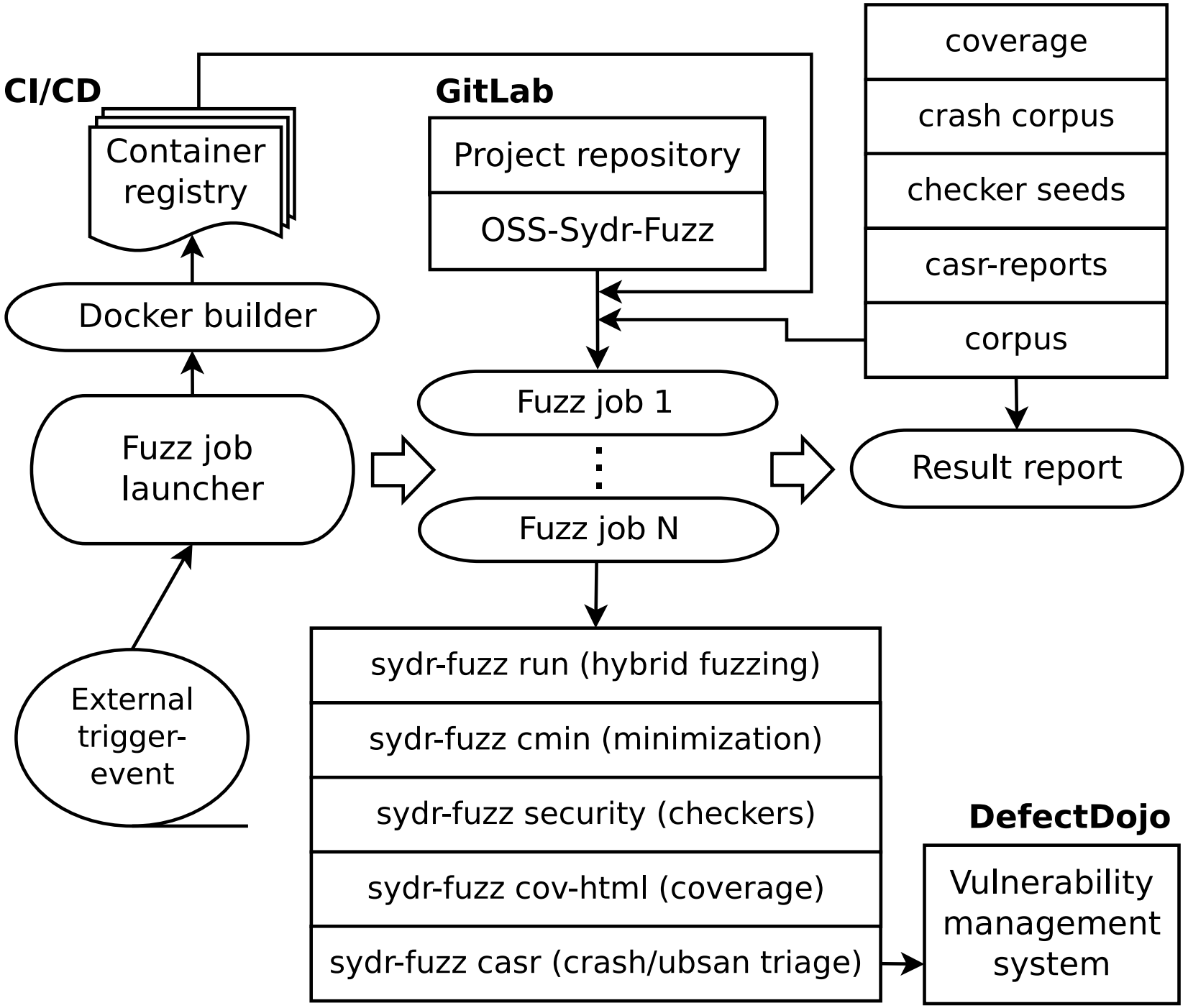
Error detection (out of bounds, integer overflow, numeric truncation, etc.) via symbolic security predicates:
`sydr-fuzz security`

4.

Collecting coverage:
`sydr-fuzz cov-html`

5.

Triaging, deduplication, and clustering of crashes and Undefined Behavior Sanitizer errors with Casr, and later upload of new and unique reports to DefectDojo:
`sydr-fuzz casr --ubsan --url <URL>`

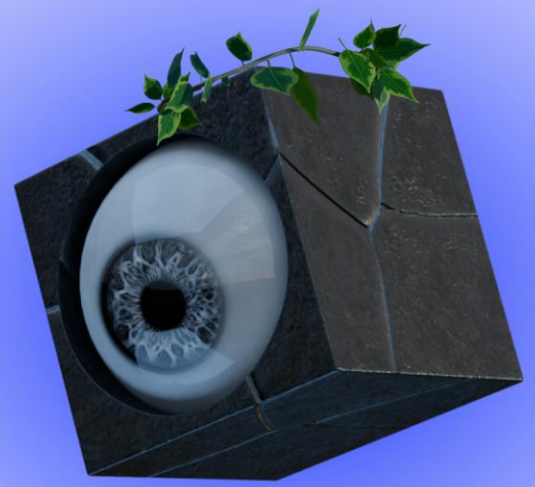

CI/CD
GitLab
Container registry
Project repository
OSS-Sydr-Fuzz
coverage
crash corpus
checker seeds
casr-reports
corpus
Docker builder
Fuzz job 1
Fuzz job launcher
Result report
Fuzz job N
External trigger-event
sydr-fuzz run (hybrid fuzzing)
sydr-fuzz cmin (minimization)
sydr-fuzz security (checkers)
sydr-fuzz cov-html (coverage)
sydr-fuzz casr (crash/ubsan triage)
DefectDojo
Vulnerability management system

Sydr & libFuzzer

- libFuzzer workers use shared corpus directory
- Sydr takes seeds to modify and puts generated seeds to the same directory
- libFuzzer immediately loads seeds generated by Sydr
- Reloaded files are logged by libFuzzer: reviews.llvm.org/D100303000
- Sydr-Fuzz removes not reloaded seeds from corpus
- Scheduling seeds for Sydr:
  - whether seed discovered new function
  - whether seed brought new coverage
  - whether seed increased libFuzzer features
  - creation time / size

Sydr & AFL++

- Sydr is launched as a fake secondary AFL worker
- Sydr is executed on seeds from AFL main worker queue
- Sydr-Fuzz uses afl-showmap to minimize seeds generated by Sydr before putting them in Sydr worker queue
- AFL main worker scans Sydr queue and imports useful seeds
- Seeds for Sydr are scheduled: new coverage, initial corpus seed, file size, novelty
- Running AFL++ in parallel mode with automatically assigned options (schedulers, MOpt, etc.)



Symbolic Security Predicates

- Out of bounds, integer overflow, etc.
- Security predicates are checked on minimized corpus after fuzzing
- Generated seeds are verified on sanitizers
- Deduplication of detected errors

Symbolic Checkers Detect Additional Bugs After Fuzzing

github.com/opencv/opencv/issues/22284

opencv/3rdparty/openjpeg/openjp2/image.c:134:

```
l_y1 = p_cp->ty0 + (p_cp->th - 1U) * p_cp->tdy; /* can't overflow */
```

Can’t overflow? But we can!

Sydr security predicate error:

opj_image_comp_header_update:/opencv/3rdparty/openjpeg/openjp2/image.c:134 - imul r15d, eax - **unsigned integer overflow**

Automatic verification with sanitizers:

/opencv/3rdparty/openjpeg/openjp2/image.c:134:40: runtime error: **unsigned integer overflow**: 2 * 4278190076 cannot be represented in type 'unsigned int'

Integer Overflow to Buffer Overflow in Rizin

```
symbols_size = (symbols_count + 1) * 2 * sizeof(struct symbol_t);
if (symbols_size < 1) {
    ht_pp_free(hash);
    return NULL;
}
if (!(symbols = calloc(1, symbols_size))) {
    ht_pp_free(hash);
    return NULL;
}
...
symbols[j].last = true;
```

CASR: Crash Triaging

- **`casr-san`** runs crashes on sanitized binary and creates reports
- Crash report contains stack trace, crash line, crash severity, assembly, source, etc.
- **`casr-cluster -d`** deduplicates crashes based on stack trace hash
- **`casr-cluster -c`** performs hierarchical clustering of crash reports
- **`casr-gdb`** generates crash reports for non-instrumented binaries
- **`casr-ubsan`** creates Casr reports for unique UBSAN errors
- **`casr-dojo`** uploads new and unique reports to DefectDojo

More at OFFZONE 2023: CASR: Your Life Vest in a Sea of Crashes

github.com/ispras/casr

```
[2022-11-13 14:00:39] [INFO] ==> <cl10>
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl10/crash-d07585811a792f15991c6ce9896f1106303ba58e
[2022-11-13 14:00:39] [INFO]   casr-san: NOT_EXPLOITABLE: SourceAvNearNull: /xlnt/source/detail/serialization/xlsx_consumer.cpp:2044:22
[2022-11-13 14:00:39] [INFO]   casr-gdb: NOT_EXPLOITABLE: SourceAvNearNull: /xlnt/source/detail/serialization/xlsx_consumer.cpp:2044
[2022-11-13 14:00:39] [INFO]   Similar crashes: 1
[2022-11-13 14:00:39] [INFO] Cluster summary -> SourceAvNearNull: 2
[2022-11-13 14:00:39] [INFO] ==> <cl11>
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl11/crash-f403e0aedb35c05126272da86b4312939bb1efc1
[2022-11-13 14:00:39] [INFO]   casr-san: NOT_EXPLOITABLE: heap-buffer-overflow(read): /xlnt/source/../source/detail/cryptography/compound_document.hpp:83:30
[2022-11-13 14:00:39] [INFO]   casr-gdb: NOT_EXPLOITABLE: AbortSignal: /xlnt/third-party/utfcpp/utf8/checked.h:216
[2022-11-13 14:00:39] [INFO]   Similar crashes: 1
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl11/crash-a02068022d89646963e409eacd5d59ea63089750
[2022-11-13 14:00:39] [INFO]   casr-san: NOT_EXPLOITABLE: heap-buffer-overflow(read): /xlnt/source/../source/detail/cryptography/compound_document.hpp:83:30
[2022-11-13 14:00:39] [INFO]   casr-gdb: No crash
[2022-11-13 14:00:39] [INFO]   Similar crashes: 1
[2022-11-13 14:00:39] [INFO] Cluster summary -> heap-buffer-overflow(read): 2 AbortSignal: 1
[2022-11-13 14:00:39] [INFO] ==> <cl12>
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl12/crash-292e7f90b9ea11b176c04f106fe2a9e439b5b40b
[2022-11-13 14:00:39] [INFO]   casr-san: PROBABLY_EXPLOITABLE: DestAvNearNull: /xlnt/source/../source/detail/binary.hpp:278:9
[2022-11-13 14:00:39] [INFO]   casr-gdb: PROBABLY_EXPLOITABLE: DestAvNearNull: /xlnt/source/detail/binary.hpp:278
[2022-11-13 14:00:39] [INFO]   Similar crashes: 2
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl12/crash-51b39d8f893faefd1d3d2003d438b82b470557e2
[2022-11-13 14:00:39] [INFO]   casr-san: EXPLOITABLE: heap-buffer-overflow(write): /xlnt/source/../source/detail/binary.hpp:278:9
[2022-11-13 14:00:39] [INFO]   casr-gdb: EXPLOITABLE: DestAv: /xlnt/source/detail/binary.hpp:278
[2022-11-13 14:00:39] [INFO]   Similar crashes: 1
[2022-11-13 14:00:39] [INFO] Cluster summary -> DestAv: 1 DestAvNearNull: 4 heap-buffer-overflow(write): 1
[2022-11-13 14:00:39] [INFO] ==> <cl13>
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl13/crash-5ba9e014e4314b5fa1f0e270938024f7c0d02d00
[2022-11-13 14:00:39] [INFO]   casr-san: EXPLOITABLE: heap-buffer-overflow(write): /xlnt/source/detail/cryptography/base64.cpp:176:32
[2022-11-13 14:00:39] [INFO]   casr-gdb: No crash
[2022-11-13 14:00:39] [INFO]   Similar crashes: 5
[2022-11-13 14:00:39] [INFO] Crash: /fuzz/sydr-fuzz-afl++-out/casr/cl13/crash-18ebf7db76ffe9fc403faaebc0003d166e0ecc44
[2022-11-13 14:00:39] [INFO]   casr-san: EXPLOITABLE: heap-buffer-overflow(write): /xlnt/source/detail/cryptography/base64.cpp:148:36
[2022-11-13 14:00:39] [INFO]   casr-gdb: No crash
[2022-11-13 14:00:39] [INFO]   Similar crashes: 3
[2022-11-13 14:00:39] [INFO] Cluster summary -> heap-buffer-overflow(write): 8
```

xlnt

Overview | Components | Metrics | Engagements 2 | Findings 60 | Endpoints | Benchmarks | Settings

sydr-fuzz 2023-06-07T16:47:18+03:00 / Sydr-Fuzz DAST Report / [Xlnt] [Load_fuzzer] Heap-Buffer-Overflow(write) in /xlnt/s... / View Finding

[Xlnt] [Load_fuzzer] Heap-Buffer-Overflow(write) in /xlnt/source/../source/detail/binary.hpp:278:9

Last Reviewed today by Admin User (admin), Last Status Update today, Created today

ID	Severity	SLA	Status	Type	Date discovered	Age	Reporter	CWE	Vulnerability Id	Found by
19	Critical	7	Active	Static	June 7, 2023	0 days	Admin User (admin)			API Test Sydr-Fuzz DAST Report

Location	Line Number
/xlnt/source/../source/detail/binary.hpp	278

Similar Findings (2)

Description

Severity: EXPLOITABLE: heap-buffer-overflow(write): Heap buffer overflow

The target writes data past the end, or before the beginning, of the intended heap buffer.

GDB severity (without ASAN): EXPLOITABLE: DestAv: Access violation on destination operand

The target crashed on an access violation at an address matching the destination operand of the instruction. This likely indicates a write access violation, which means the attacker may control the write address and/or value.

Command: /load_fuzzer -artifact_prefix=/fuzz/sydr-fuzz-out/crashes/ -verbosity=2 -rss_limit_mb=8192 -timeout=10 -close_fd_mask=1 /fuzz/sydr-fuzz-out/crashes/crash-3cdf0550a8765c86422c76eb5c123c2e3c67fe10

OS: Ubuntu 20.04

Architecture: amd64

Source

```
    274           {
    275               throw xlnt::exception("reading past end");
    276           }
    277
--->278           std::memcpy(data_->data() + offset_, reader.data() + reader.offset(), reader_element_count * sizeof(U));
    279           offset_ += reader_element_count * sizeof(U) / sizeof(T);
    280       }
```

Questions?

sydr-fuzz.github.io

Telegram:

@ispras_natch @sydr_fuzz