The International Conference on Fundamental Approaches to Software Engineering
(FASE 2022)

Artifacts Available EAPLS V1.1

Artifacts Evaluated EAPLS Functional V1.1

Estimating Worst-case Resource Usage by Resource-usage-aware Fuzzing

Liqian Chen, Renjie Huang, Dan Luo, Chenghu Ma, Dengping Wei, Ji Wang

National University of Defense Technology, Changsha, China

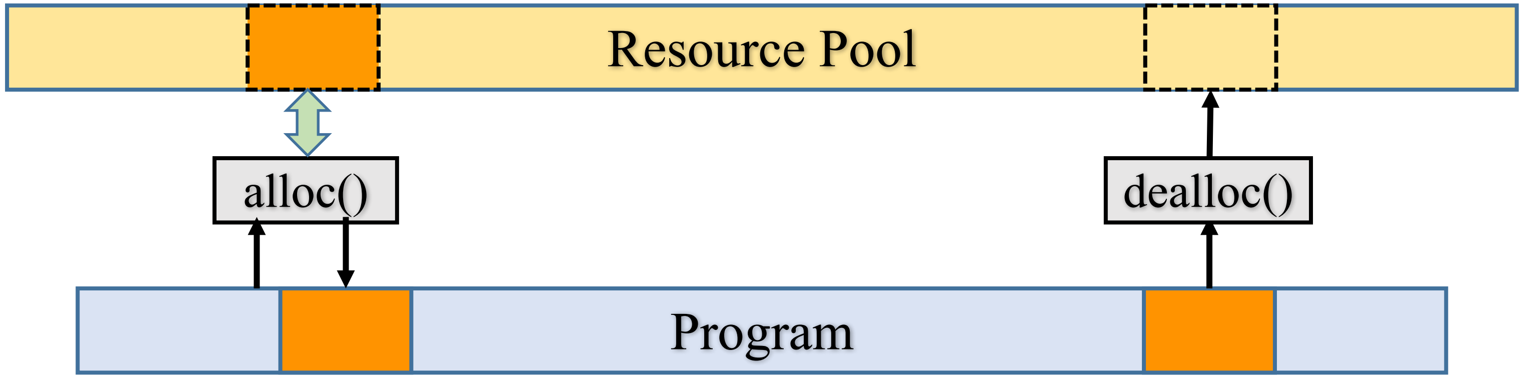
2022-04-04

Overview

- Motivation
- Approach
- Experiment
- Conclusion

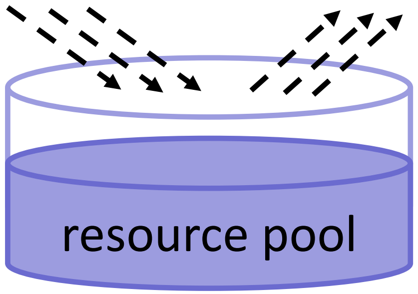
Motivation

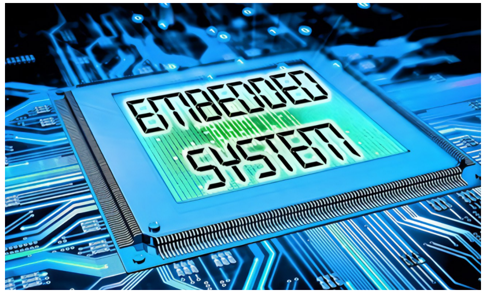
- **Resources**: any abstractions offered to a process by system calls
  - typical resources: heap/stack memory, sockets, file descriptors, threads, database connections, gas in smart contracts, etc.
  - user-defined application-dependent resources: buffers, memory pools, number of licenses consumed, etc.
- **Resource usage**: via APIs



Motivation

- Worst-case resource usage
  - a useful guidance in the design, configuration and deployment of software
  - especially when the software runs with limited amount of resources, e.g., in modern CPSs, mobile systems and IoT devices, etc.





Motivation

- Unexpected or uncontrolled resource usage may degrade program performance, or even lead to CWE vulnerabilities
  - CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')
  - CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak')
  - CWE-404: Improper Resource Shutdown or Release
  - CWE-405: Asymmetric Resource Consumption (Amplification)
  - CWE-410: Insufficient Resource Pool
  - CWE-674: Uncontrolled Recursion
  - CWE-769 File Descriptor Exhaustion
  - CWE-770: Allocation of Resources Without Limits or Throttling
  - CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime
  - CWE-789: Uncontrolled Memory Allocation
  - CWE-920 Improper Restriction of Power Consumption
  - …

Related Work

- Static resource-bound analysis [Gulawani et al. POPL09] [Albert et al, TCS12] [Carbonnneaux et al. PLDI15]
  - +provide sound upper bounds of worst-case resource usage
  - - may provide too conservative, even unbounded, results
  - - complex syntactic constructs in programs are usually being abstracted away
  - - the actual usage amount of resources may depend on the running system environment (e.g., malloc())

Related Work

- Dynamic methods [Antunes et al. ISSRE08] [Lemieux et al. ISSTA18] [Petsios et al. CCS17] [Weiet al. FSE18]
  - not sound
  + useful for estimating resource bounds and detecting vulnerability
  + practical for realistic software

- MemLock [Wen et al. ICSE20]
  - technique: memory usage guided fuzzing
  - use default branch coverage together with memory consumption to guide fuzzing
  - consider only memory resources (heap, stack)

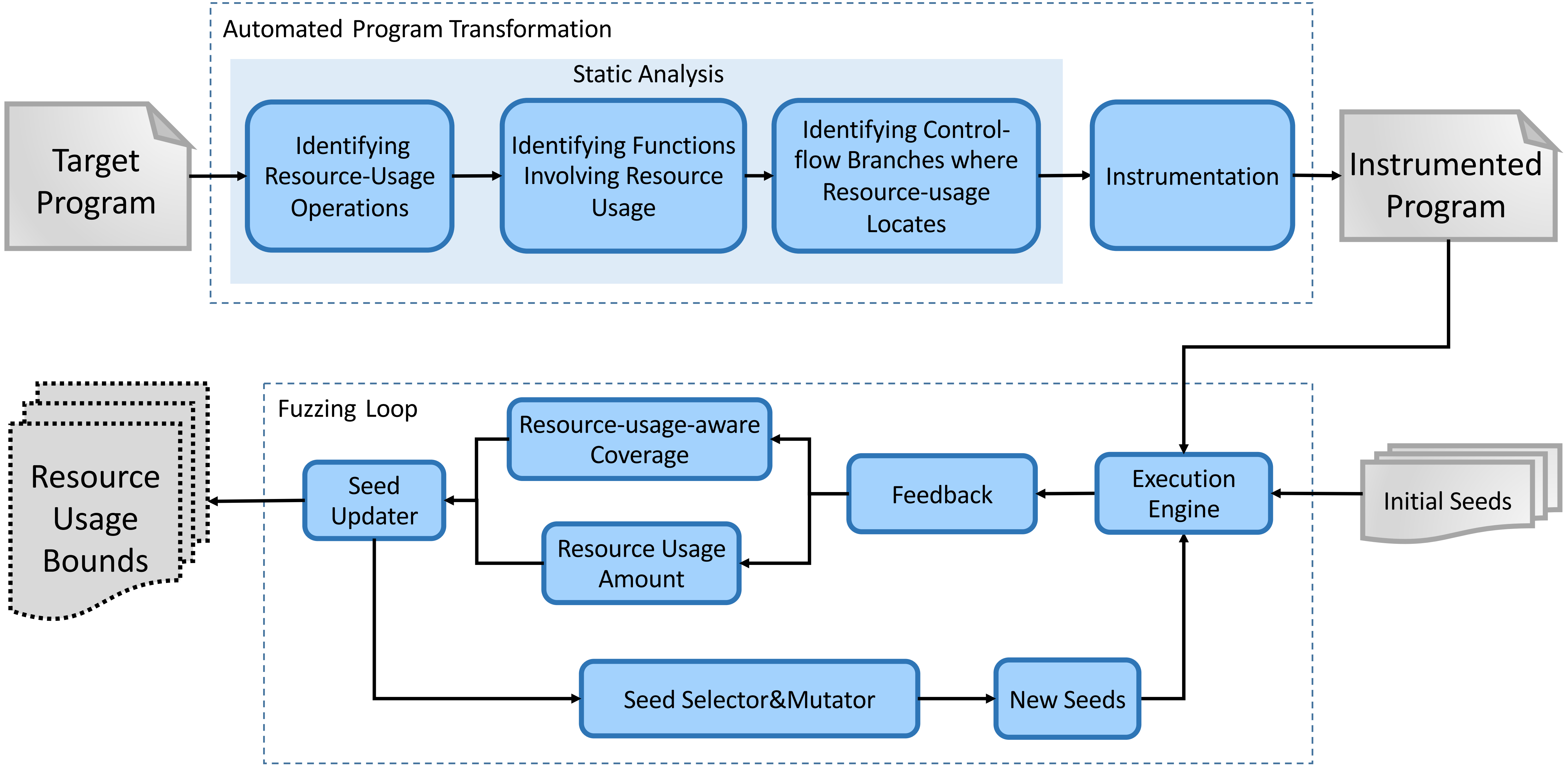
Main Idea

- **Resource-usage-aware fuzzing**
  - **goal**: to estimate worst-case resource usage for general resources
    - including memory, file descriptors, socket connections, user-defined resources, etc.
  - **approach**: employ resource-usage amount and resource-usage-aware coverage to guide fuzzing

Overview

- Motivation
- Approach
- Experiment
- Conclusion

Workflow



Automated Program Transformation
Static Analysis
Target Program
Identifying Resource-Usage Operations
Identifying Functions Involving Resource Usage
Identifying Control-flow Branches where Resource-usage Locates
Instrumentation
Instrumented Program
Fuzzing Loop
Resource Usage Bounds
Seed Updater
Resource-usage-aware Coverage
Resource Usage Amount
Feedback
Execution Engine
Initial Seeds
Seed Selector&Mutator
New Seeds

Step 1: Static analysis and instrumentation

- 1.1: Identifying and modeling resource-usage operations
- 1.2: Identifying functions involving resource usage
- 1.3: Identifying control-flow branches where resource-usage locates
- 1.4: Instrumentation

Step 1: Static analysis and instrumentation

- 1.1: Identifying and modeling resource-usage operations
  - resource-usage operations: APIs provided by systems/libraries, programmer-defined APIs
  - modeling via two unified functions
    - __RAlloc(int n): to model allocating *n* number of resources
    - __ RDealloc(int n): to model deallocating *n* number of resources

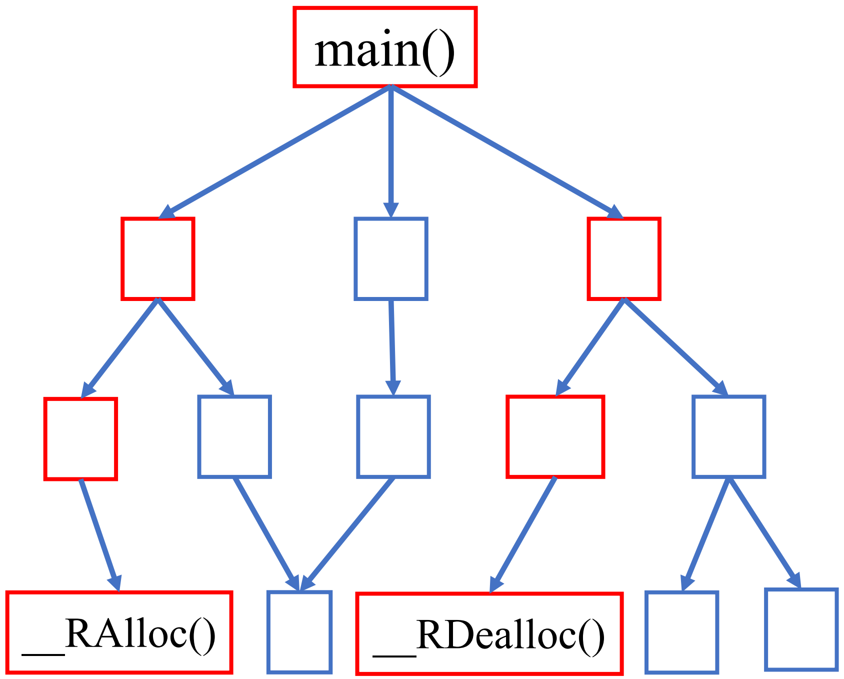
Resource operation	Resource modeling statements
pFile = fopen(…);	pFile = fopen(…); __RAlloc(pFile != NULL? 1 : 0);
fclose(pFile);	__RDealloc(pFile != NULL? 1 : 0); fclose(pFile);
p = malloc(…);	p = malloc(…); __RAlloc(malloc_usable_size(p));
free(p);	__RDealloc(malloc_usable_size(p)); free(p);

The fuzzer will track and capture the parameters of __RAlloc/__RDealloc() to maintain the amount of resource usage

Step 1: Static analysis and instrumentation

- 1.2: Identifying functions involving resource usage
  - **insight**: many functions and basic blocks in the program are not relevant to resource usage
  - **goal**: guide fuzzing to cover functions and basic blocks that are relevant to resource usage
    - use call graph to identify all functions that directly or indirectly invoke resource-usage operations
    - instrument coverage-label function __covl() before the invocation of these functions

__covl(): to label basic blocks that involve resource usage



Function call graph

Step 1: Static analysis and instrumentation

- 1.3: Identifying control-flow branches where resource-usage locates
  - for each program block containing invocations of __RAlloc(), _RDealloc(), __covl () or exit(), we instrument label function __covl ()
    - before the control-flow branch where this block locates in (e.g., in the then branch) and
    - at the beginning of the block in the other branch (e.g., the else branch).

```
+ __covl ();
  if (…){
     …
      __RAlloc(…);
  }
  else{
+    __covl ();
     …
  }
  …
```

Step 1: Static analysis and instrumentation

- 1.4: Instrumentation
  - use program transformation tool Coccinelle, to automatically instrument statements invoking resource-usage modeling functions __RAlloc/_RDealloc() as well as coverage-label function __covl () into the original program

https://coccinelle.gitlabpages.inria.fr/website/

- Coccinelle
  - a program matching and transformation engine
  - providing the language SmPL (Semantic Patch Language) for specifying desired matches and transformations in C code

Step 1: Static analysis and instrumentation

Example illustration

```
static SVCXPRT *makefd_xprt(int fd,  u_int sendsize,
                                     u_int recvsize)
{
  . . .
  if (fd >= FD_SETSIZE) { . . . ; return NULL; }
  . . .
  return (xprt);
}

static bool rendezvous_request(SVCXPRT *xprt)
{
   . . .
   if ((sock = accept(xprt->xp_fd, (struct sockaddr *)
           (void *)&addr, &len))   < 0) {. . . ; return false; }
   . . .
   newxprt = makefd_xprt(sock, r->sendsize, r->recvsize);
   if (newxprt==NULL){
       raise(SIGSEGV); //simulating CVE-2018-14622
    }
    . . .
}
```

(a) Original Program (extracted from *libtirpc*)

```
@ accept @
type T;
expression E;
identifier id;
@@
(
if ((E = accept(...)) < 0){ ... }
+     __covl();
+     __RAlloc(1);
|
...
)
```

(b) Semantic Patch

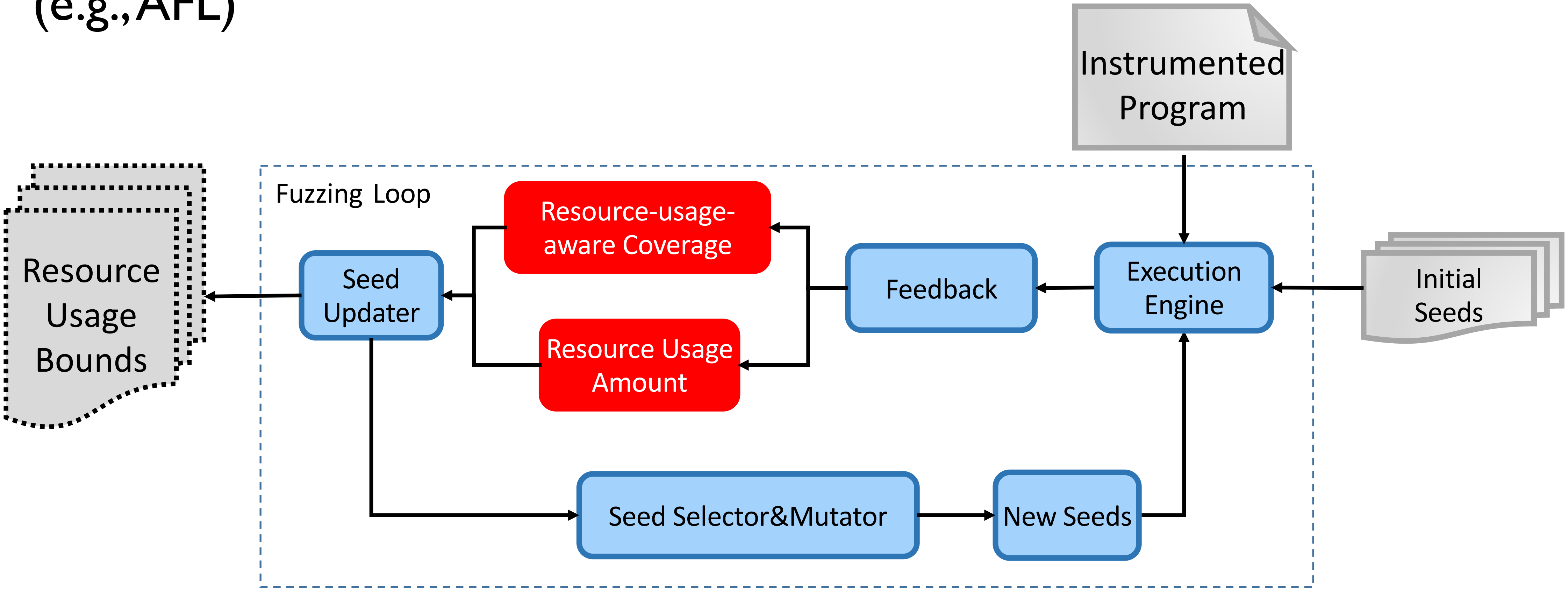
```
static SVCXPRT *makefd_xprt(int fd,  u_int sendsize,
                                     u_int recvsize)
{
  . . .
  if (fd >= FD_SETSIZE) { . . . ; return NULL; }
  . . .
  return (xprt);
}

static bool rendezvous_request(SVCXPRT *xprt)
{
   . . .
  if ((sock = accept(xprt->xp_fd, (struct sockaddr *)
          (void *)&addr, &len))   < 0) {. . . ; return false; }
  __covl();
  __RAlloc(1);
  . . .
  newxprt = makefd_xprt(sock, r->sendsize, r->recvsize);
  __covl();
  if (newxprt==NULL){
      __covl();
      raise(SIGSEGV); //simulating CVE-2018-14622
   }
   . . .
}
```

(c) Instrumented Program

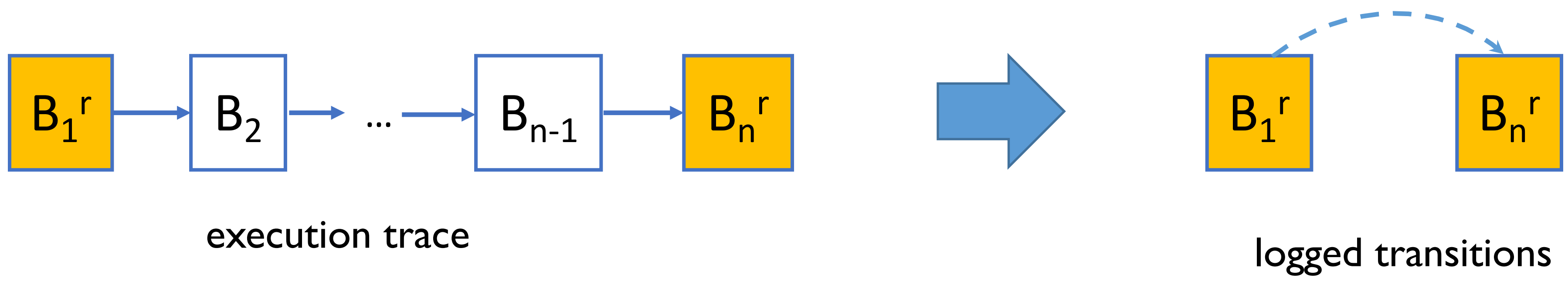
Step 2: Fuzzing loop

- Similar to the process of traditional coverage-based grey-box fuzzers (e.g., AFL)



Step 2: Fuzzing loop

- Resource-usage aware coverage
  - traditional coverage-based grey-box fuzzers capture basic block transitions, and log edge coverage information during runtime
  - **insight**: many basic blocks in the program are not relevant to resource usage
  - **idea**: resource-usage-aware edge coverage
    - log only transitions between those basic blocks that contain resource-usage modeling functions, coverage-label function __covl () and exit() function



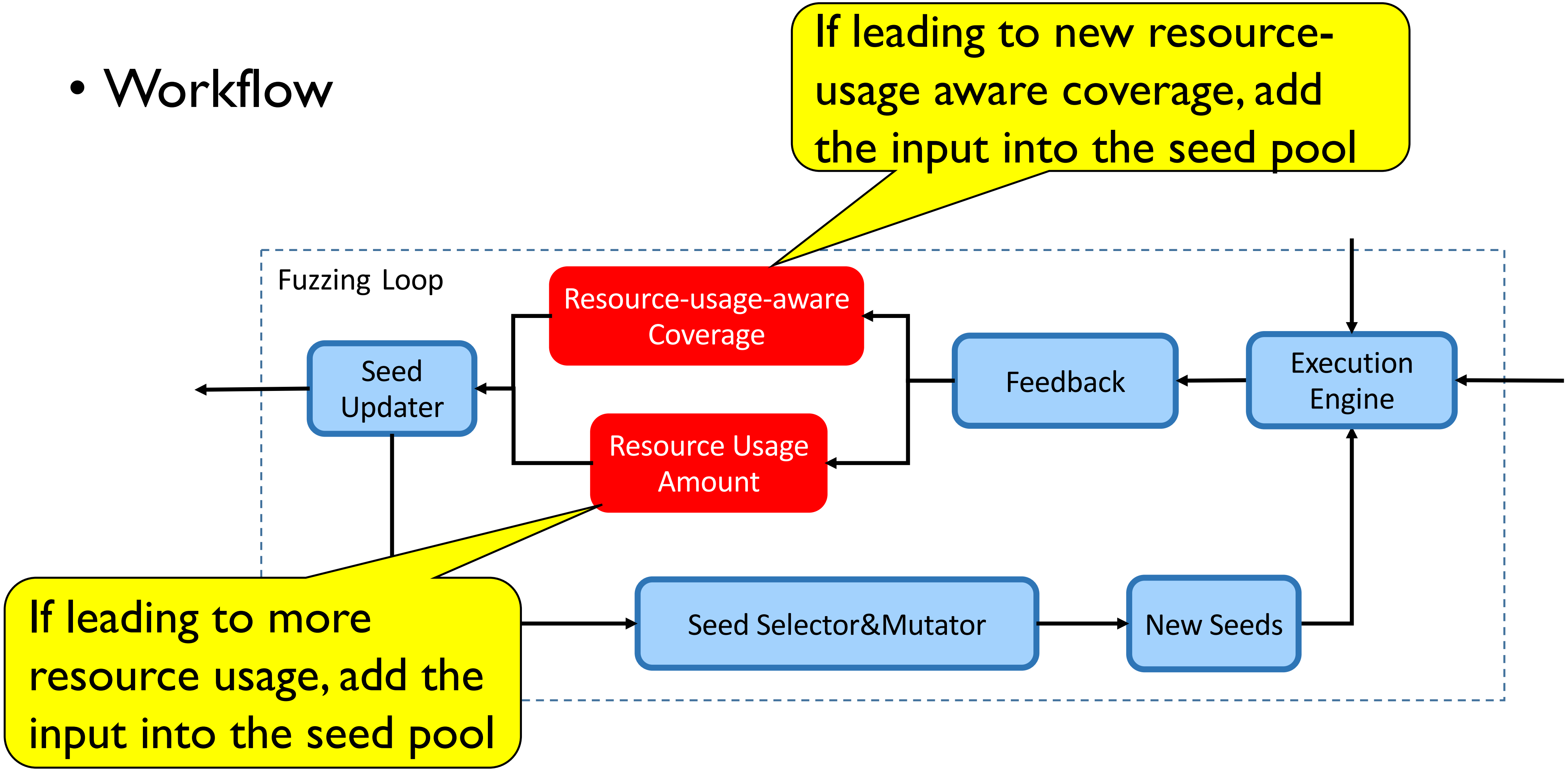
Step 2: Fuzzing loop

- Resource-usage amount guidance
  - the fuzzer collects resource-usage amount, by maintaining
    - resc_cur: the current amount
    - resc_peak: the historical peak amount of resource usage
  - the fuzzer captures the parameters of __RAlloc/__RDealloc(n), and updates resc_cur and resc_peak

Resource operation	resource-usage amount change
__RAlloc(n);	resc_cur += n; if(resc_cur > resc_peak) resc_peak = resc_cur;
__RDealloc(n);	resc_cur -= n;

Step 2: Fuzzing loop

- Workflow



Overview

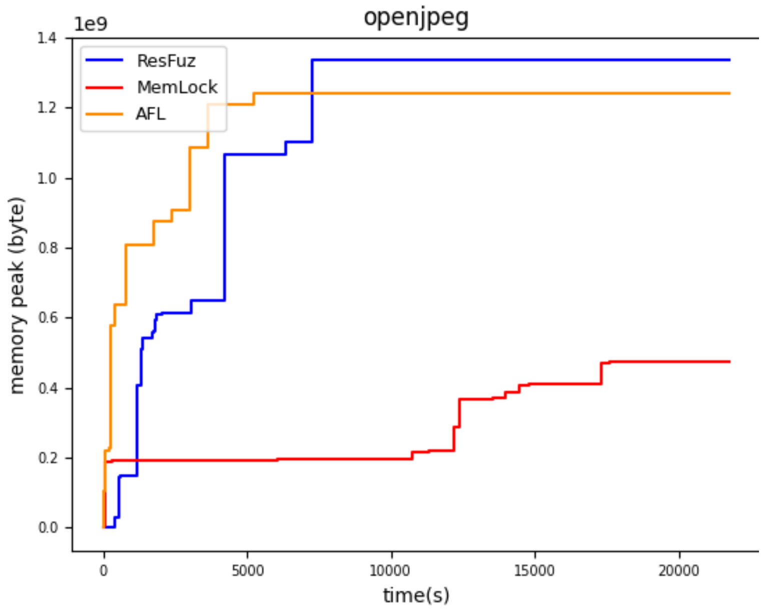
- Motivation
- Approach
- Experiment
- Conclusion

Experiment

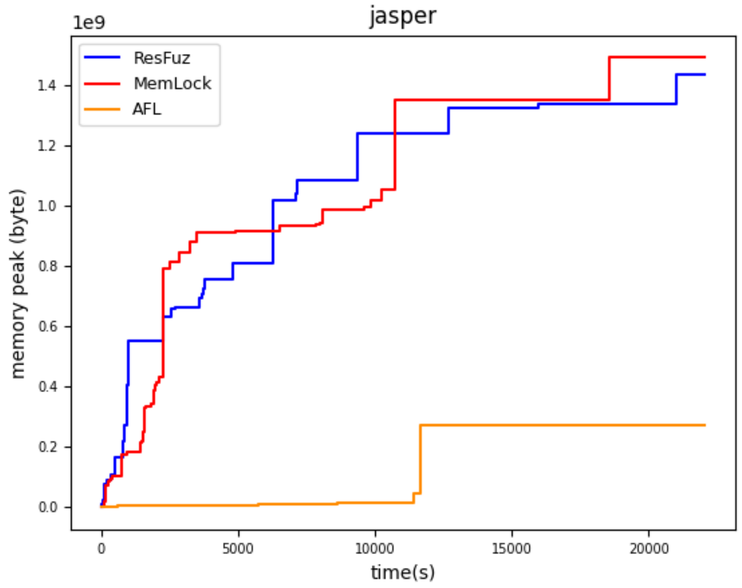
- Implementation: ResFuz
  - build on top of MemLock [Wen et al. ICSE20]
  - use Coccinelle to conduct program instrumentation
- Benchmark
  - for heap: jasper, openjpeg
  - for stack: yara
  - for socket connections: *libtirpc_slice* extracted from an old version of *libtirpc*
  - for user-defined resources
    - jasper: jas_malloc(), jas_free() to manage a heap memory pool with a user-configurable size
    - openjpeg: opj_malloc(); opj_free() to manage a specic type of heap memory
- Baseline fuzzers:
  - AFL [AFL 2.52b]
  - MemLock [Wen et al. ICSE20]

Preliminary Experimental Results

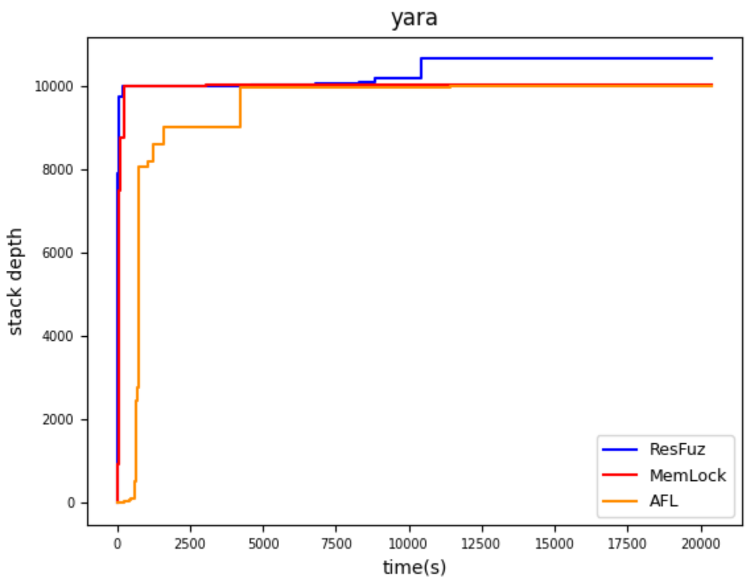
general heap



general heap



stack depth

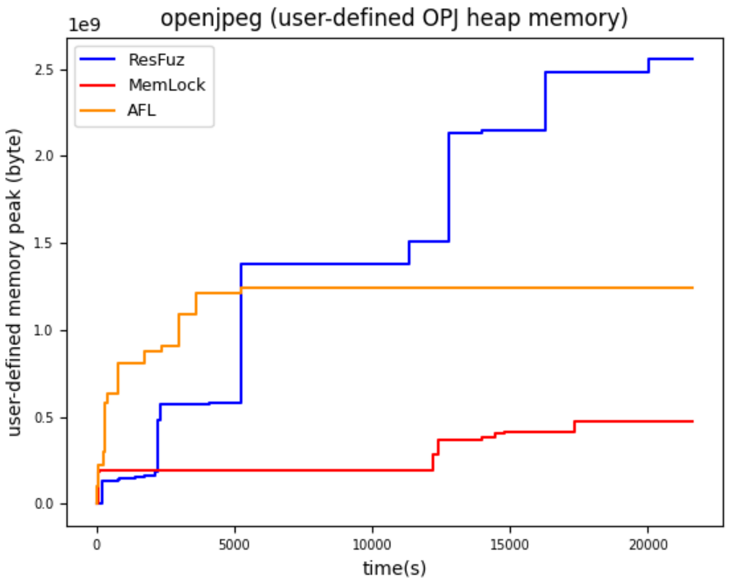


The growth trend of resource usage

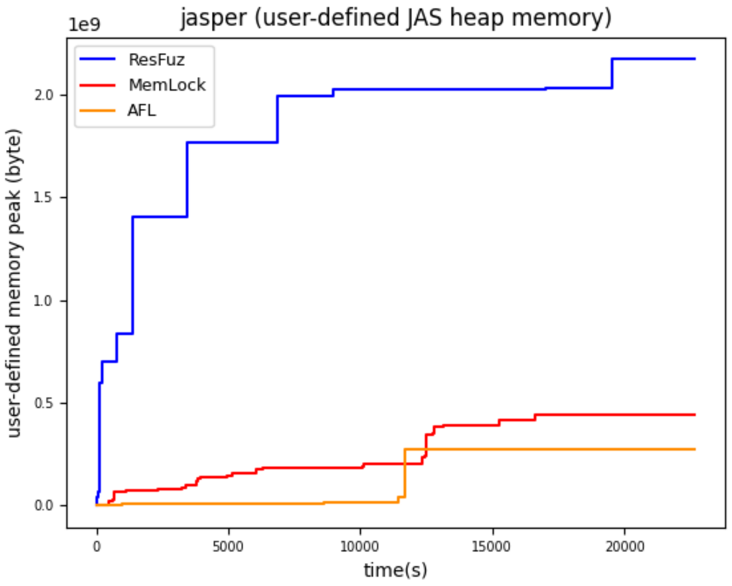
- ResFuz performs stably well
  - MemLock uses default branch coverage and memory consumption to guide fuzzing
  - ResFuz uses resource-usage-aware coverage and resource-usage amount to guide fuzzing
  - AFL uses default branch coverage to guide fuzzing

Preliminary Experimental Results

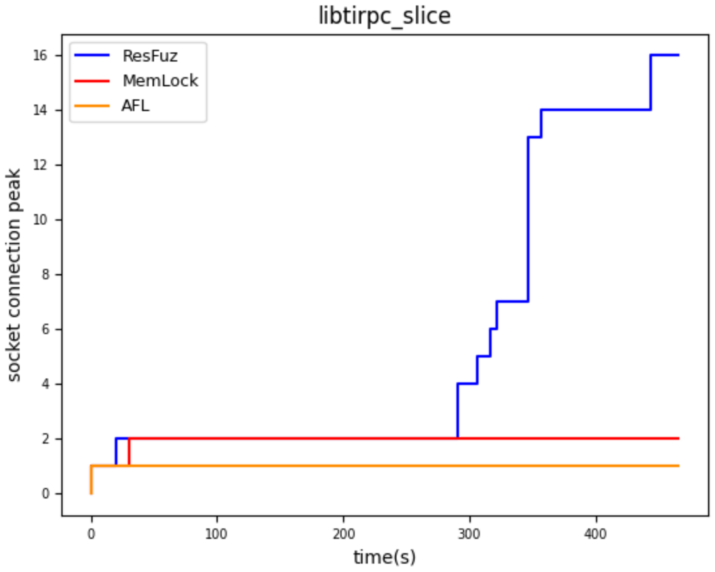
user-defined OPJ heap



user-defined JAS heap



socket connections



The growth trend of resource usage

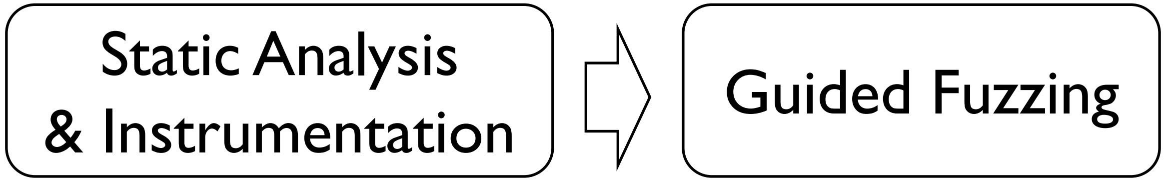
- ResFuz performs much better results than MemLock and AFL
  - MemLock uses default branch coverage and memory consumption to guide fuzzing
  - ResFuz uses resource-usage-aware coverage and resource-usage amount to guide fuzzing
  - AFL uses default branch coverage to guide fuzzing

Overview

- Motivation
- Approach
- Experiment
- Conclusion

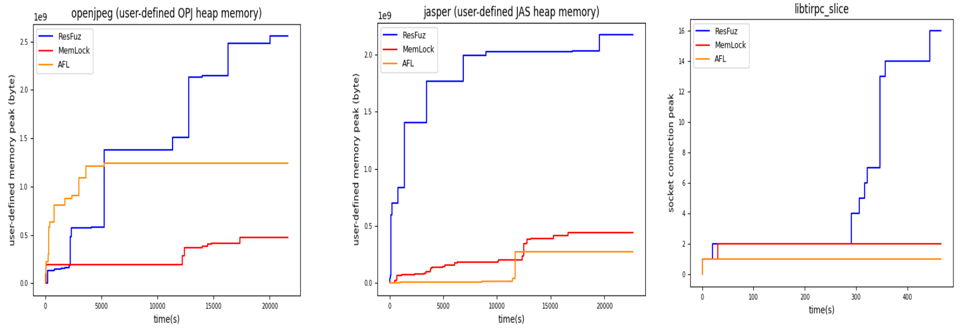
Summary

- **Approach**: Resource-Usage-Aware Fuzzing
  - leverage semantic patch to make use of static analysis information for instrumentation
  - employ resource-usage amount and resource-usage-aware coverage to guide fuzzing



- **Tool**: ResFuz
  - https://doi.org/10.5281/zenodo.5894821

- **Experiments**



Future Work

- More real-world programs and more kinds of resources
- Compare with more state-of-the-art fuzzing tools
- Evaluate our approach in detecting resource-usage bugs and vulnerabilities in real-world programs

Thank you
Any Questions?