Spitfire Design and Specs

Tim Leek

Andy Davis

Heather Preslier

Chris Connelly

12/5/2019

Introduction

Spitfire is a fuzzer framework. Its purpose is to enable fuzzing research.

- Experiments are scripted and curated
- Existing tools (Angr, PANDA, Triton, Mayhem, etc) useable, wrapped to match an interface
- Code versions, config files, all captured
- Analytic results made available in a `knowledge_base`
- Everything is in the repo; experiments can be checked-out and run
- Smart fuzzers from literature can be *implemented* with Spitfire
- Kubernetes-based so scaling is just a matter of $$

Fuzzing manager

Orchestrates the fuzzing campaign

- This is the fuzzing strategy at highest level of abstraction
- Run by a Kubernetes cron job, every N minutes
    i. Take stock of the state of the fuzzing campaign by consulting the `knowledge_base`
    ii. Decide what tools to deploy (fuzzer, symbolic exec, taint, etc) with what parameters, and then launchesKubernetes jobs
    iii. Exit, to be run again by *cron*
- To *implement* a smart fuzzer from a paper, one would write a new fuzzing manager

Persistent `knowledge_base`

Spitfire compute will happen in the cloud. Each (wrapped) tool run is required to be finite in time and will be deposit its output a persistent `knowledge_base`.

- `knowledge_base` could be SQL or NoSQL database or a key-value store or whatever
- Lincoln will implement libraries in C and Python to store the various data types (`Interest`, `Coverage`, `FileExtent`, etc) to at least one `knowledge_base`. Probably postrgres. Other back ends will come later
- Once data is in the `knowledge_base`, there will be simple interfaces to access it (read and modify) that are independent of how the `knowledge_base` is implemented

Persistent filesystem

We also need a filesystem, for things that don't belong in a `knowledge_base`. All cloud instances will mount this volume and will be able to refer to files with consistent path names.

- Seed input corpora
- Binaries for programs to be fuzzed
- All output `interesting_files` from fuzzing or symbolic exec / solving
- PANDA recordings and qcows

Mutational fuzzer tool Interface

Fuzzes a single file, collecting **`Interesting`** results

I/O	Name	Type	Required?	Location	Comment
input	`input_file`	`File`	Yes	`file_system`	
input	`max_iterations`	`Integer`	No	`config`	
output	`interesting_files`	`File Array`	Yes	`file_system` and `knowledge_base`	
output	`why_interesting`	`Interest Array`	Yes	`knowledge_base`	one per `interesting_file`
output	`marginal_coverage`	`Coverage Array`	No	`knowledge_base`	one per `interesting_file`
output	`global_coverage`	`Coverage`	No	`knowledge_base`	

Notes:

- `max_iterations` is the number of fuzzings to try
- `Interest` is a set of possible detector outputs (new coverage, exceptions, ASAN, assertions, etc)
- `marginal_coverage` is the coverage unique to this file
- `global_coverage` is the union of coverage for all `interesting_files`

Grammar-based fuzzer tool Interface

Generates draws from a grammar, collecting **`Interesting`** results

I/O	Name	Type	Required?	Location	Comment
input	`max_iterations`	`Integer`	No	`config`	
output	`interesting_files`	`File Array`	Yes	`file_system` and `knowledge_base`	
output	`why_interesting`	`Interest Array`	Yes	`knowledge_base`	one per `interesting_file`
output	`marginal_coverage`	`Coverage Array`	No	`knowledge_base`	one per `interesting_file`
output	`global_coverage`	`Coverage`	No	`knowledge_base`	

Notes:

- `max_iterations` is the number of draws from the grammar
- The only input is the number of fuzzings to try: `num_fuzzed_files`
- There is a grammar, but this isn't really an input and there's no benefit in standardizing format
- Might need to add some notion of *grammar* coverage?
- Output is same as mutational fuzzer

Taint analysis tool Interface

Labels taint on **`input_file`**, then tracks taint, determining mapping from **`file_extents`** to **`tainted_instructions`**

I/O	Name	Type	Required?	Location	Comment
input	`input_file`	`File`	Yes	`file_system`	
output	`tainted_intructions`	`TaintedInstruction Array`	Yes	`knowledge_base`	
output	`file_extents`	`FileExtent Array`	Yes	`knowledge_base`	
output	`taint_map`	`TaintedInstruction * FileExtent Array`	Yes	`knowledge_base`	
output	`comp_dist`	`Integer Array`	No	`knowledge_base`	

Notes:

- `TaintedInstruction` is an instruction in the target program seen to be tainted, probably not just program counter but also instruction type
- `FileExtent` is some set of positional bytes in the input file that are seen to taint an instruction at some instant in the trace
- `taint_map` is a sparse matrix mapping `FileExtent`s to `TaintedInstruction`s
- `comp_distance` is a companion to `taint_map` (same size) indicating computational distance of tainted instruction from inputs

Symbolic execution tool Interface

Makes **`input_file`** bytes symbolic, executes symbolically, solve branches etc. New files vetted for **`Interest`**

I/O	Name	Type	Required?	Location	Comment
input	`input_file`	`File`	Yes	`file_system`	used to set input length
output	`interesting_files`	`File Array`	Yes	`file_system` and `knowledge_base`	
output	`why_interesting`	`Interest Array`	Yes	`knowledge_base`	one per `interesting_file`
output	`path_constraints`	`ConstraintSet Array`	No	`knowledge_base`	one per `solve_file`

Notes:

- `ConstraintSet` is a companion array to `interesting_files` (same length), providing path constraints fed to solver
- Should we distinguish concolic vs symbolic?
- Is search strategy a useful and agreed-upon idea s.t. we can expose it as a generic input?

Coverage

The `Coverage` type is a tuple of various kinds of coverage.
Each element in the tuple is optional. However, it is required that at least one kind of coverage is provided.

- `BlockCoverage` is set of program counters, with counts
- `EdgeCoverage of Integer` is the set of n-edges covered, with counts. An `Edge` is a pair of program counters representing a transition observed between two basic blocks, an n-edge represents n transitions.

Other kinds of coverage are possible, including state coverage. We will tackle these later.

Tools that output coverage will do so in a standard binary file format. Lincoln will write C and Python libraries to marshal this format to the `knowledge_store`.

Taint

The `FileExtent` and `TaintedInstruction` types will also output by taint analysis tools in a standard binary file format. Lincoln will write C and Python libraries to marshal to this format to the `knowledge_store`.

Seed corpus

An initial corpus of `input_files` is needed. It will be selected to span the input space of the program to be fuzzed, along various axes.

- File size
- Program features exercised

Corpora will be part of the Spitfire repo. There may be more than one corpus for a binary target to be fuzzed.

Comments!

Our interest in specifying and building this framework is to facilitate. We are maximally interested in having this used by others.

Let's discuss!