Ghosts of Christmas Past

Fuzzing Language Interpreters Using Regression Tests

Sean Heelan – Persistence Labs

Infiltrate 2014

A note for those of you reading this post-Infiltrate:

Herein you'll find the results of my early investigations into fuzzing interpreters, using test cases generated from existing source code samples.

This is not a comprehensive final report, but an overview of what has and has not worked so far, a high level overview of the algorithms involved, and an indication of where the research is going. The results and conclusions are based on work performed up to the end of April 2014.

Feel free to contact me with follow-up questions. I'll pre-empt the first one with “No, the code is not currently available” =)

Sean

Introduction

About Me

- Founder of Persistence Labs
- Former security researcher at Immunity Inc.
- Interested in building better tools for program analysis

Motivation

- We build program analysis tools
  - If you build program analysis tools and don't do program analysis, you will fail
  - We'd like to maintain a fairly regular stream of analysis problems in order to test existing tools and inspire new ones

Motivation

- Language interpreters are a lucrative target but have significant pain points for fuzzing
- Vast APIs
  - How do we discover them?
  - How do we figure out how to correctly use them?
    - Object instantiation, ordering of API calls, argument counts & types, inter-function/inter-object interaction, environment features & requirements...
  - How do we keep up with new features?
- Manual encoding of API semantics isn't something we're inclined to pursue, so ...

This talk ...

- How do we find new and useful bugs using tests for old and dead bugs
- Impart some “best practice” info on interpreter fuzzing/fuzzing in general
- Document the many ways in which I have failed as an engineer, scientist, and human being

Talk Outline

- What makes for “good fuzzing”?
- The past and present of interpreter fuzzing
- Fuzzing stand-alone interpreters with regression tests
- FragFuzz : A new approach to reference fuzzing using regression tests
- Conclusions & future work

Things That Will Make Your Life Suck Less ...

What makes for good fuzzing?

- Atte Kettunen @ 44Con - “Stay green”
  - Target: New software, new features of that software, or new platforms for older software
  - Approach: New input generation methods, greater scale
  - Auxiliary tools: New/Better instrumentation, crash detection
- Often sufficient to hit 'green' in just one of these categories to find new bugs

What makes for bad fuzzing?

- Ben Nagy @ Blackhat/DailyDave
  - Test generation != fuzzing
    - Fuzzing is not test case generation, test case generation is *one part* of fuzzing

So … Fuzzing

- You can skimp on test generation “smarts”, just ensure replacement is a config file change
- You can't skimp on automation of test delivery, crash detection, and master/slave deployment
- Crash filtering, bucketing, and prioritisation are 3 different problems and all three need to be completely automated, or you'll hate your life (as with TG, you can start dumb)
  - Filtering – Do we even record this as an issue?
  - Bucketing (aka. stage 0 root cause analysis) – What other crashes are likely from the same issue?
  - Prioritisation – In what order am I going to feed these to my next-stage analyser, meatsack or otherwise

A Brief, Inaccurate and Incomplete Introduction to Interpreter Fuzzing

Interpreter Fuzzing

- Quite a lot of prior art that's potentially useful
- Worth enumerating to get the lay of the land
  - “Stay green”
- What can we reuse
  - End goal – find bugs, ideally as little activity that resembles “research” or “work” as possible
  - **Fail #1**: I'm here. Clearly something went wrong.

PHP

- The worlds most popular remote access tool *(now with limited support for web development)*
- Pre-5.4 recipe for insta-win
  - Enumerate global methods
  - Generate a few basic objects
  - Call random functions with multiple references to the same objects + random data
  - All the double-frees you could ever want

PHP

- Post-5.4
  - Pass by reference removed
  - Global functions + random args still finds some bugs, but far fewer
- Problem: Hitting new code requires correct object initialisation, sequences of API calls, interplay between various objects
  - We don't want to manually write initialisation code for each PHP object/component we want to test
  - Too much structure to randomly generate

Javascript

- lcamtuf – ref_fuzz (2010)
  - Strategy (sort of):
    0. Begin with default list of DOM objects
    1. Enumerate new methods across objects
    2. Call methods to generate new objects, feeding in existing objects as arguments
       - If return value is an object, save it
    3. Randomly store objects as attributes of other objects
    4. Randomly delete some stored objects
    5. Go to step 1

Ref-Fuzz

- Good
  - Broke *
  - Provides a decent core algorithm for reference fuzzing
    - Definitely an idea worth stealing if we can find new objects to construct, or find ways to put already covered objects into new states

Ref-Fuzz

- Bad
  - Absolute nightmare to extract the root cause
  - No record of injected tests
    - Makes people resort to craziness like post-failure memory greping for the crashing file
  - Every bug that it can find, that is reasonably traceable, is dead

Javascript

- Far fewer test case generators released in the past few years
- Inference from public presentations – large numbers of people doing *ref_fuzz* style fuzzing, but in a more sane architecture
  - Browsers have a constantly growing attack surface, so the 'green' can be found here, at the very least

What can we salvage?

- The reference fuzzing algorithm is nice, but where will we get our green?
- We need to apply the algorithm to objects not covered by the original ref_fuzz, or subsequent work
- Same problem as with PHP
  - We need to put together structured, chains of API calls, in order to generate new objects and interact with them
  - We need to do this automatically

Javascript

- Christian Holler – Langfuzz (2012)
  - One of the few publicly discussed, and interesting, methods of test case generation from the past few years
  - Key insight:
    - We already have available extensive examples of syntactically, and semantically, valid ways to exercise a huge amount of the attack surface of web browser – regression tests
    - Usually standalone, fairly self-contained, target things known to have broken in the past

Langfuzz

- Requirements
  - Language grammar
  - A large and diverse set of sample tests

Langfuzz

- Algorithm

  0. Feed the tests though a “learning” stage that uses the language grammar to learn examples of valid “tokens”

  1. Produce new tests by starting from an existing test, and replacing random tokens with tokens of the same type, learned in step 0

  1.1. Alternatively, tests may be created purely generatively by traversing the grammar and randomly selecting a learned expansion for each non-terminal encountered

Langfuzz

```
var a = getA();
var x = y.foo(a,b,c);
var z = x.bar(10, b);

if (a.i > 0) {
    a.meh(x, b);
} else if (a.i > 9) {
    a.meh(x, c);
}
```

- Example Tokens:
  - (a, b, c)
  - y.foo(..., …, …)
  - a.i
  - a.i > 0
  - a.i > 9
  - if (…) {
    a.meh(x, b);
    } ...

Langfuzz

```
var a = getA();
a.i = foo();
if (a.i < 9) {
    a.heh();
}
```

- Example tokens:
  - a.i = foo()
  - a.i < 9
  - a.heh()

Langfuzz

```
var x = y.foo(a,b,c);
var z = x.bar(10, b);

if (a.i > 0) {
    a.meh(x, b);
} else if (a.i > 9) {
    a.meh(z, c);
}
```

```
var x = y.foo(a,b,c);
var z = x.bar(10, b);

if (a.i > 10) {
    a.meh(z, b);
    a.heh();
} else if (a.i > 0) {
    a.meh(x, c);
}
```

Langfuzz Results

- 18 Chromium security rewards
- 12 Mozilla security rewards
- 18 bugs in PHP
- Quite a few more found since

Ghosts of Christmas Past

- Regression tests can solve our primary problems
  - Provide semantically valid examples of object instantiation
  - Provide examples of the APIs provided by different object types, as well as correct patterns for their usage
  - Require zero knowledge of any of the tested subsystems, on our behalf
    - Mimicking the set-up of their testing harness is sufficient

Fuzzing Stand-Alone Interpreters

Prototype

- Targets: PHP, Ruby, D8, Firefox JS shell
- Positives:
  - Fast to launch
  - Self-contained
  - Allows for a straight-forward fuzzing architecture
- Negatives
  - Reduced attack surface
  - The targets we really care about (FF JS Shell, D8) have been extensively tested already

Where's our Green?

- Using regression tests separates us from people doing ref_fuzz style testing
  - How do we separate ourselves from Langfuzz?
- PHP
  - Langfuzz only worked with 300 or so of the PHP tests but there are actually close to 10k
- Ruby
  - Untested via langfuzz

D8/Firefox JS Shell

- Firefox comes with two sets of Javascript-only tests
  - JIT tests (js/src/jit-test) and the jsreftests (js/src/tests/)
- Langfuzz used both sets
- Our green needs to come from somewhere else
  - Test generation

Extraction

- *“You're wasting your time, I ran all the .js tests through the same algorithm and found nothing”* - Nameless accomplice
- You need to provide the same environment as the test harness
  - Env. variables
  - Command line options
  - Library code
  - Resource access

Extraction: PHP

- Tests scattered throughout the repository
- Many of the tests contain some boilerplate, which needs to be removed in order to get a standalone .php
- Solution: find + custom Python script to strip boilerplate!
    - Came up with about 1500 tests
    - Boilerplate removal broke quite a few

Observation Fail

- `./run-tests.php`
  - Presumably the same mistake the langfuzz guys made
  - `php run-tests.php  --temp-source `pwd` --temp-target /tmp/phptests/ -p ./sapi/cli/php -q`
  - End result:
    - ~8.5k boiler-plate free .php tests in /tmp/phptests

Extraction: Ruby

- Again, tests scattered through the repository
- Findable via `make test-all`
  - 658 in total
- Each test relies on some environment variables
  - RUBY, RUBYLIB
- Unlike PHP, each test embedded in a unit-test style harness
  - Individual tests are smaller and give us less information on API interaction and structure
  - Many tests bundled together in a single .rb file
  - Blind manipulation of the test risks breaking the harness

Extraction: Firefox

- Both the JIT tests and the jsreftests come with a .py to run the tests
  - Running in verbose modes spits out all the configuration options required for the Firefox JS shell
  - The D8 equivalent options are easily found by reading it's –help output
- JIT tests – ~4k total
- jsreftests – ~6.5k total

Mutation

- Radamsa from OUSPG chosen as our baseline
  - Input format agnostic
  - Uses standard parsing rules to try to identify nodes in a file
  - Generic, so applicable to PHP, Ruby and JS
  - Lots of input manipulation strategies – you'll want to disable things more likely to result in syntactic issues (e.g. byte flipping)

Radamsa

- Weaknesses vs Langfuzz
  - Not language aware, so a large number of tests will end up being syntactically invalid
  - Far weaker form of cross-test interpolation
- Strengths vs Langfuzz
  - “Solves” test generation immediately so we can focus on the remainder of the framework
    - If test generation is a plugin, which it should be, then we can come back to it later

Test Delivery & Crash Detection

- PHP, Ruby, D8, FF JS shell
  - Standalone, command line apps
  - We've already figured out the required environment variables etc
  - Each builds cleanly using ASan
  - Filtering, bucketing and prioritisation hacked up based on the available info from ASan

Aside: Address Sanitiser (ASan)

```
=================================================================
==16111==ERROR: AddressSanitizer: heap-use-after-free on address 0xb7fafe48 at
pc 0x8687f49 bp 0xbfff4ef8 sp 0xbfff4ef0

READ of size 4 at 0xb7fafe48 thread T0

    #0 0x8687f48
(/home/user/Documents/Testing/php/builds_5208e8/asan/sapi/cli/php+0x63ff48)

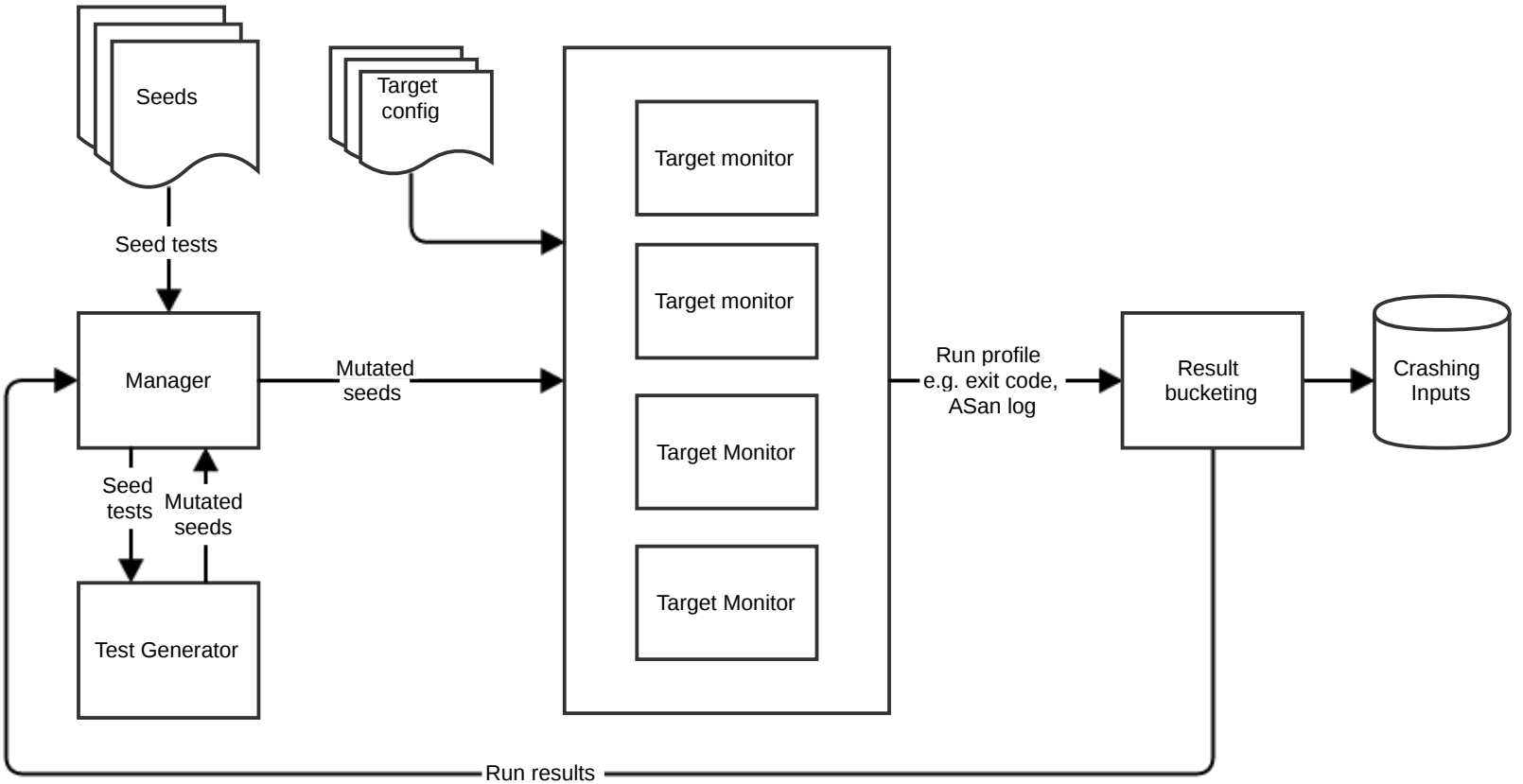
    #1 0x8687b82
(/home/user/Documents/Testing/php/builds_5208e8/asan/sapi/cli/php+0x63fb82)

    #2 0x861fded
(/home/user/Documents/Testing/php/builds_5208e8/asan/sapi/cli/php+0x5d7ded)

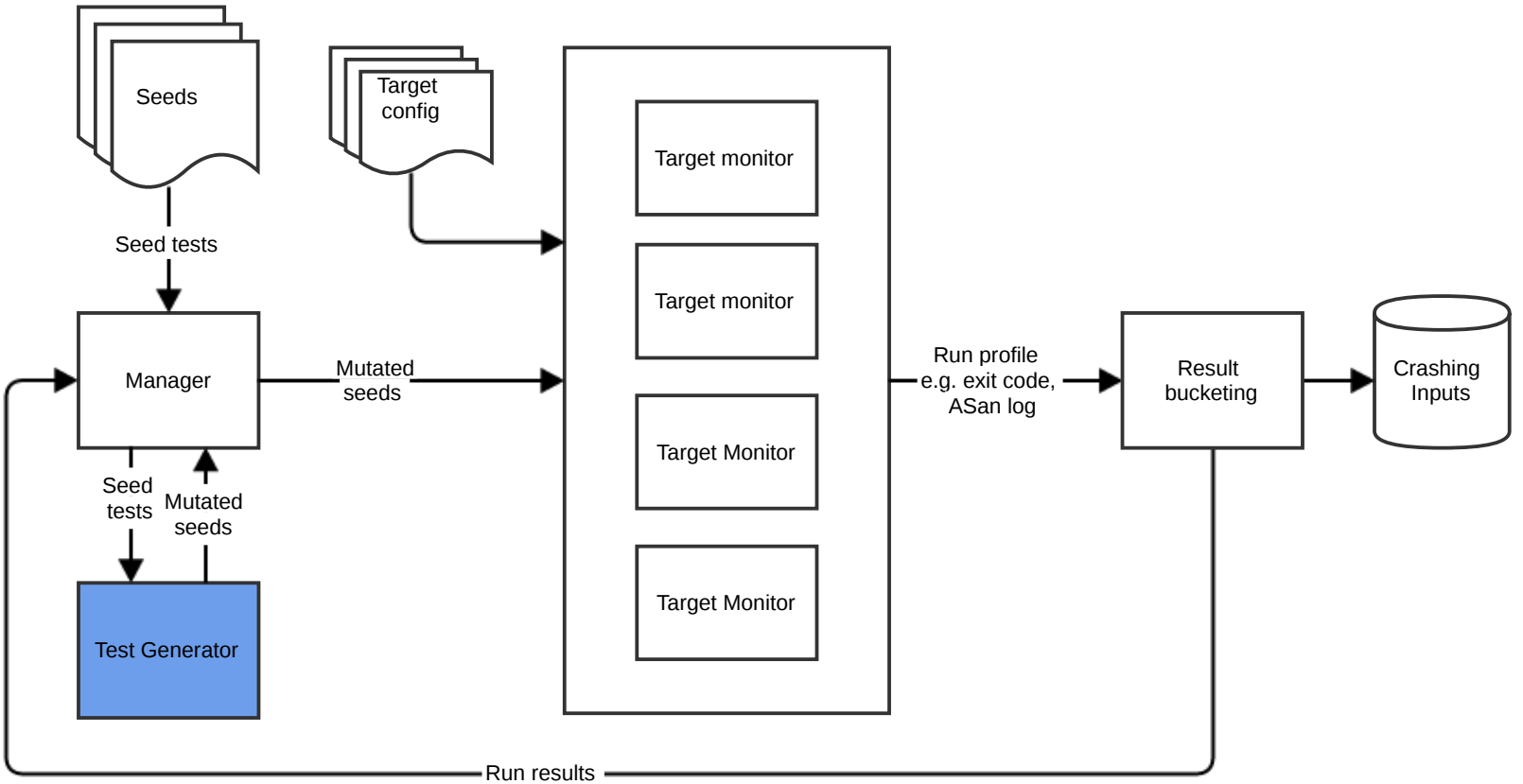
    #3 0xaf3c394
(/home/user/Documents/Testing/php/builds_5208e8/asan/sapi/cli/php+0x2ef4394)

    #4 0xaa3e3e4
(/home/user/Documents/Testing/php/builds_5208e8/asan/sapi/cli/php+0x29f63e4)
```

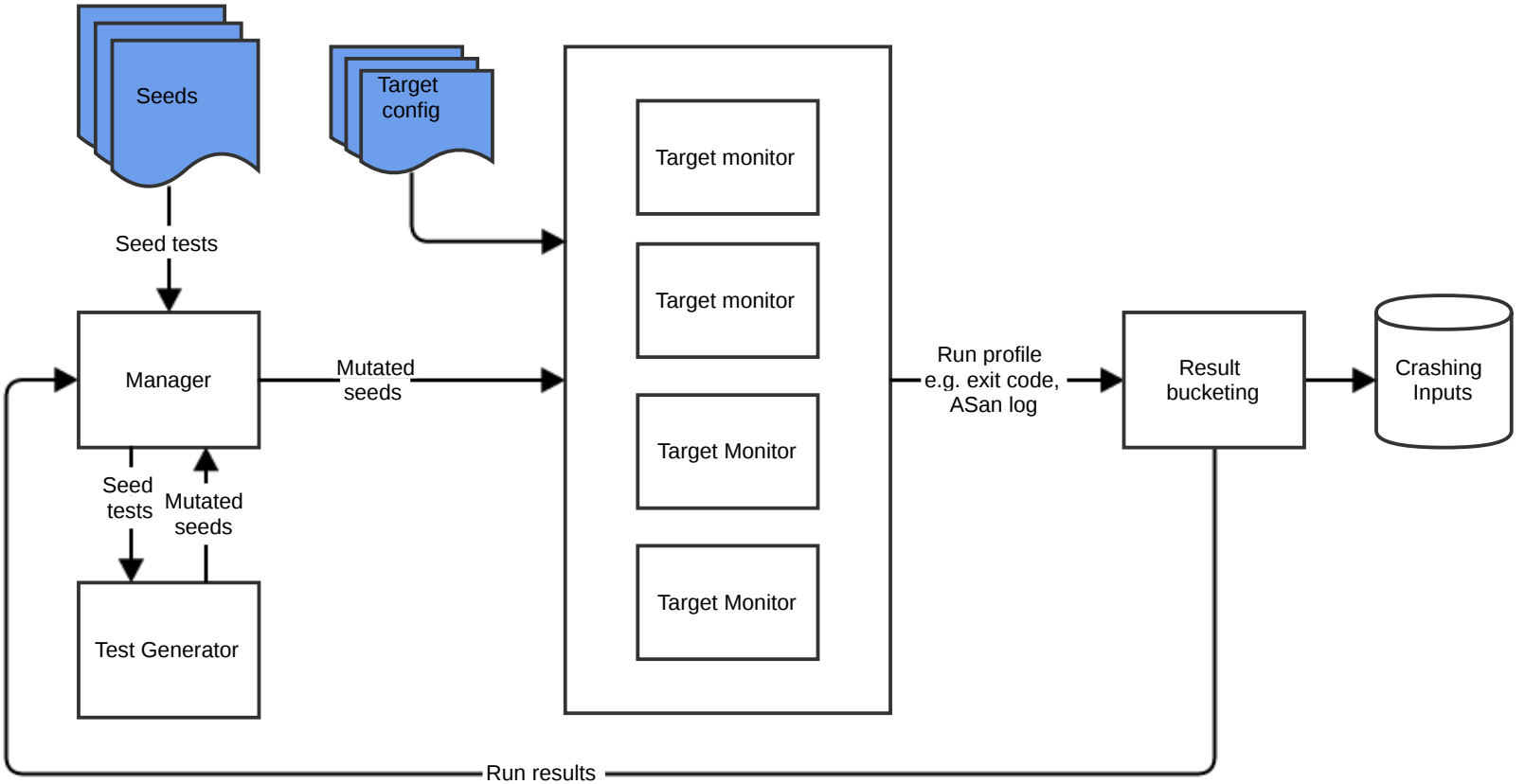
Malamute Architecture



Architecture: Changing Mutation Method



Architecture: Retargeting



Results

Example #1: PHP Seed

```
<?php
$db = new SQLite3(':memory:');

$db->exec('CREATE TABLE foo (id INTEGER, bar STRING)');
$db->exec("INSERT INTO foo (id, bar) VALUES (1, 'This is a test')");

$stmt = $db->prepare('SELECT bar FROM foo WHERE id=:id');
$stmt->bindValue(':id', 1, SQLITE1_INTEGER);
$stmt->reset(“dummy”);
$stmt->reset();

//var_dump($db);
//var_dump($db->close());
echo "Done\n";
?>
```

Example #1: PHP UAF

```
<?php
$db = new SQLite3(':memory:');

$db->exec('CREATE TABLE foo (id INTEGER, bar STRING)');
$db->exec("INSERT INTO foo (id, bar) VALUES (1, 'This is a test')");

$stmt = $db->prepare('SELECT bar FROM foo WHERE id=:id');
$stmt->bindValue(':id', 1, SQLITE1_INTEGER);
$stmt->reset(($db->close()));
$stmt->reset();

//var_dump($db);
//var_dump"dummy";
echo "Done\n";
?>
```

Example #1: Required Structure

```
<?php
$db = new SQLite3(':memory:');

$db->exec('CREATE TABLE foo (id INTEGER, bar STRING)');
$db->exec("INSERT INTO foo (id, bar) VALUES (1, 'This is a test')");

$stmt = $db->prepare('SELECT bar FROM foo WHERE id=:id');
$stmt->bindValue(':id', 1, SQLITE1_INTEGER);
$stmt->reset(($db->close()));
$stmt->reset();

//var_dump($db);
//var_dump"dummy";
echo "Done\n";
?>
```

Example #2: PHP Seed

```
<?php
$XML = <<<XML
<?xml version="1.0"?>
<ns1:listOfAwards xmlns:ns1="http://www.fpdsng.com/FPDS">
...
</ns1:listOfAwards>
XML;
…

function startElement($parser, $name, $attribs) { echo $name . PHP_EOL; }
function endElement($parser, $name) { echo $name . PHP_EOL; }

$xml_parser = xml_parser_create();
xml_set_element_handler($xml_parser, 'startElement', 'endElement');
xml_parser_set_option($xml_parser, XML_OPTION_SKIP_TAGSTART, 4);

xml_parse($xml_parser, $XML);
?>
```

Example #2: PHP Info Leak

```
<?php
$XML = <<<XML
<?xml version="1.0"?>
<ns1:listOfAwards xmlns:ns1="http://www.fpdsng.com/FPDS">
...
</ns1:listOfAwards>
XML;
…

function startElement($parser, $name, $attribs) { echo $name . PHP_EOL; }
function endElement($parser, $name) { echo $name . PHP_EOL; }

$xml_parser = xml_parser_create();
xml_set_element_handler($xml_parser, 'startElement', 'endElement');
xml_parser_set_option($xml_parser, XML_OPTION_SKIP_TAGSTART, -33543462);

xml_parse($xml_parser, $XML);
?>
```

Example #2: Required Structure

```
<?php
$XML = <<<XML
<?xml version="1.0"?>
<ns1:listOfAwards xmlns:ns1="http://www.fpdsng.com/FPDS">
...
</ns1:listOfAwards>
XML;
…

function startElement($parser, $name, $attribs) { echo $name . PHP_EOL; }
function endElement($parser, $name) { echo $name . PHP_EOL; }

$xml_parser = xml_parser_create();
xml_set_element_handler($xml_parser, 'startElement', 'endElement');
xml_parser_set_option($xml_parser, XML_OPTION_SKIP_TAGSTART, -33543462);

xml_parse($xml_parser, $XML);
?>
```

Example 3 ... N: Anything Involving Serialisation

[Censored for Publication ;)]

Testing Summary

- Malamute(Radamsa) vs PHP/Ruby
  - 8 cores, 8 GB Ram
    - Translates to 16 monitors (16 concurrent tests)
  - Approx. 7 seconds to process 1000 tests
    - 514,000 tests per hour
    - 12 million tests per day
  - Total run time (1000 tests per seed, processed each seed once)
    - PHP : 2500 out of the 8.5k tests, < 5 hours
    - Ruby : 658 tests, just over 1 hour
  - Bugs
    - 100s of crashes per project
    - PHP uniques: 30-ish
    - Ruby uniques: 10-ish

Testing Summary

- Malamute(Radamsa) vs PHP/Ruby
  - 8 cores, 8 GB Ram
    - Translates to 16 monitors (16 concurrent tests)
  - Approx. 7 seconds to process 1000 tests
    - 514,000 tests per hour
    - 12 million tests per day
  - Total run time (1000 tests per seed, processed each seed once)
    - PHP : 2500 out of the 8.5k tests, < 5 hours
    - Ruby : 658 tests, just over 1 hour
  - **Bugs**
    - **100s of crashes per project**
    - **PHP uniques: 30-ish**
    - **Ruby uniques: 10-ish**

Testing Summary

- Malamute(Radamsa) vs Firefox JS Shell/D8 (Attempt 1)
  - 8 cores, 8 GB Ram
    - Translates to 16 monitors (16 concurrent tests)
  - JSRefTests (~ 6.5k)
    - Approx. 20 seconds to process 1000 tests
    - 180,000 tests per hour
    - Just under 4.5 million tests per day
  - JIT-Tests (~4k)
    - Approx. 10 seconds to process 1000 tests
    - 360,000 tests per hour
    - Just over 8.5 million tests per day
  - Total run time (1000 tests per seed, processed each seed once)
    - JSRefTests: ~33 hours for each interpreter
    - JIT Tests: ~20 hours for each interpreter

Testing Summary

- 1000 Radamsa-generated tests per seed turned up nothing useful
- Malamute(Radamsa) vs Firefox JS Shell (Attempt 2)
  - JIT-Tests
    - Approx. 190 hours of runtime
    - Approx. 70,000,000 tests
  - Bugs
    - 1 heap-based buffer overflow
    - A few more NULL pointer dereferences

Conclusions

- PHP/Ruby - “Solved” - We're done here
- D8/FF JS shell – Poor results
  - How likely is Radamsa to generate a test that covers code not covered by a test generated by Langfuzz, jsfunfuzz, and other Mozilla internal projects?
    - Empirically, “not very”

Conclusions

- Problems
  - Many syntactically invalid tests
  - Low percentage of tests containing interesting “looking” reference manipulation
  - Extremely low percentage of tests containing syntactically & semantically valid combinations of multiple seeds
- Causes
  - Lack of a language grammar
  - Generic manipulation techniques (not a fault of Radamsa's, it's purposefully generic, but we've hit a limitation of that strategy)
  - No capacity to deduce functionality not explicitly found in the regression tests
  - Limited capacity to combine functionality described across multiple tests

Going Forward

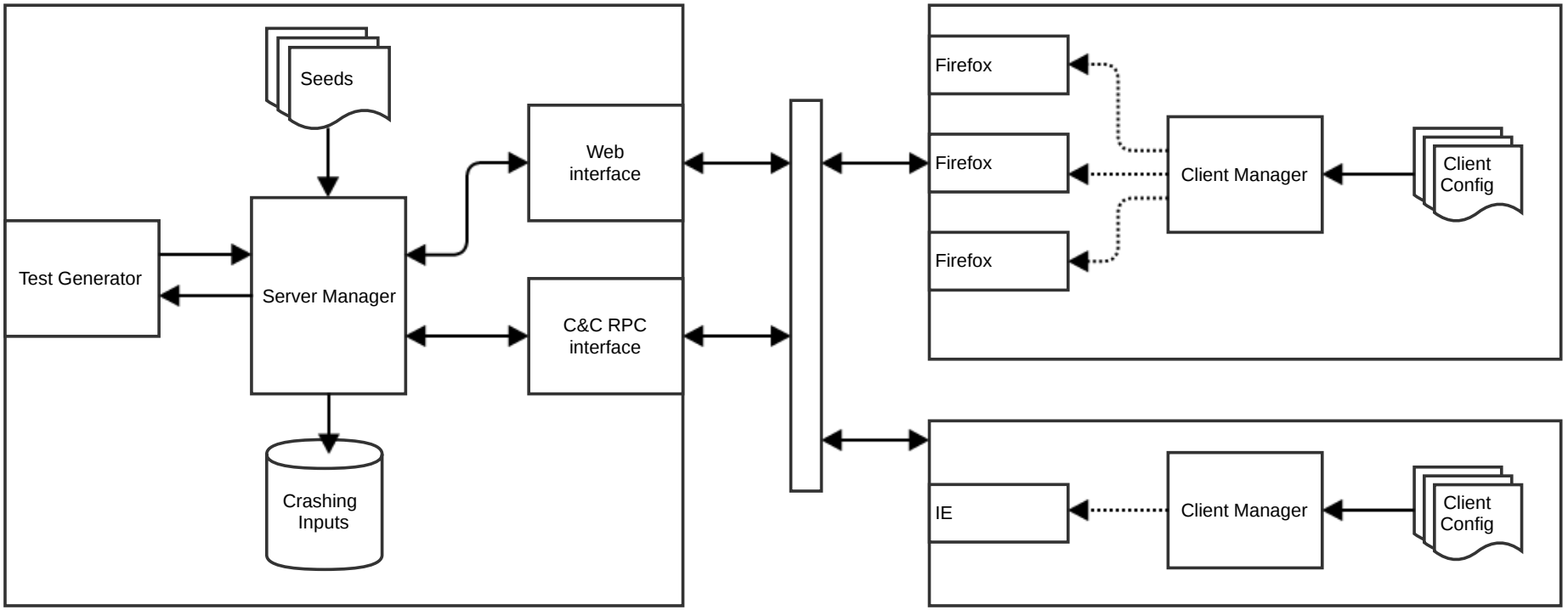
- Where can we get our “green”?
  - Target
    - Full Firefox/Chrome browsers versus the JS shells
    - IE, Safari, Opera
  - Test case generation
    - Seek out HTML/JS tests not used by Langfuzz and apply Radamsa
    - Build a better test generator that addresses our limitations and makes better use of the information available in tests

Fuzzing Web Browsers

Plan

- Browsers will require a retooling
  - 1 fuzz server, multiple client launchers
  - Server is agnostic to the browsers being run, or the host on which they're running
  - Avoiding the ref_fuzz chaos
    - Client detects a crash, can request all files served to that browser
    - Automated triage that can reduce these N files to those producing the crash

Architecture



Architecture

- As before
  - Swapping target just requires a new config file
  - Changing mutation strategy just requires a new test generator
- The “Client Manager” component contains the crash detection logic
  - Can make use of ASan, Pageheap, exit codes, etc
  - Doesn't support a local debugger yet, but should for sanity
- RPC interface to manage test session
  - Notify server of new browser/Tell server that a browser has finished
  - Push testing results from client to server e.g. ASan logs

Test Consumption

- Prior to starting, the client manager makes an RPC call telling the server to expect a new browser, and how many tests to serve it
- A new browser is then spawned which is served a HTML file and Javascript driver
  - The driver requests new tests from the server and renders each one in an IFrame or new window
  - Each test request contains that browser instance' UUID

Test Generation

Further Firefox Tests

- Crashtests
  - ```
    find . -wholename "*crashtests*.html" \
         -o -wholename "*crashtests/*.xhtml" \
         -o -wholename "*crashtests/*.svg"
    ```
  - Typically standalone
    - Don't src any other Javascript or reference any other resources
  - Approx. 2500 in total
  - Approx. 1500 contain Javascript
    - The remaining are mostly plain HTML and were ignored
    ```

Further Firefox Tests

- Mochitest
  - Uses mochitest.ini files to describe tests
  - Approx. 6k in total
    - Rough estimate

```
for f in `find . -name "mochitest.ini" -type
f`; do grep -E "*.[x|h]tml" $f; done | wc -l
```

  - Depend on some fairly standard support files provided by the Mochitest library

Anything Else?

- There are other HTML/JS files in the Firefox repository
- Some are tests, some aren't
- Might be usable
  - All we care about is whether we can provide all of the required dependencies for them to run correctly
  - Don't actually need to be “tests”
  - Anything that provides valid HTML + Javascript API usage will do

Test Generation Approaches

- Regression tests + ref_fuzz (FragFuzz)
  - 'Learning' object creation patterns, followed by embedding them in the standard ref_fuzz algorithm
- Radamsa + single seed
  - Mutations of the seed based purely on data from that seed
- Radamsa + multiple seeds
  - Relying on Radamsa's native ability to combine information from multiple seeds

FragFuzz

- Two stages
  - First we “learn”, from the tests, how to construct interesting objects
  - Second, we run the standard ref_fuzz algorithm, but using the above objects instead of the default DOM objects that ref_fuzz uses

Fragment Extraction

- Input : A JS or HTML file
- Output : Self-contained JS functions (and any HTML element dependencies) that each instantiate a single object found in the input

Implementation

- Straightforward intra-procedural dataflow analysis algorithm based on use-def chains
  - Recursively discover the program statements required to create a variable based on the statements required to create its parents, and so on
- A couple of thousand lines of Go, built from scratch
  - Javascript grammar + parser
  - AST construction
  - Dataflow analysis on top of that

Fragment Extraction : Input

```
<html>
<body>
<canvas id="c" width="500" height="1000"></canvas>
<script>
var canvas = document.getElementById('c');
var gl = canvas.getContext("experimental-webgl");

gl.texImage2D(0, 0, 0, 0, 0, { width: 10, height: 10,
    data: 7 });

</script>
</body>
</html>
```

Fragment Extraction : Output

```
function get_obj0() {
    var canvas = document.getElementById('c');
    return canvas;
}

function get_obj1() {
    var canvas = document.getElementById('c');
    var gl = canvas.getContext("experimental-webgl");
    return gl;
}

-----------------------------------------------------

<canvas id="c" width="500" height="1000"></canvas>
```

Fragment Extraction : Input

```
var StructType = TypedObject.StructType;
var uint8 = TypedObject.uint8;

function runTests() {
    …

    var Color = new StructType({'r': uint8, 'g': uint8,
        'b': uint8});
    var Rainbow = Color.array(7);

    …

    var x = new Rainbow([{'r': 0, 'g': 0, 'b': 0}, … ]);

    …
}
```

Fragment Extraction : Output

```
function get_obj2() {
    var StructType = TypedObject.StructType;
    var uint8 = TypedObject.uint8;
    var Color = new StructType({'r': uint8, 'g': uint8,
                    'b': uint8});
    var Rainbow = Color.array(7);
    var x = new Rainbow([{'r': 0, 'g': 0, 'b': 0}, … ]);
    return x;
}
```

(As well as fragments for Color and Rainbow)

Fragment Usage (Modified ref_fuzz)

```
var state = 0;
var round = 0;
var curr_obj;

function generate_references() {
    return [get_obj0(), get_obj1(), get_obj2(), …];
}

function run_tests() {
    setInterval(
        'try {next_step() } catch (e) {location.reload();}',
        1
    );
}
```

Fragment Usage (Modified ref_fuzz)

```
function next_step() {
   round++;
    switch (state) {
    case 0:
        if (refs == undefined || R(RESET_ODDS) == 0) {
             refs = generate_references();
             …
        }
        curr_obj = rand_select(refs);
    case 1:
        /* Crawl references, randomise object state */
    case 2:
        /* Poke stale references */
    case 3:
        /* Discard object, try to force GC */
    }

    state = (state + 1) % 4;
}
```

Reference Crawling

- On the fly discovery of methods and attributes of the objects we've created
  - Call the methods
  - Tweak the attributes
  - Stores any newly created objects alongside our originals
- Objective is to allocate objects and propagate references
- Key point: Runtime introspection allows us to find and use methods that never appear in the regression tests

Bugs! (Early-stage Results)

- Radamsa (single-seed, multi-seed)
  - 48 hours of runtime
  - 1.2 million tests
  - 0 bugs in Firefox and IE, 1-5 in Opera (uncertainty due to the potential for duplicates)
- FragFuzz
  - 48 hours of runtime
  - Approx. 150 million *next_step* iterations
  - Firefox : At least 3 uniques so far
  - IE : At least 2 uniques so far

FragFuzz Summary

- Advantages
  - Over ref_fuzz : Learned fragments allow us access to any object we can find instantiated in the tests
  - Over LangFuzz : The introspection phases discover methods and attributes never referenced in the regression tests
  - One time preprocessing cost, and fast test generation from then on

FragFuzz Summary

- Disadvantages
  - We have a slightly less chaotic version of the original crash triage problem presented by ref_fuzz
    - Although, because we can have the server automatically prune the reference constructing functions, we can do automatic reduction of the test case
  - Our algorithm doesn't always succeed in producing valid Javascript
    - Missed HTML dependencies
    - Lack of inter-function data-flow tracking

Overall Summary

- Regression tests provide a great source of information on object instantiation methods, API usage patterns, and interesting control flow
- On soft targets, Radamsa's capabilities are sufficient to find bugs based on this information
- Against harder targets, we need to do more work
  - Language specific grammars and further processing to enable things like LangFuzz and FragFuzz
- Approx. 30 days of work in total - the pay-off appears worthwhile

The Future?

- Fragment extraction improvements
  - Could also be used to extract information on function call argument counts/types
  - Currently ignores attributes set and functions called on the objects that make up the dependency chain
- Longer test runs
  - FragFuzz was finished in the past month and the longest consecutive testing period has been 48 hours
- More targets
  - The JS shells, Safari, Chrome

The Future?

- Unnecessary research fail
  - There are probably quite a few bugs to be found by just directly recreating Langfuzz but targeting full browsers, instead of the JS shells
- There's still quite a lot of scope for using regression tests as API pattern providers, in combination with things like ref_fuzz/cross_fuzz
- New tests are added regularly

Questions?

(May also be sent to @seanhn or sean@persistencelabs.com)