symbolically executing a fuzzy tyrant

or, how to fuck literally anything

a tragedy in four symbolic acts of Verdi's Nabucco

dramatis personae

```
[lojikil.com]
Stefan Edwards (lojikil) is not presently logged in.

- Assurance Practice Lead at Trail of Bits
- Twitter/Github/Lobste.rs: lojikil
- Works in: Defense, FinTech, Blockchain, IoT, compilers,
vCISO services, threat modeling
- Previous: net, web, adversary sim, &c.
- Infosec philosopher, programming
language theorist, everyday agronomer, father.
- As heard on Absolute AppSec (multiple) and Risky
Business (No. 559).

WARNING: DEAF
WARNING: Noo Yawk
```

overture

our traged ies:

1. prologos (Jerusalem)
2. the traditional kingdoms (The Impious Ones)
    i. what are they & how do they work
    ii. coverage?
3. a fuzzy tyrant (The Prophecy)
    i. of fuzzing and traditional testing
    ii. understanding property coverage
4. his symbolic execution (The Broken Idol)
    i. program space & analysis
    ii. concolic and symbolic

prologos: Jerusalem

this talk covers three main items:

1. how can we "do better" than traditional tooling?
2. what does this look like?
3. can we make "formal" tools more accessible?

prologos: Jerusalem (*or, what the actual fuck, loji?*)

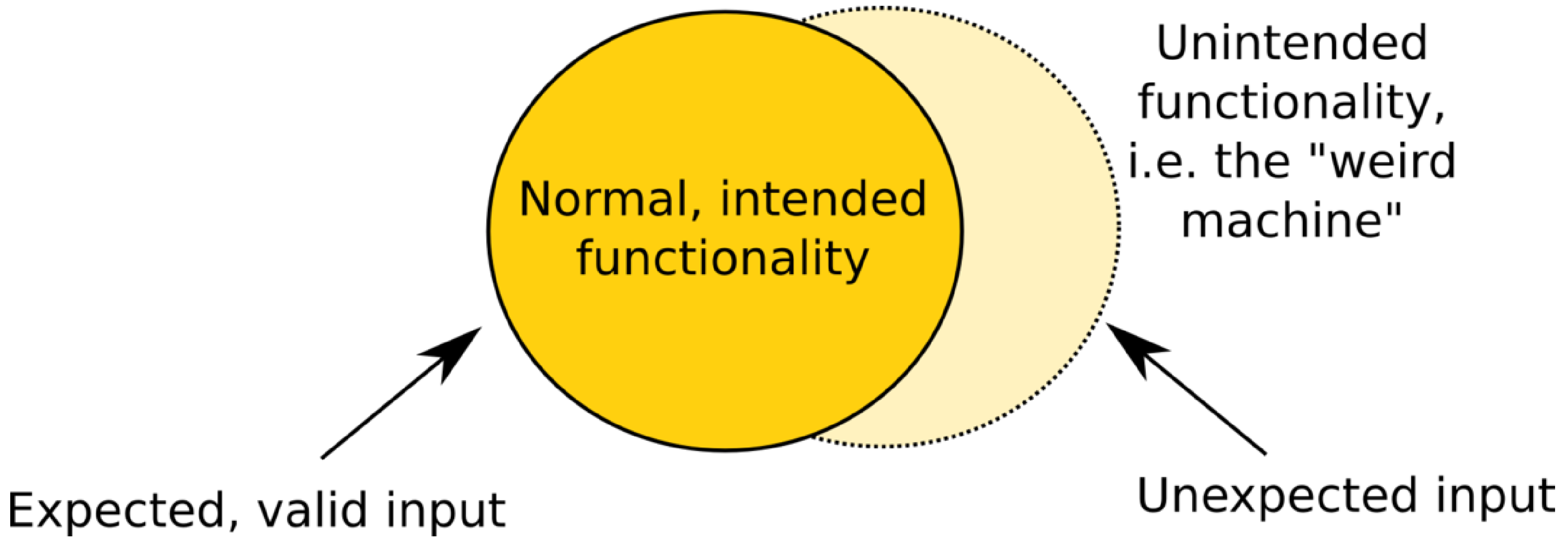
three main take aways:

1. traditional tools have a traditional place
2. formal verification techniques **are accessible for *everyone***
3. a rough intro to program analysis

prologos: Jerusalem

- program analysis?
  - programs have a "space"
  - intended actions vs unintended
  - many techniques to discover
- effectively: formalized & detailed debugging

prologos: Jerusalem

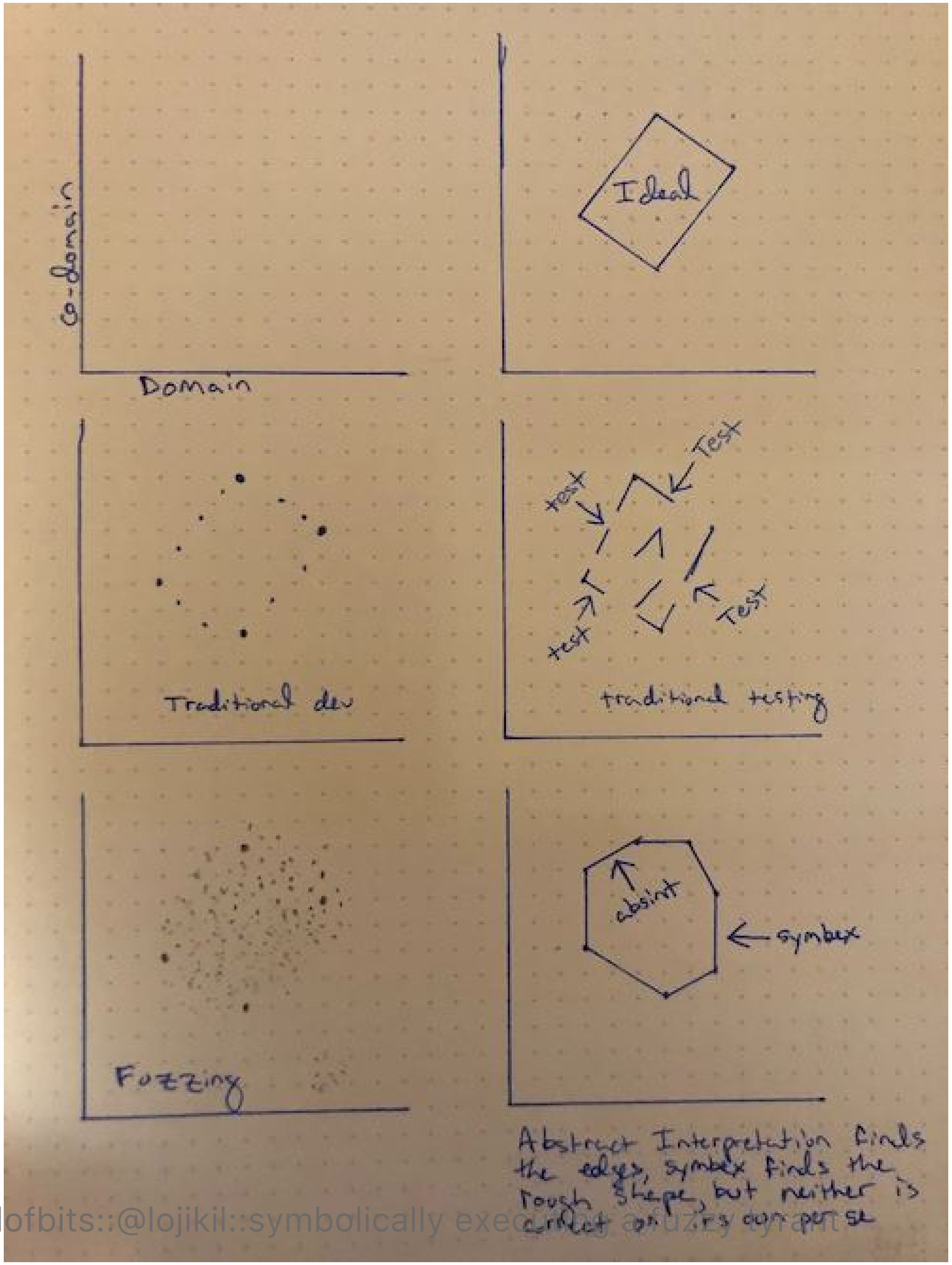


source: Weird Machines

prologos: Jerusalem

- many, many types of "weird machines"
  - using `MOV` as a OISC on x86
  - Python's `pickle`
  - ROP gadgets

prologos: Jerusalem



prologos: Jerusalem

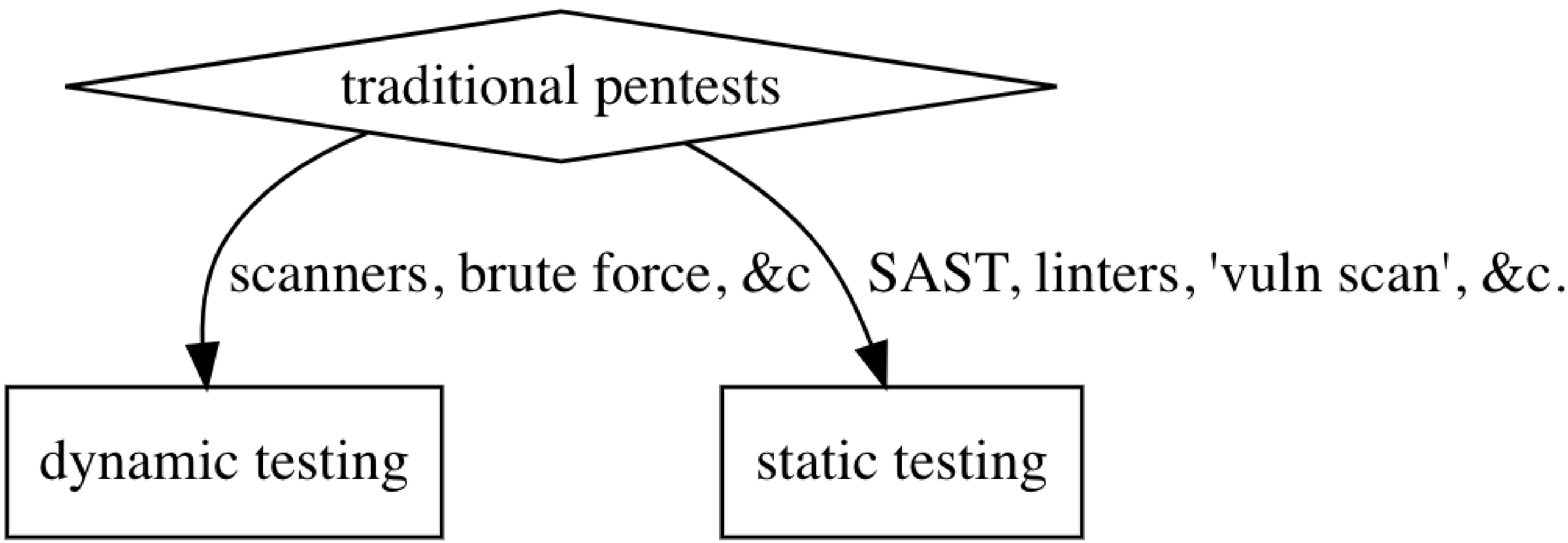
- more than anything: this talk is about understanding code
    - Malware
    - White/Clear box testing
    - Stolen/RE'd code

act 1: traditional testing

scene 1: Traditional infosec testing techniques and their forebearance upon our understanding of systems

sennet: Enter: *certain* traditional tools

a1s1: our traditional dichotomy



a1s1: what are they

- static: linters, code formatters, unsafe function checkers,...
- dynamic: runners, sandboxes, various execution environments...
- basically: the most simple sorts of tests possible
- low barrier to entry, low quality of bugs caught

a1s1: example code

```c
int
main(void) {
     char *foo = nil, bar[64] = {0};

     foo = malloc(sizeof(char) * 128);

     if(!foo) {
        printf("foo is nil\n");
     }

     foo = gets(foo);

     strcpy(foo, bar);

     printf("%s\n", bar);

     free(foo);
     return 0;
}
```

a1s1: `rats`

```
% rats splintex.c
Entries in perl database: 33
Entries in ruby database: 46
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing splintex.c
splintex.c:16: High: gets
Gets is unsafe!! No bounds checking is performed, buffer
      is easily overflowable by user. Use fgets(buf, size, stdin) instead.

splintex.c:18: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 25
Total time 0.002147 seconds
11644 lines per second
```

a1s1: `rats`

- we get two hits: `gets` and `strcpy`
- `fgets` rec is good
- `strcpy` rec ... not as much
- about as simple as we can get
  - code in
  - list of findings out

```
% splint splintex.c
Splint 3.1.2 --- 13 Sep 2018

splintex.c: (in function main)
splintex.c:9:34: Initializer block for bar has 1 element, but declared as char
                    [64]: 0
  Initializer does not define all elements of a declared array. (Use
  -initallelements to inhibit warning)
splintex.c:9:35: Initial value of bar[0] is type int, expects char: 0
  Types are incompatible. (Use -type to inhibit warning)
splintex.c:16:12: Use of gets leads to a buffer overflow vulnerability.  Use
                     fgets instead: gets
  Use of function that may lead to buffer overflow. (Use -bufferoverflowhigh to
  inhibit warning)
splintex.c:16:17: Possibly null storage foo passed as non-null param:
                     gets (foo)
```

a1s1: `splint`

- better: we get six hits (FP) initializer x 2, `gets`, NPE, potential memory leak
- but `strcpy` tho?
- still, p simple:
    - code in
    - list of findings

a1s1: issues

- lots of FPs
- easily fooled (ever seen `nopmd` in Java code?)
- completely misses intent:
  - `strcpy(foo, bar)` is wrong

```
char *
strcpy(char * dst, const char * src);
```

- same for naive dynamic testing: easily fooled

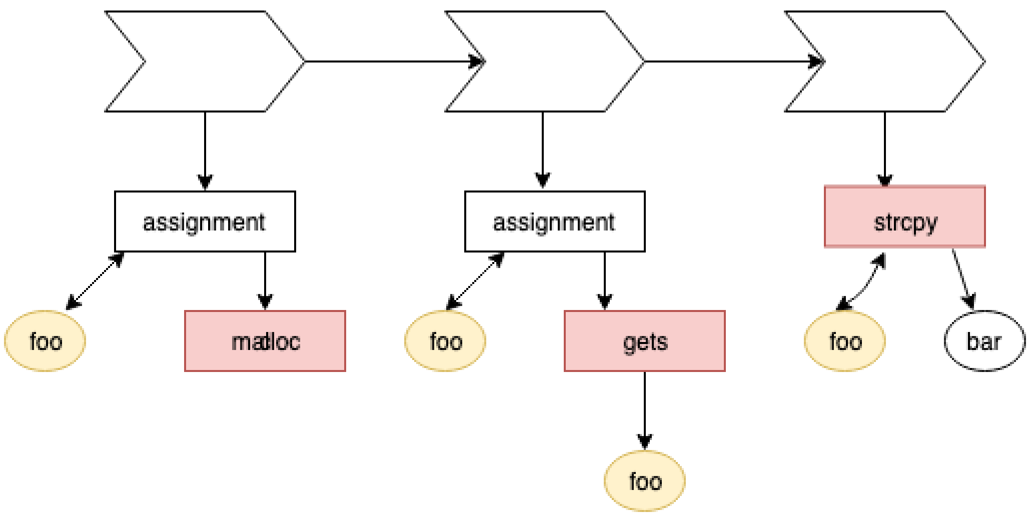
a1s2: how they work

- this all goes back to how these tools work
- very simple models for code

```
if(!foo) {
    printf("foo is nil\n");
}
foo = gets(foo);

strcpy(foo, bar);
```

a1s2: how they work

- Splint builds a more informationally-dense model of code



a1s2: how they work => coverage

- the model of a thing impacts what we can test

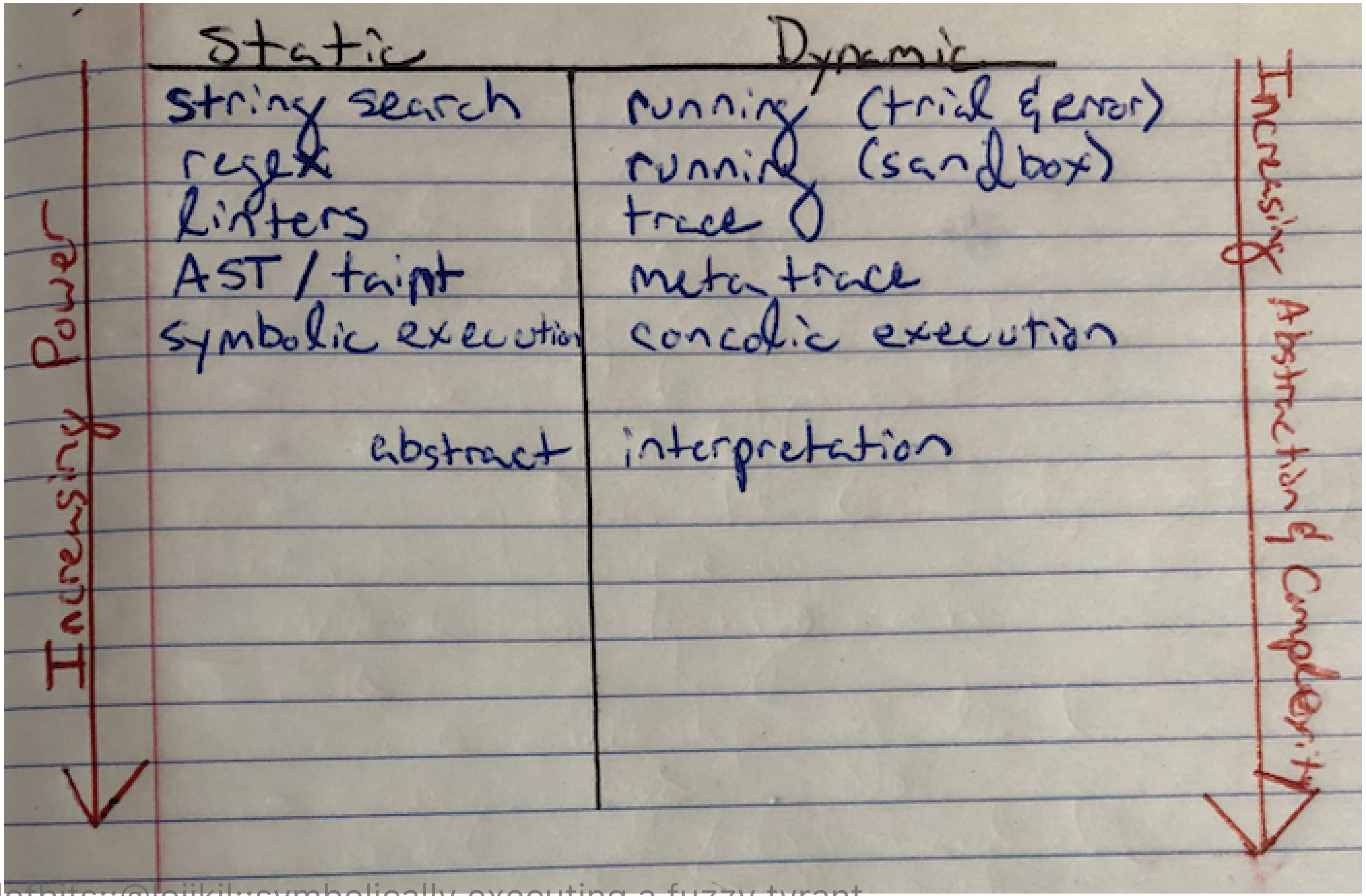
```
int
main(void) {
    char buf[7] = "\0\0\0\0\0\0", foo[7] = "GrrCon";

    strcpy(buf, foo);

    printf("%s\n", buf);

    return 0;
}
```

a1s3: coverage

```
% splint foo.c
Splint 3.1.2 --- 13 Sep 2018

Finished checking --- no warnings
% rats foo.c
Entries in perl database: 33
Entries in ruby database: 46
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing foo.c
foo.c:3: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

foo.c:5: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 11
Total time 0.001117 seconds
9847 lines per second
```

a1s3: coverage (or, why do I care?)

- as {program, malware, ...} analysts, we need to model our code
- adversaries will have decent understanding of their intent
- ... which we must discover



act 2: a fuzzy tyrant

scene 1: On the differences between what is oft referred to as fuzzing and what we mean by fuzzing

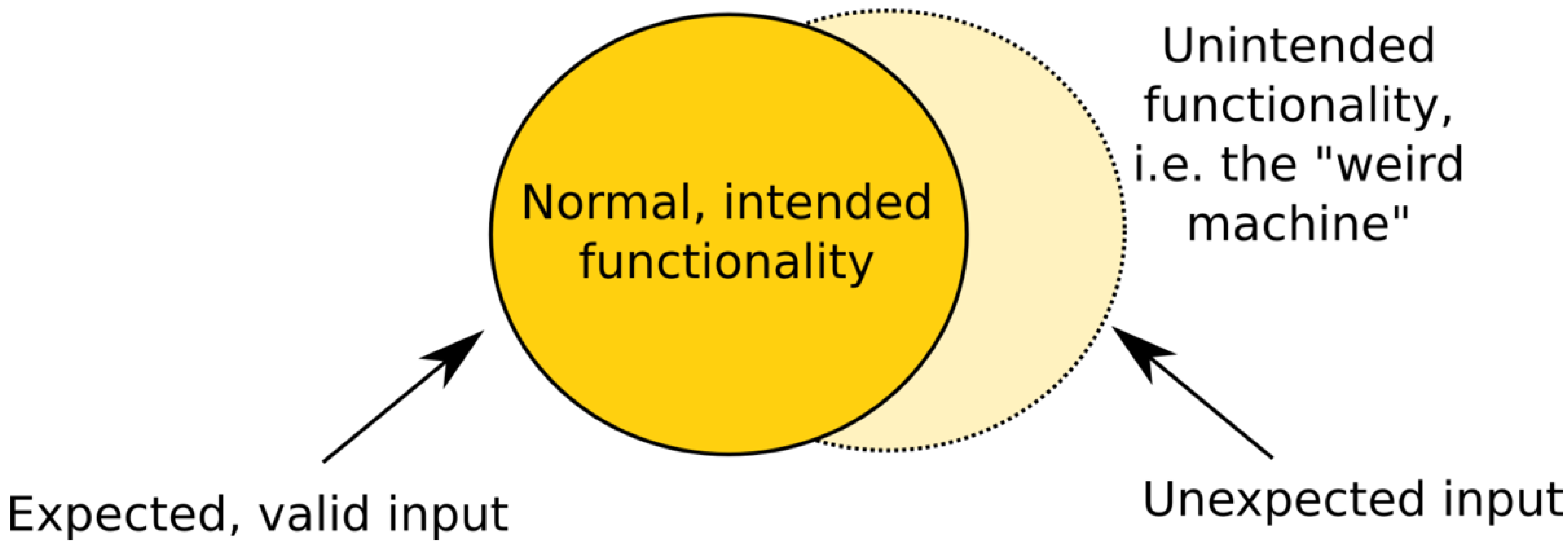
sennet: Enter: *modern* fuzzy tyrants

a2s1: on fuzzing

- traditionally: throwing random data at an app
- what we mean: random mutation testing, property testing
  - can be mutated at the string level
  - can be highly structured

a2s1: on fuzzing

- def in the toolbox: fuzzdb, SecLists, IntruderPayloads...
- missing: mutation of program state
- remember our weird machines



a2s1: on fuzzing

- what we expect: `string`
- naive: random bytes
- mutation: accept valid data, and output N variants
- grammar: accept a definition of data, generate random data
- property testing: define functions, mutate data
- perhaps with instrumentation into program state

a2s1: on fuzzing

- the goal: greater depth of coverage
- beyond what humans can see
- results speak for themselves:
  - personally, 50+ significant bugs from Radamsa in 2 years
  - afl has a repo, with at least 332 CVEs listed
- clearly random testing finds serious issues
- ... but...

a2s2: a fuzzy notion of coverage

- what do we get coverage wise?
- we generate data and watch program result
- want: program to walk other paths
- get: *deeply shrugging man emoji*

a2s2: a fuzzy notion of coverage

- different ways to increase coverage:
  - reach into the binary/system (afl)
  - deeply specify program invariants (property testing)
  - newer techniques, such as grey-box fuzzing
- discover new territory within a program

a2s2: a fuzzy notion of coverage

- fundamental point: we need to uncover paths
- programs themselves are just graphs
  - constrained by conditions
  - constrained by input
- can we discover & graph all paths?

act 3: his symbolic execution

scene 1: my dear, the depths of your program space run far and wide, let me explore the paths and constraints of your heart as a symbol of our love

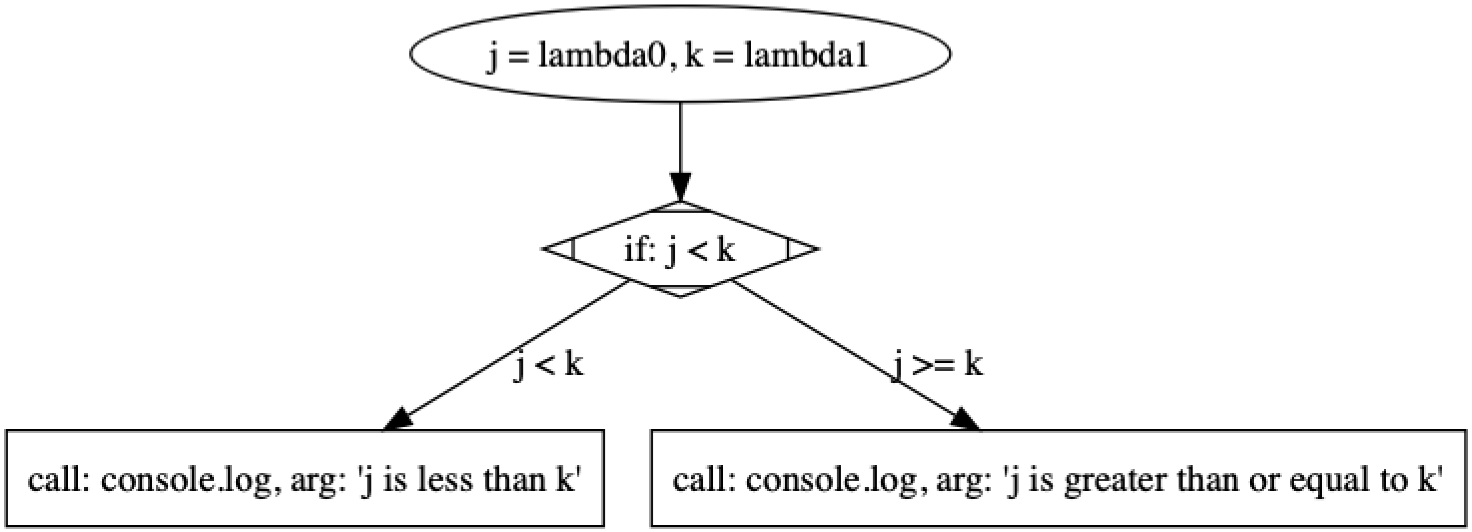
sennet: Enter: a constrained guillotine

a3s1: program space

- at their hearts, programs are just graphs:
  - nodes represent actions
  - edges represent constraints

```
if(j < k) {
    console.log("j is less than k");
} else {
    console.log("j is greater  than or equal to k");
}
```

a3s1: program space



a3s1: program space

- symbolic execution (and related techniques) provide us these graphs
- generate graphs & constraints, then solve them
    - by various means
- **extremely** useful for security
    - KLEE, Manticore, Mythril, &c

a3s1: program space

- the problem: work on binary code
- as malware analysts, we may not always have binary
    - VBA/VBScript, JScript, JavaScript, PowerShell
    - esp. useful for uncovering hosts, second stage, &c
    - most solutions are fancy sandboxes
    - require complete code for execution

a3s1: program space

- decided to fix that: github.com/lojikil/uspno.9
  - "Unnamed Symbex Project No. 9"
- focus on HLLs
  - primarily JS & VBScript
- works on partial code
- safe by default
- **very** new: began life 26 SEP 2019
- Basically: an ugly Scheme-dialect + Python Library

a3s2: concolic & symbolic

- concolic execution: execution with specific (concrete) values
- symbolic execution:

a3s2: concolic & symbolic

- we want to map program space
  - tags (UUIDs) show unique locations
  - traces show values + tags that created data

```
% python
Python 2.7.16 (default, Apr 29 2019, 10:26:08)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from uspno9 import ValueAST
>>> f = ValueAST.new_integer(10)
>>> g = ValueAST.new_integer(11)
>>> h = f + g
>>> h.trace
['(10 tag: 4f7b70b8-7329-4291-b804-ef95697480a8)', '+', '(11 tag: beb4eca0-0508-4e27-a9b7-5dc03ab1bc04)']
>>> h.value
21
>>>
```

a3s2: concolic & symbolic

- but more importantly… *unknown* (symbolic) data

```
% python
Python 2.7.16 (default, Apr 29 2019, 10:26:08)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from uspno9 import ValueAST
>>> f = ValueAST.new_integer(10)
>>> g = ValueAST.new_symbolic_integer()
>>> h = f + g
>>> h.trace
['(10 tag: fdd37c61-93ef-46c7-9759-d34c3f97e3ca)', '+', '(54e74c4d-5c2d-4c4c-85f8-c4560f874cc6)']
>>> h.value
UUID('8d0f6250-a99d-4961-b753-04ae93ea84d8')
>>> g.value
UUID('54e74c4d-5c2d-4c4c-85f8-c4560f874cc6')
>>>
```

a3s2: concolic & symbolic

- but who cares? consider:

```
(if (variable foo ::pure-symbolic)
    (value 12 ::int trace: 12 tag: 91ac...)
    (value 13 ::int trace: 13 tag: e8ab...))
```

- we know nothing about `foo`
- we do know sometimes we get 12, sometimes 13

a3s2: concolic & symbolic

- ask questions

```
>>> test.vm0.microexecute(test.if1)
(PathExecution((value 12 ::int trace: 12 tag: 91ac4b9d-fcd6-4f8e-a98e-5a025d98f0d8), <uspno9.VarRefAST object at 0x10a959c10>), [], <uspno9.EvalE
>>> test.vm1.microexecute(test.if1)
(<uspno9.ForkPathExecution object at 0x10a7d99d0>, [], <uspno9.EvalEnv object at 0x10a96b7d0>)
>>> j = _[0]
>>> j.constraints
[False, True]
>>> [x.to_sexpr() for x in j.asts]
['(value 12 ::int trace: 12 tag: 91ac4b9d-fcd6-4f8e-a98e-5a025d98f0d8)', '(value 13 ::int trace: 13 tag: e8abca4d-1b02-4d46-bc49-0f9a8b9699d0)']
>>>
```

- `PathExecution` gives a value/code under a specific true constraint
- `ForkPathExectution` gives us **both** sides of an execution path

a3s2: concolic & symbolic

- find the constraints underwhich code executes
- **coming soon**: generate reasonable strategies for the same
- execute code both concretely & symbolically
- with both micro-execution & standard execution models

quick break: micro-execution

- given an {env, stack, ...}, execute **one** instruction/form
- helpful for understanding impact of an instruction/form
- https://github.com/lifting-bits/microx
- https://patricegodefroid.github.io/public_psfiles/icse2014.pdf

a3s2: concolic & symbolic

- **lots** to do
    1. flesh out the JS parser
    2. fix ANF & lambda lifting
    3. more tests
    4. more strategies (for generation, &c)
- my use: understanding constraints in gnarly code
- my future use: exercising them

fin

- thanks for coming
- questions?