NYU TANDON SCHOOL OF ENGINEERING

LAVA:
Large-scale Automated Vulnerability Addition

LAVA

Engin Kirda
Andrea Mambretti
Wil Robertson
Northeastern University

Brendan Dolan-Gavitt
NYU Tandon

Patrick Hulin, Tim Leek,
Fredrich Ulrich, Ryan Whelan
MIT Lincoln Laboratory

This Talk

- In this talk, we explore how to ***automatically add large numbers of bugs to programs***
- Why would we want to do this?
  - Computer programs don't have enough bugs
  - We want to put backdoors in other people's programs

This Talk

- In this talk, we explore how to ***automatically add large numbers of bugs to programs***
- Why would we want to do this?
  - ~~Computer programs don't have enough bugs~~
  - We want to put backdoors in other people's programs

This Talk

- In this talk, we explore how to ***automatically add large numbers of bugs to programs***
- Why would we want to do this?
  - ~~Computer programs don't have enough bugs~~
  - ~~We want to put backdoors in other people's programs~~

NYU

Vulnerability Discovery

- Finding vulnerabilities in software automatically has been a major research and industry goal for the last 25 years

Academic

An Empirical Study of the Reliability of UNIX Utilities

Barton P. Miller, bart@cs.wisc.edu
Lars Fredriksen, L.Fredriksen@att.com
Bryan So, so@cs.wisc.edu

Fuzzing (1989)

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

KLEE (2005)

Driller: Augmenting Fuzzing Through Selective Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{stephens,jmg,salls,dutcher,fish,jacopo,yans,chris,vigna}@cs.ucsb.edu

Driller (2015)

Commercial

klocwork — a Rogue Wave Company
∀.Secure
FORTIFY
VERACODE
coverity

NYU

Vulnerability Discovery

- Finding vulnerabilities in software automatically has been a major research and industry goal for the last 25 years

Academic

An Empirical Study of the Reliability of UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu

Fuzzing (1989)

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

KLEE (2005)

Driller: Augmenting Fuzzing Through Selective Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{stephens,jmg,salls,dutcher,fish,jacopo,yans,chris,vigna}@cs.ucsb.edu

Driller (2015)

Commercial

klocwork — a Rogue Wave Company

∀Secure

FORTIFY

VERACODE

coverity®

Does this work??

Debugging the Bug Finders

- Lots of work that claims to find bugs in programs
- **Lack of ground truth** makes it very difficult to evaluate these claims
- If Coverity finds 22 bugs in my program, is that good or bad?
- What are the false positive and **false negative** rates?

Existing Test Corpora

Some existing bug corpora exist, but have many problems:

- Synthetic (small) programs
- Don't always have triggering inputs
- Fixed size – tools can "overfit" to the corpus

What About Real Vulnerabilities?

- Real vulnerabilities with proof-of-concept exploits are essentially what we want
- But there just aren't that many of them. And finding new ones is expensive!

ADOBE READER	$5,000–$30,000
MAC OSX	$20,000–$50,000
ANDROID	$30,000–$60,000
FLASH OR JAVA BROWSER PLUG-INS	$40,000–$100,000
MICROSOFT WORD	$50,000–$100,000
WINDOWS	$60,000–$120,000
FIREFOX OR SAFARI	$60,000–$150,000
CHROME OR INTERNET EXPLORER	$80,000–$200,000
IOS	$100,000–$250,000

Forbes, 2012

NYU

Debugging the Bug Finders

- Existing corpora are fixed size and static – it's easy to optimize to the benchmark
- Instead we would like to **automatically create bug corpora**
- Take an existing program and *automatically* add new bugs into it
- Now we can measure how many of our bugs they find to estimate **effectiveness** of bug-finders

Goals

- We want to produce bugs that are:
  - **Plentiful** (can put 1000s into a program easily)
  - **Distributed** throughout the program
  - Come with a **triggering input**
  - Only manifest for a **tiny fraction of inputs**
  - Are likely to be **security-critical**

Sounds Simple... But Not

- Why not just change all the `strncpys` to `strcpys`?
  - Turns out this breaks most programs for *every* input – trivial to find the bugs
  - We won't know how to trigger the bugs – hard to prove they're "real" and security-relevant
  - This applies to most **local**, random mutations

Our Approach: DUAs

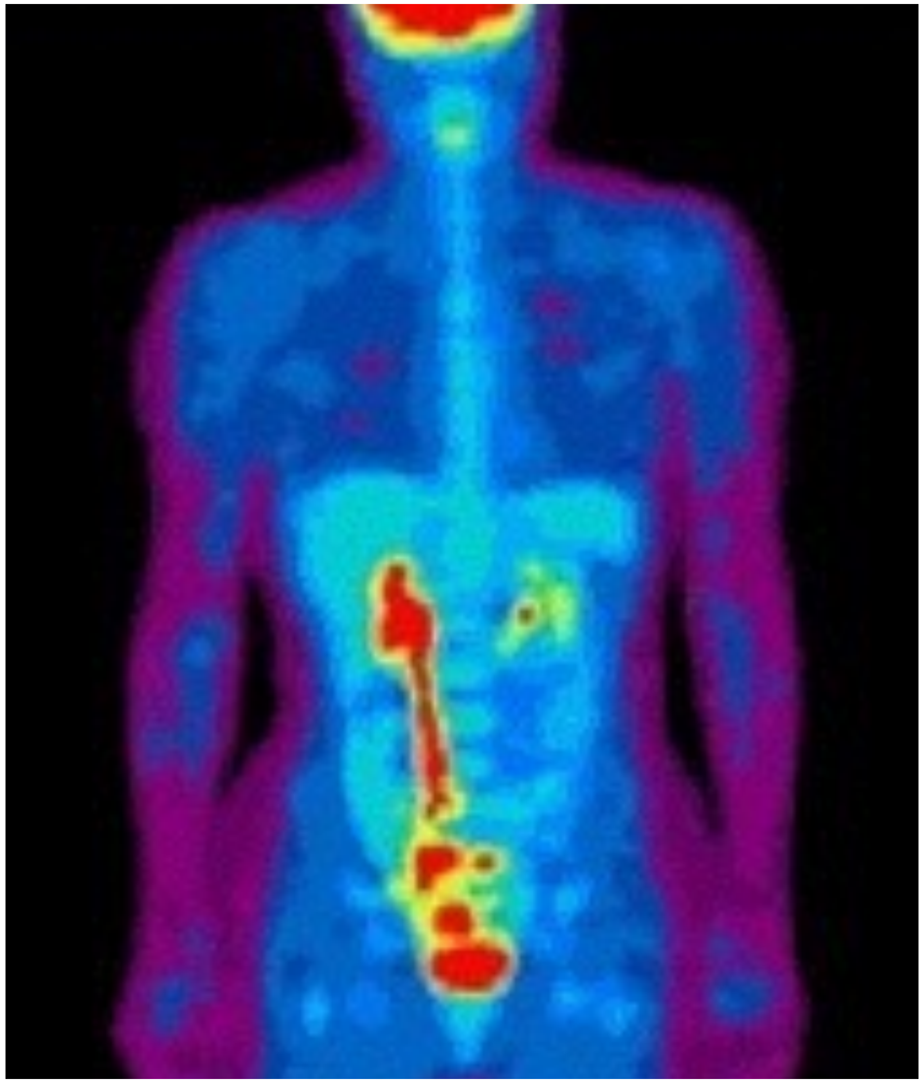
- We want to find parts of the program's input data that are:
  - **Dead:** not currently used much in the program (i.e., we can set to arbitrary values)
  - **Uncomplicated:** not altered very much (i.e., we can predict their value throughout the program's lifetime)
  - **Available** in some program variables
- These properties try to capture the notion of ***attacker-controlled data***
- If we can find these **DUAs**, we will be able to add code to the program that uses such data to trigger a bug

New Taint-Based Measures

- How do we find out what data is **dead** and **uncomplicated**?
- Two new taint-based measures:
  - *Liveness*: a count of how many times some input byte is used to decide a branch
  - *Taint compute number*: a measure of how much computation been done on some data

Dynamic Taint Analysis

- We use ***dynamic taint analysis*** to understand the effect of input data on the program
- Our taint analysis requires some specific features:
  - Large number of labels available
  - Taint tracks *label sets*
  - Whole-system & fast (enough)
- Our open-source dynamic analysis platform, **PANDA**, provides all of these features



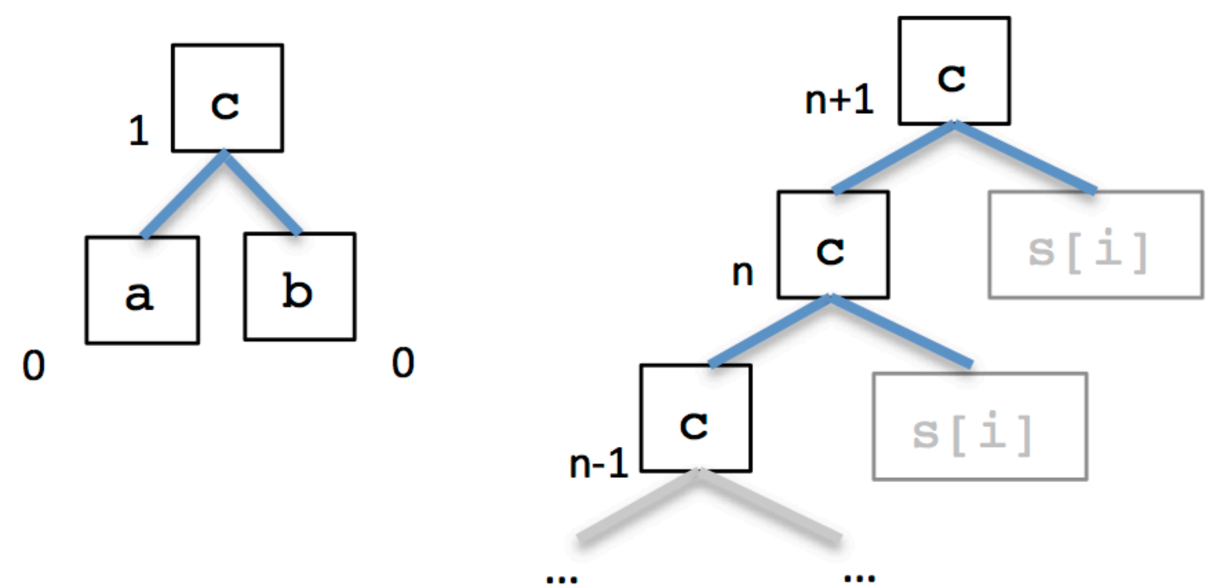
c = a + b ; a: {w,x} ; b: {y,z}

c ← {w,x,y,z}

https://github.com/moyix/panda

Taint Compute Number (TCN)

```
// a,b,n are inputs
int c = a+b;
if (a != 0xdeadbeef)
      return;
for (int i=0; i<n; i++)
      c+=s[i];
```



TCN measures how much computation has been done on a variable at a given point in the program

Liveness

```
   // a,b,n are inputs
1: int c = a+b;
2: if (a != 0xdeadbeef)
3:       return;
4: for (int i=0; i<n; i++)
5:       c+=s[i];
```

b: bytes {0..3}
n: bytes {4..7}
a: bytes {8..11}

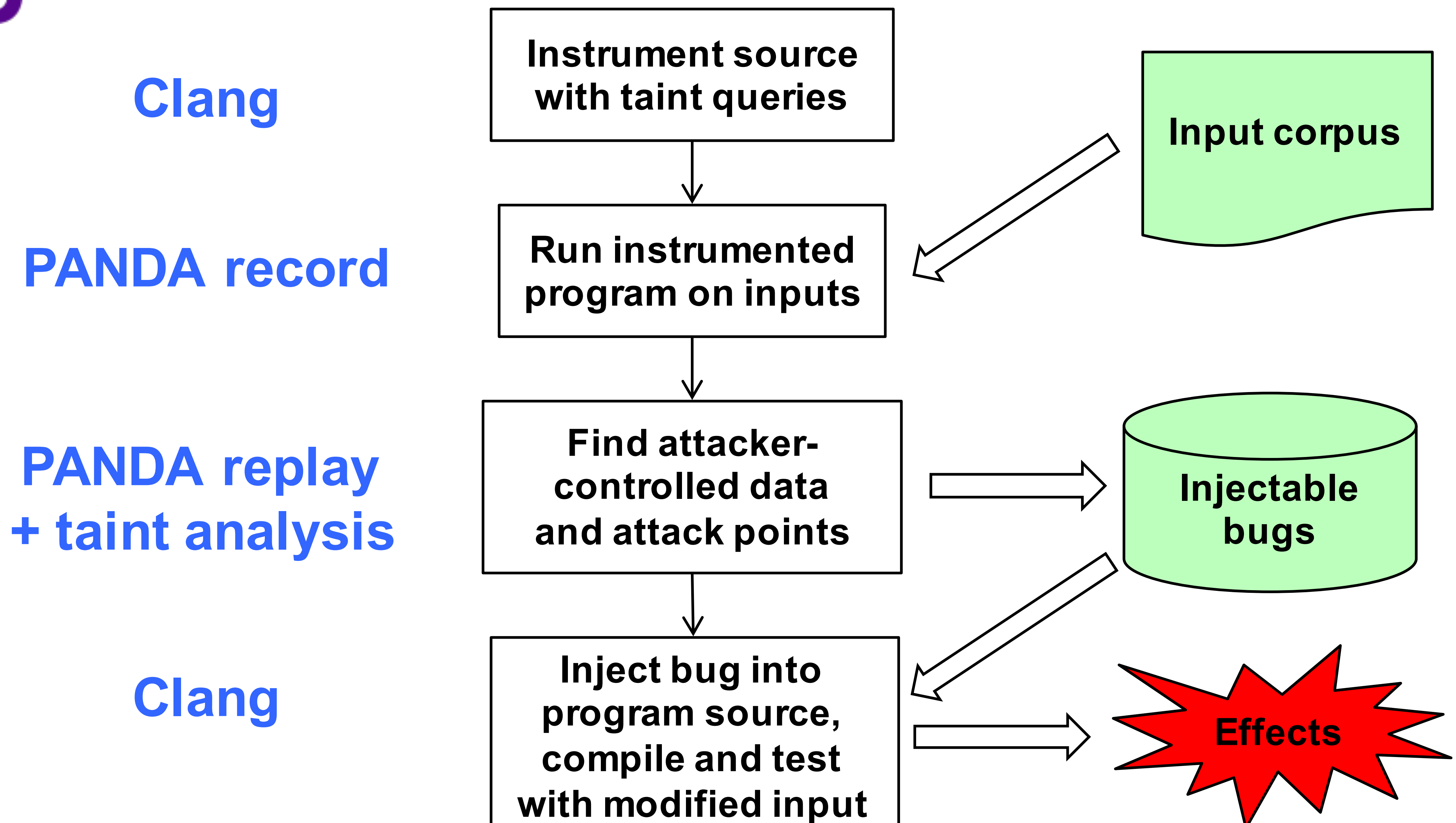
Bytes	Liveness
{0..3}	**0**
{4..7}	**n**
{8..11}	**1**

Liveness measures how many branches use each input byte

Attack Point (ATP)

- An Attack Point (ATP) is any place where we may want to use attacker-controlled data to cause a bug
- Examples: pointer dereference, data copying, memory allocation, ...
- In current LAVA implementation we just modify pointer dereferences to cause buffer overflow

Approach: Overview



LAVA Bugs

- Any (DUA, ATP) pair where the DUA occurs before the attack point is a potential bug we can inject
- By modifying the source to add new data flow the from DUA to the attack point we can create a bug

DUA + ATP = 

NYU

LAVA Bug Example

- PANDA taint analysis shows that bytes 0-3 of `buf` on line 115 of `src/encoding.c` is attacker-controlled (dead & uncomplicated)
- From PANDA we also see that in `readcdf.c` line 365 there is a read from a pointer – if we modify the pointer value we will likely cause a bug in the program

Attacker controlled data

```
encoding.c 115: } else if (looks_extended(buf, nbytes,
                *ubuf, ulen)) {
```

Corruptible pointer

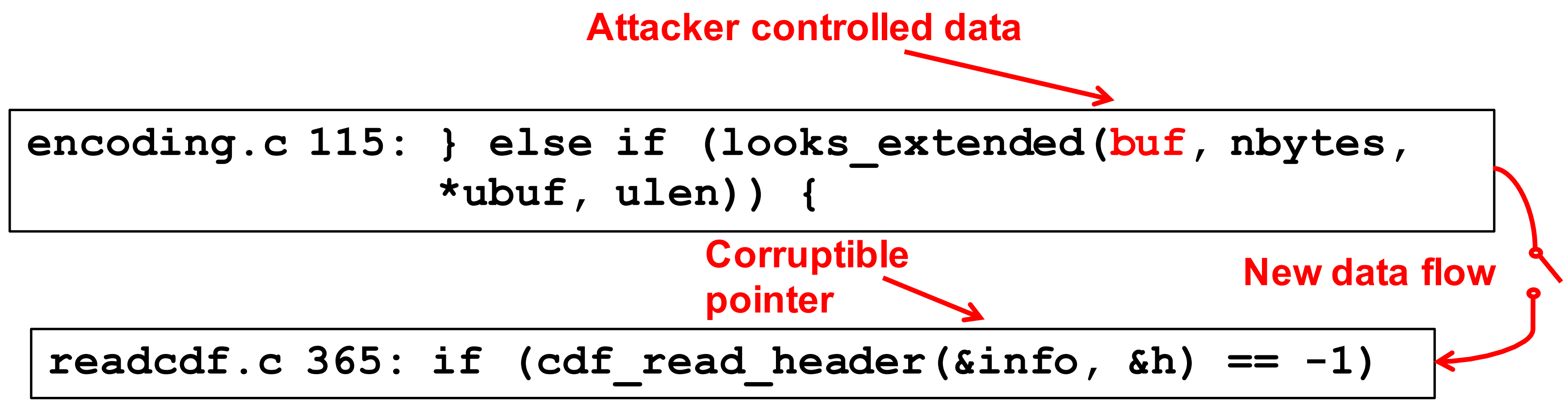
New data flow

```
readcdf.c 365: if (cdf_read_header(&info, &h) == -1)
```

NYU

LAVA Bug Example

- PANDA taint analysis shows that bytes 0-3 of `buf` on line 115 of `src/encoding.c` is attacker-controlled (dead & uncomplicated)
- From PANDA we also see that in `readcdf.c` line 365 there is a read from a pointer – if we modify the pointer value we will likely cause a bug in the program



NYU

LAVA Bug Example

```
// encoding.c:
} else if
  (({int rv =
       looks_extended(buf, nbytes, *ubuf, ulen);
     if (buf) {
       int lava = 0;
       lava |= ((unsigned char *)buf)[0];
       lava |= ((unsigned char *)buf)[1] << 8;
       lava |= ((unsigned char *)buf)[2] << 16;
       lava |= ((unsigned char *)buf)[3] << 24;
       lava_set(lava);
    }; rv; })) {
```

```
// readcdf.c:
if (cdf_read_header
    ((&info) + (lava_get()) *
      (0x6c617661 == (lava_get()) || 0x6176616c == (lava_get())),
       &h) == -1)
```

When the input file data that ends up in buf is set to 0x6c6176c1, we will add 0x6c6176c1 to the pointer info, causing an out of bounds access

Evaluation: How Many Bugs?

Name	Version	Num Src Files	Lines C code	N(DUA)	N(ATP)	Potential Bugs	Validated Bugs	Yield	Inj Time (sec)
file	5.22	19	10809	631	114	17518	774	38.7%	16
readelf	2.25	12	21052	3849	266	276367	1064	53.2 %	354
bash	4.3	143	98871	3832	604	447645	192	9.6%	153
tshark	1.8.2	1272	2186252	9853	1037	1240777	354	17.7%	542

- We ran four open-source programs each on a single input and generated candidate bugs
- Because validating all possible bugs would take too long, we instead validated a random sample of 2000 per program
- **Result**: extrapolating from the yield numbers, a single run gives us up to **~200,000** real bugs

Evaluation: What Influences Yield?

$mTCN$	$mLIV$ $[0, 10)$	$[10, 100)$	$[100, 1000)$	$[1000, +\inf]$
$[0, 10)$	51.9%	22.9%	17.4%	11.9%
$[10, 100)$	–	0	0	0
$[100, +\inf]$	–	–	–	0

- TCN strongly affects yield
  - *No* bugs that involved TCN greater than 10 were useable
- Liveness has a weaker correlation with yield – even fairly live data can be sometimes be used if TCN is low

Evaluation: Can Tools Find Them?

- We took two open-source bug-finding tools and tried to measure their success at finding LAVA bugs
  - A *coverage-guided fuzzer* (FUZZER)
  - A *symbolic execution and constraint solving* tool (SES)
  - (Actual names withheld since this is just a preliminary study)

Results: Specific Value

Program	Total Bugs	Unique Bugs Found		
		FUZZER	SES	Combined
uniq	28	7	0	7
base64	44	7	9	14
md5sum	57	2	0	2
who	2136	0	18	18
Total	2265	16	27	41

Less than 2% of injected bugs found

Results: Range-Triggered Bugs

Tool	Bug Type					
	Range					
	2^0	2^7	2^{14}	2^{21}	2^{28}	KT
FUZZER	0	0	9%	79%	75%	20%
SES	8%	0	9%	21%	0	10%

Evaluation: Realism

- The burning question in everyone's mind now: are these bugs **realistic**?
- This is hard to measure, in part because realism is not a well-defined property!
- Our evaluation looks at:
  - How injected bugs are distributed in the program
  - What proportion of the trace has normal data flow
- Ultimately, the best test of realism will be whether it helps bug-finding software get better

Results: Realism

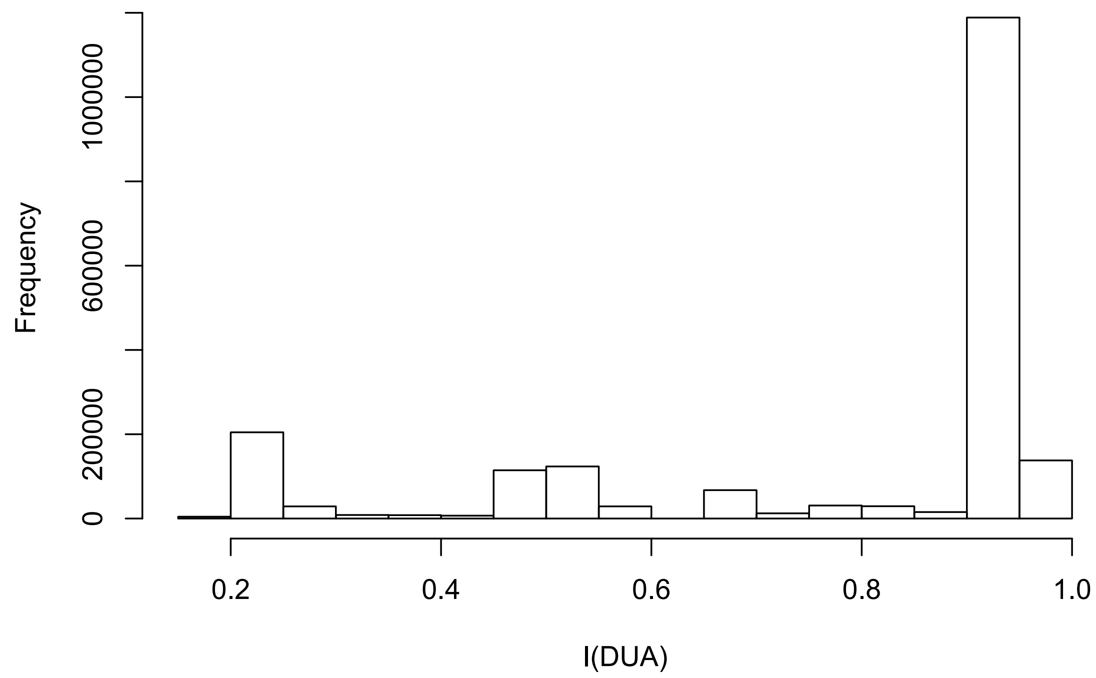


Fig. 10: Normalized DUA trace location

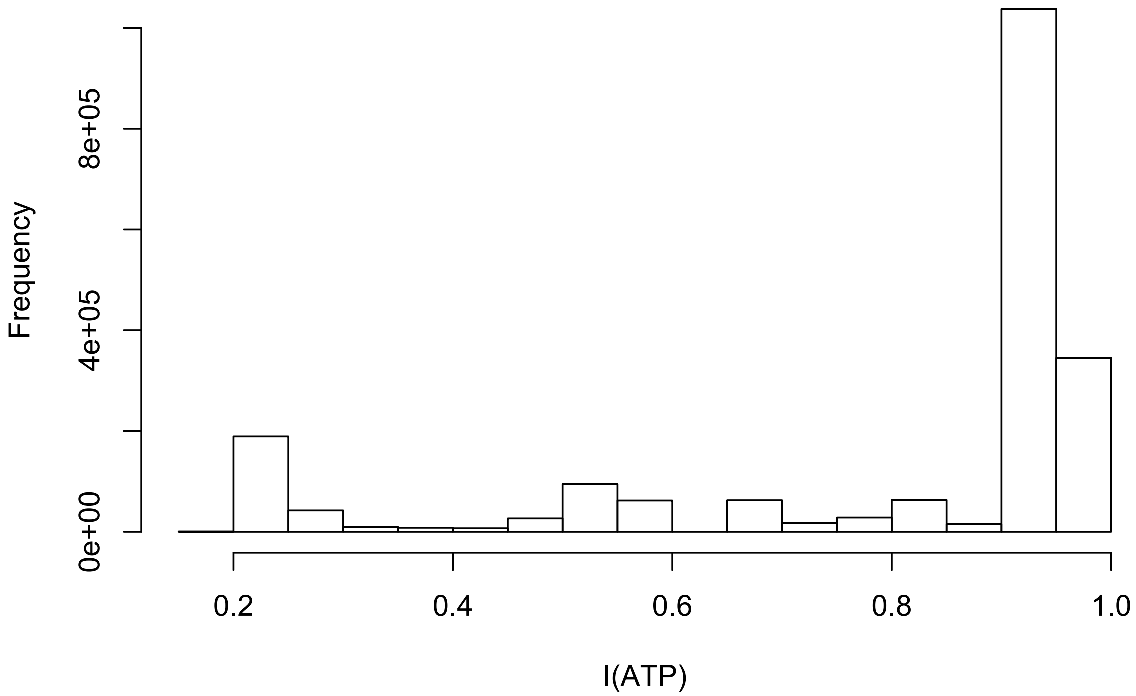
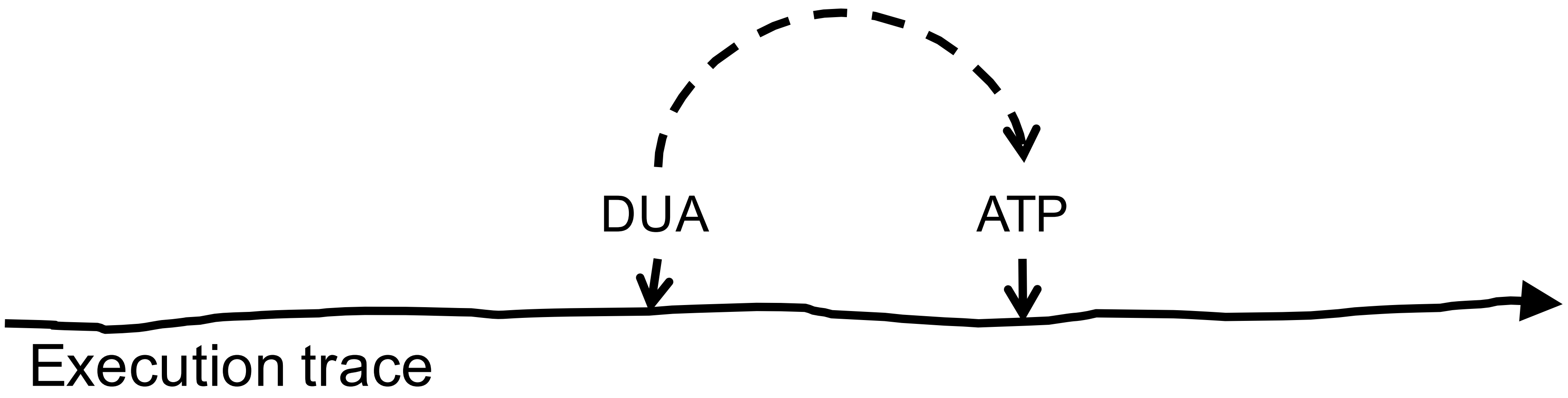


Fig. 11: Normalized ATP trace location



Limitations and Caveats

- General limitations:
  - Some types of vulnerabilities probably can't be injected using this method – e.g., weak crypto bugs
  - More work is needed to see if these bugs can improve bug-finding software
- Implementation limits:
  - Currently only works on C/C++ programs in Linux
  - Only injects buffer overflow bugs
  - Works only on source code

Future Work

- Continuous on-line competition to encourage self-evaluation

- Use in security competitions like Capture the Flag to re-use and construct challenges on-the-fly

- Improve and assess realism of LAVA bugs
- More types of vulnerabilities (use after free, command injection, ...)
- More interesting effects (prove exploitability!)

Conclusions

- Presented a new technique that is capable of **quickly** injecting massive numbers of bugs
- Demonstrated that current tools are not very good at finding these bugs
- If these bugs prove to be good stand-ins for real-world vulnerabilities, we can get **huge, on-demand bug corpora**

NYU

Questions?