Fuzzing Low-Level Code

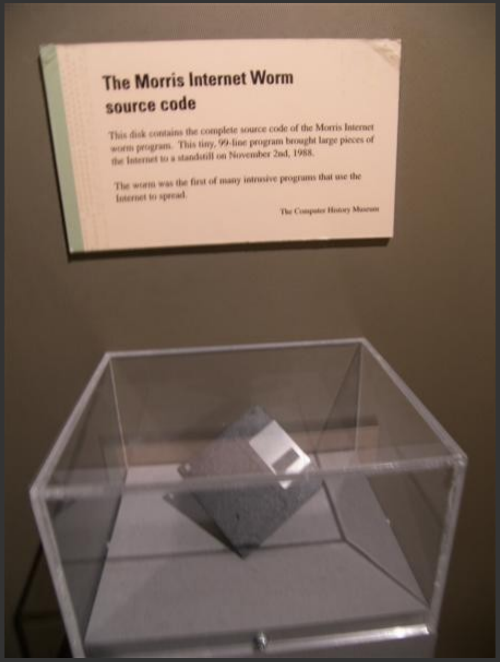
Mathias Payer <mathias.payer@epfl.ch>
https://hexhive.github.io

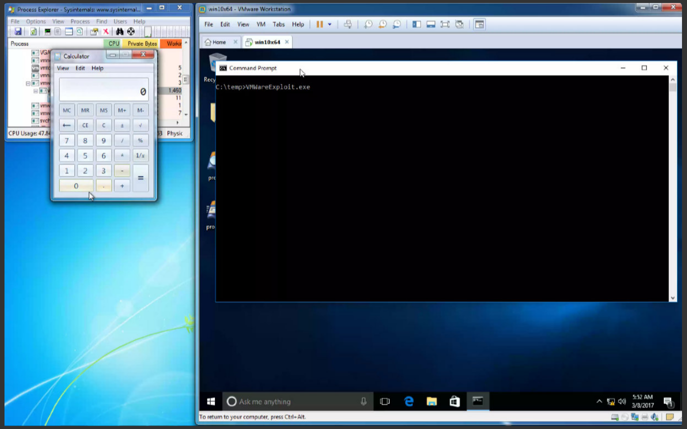
HexHive is hiring!

erc
European Research Council
Established by the European Commission

EPFL

Challenge: vulnerabilities everywhere

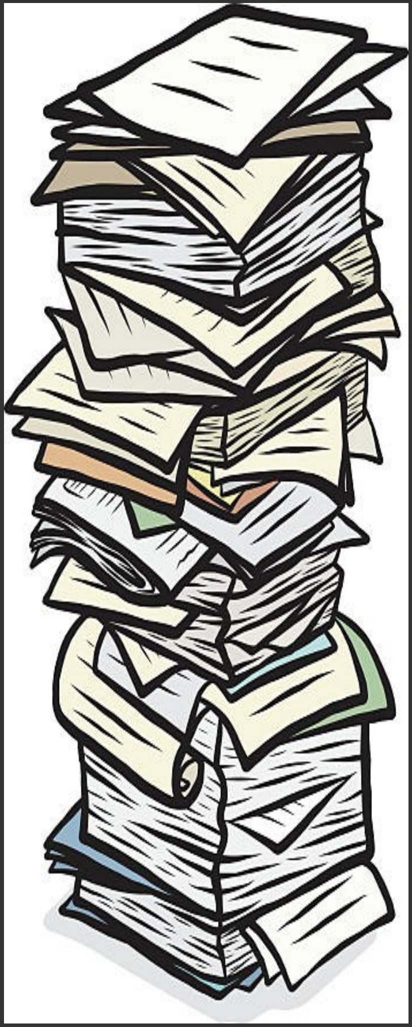




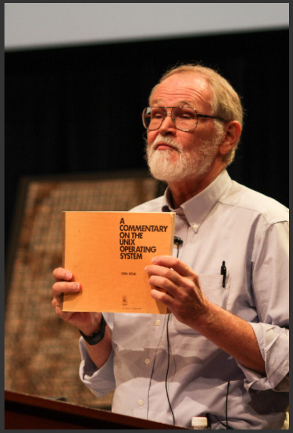
Challenge: software complexity

Google Chrome:	**76 MLoC**
Gnome:	**9 MLoC**
Xorg:	**1 MLoC**
glibc:	**2 MLoC**
Linux kernel:	**17 MLoC**

Chrome and OS
~100 mLoC,
27 lines/page,
0.1mm/page ≈ 370m



Margaret Hamilton with code for Apollo Guidance Computer (NASA, ‘69)



Brian Kernighan holding Lion’s commentary on BSD 6 (Bell Labs, ‘77)

Defense: Testing *OR* Mitigating?

Software Testing

Mitigations

```
vuln("AAA");
```

```
vuln("ABC");
```

```
vuln("AAAABBBB");
```

```
strcpy_chk(buf, 4, str);
```

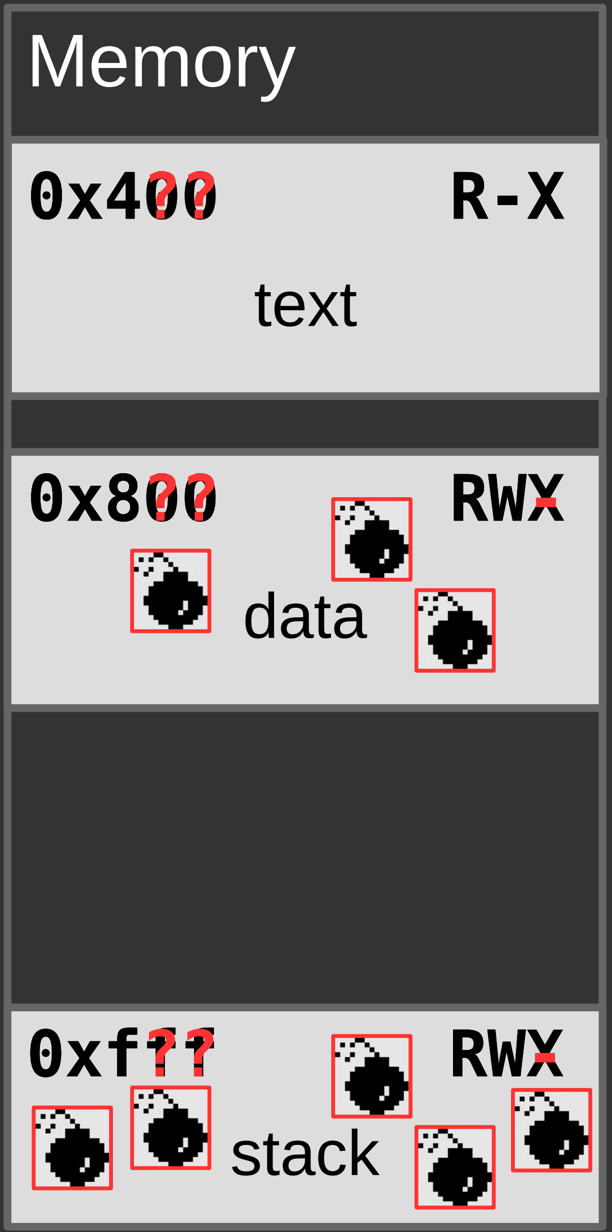
C/C++

```
void log(int a) {
  printf("A: %d", a);
}

void vuln(char *str) {
  char *buf[4];
  void (*fun)(int) = &log;
  strcpy(buf, str);

  fun(15);
}
```

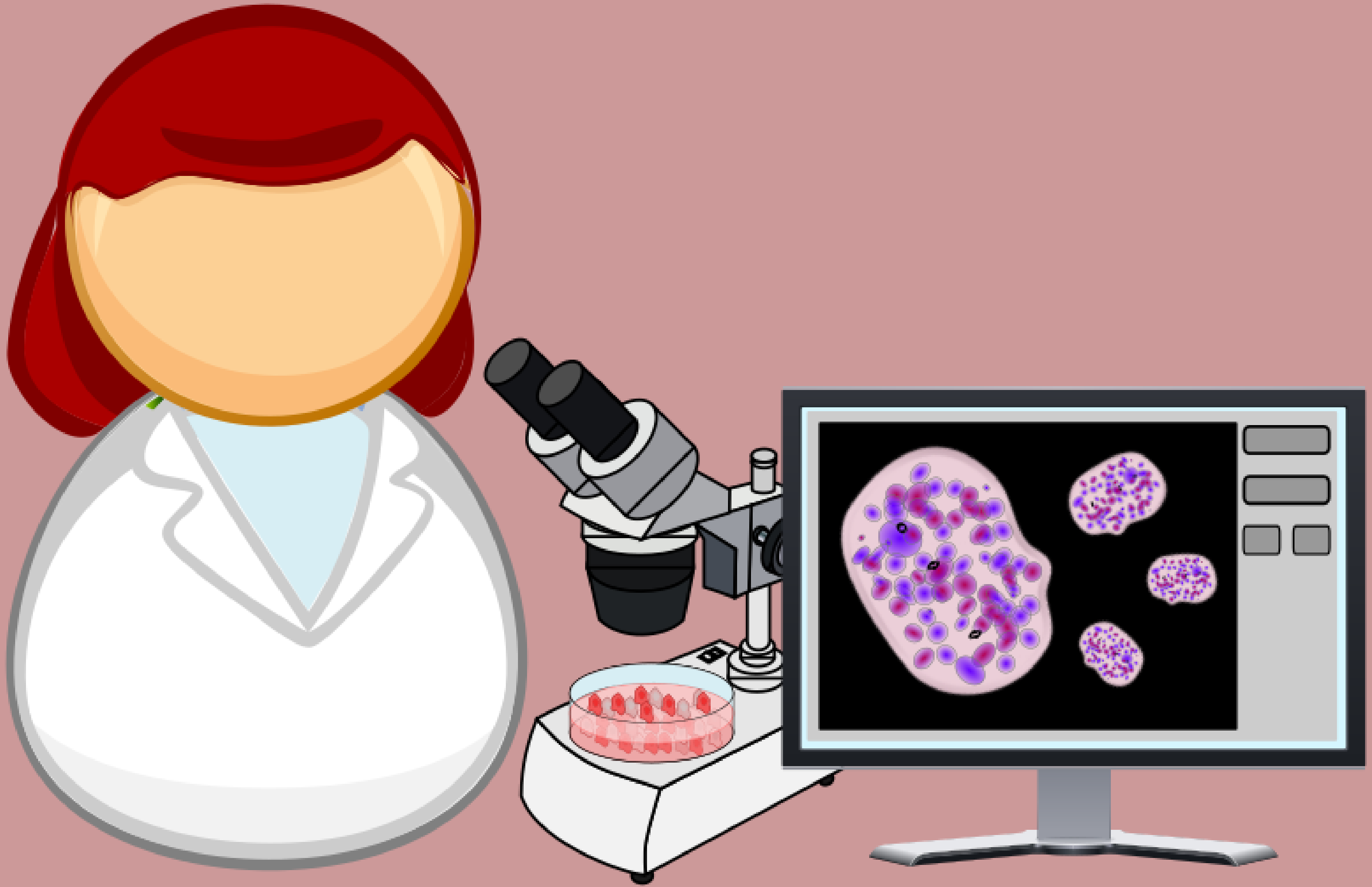
```
CHECK(fun, tgtSet);
```

Status of deployed defenses

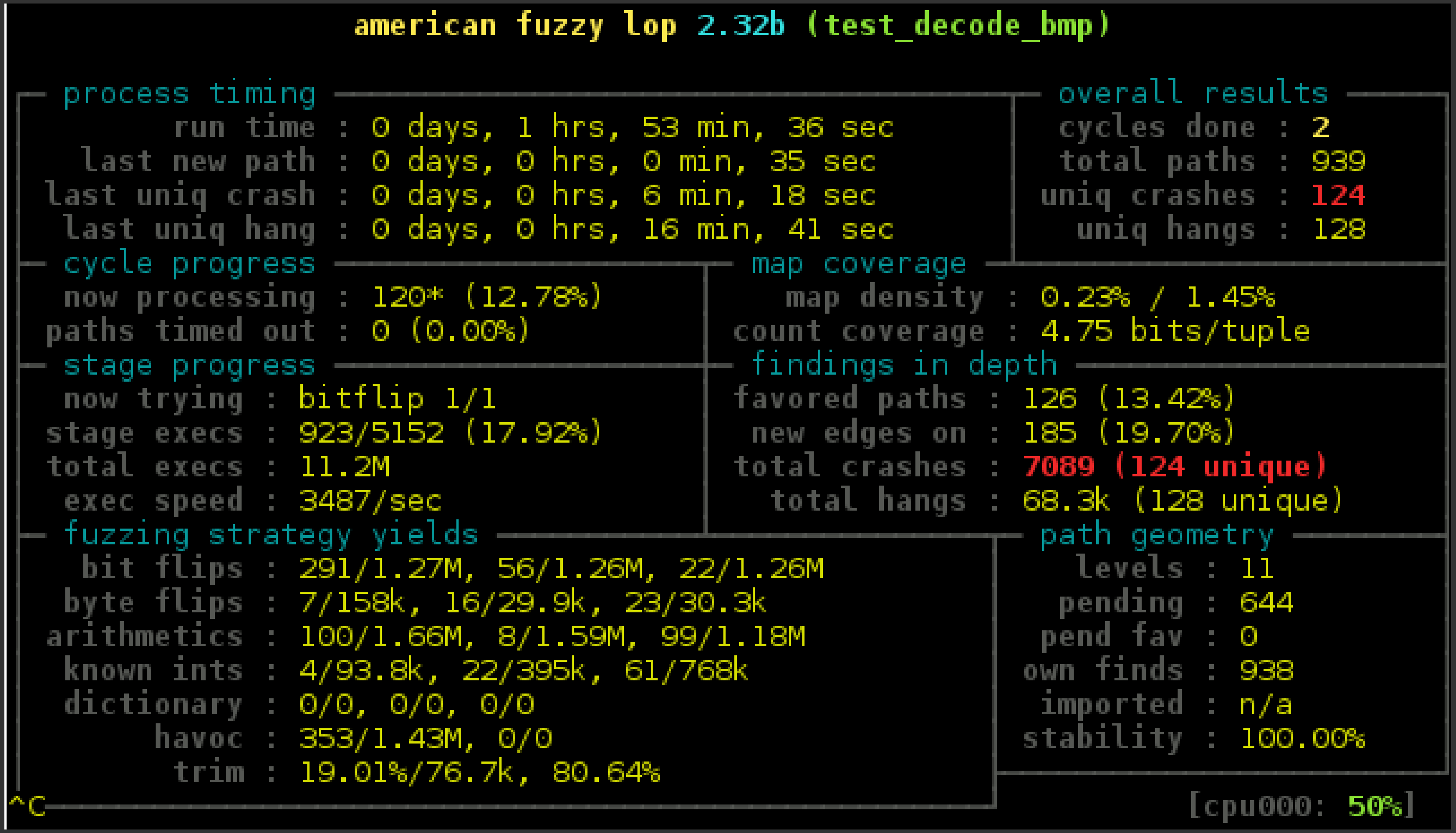
- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack canaries
- Safe exception handlers
- Control-Flow Integrity (CFI): Guard indirect control-flow



Assessing exploitability



Which crash to focus on first?



Residual Attack Surface Probing

- State-of-the-art mitigations complicate attacks
  - Mitigations have limitations but these are hard to assess and explore systematically (and globally)

- Let’s infer the ***Residual Attack Surface***
  - Given a crash/bug what can an adversary still do?
  - Residual attack surface depends on program, environment, and input

Block Oriented Programming: Automating Data-Only Attacks
Kyriakos Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer.
In CCS'18: ACM Conference on Computer and Communication Security, 2018

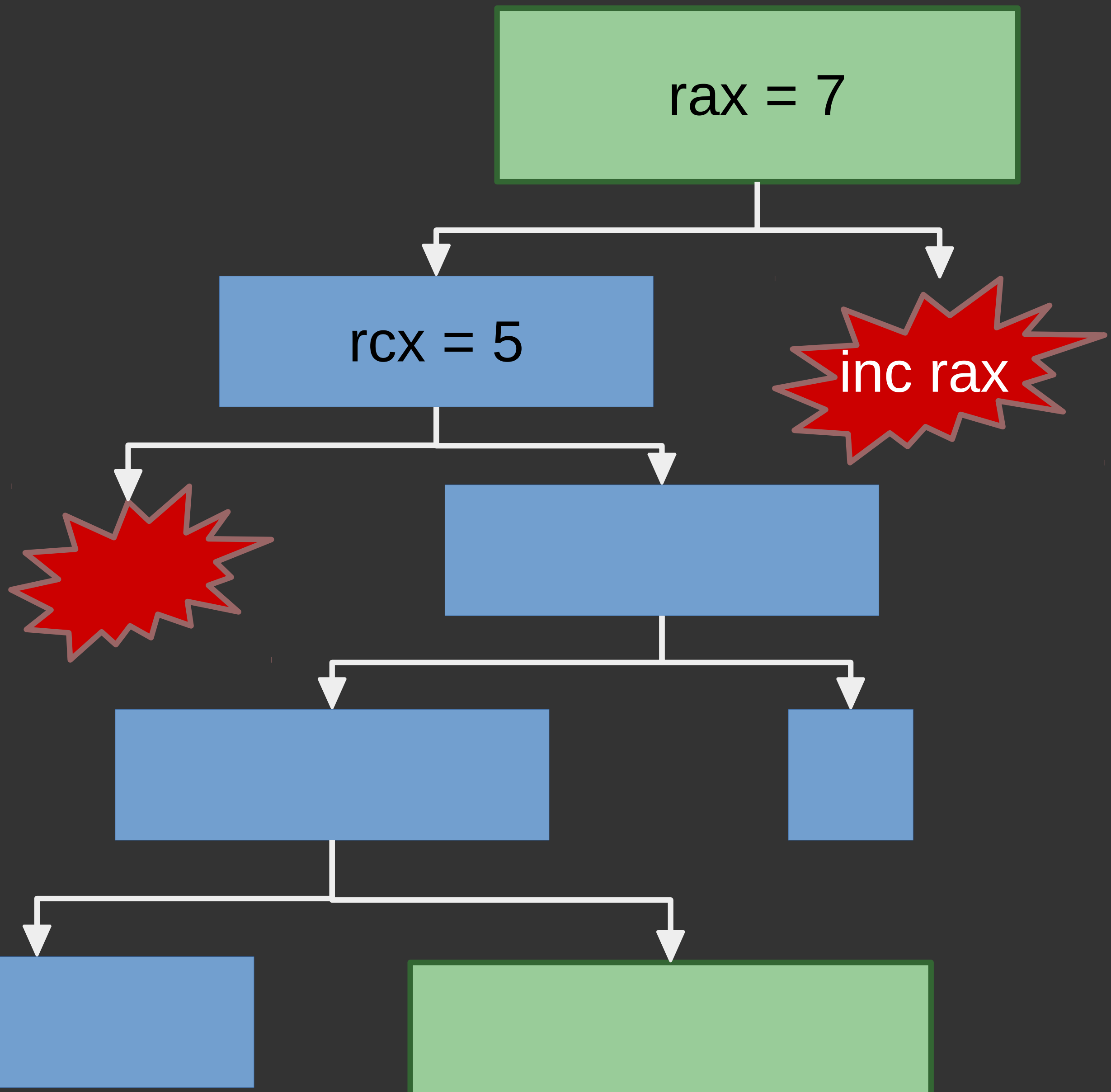
Approach in a nutshell

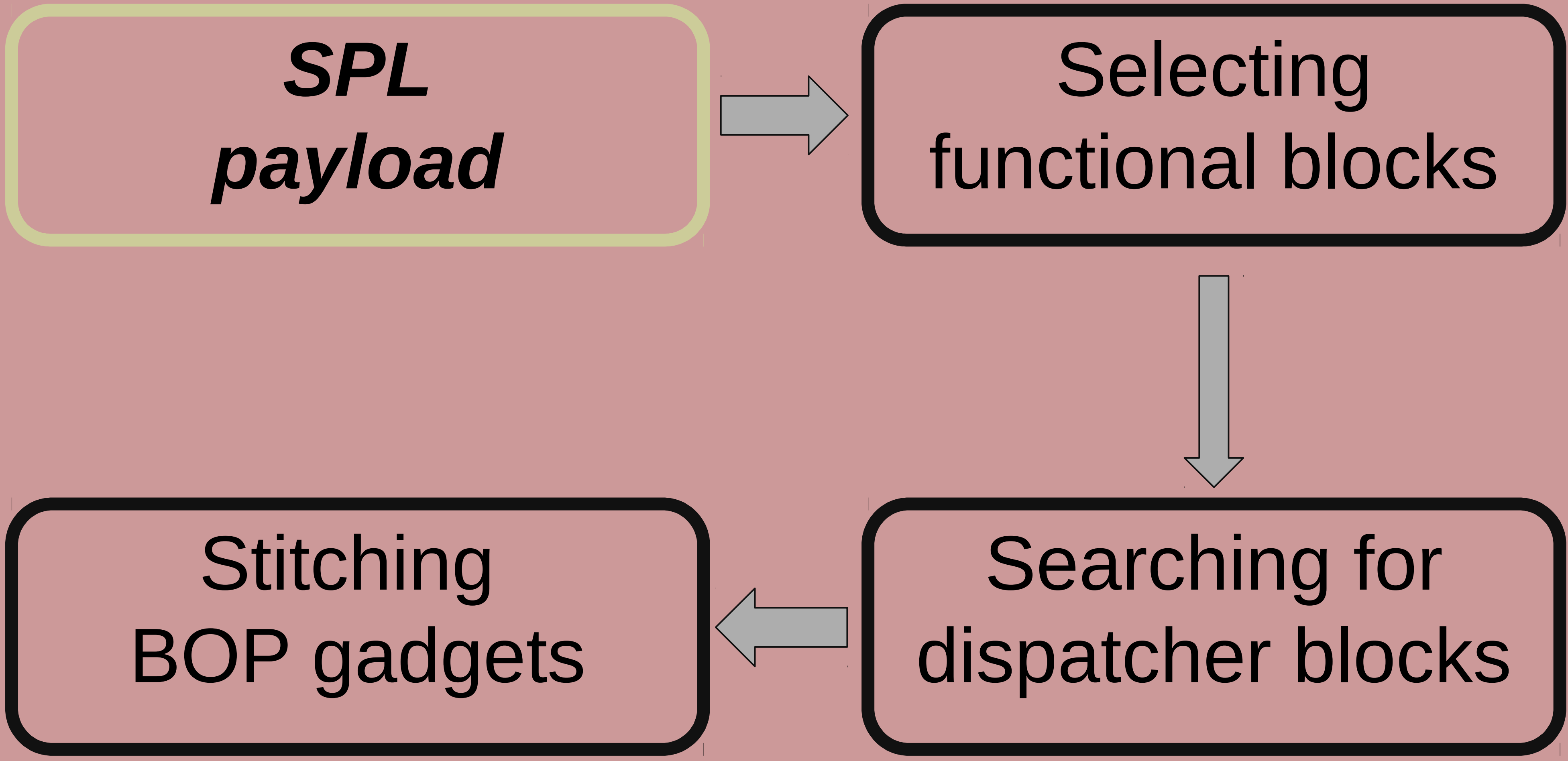
- Given: crash that results in arbitrary write
- Assume: mitigations make exploitation hard

- Perform ***Code Reuse*** using ***Data-Only Attack***
  - Leverage memory corruption to corrupt state
  - Build Turing-complete payloads as execution traces
  - Express execution traces as memory writes

BOP Gadget: basic block sequence

- **Functional:** compute (rax = 7)
- **Dispatcher:** connect functional blocks
- **Clobbering:** destroy context

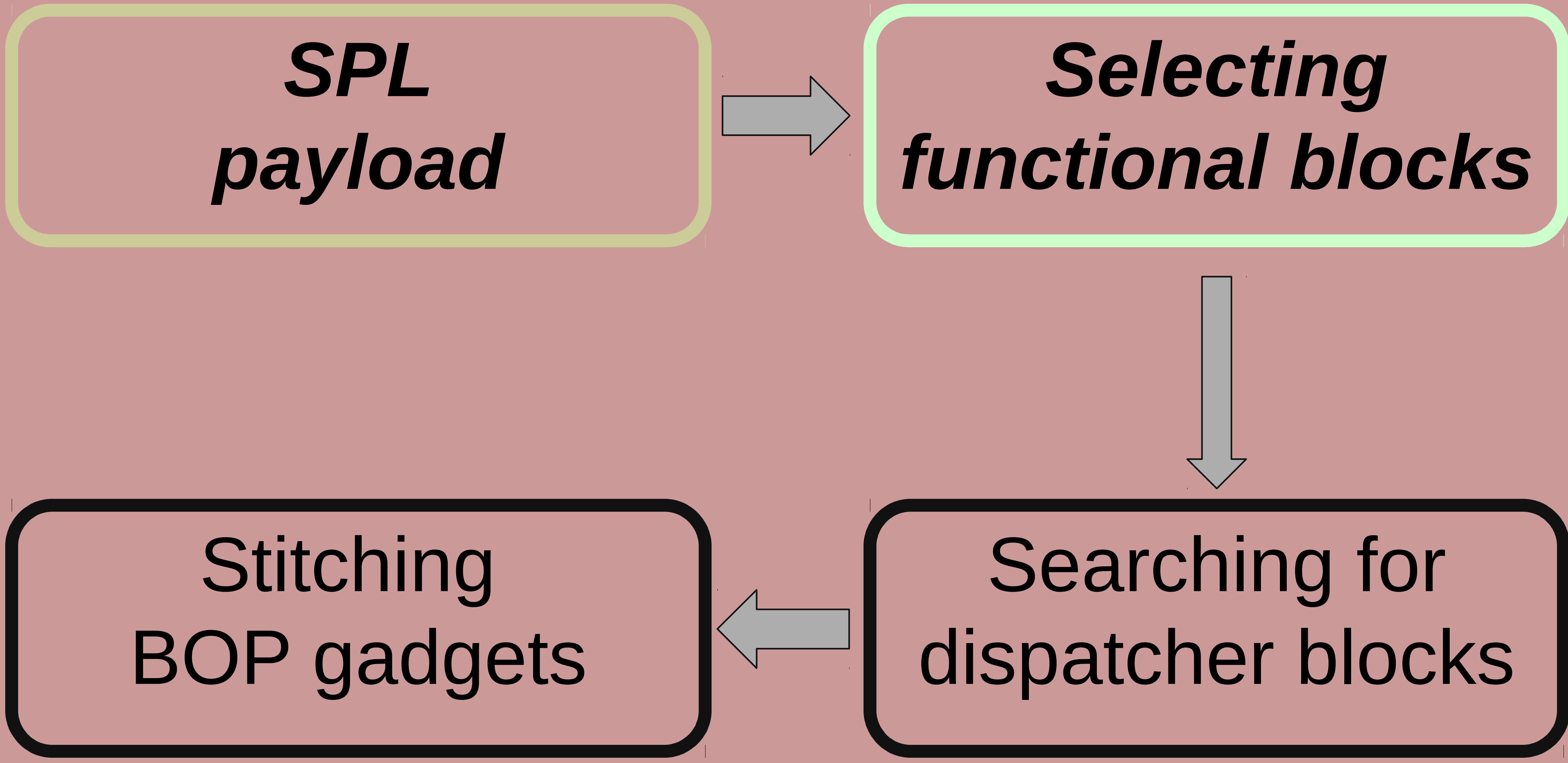



SPL payload
Selecting functional blocks
Searching for dispatcher blocks
Stitching BOP gadgets

SPL payload

- Payload language
- Subset of C
- Library Calls
- Abstract registers as volatile vars

```
void payload() {
  string prog = "/bin/sh\0";
  int64* argv = {&prog, 0x0};

  __r0 = &prog;
  __r1 = &argv;
  __r2 = 0;

  execve(__r0, __r1, __r2);
}
```


SPL payload
Selecting functional blocks
Searching for dispatcher blocks
Stitching BOP gadgets

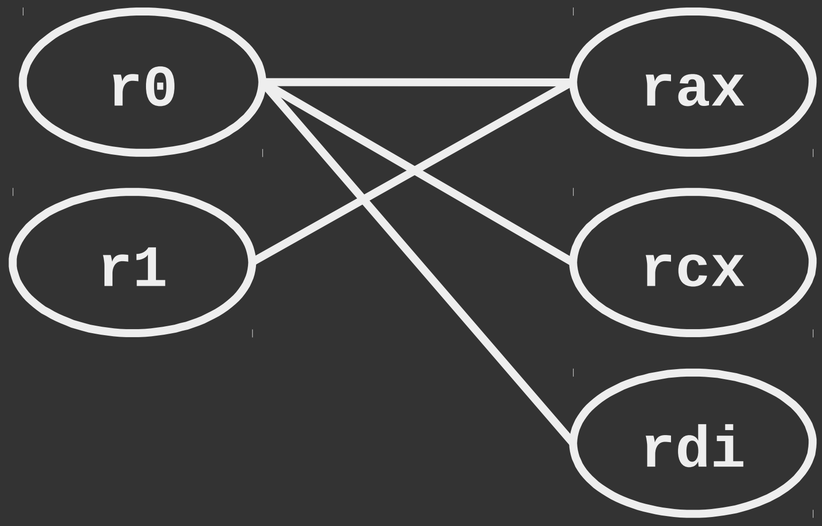
Functional block selection

- Find set of candidate blocks for SPL statement
- Candidate blocks ***“could be”*** functional blocks as the execute the correct computation

- What about other side effects? What about chaining functional blocks?

Functional block selection (example)

```
__r0 = 10;
__r1 = 20;
```



rax = 10

rdi = 10

rax = 20

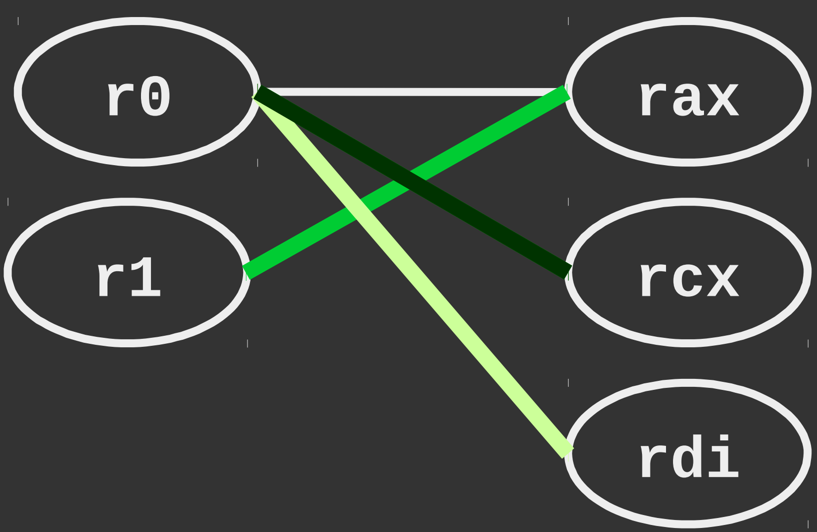
rcx = 10

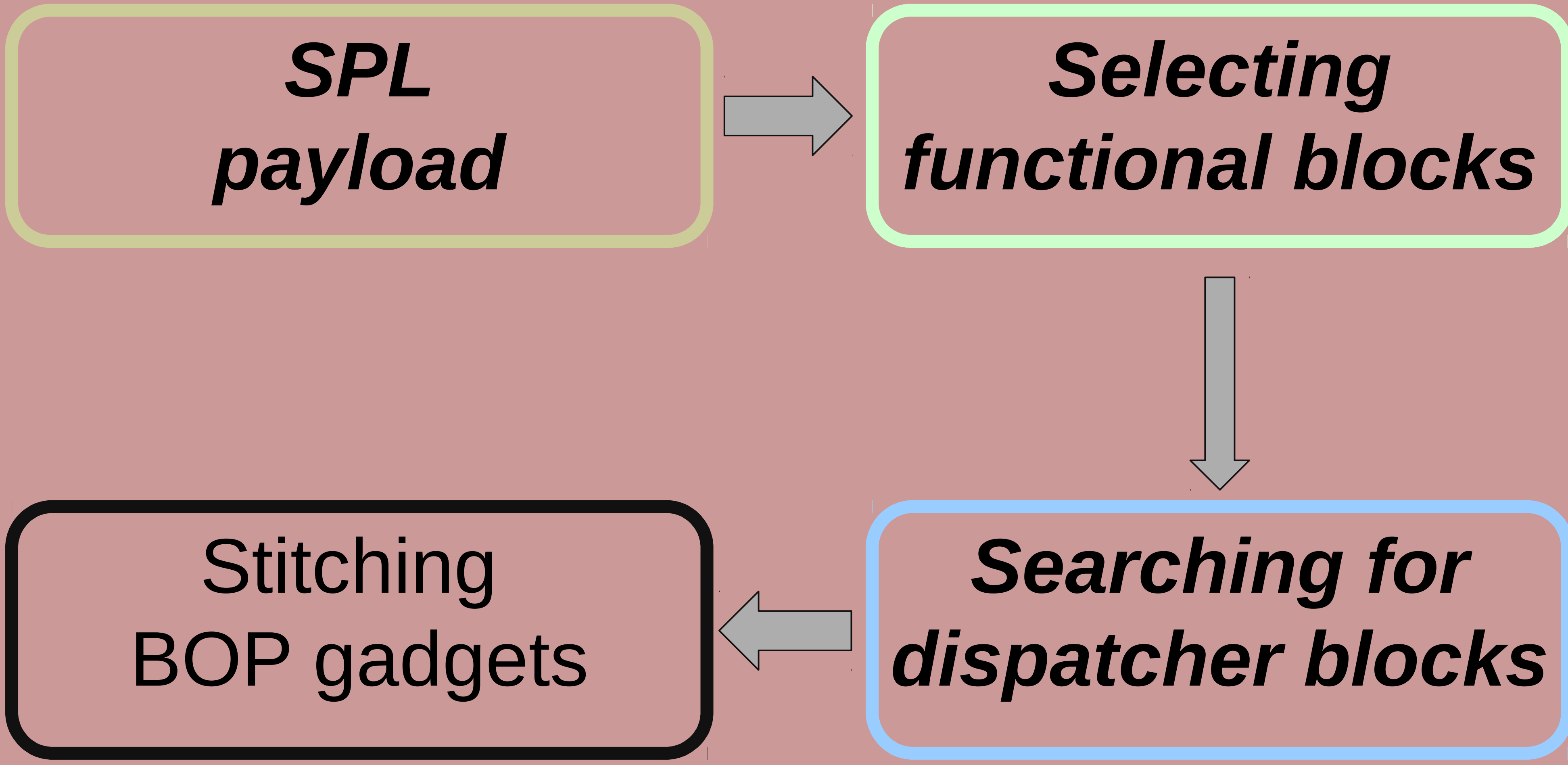
rcx = 30

Functional block selection (example)

```
__r0 = 10;
__r1 = 20;
```

rax = 10	Clobbering	Clobbering
rdi = 10	Dispatcher	Functional
rax = 20	Functional	Functional
rcx = 10	Functional	Dispatcher
rcx = 30	Clobbering	Dispatcher

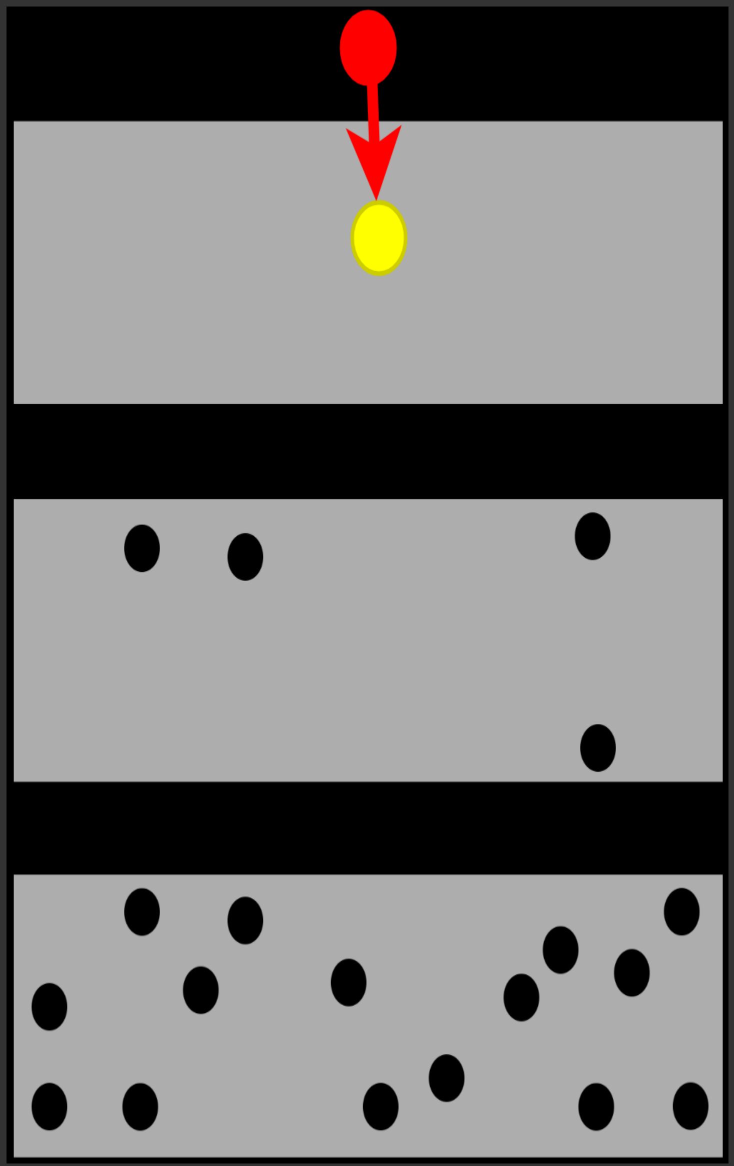



SPL payload
Selecting functional blocks
Searching for dispatcher blocks
Stitching BOP gadgets

Dispatcher block search

- BOP gadgets are ***brittle***
- Side-effects make gadgets hard to chain
  - Stitching gadgets is NP-hard
  - There is no approximative solution
- Our approach: back tracking and heuristics

BOP gadgets are brittle



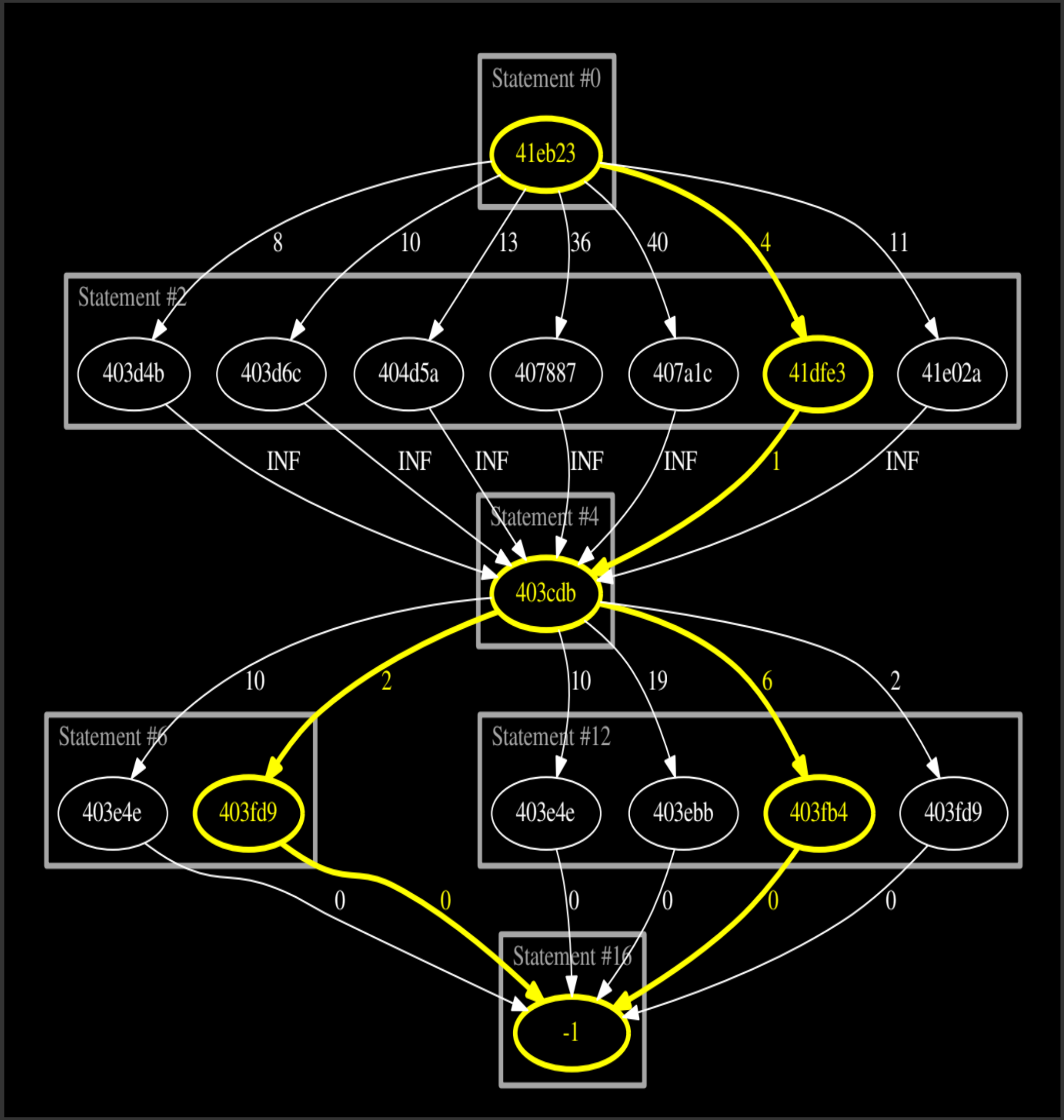
Statement #1

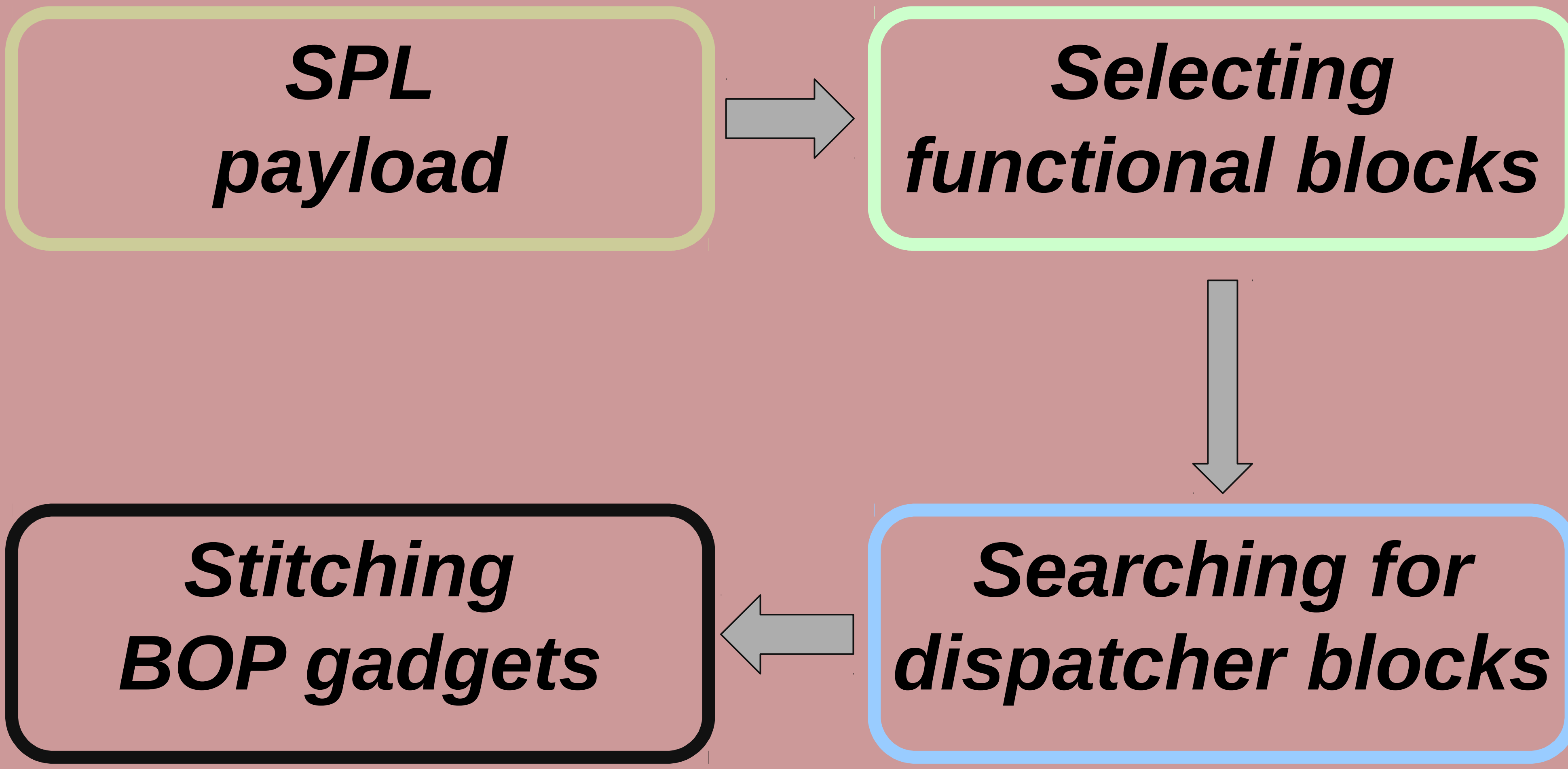
Statement #2

Statement #3

Delta Graph: keeping track of blocks

- Squares: Functional blocks for SPL statements
- Nodes: Functional blocks
- Edges: Length of dispatcher chain
- Goal: Select one “node” from each layer (yellow)




SPL payload
Selecting functional blocks
Searching for dispatcher blocks
Stitching BOP gadgets

Stitching BOP gadgets

- Each path is a candidate exploit
- Check and validate constraints along paths
  - Goal: find a valid configuration
  - Constraints come from environment, SPL program, or execution context
  - Verify using concolic execution & constraint solving

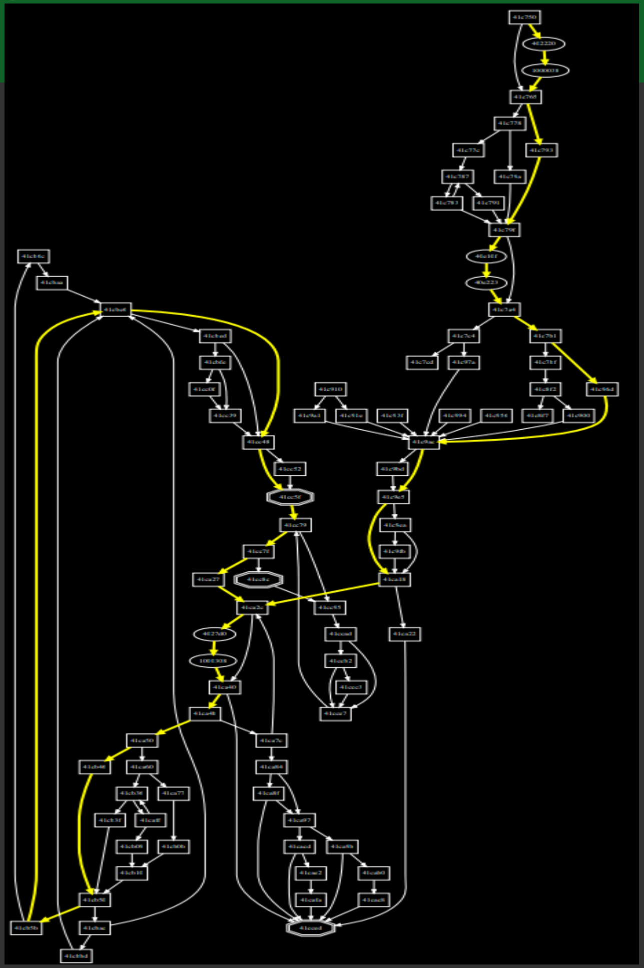
Payload synthesis

Program	SPL payload												
	regset4	*regref4*	*regset5*	*regref5*	*regmod*	*memrd*	*memwr*	*print*	*execve*	*abloop*	*infloop*	*ifelse*	*loop*
ProFTPd	✓	✓	✓	✓	✓	✓	✓	✓ 32	✗$_1$	✓ 128+	✓ ∞	✓	✓ 3
nginx	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✓	✓ 128+	✓ ∞	✓	✓ 128
sudo	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✓ 128+	✗$_4$	✗$_4$
orzhttpd	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_1$	✗$_4$	✓ 128+	✗$_4$	✗$_3$
wuftdp	✓	✓	✓	✓	✓	✓	✓	✓	✗$_1$	✓ 128+	✓ 128+	✗$_4$	✗$_3$
nullhttpd	✓	✓	✓	✓	✓	✓	✗$_3$	✗$_3$	✓	✓ 30	✓ ∞	✗$_4$	✗$_3$
opensshd	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_4$	✗$_4$	✓ 512	✓ 128+	✓	✓ 99
wireshark	✓	✓	✓	✓	✓	✓	✓	✓ 4	✗$_1$	✓ 128+	✓ 7	✓	✓ 8
apache	✓	✓	✓	✓	✓	✓	✓	✗$_4$	✗$_4$	✓ ∞	✓ 128+	✓	✗$_4$
smbclient	✓	✓	✓	✓	✓	✓	✓	✓ 1	✗$_1$	✓ 1057	✓ 128+	✓	✓ 256

- ✓ **The SPL payload was successfully executed on the target binary**
- ✗$_1$ **Not enough candidate blocks**
- ✗$_2$ **No valid register/variable mappings**
- ✗$_3$ **No valid paths between functional blocks**
- ✗$_4$ **Un-satisfiable constraints or solver timeout**

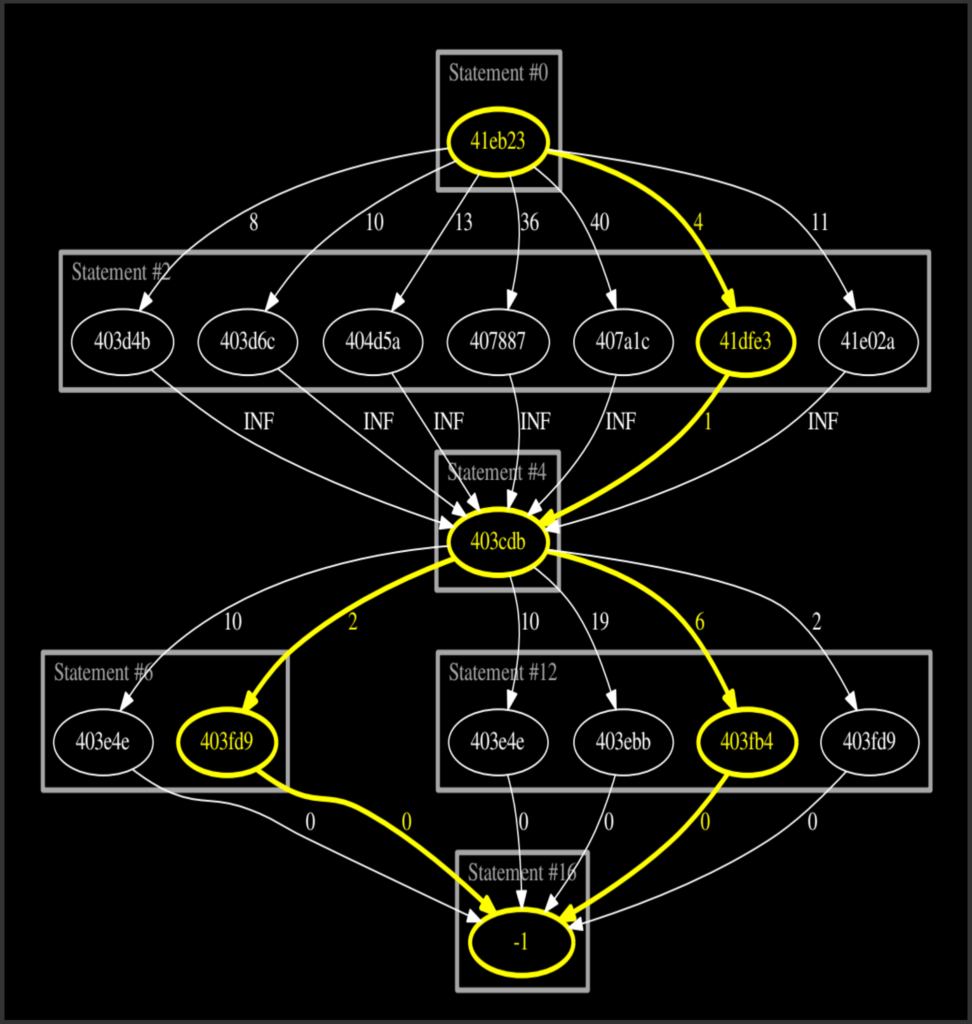
Success Rate: *81%*

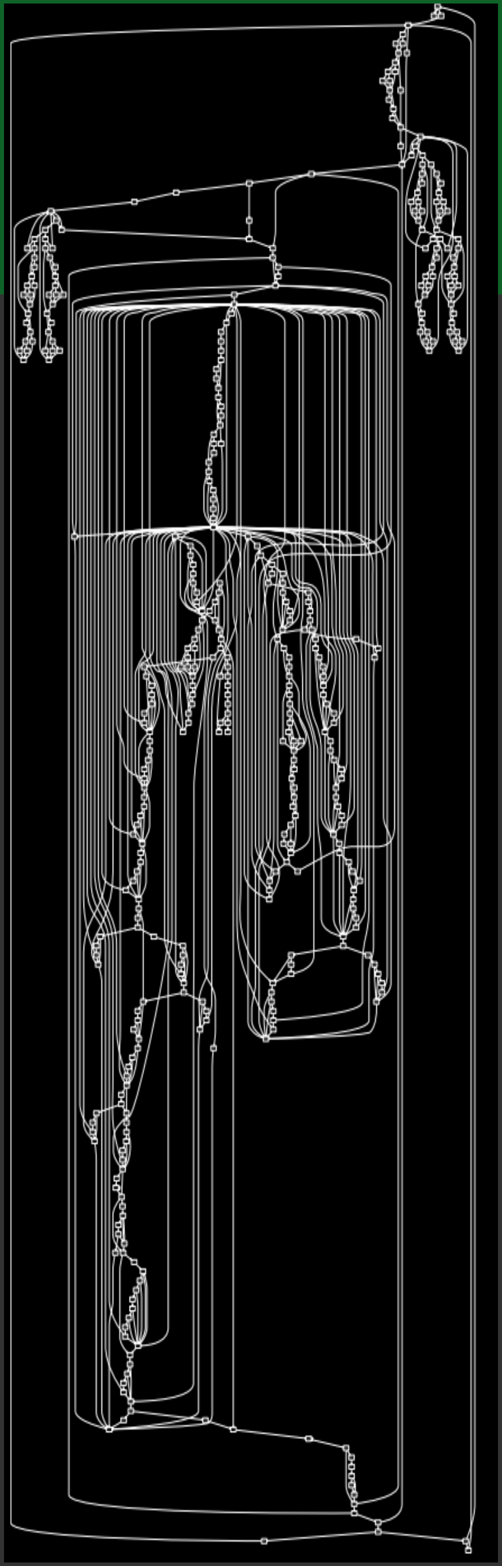
Case study: inf loop on nginx



```
ngx_signal_handler()
41C765: signals.signo == 0
40E10F: ngx_time_lock != 0
41C7B1: ngx_process 3 > 1
41C9AC: ngx_cycle = $alloc_1
        $alloc_1 > log = $alloc_2
        $alloc_2 > log_level <= 5
41CA18: signo == 17
41CA4B: waitpid() return value != {0, 1}
41cA50: ngx_last_process == 0
41CB50: *($stack 0x03C) & 0x7F != 0
41CB5B: $alloc_2 > log_level <= 1
41CBE6: *($stack 0x03C + 1) != 2
41CC48: ngx_accept_mutex_ptr == 0
41CC5F: ngx_cycle> shared_memory.part.elts = 0
        __r0 = r14 = 0
41CC79: ngx_cycle > shared_memory.part.nelts <= 0
41CC7F: ngx_cycle > shared_memory.part.next == 0
```

Case study: if-else in nginx





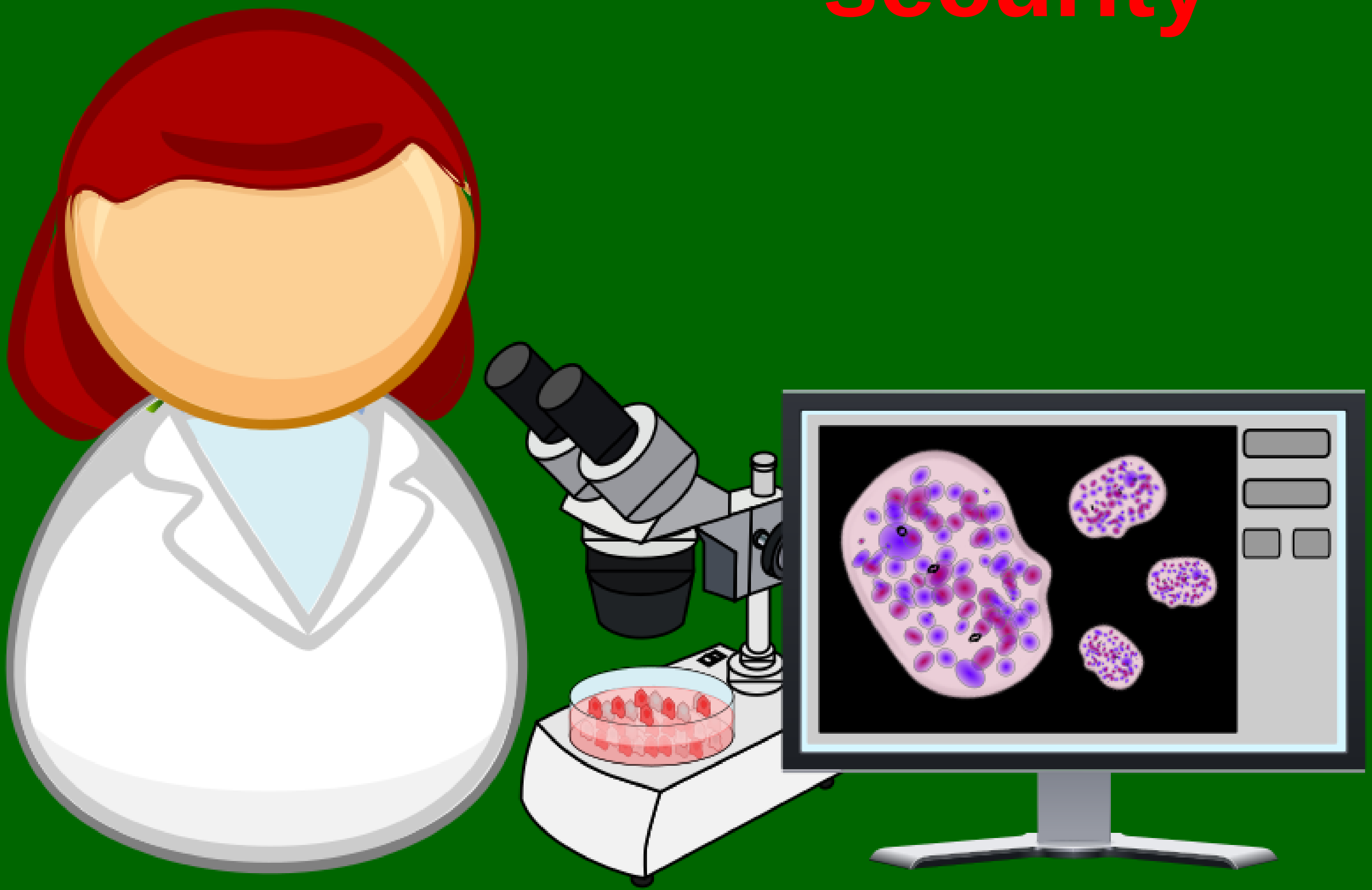
BOP summary

- Block Oriented Programming
  - Automates Data-Only attacks
  - SPL: A language to express exploit payloads
  - Concolic execution algorithm stitches BOP gadgets
- We build exploits for 81% of the case studies
- Open source implementation (~14,000 LoC)

Block Oriented Programming: Automating Data-Only Attacks
Kyriakos Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer.
In CCS'18: ACM Conference on Computer and Communication Security, 2018

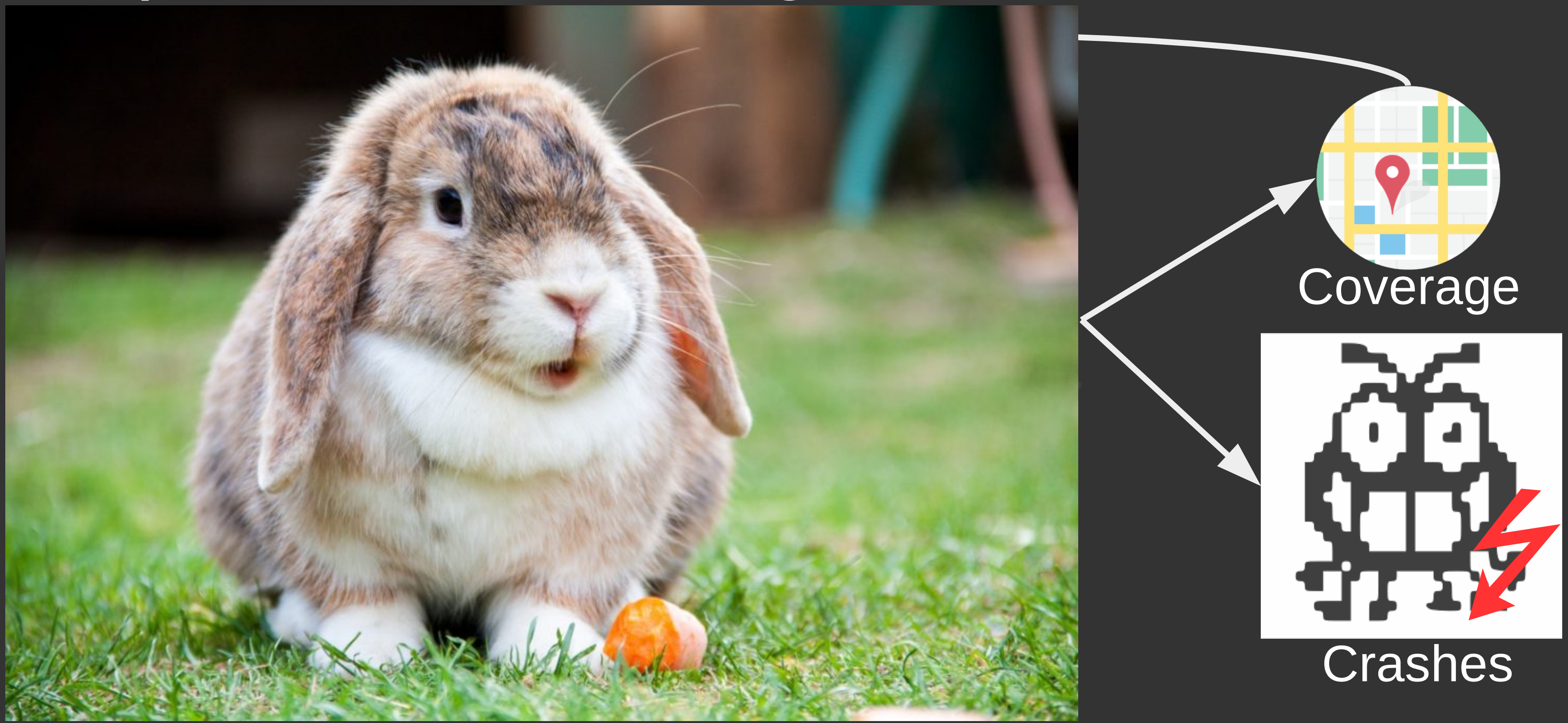
Software testing: discover bugs

security



Fuzz testing

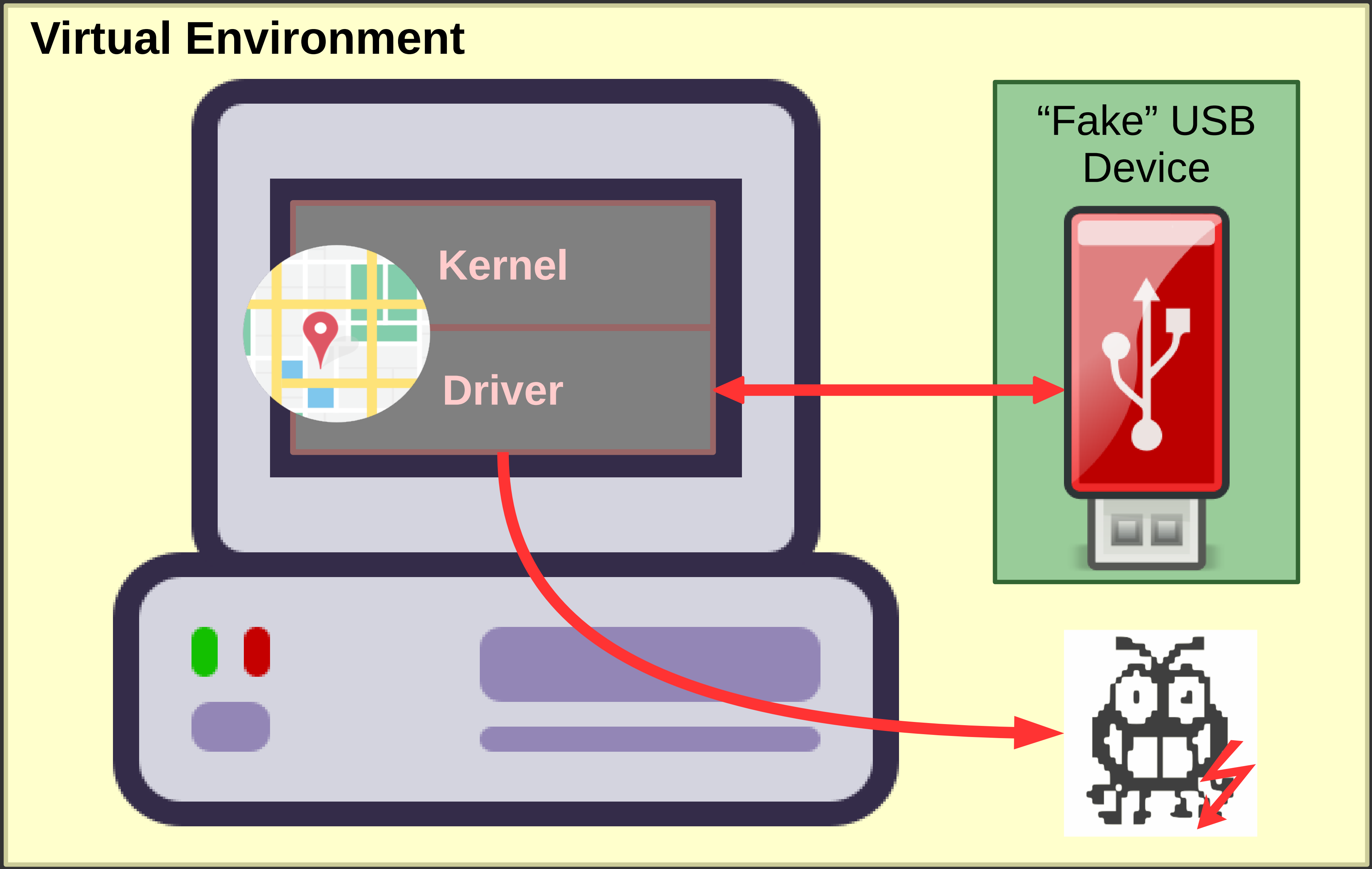
- A random testing technique that mutates input to improve test coverage



- State-of-art fuzzers use coverage as feedback to evolutionarily mutate the input

Academic fuzzing research

USBFuzz: explore peripheral space



USBFuzz Evaluation

- ~60 new bugs discovered in recent kernels
- 36 memory bugs (UaF / BoF)
- ~12 bugs fixed (with 9 CVEs)
- Bug reporting in progress

Type	Bug Info	#
Memory Bugs (36)	double-free	2
	NULL pointer dereference	8
	general protection	6
	slab-out-of-bounds access	6
	user-after-free access	16
Unexpected state reached (17)	INFO	6
	WARNING	9
	BUG	2

Security testing hard-to-reach code

- Fuzzing is an effective way to automatically test programs for security violations (crashes)
  - Key idea: optimize for throughput
  - Coverage guides mutation
- BOP: assess ***exploitability***
- USBFuzz: fuzz ***peripherals***

https://hexhive.epfl.ch
https://github.com/HexHive

EPFL

hexhive

Vulnerable apps

Program	Vulnerability	Nodes	RegSet	RegMod	MemRd	MemWr	Call	Cond	Total
ProFTPd	CVE-2006-5815	27,087	40,143	387	1,592	199	77	3,029	45,427
nginx	CVE-2013-2028	24,169	31,497	1,168	1,522	279	35	3375	37,876
sudo	CVE-2012-0809	3,399	5,162	26	157	18	45	307	5715
orzhttpd	BID 41956	1,345	2,317	9	39	8	11	89	2473
wuftpd	CVE-2000-0573	8,899	14,101	62	274	11	94	921	15,463
nullhttpd	CVE-2002-1496	1,488	2,327	77	54	7	19	125	2,609
opensshd	CVE-2001-0144	6,688	8,800	98	214	19	63	558	9,752
wireshark	CVE-2014-2299	74,186	124,053	639	1,736	193	100	4555	131276
apache	CVE-2006-3747	18,790	33,615	212	490	66	127	1,768	36,278
smbclient	CVE-2009-1886	166,081	265,980	1,481	6,791	951	119	28,705	304,027

RegSet: Register Assignment Gadgets
RegMod: Register Modification Gadgets
MemRd: Memory Read Gadgets
MemWr: Memory Write Gadgets
Call: Function/System Call Gadgets
Cond: Conditional Statement Gadgets
Total: Total number of Functional Gadgets

SPL payloads

Payload	Description
regset4	Initialize 4 registers with arbitrary values
regref4	Initialize 4 registers with pointers to arbitrary memory
regset5	Initialize 5 registers with arbitrary values
regref5	Initialize 5 registers with pointers to arbitrary memory
regmod	Initialize a register with an arbitrary value and modify it
memrd	Read from arbitrary memory
memwr	Write to arbitrary memory
print	Display a message to stdout using write
execve	Spawn a shell through execve
abloop	Perform an arbitrarily long bounded loop utilizing regmod
infloop	Perform an infinite loop that sets a register in its body
ifelse	An if-else condition based on a register comparison
loop	Conditional loop with register modification