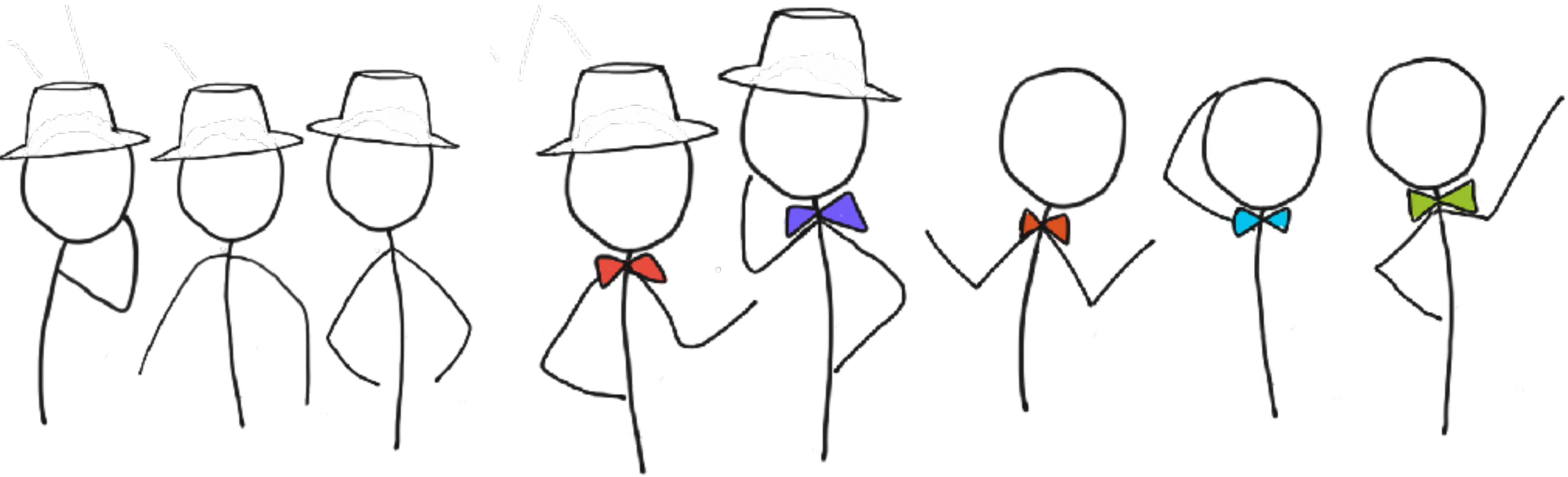
Using model checking to triage the severity of security bugs in the Xen hypervisor.

Should we wake the developer up?

Byron Cook[1,2], Björn Döbel[1], Daniel Kroening[1,3], Norbert Manthey[1], Martin Pohlack[1], Elizabeth Polgreen[5,6], Michael Tautschnig[1,4], Pawel Wieczorkiewicz[1]

[1] Amazon Web Services
[2] University College London
[3] University of Oxford
[4] Queen Mary University of London
[5] UC Berkeley
[6] Edinburgh University



Byron Cook[1,2], Björn Döbel[1], Daniel Kroening[1,3], Norbert Manthey[1],
Martin Pohlack[1], Elizabeth Polgreen[5,6], Michael Tautschnig[1,4], Pawel Wieczorkiewicz[1]

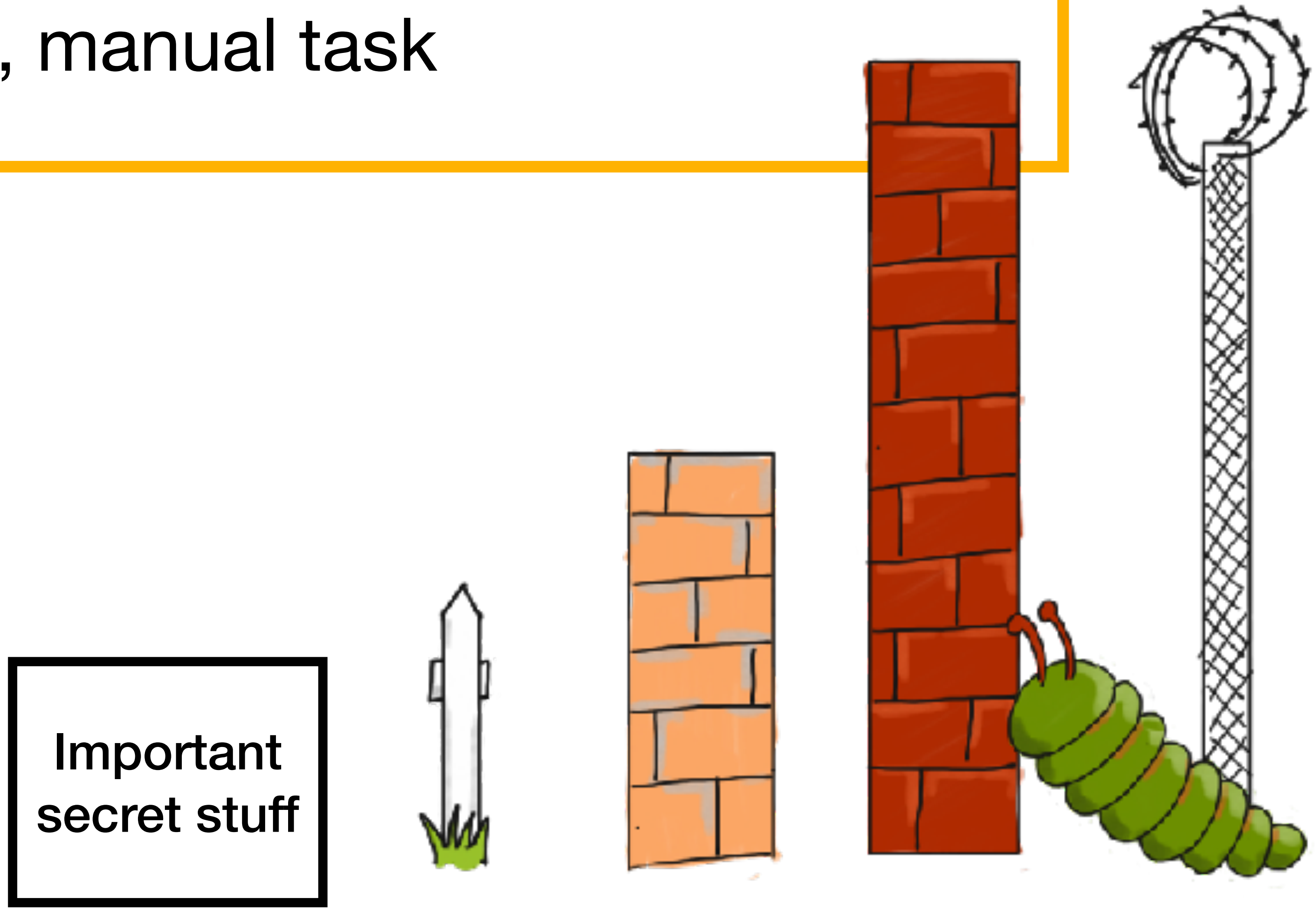
[1] Amazon Web Services
[2] University College London
[3] University of Oxford
[4] Queen Mary University of London
[5] UC Berkeley
[6] Edinburgh University

Problem:

- Most systems have layers of security
- Most bugs are not critical security issues
- BUT determining which ones are is a difficult, manual task

Problem:

- Most systems have layers of security
- Most bugs are not critical security issues
- BUT determining which ones are is a difficult, manual task



Problem:

- Most systems have layers of security
- Most bugs are not critical security issues
- BUT determining which ones are is a difficult, manual task

Solution:

- We show how to use model checking to triage the severity of security bugs
- We make adaptations to CBMC, a bounded model checker for C programs, so that it scales to big code bases
- Case study: Xen

Contents

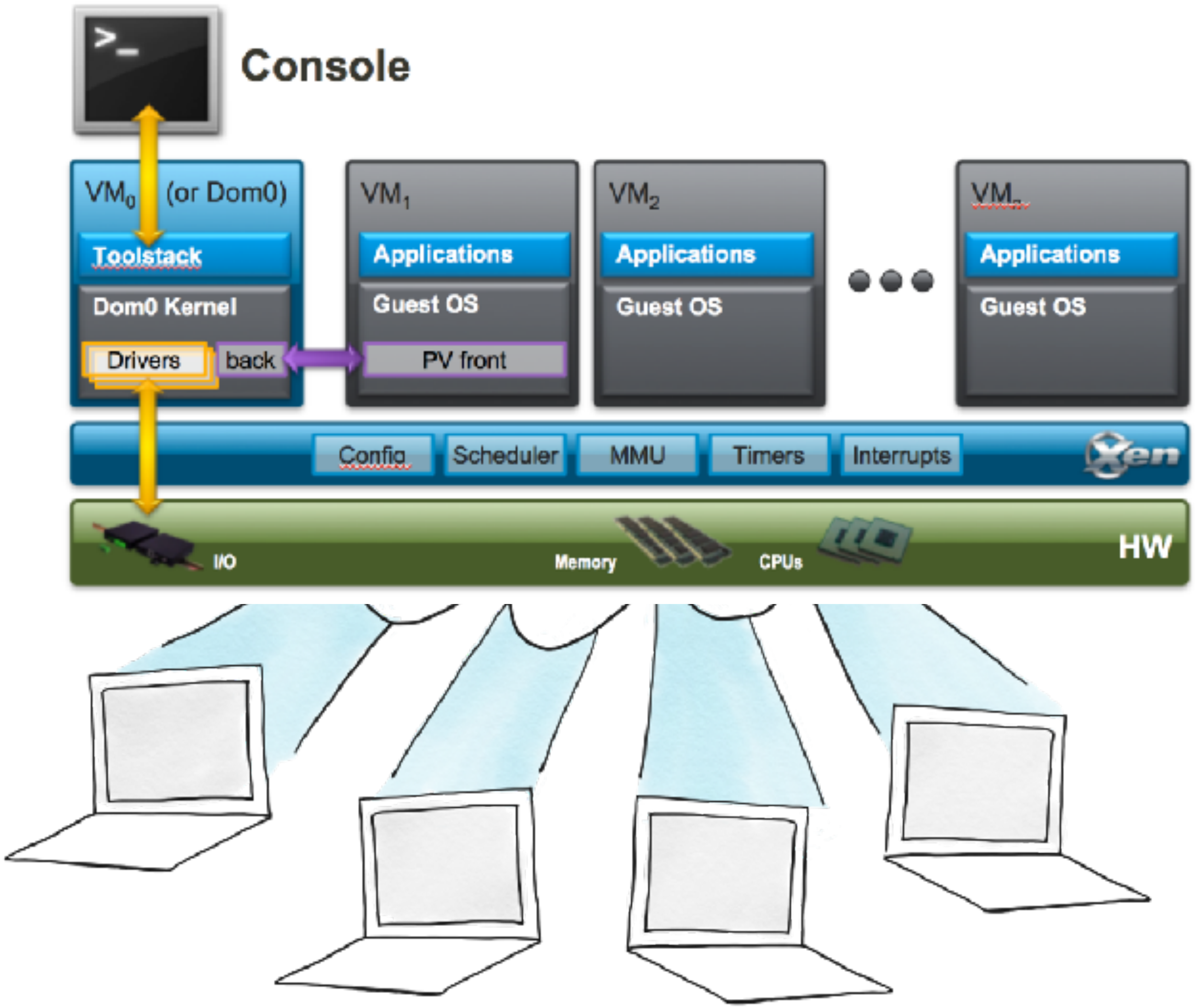
- What is Xen?
- Manual triaging of security issues in Xen.
- Why model checking Xen is hard.
- Adaptations to CBMC to make it possible.
- Conclusions

What is Xen?

Hypervisor: creates and runs virtual machines

Amazon use a custom version of Xen on
some EC2 servers

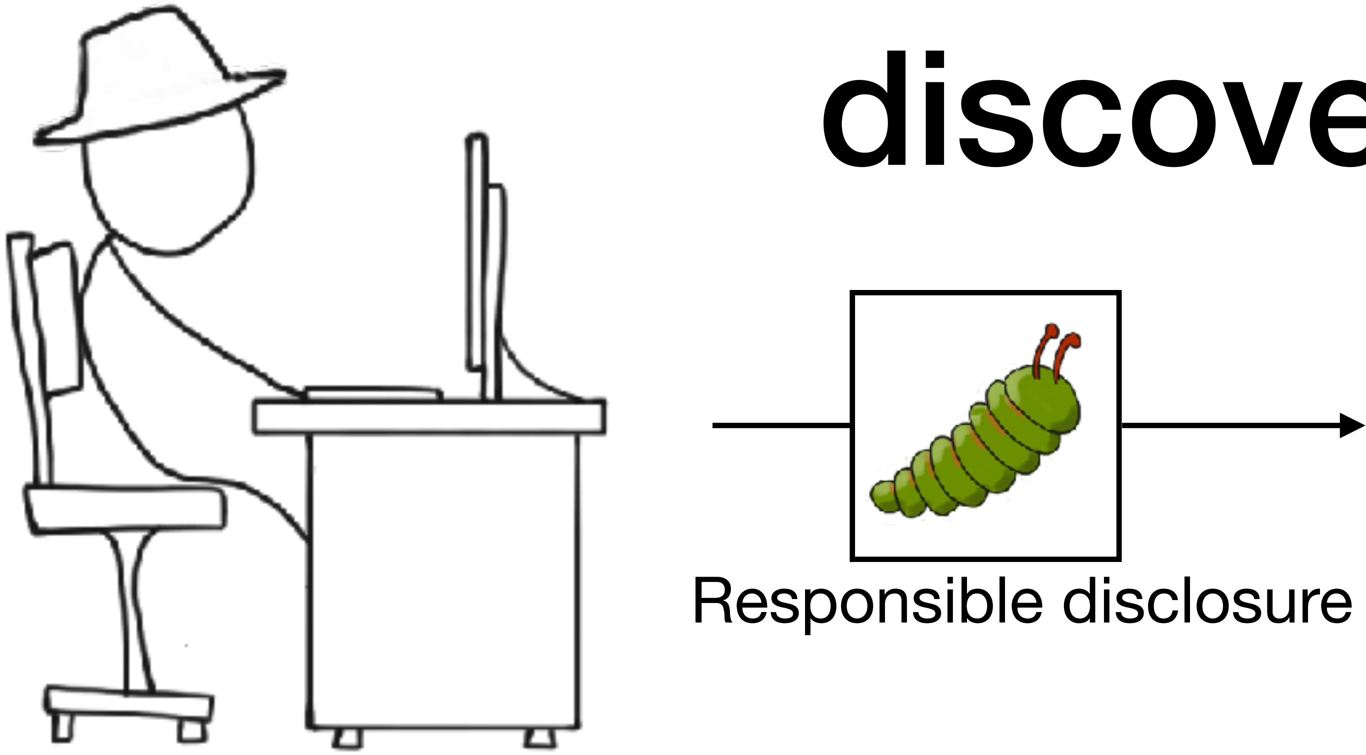
What is Xen?



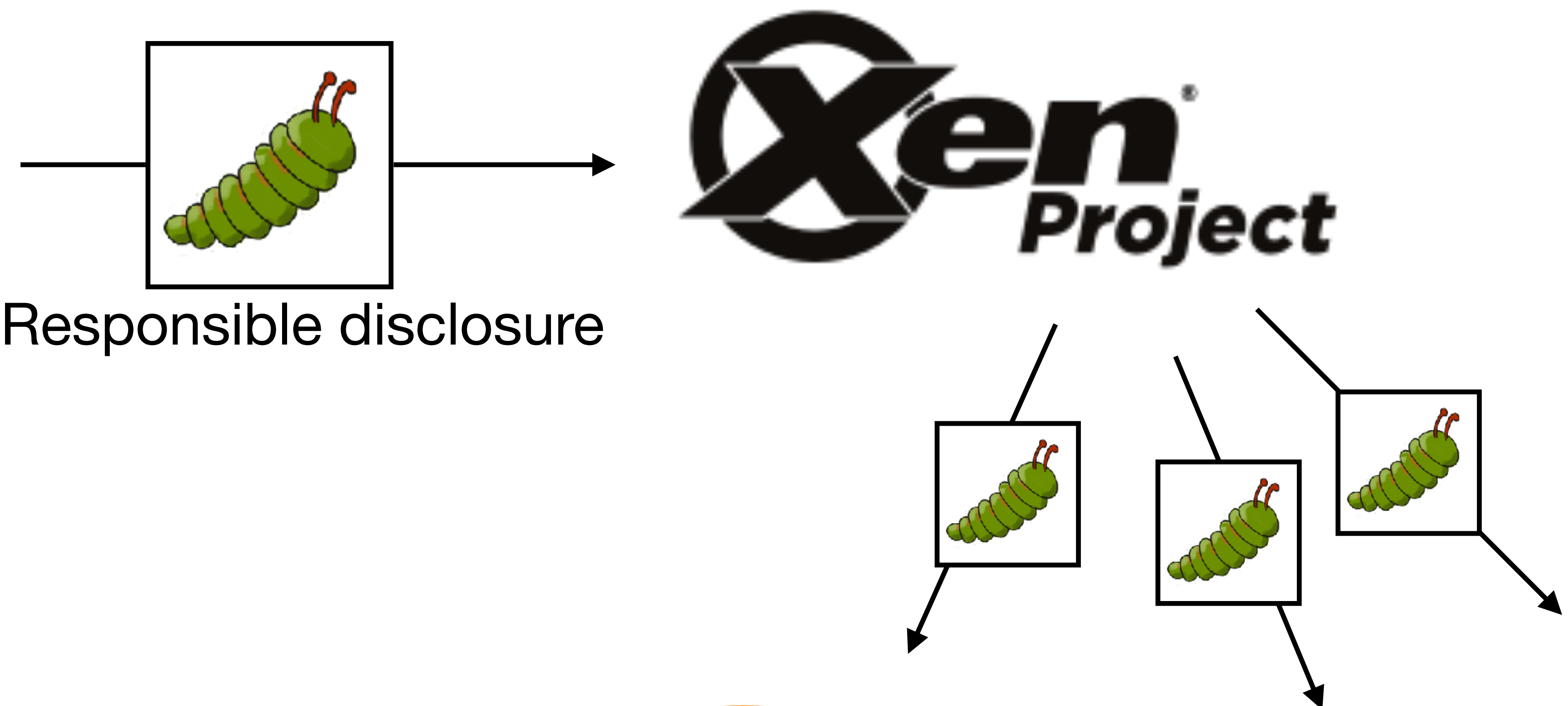
Console
VM0 (or Dom0)
Toolstack
Dom0 Kernel
Drivers
back
VM1
Applications
Guest OS
PV front
VM2
Applications
Guest OS
VMn
Applications
Guest OS
Config
Scheduler
MMU
Timers
Interrupts
Xen
I/O
Memory
CPUs
HW

What happens when a bug is discovered?

What happens when a bug is discovered?



What happens when a bug is discovered?



Responsible disclosure

Members of the Xen project

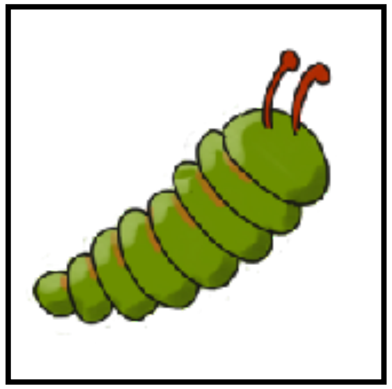
amazon web services

arm

intel

XSA: Xen Security Announcement

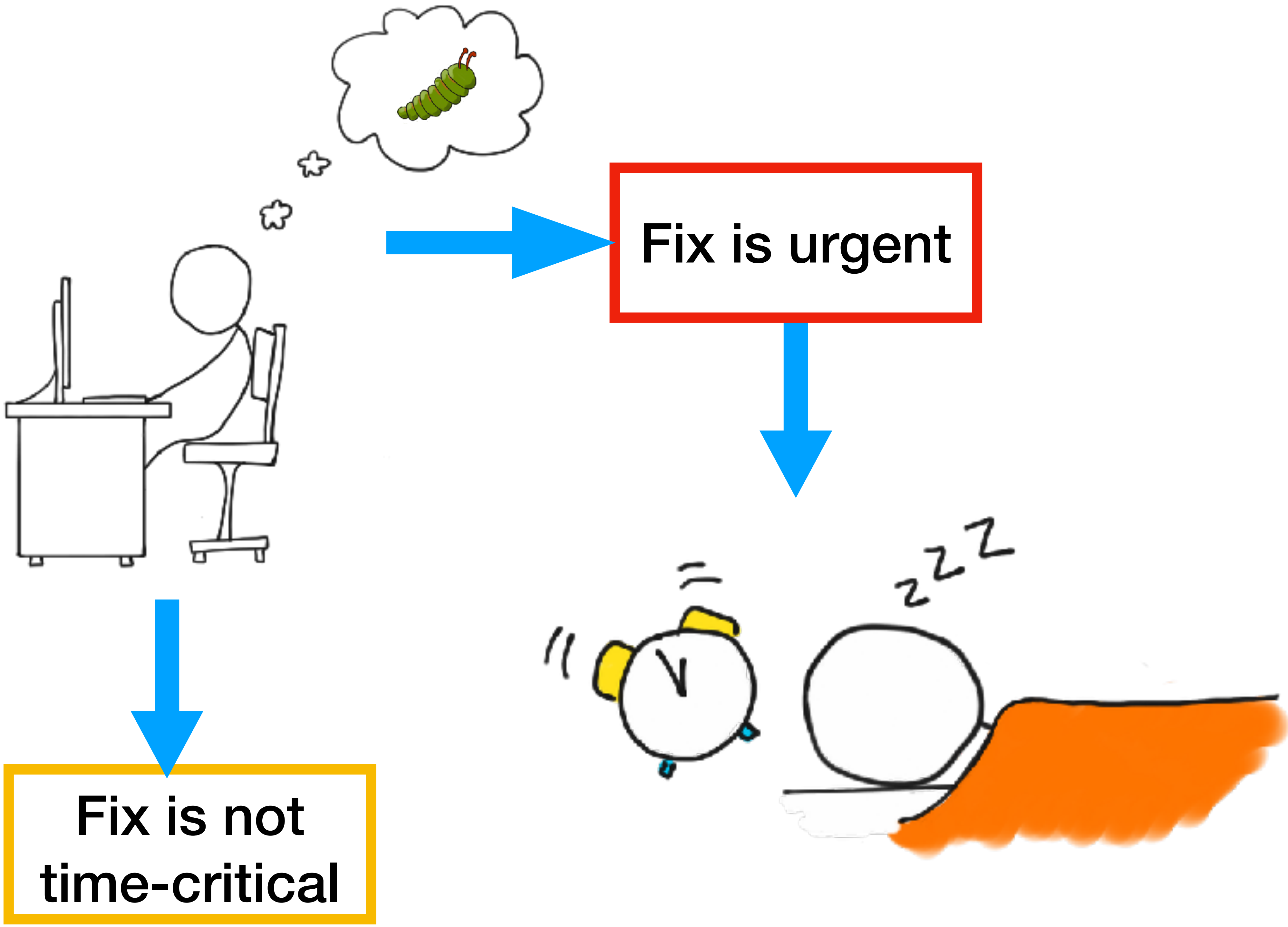
```
ISSUE DESCRIPTION
=================

The x86 instruction CMPXCHG8B is supposed to ignore legacy operand
size overrides; it only honors the REX.W override (making it
CMPXCHG16B).  So, the operand size is always 8 or 16.

When support for CMPXCHG16B emulation was added to the instruction
emulator, this restriction on the set of possible operand sizes was
relied on in some parts of the emulation; but a wrong, fully general,
operand size value was used for other parts of the emulation.

As a result, if a guest uses a supposedly-ignored operand size prefix,
a small amount of hypervisor stack data is leaked to the guests: a 96
bit leak to guests running in 64-bit mode; or, a 32 bit leak to other
guests.
```

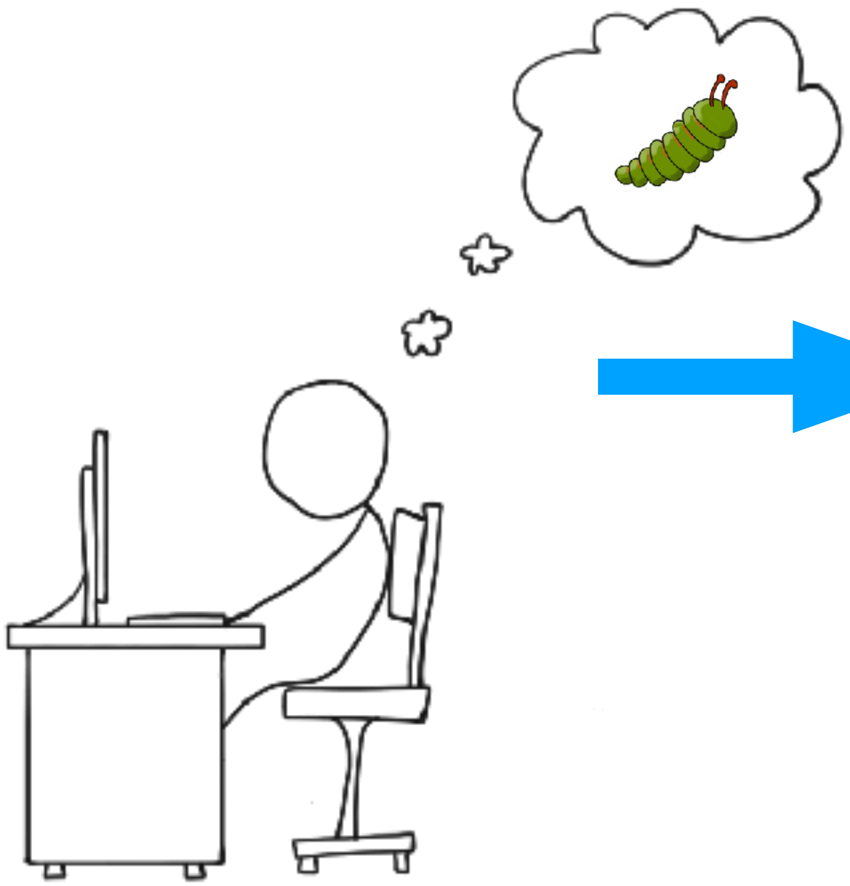


Advisories, publicly released or pre-released

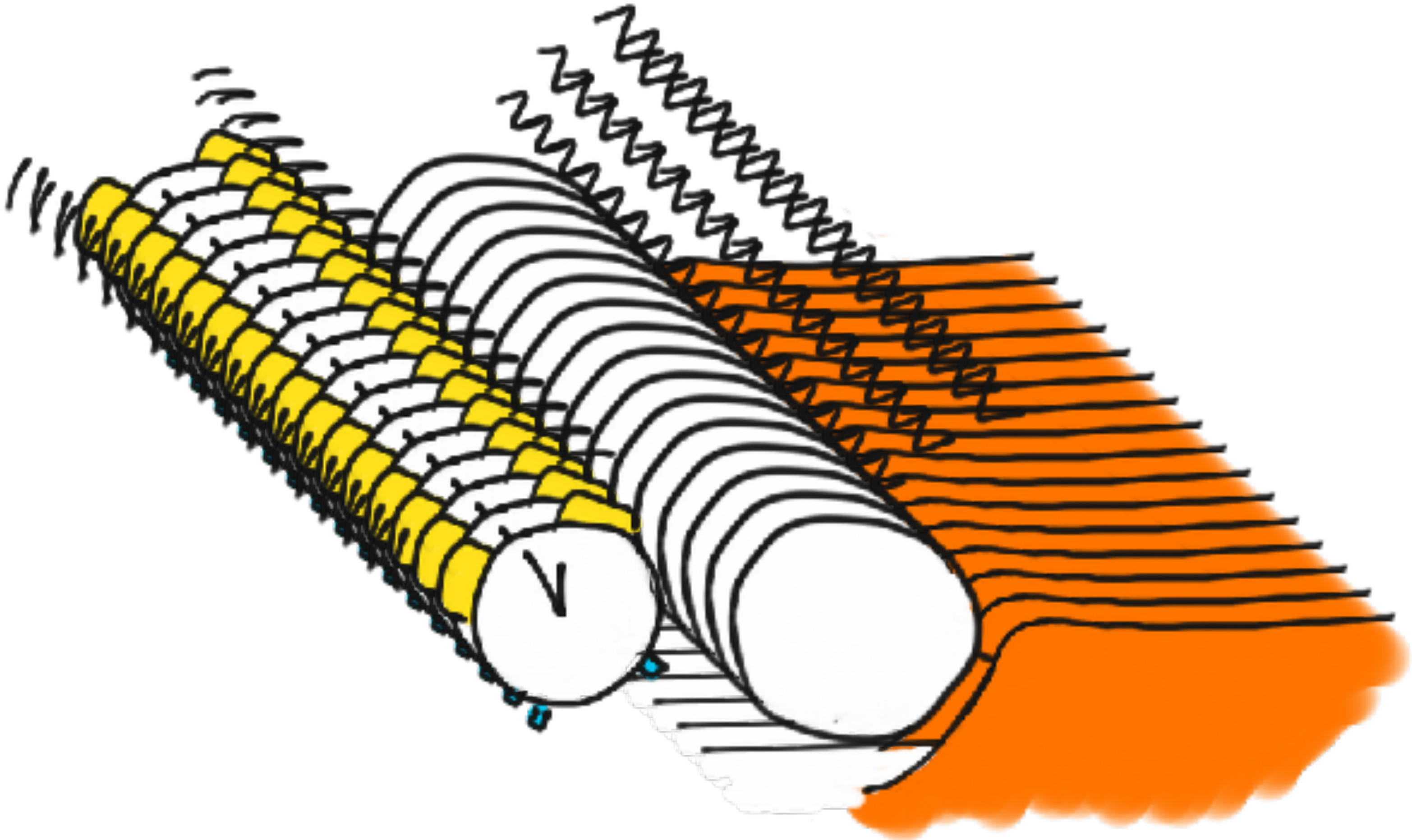
All times are in UTC. For general information about Xen and security see the Xen Project website and security policy. A JSON document listing advisories is also available.

Advisory	Public release	Updated	Version	CVE(s)	Title
XSA-344	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-343	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-342	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-341	2020-09-08 15:35		-	-	Unused Xen Security Advisory number
XSA-340	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-339	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-338	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-337	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-336	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-335	2020-08-24 12:00	2020-08-24 12:17	2	CVE-2020-14364	QEMU: usb: out-of-bounds r/w access issue
XSA-334	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-333	2020-09-22 12:00			none (yet) assigned	(Prereleased, but embargoed)
XSA-329	2020-07-16 12:00	2020-07-21 11:00	3	CVE-2020-15852	Linux ioperm bitmap context switching issues
XSA-328	2020-07-07 12:00	2020-07-07 12:23	3	CVE-2020-15567	non-atomic modification of live EPT PTE
XSA-327	2020-07-07 12:00	2020-07-07 12:23	3	CVE-2020-15564	Missing alignment check in VCPUOP_register_vcpu_info
XSA-321	2020-07-07 12:00	2020-07-07 12:21	3	CVE-2020-15565	insufficient cache write-back under VT-d
XSA-320	2020-06-09 16:33	2020-06-11 13:09	2	CVE-2020-0543	Special Register Buffer speculative side channel
XSA-319	2020-07-07 12:00	2020-07-07 12:18	3	CVE-2020-15563	inverted code paths in x86 dirty VRAM tracking
XSA-318	2020-04-14 12:00	2020-04-14 12:00	3	CVE-2020-11742	Bad continuation handling in GNTTABOP_copy
XSA-317	2020-07-07 12:00	2020-07-07 12:18	3	CVE-2020-15566	Incorrect error handling in event channel port allocation
XSA-316	2020-04-14 12:00	2020-04-14 12:00	3	CVE-2020-11743	Bad error path in GNTTABOP_map_grant

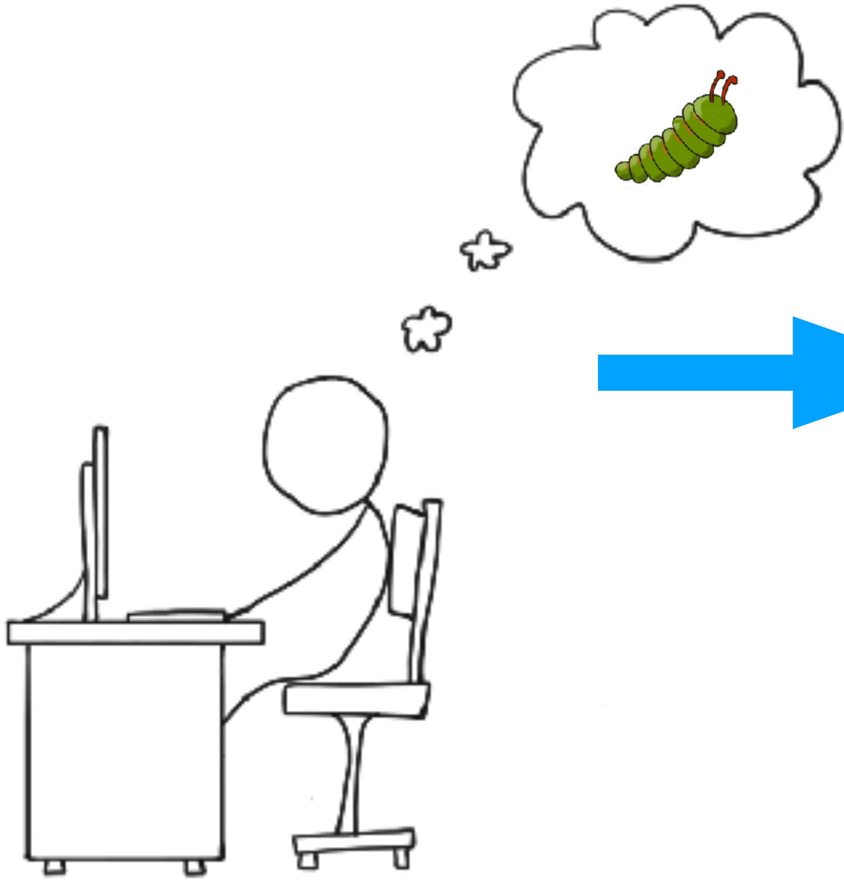

Fix is urgent
Fix is not time-critical



Fix is urgent

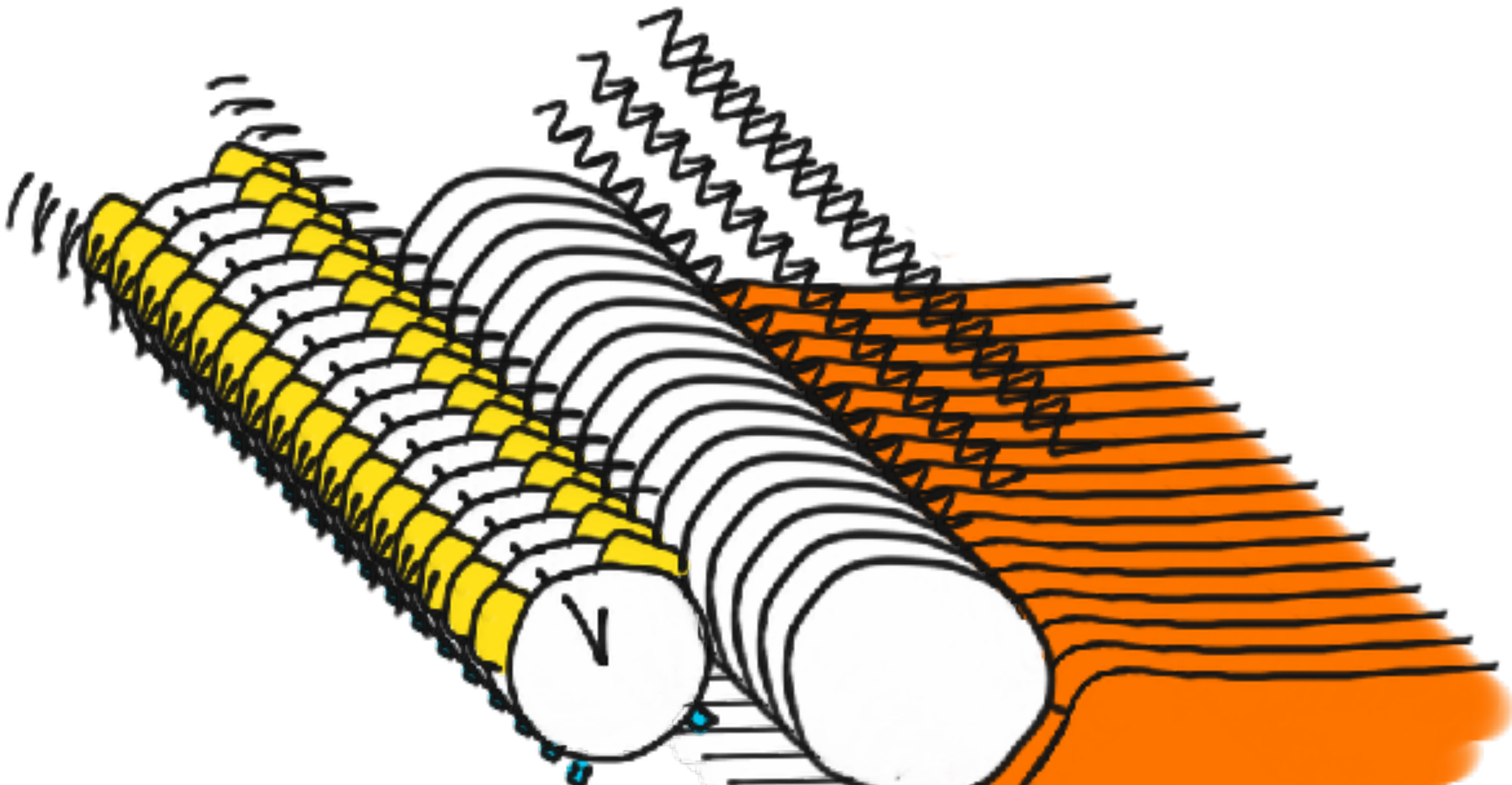


Fix is not time-critical



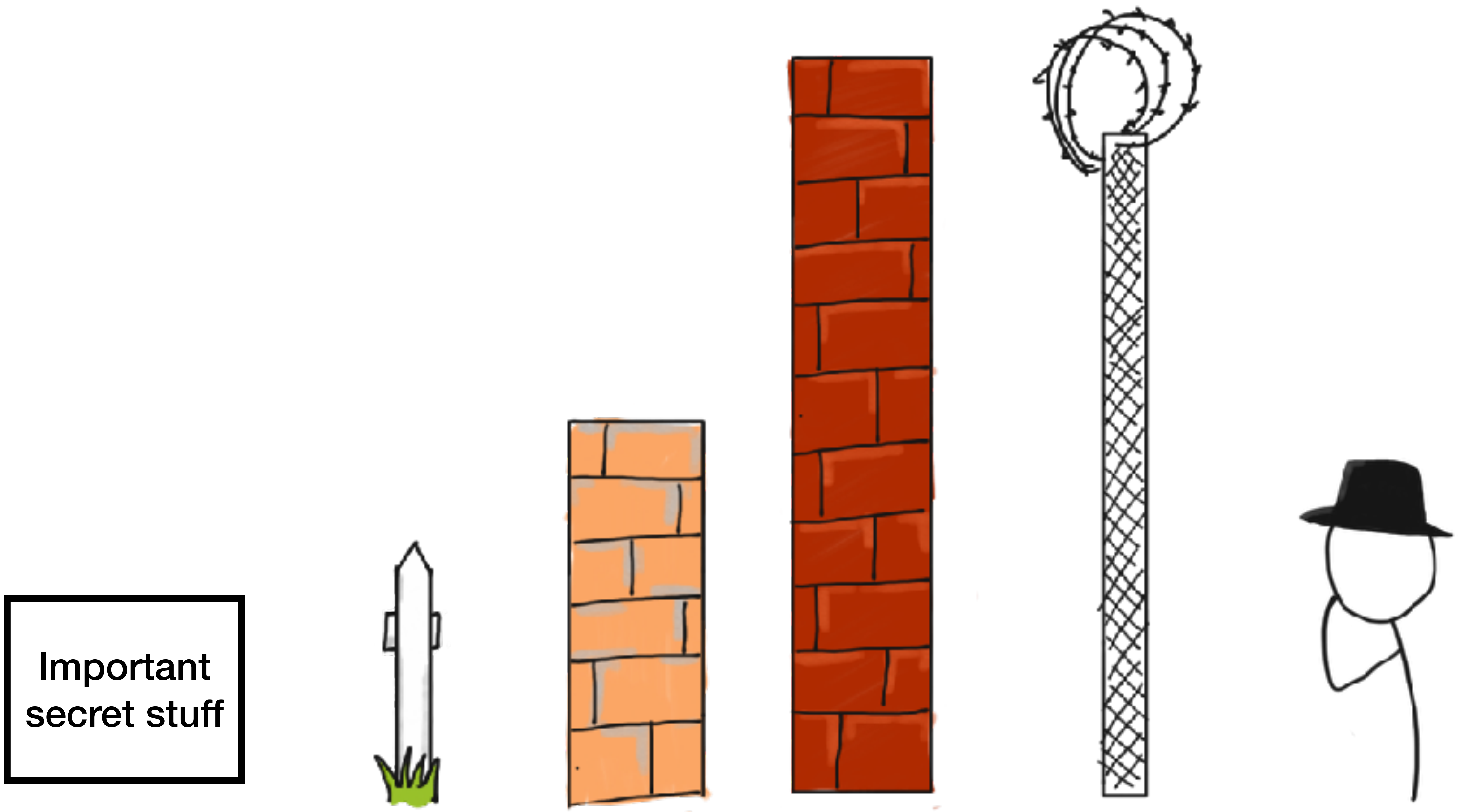
Fix is urgent

Fix is not time-critical

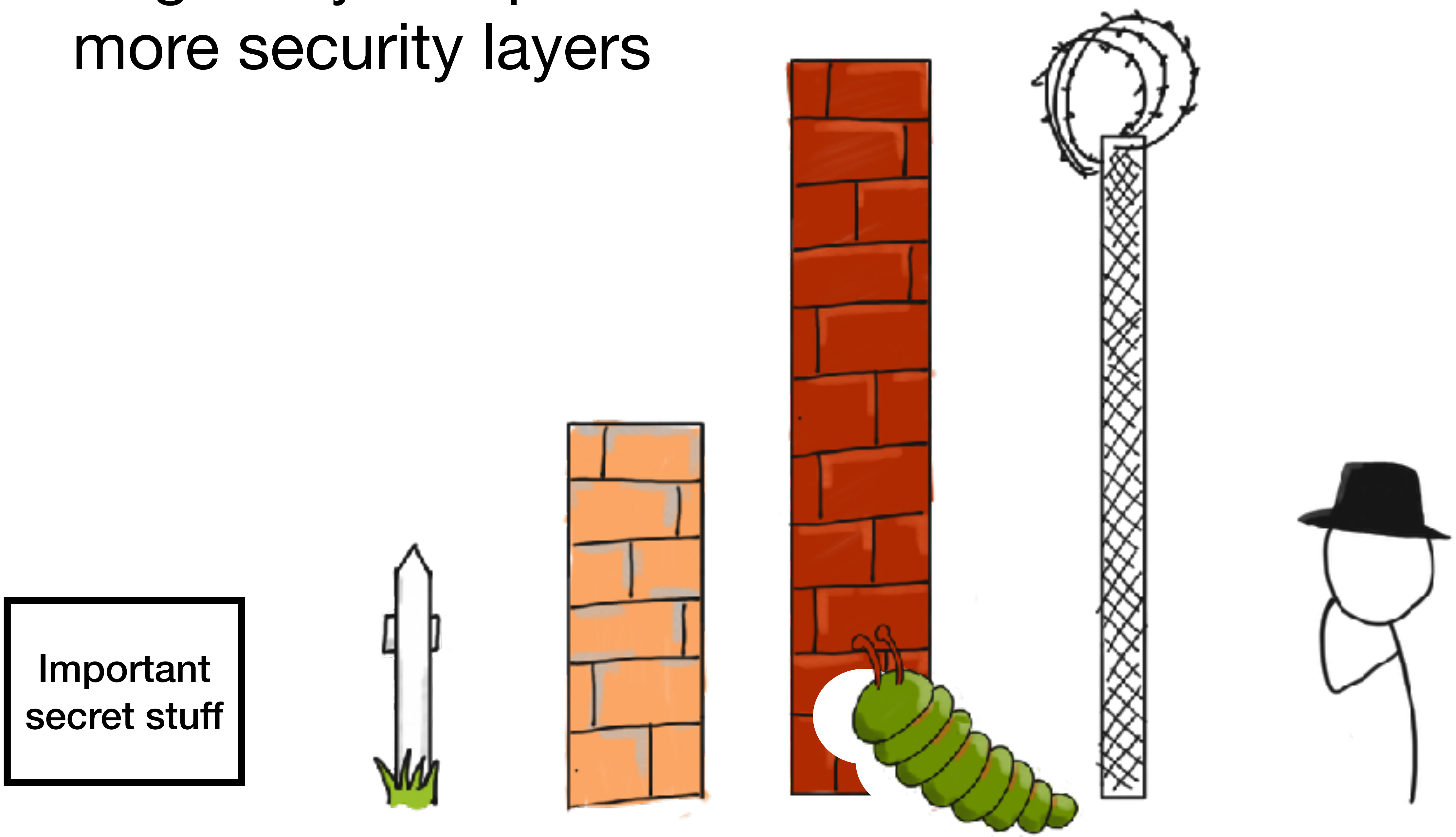


When is a fix urgent?

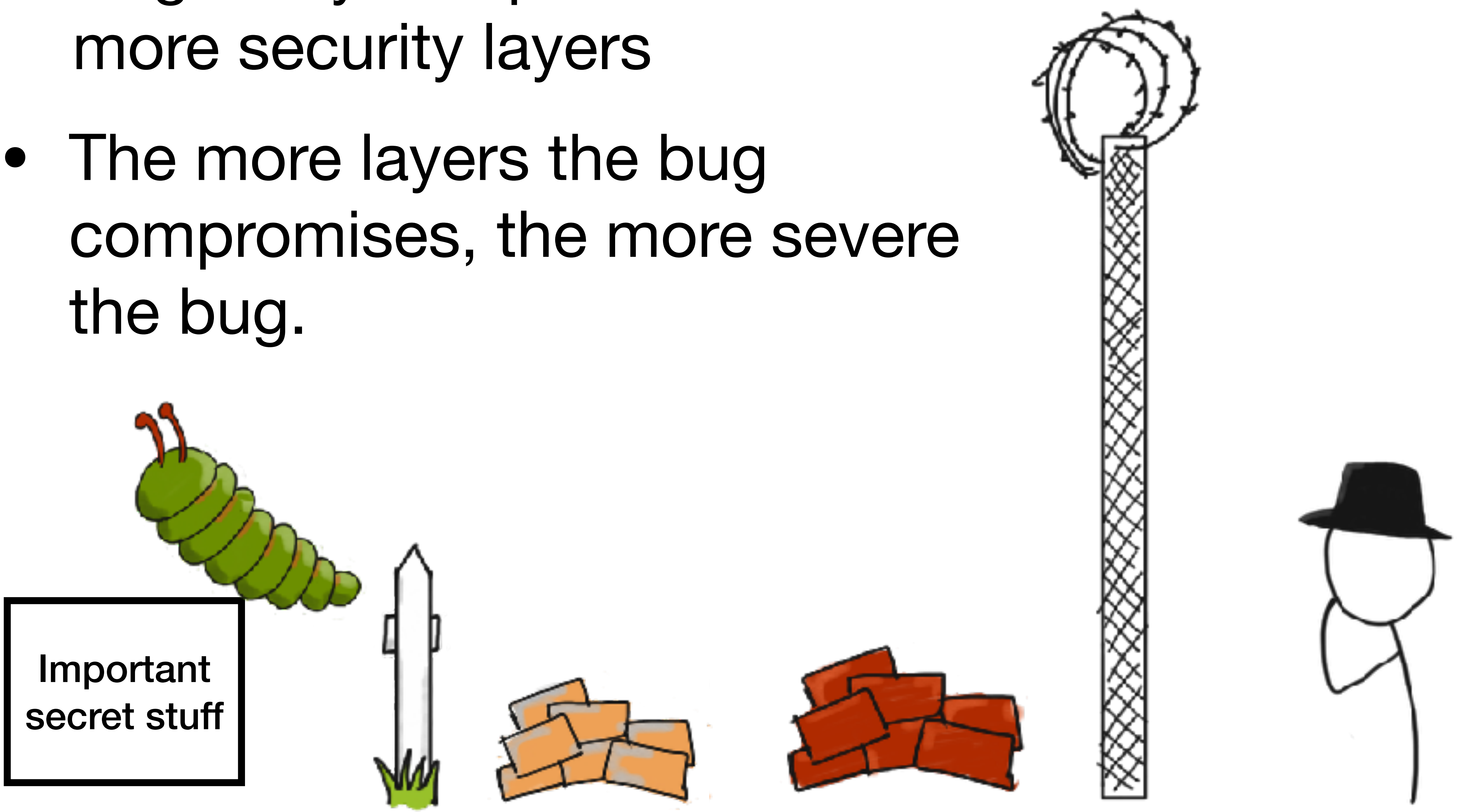
- Well-engineered systems are built with defence in depth



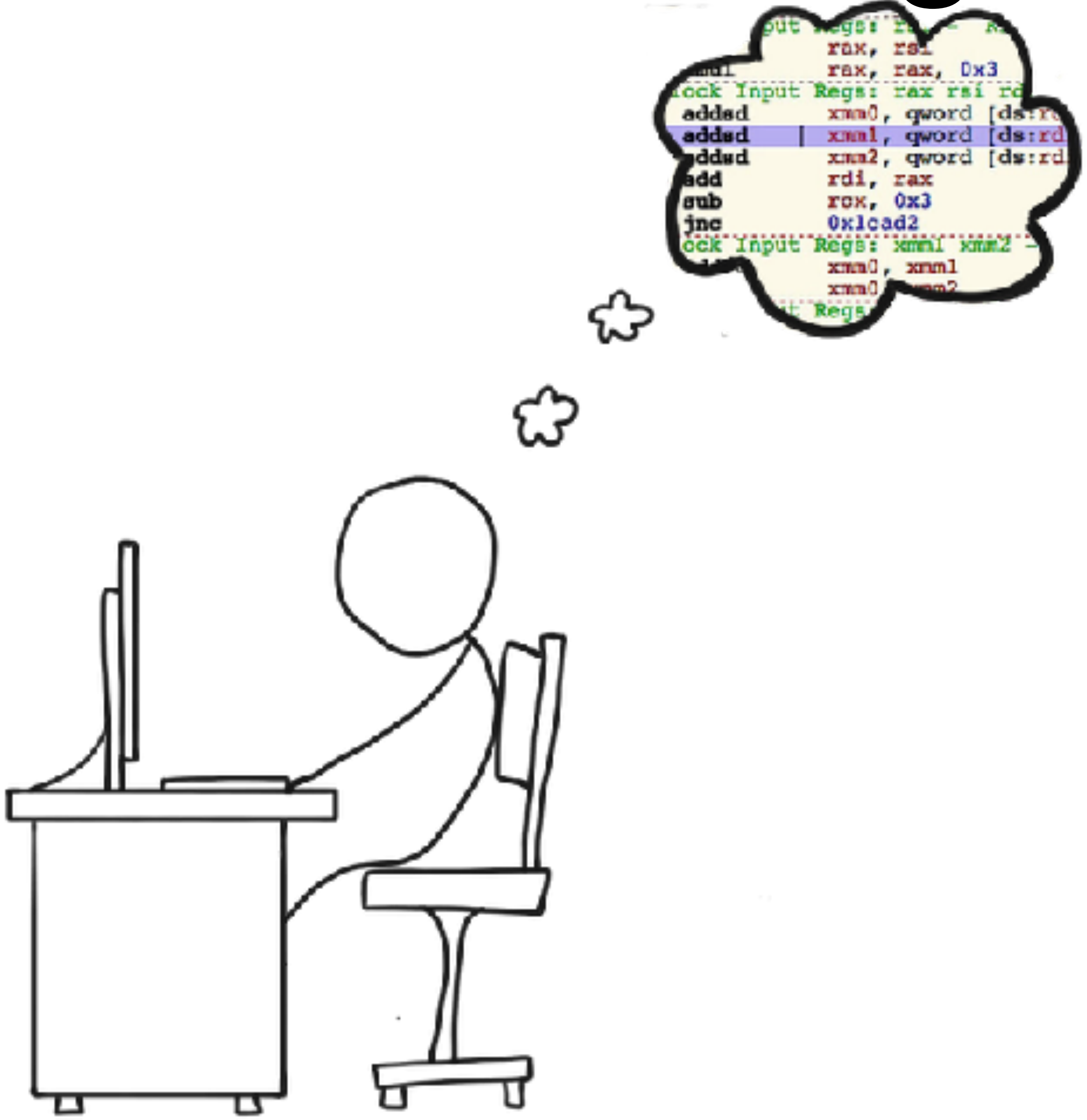
- Well-engineered systems are built with defence in depth
- Bugs may compromise one or more security layers



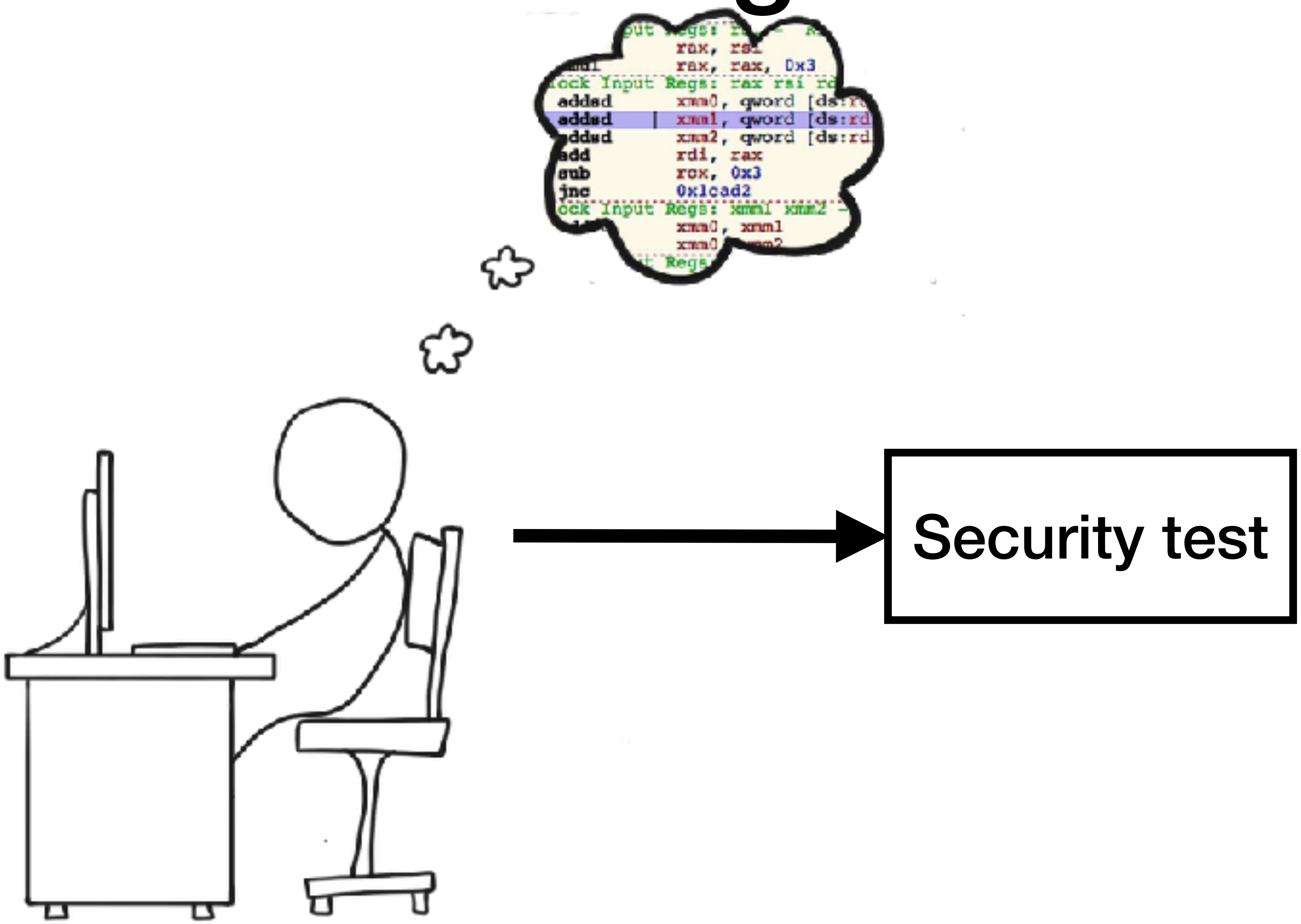
- Well-engineered systems are built with defence in depth
- Bugs may compromise one or more security layers
- The more layers the bug compromises, the more severe the bug.



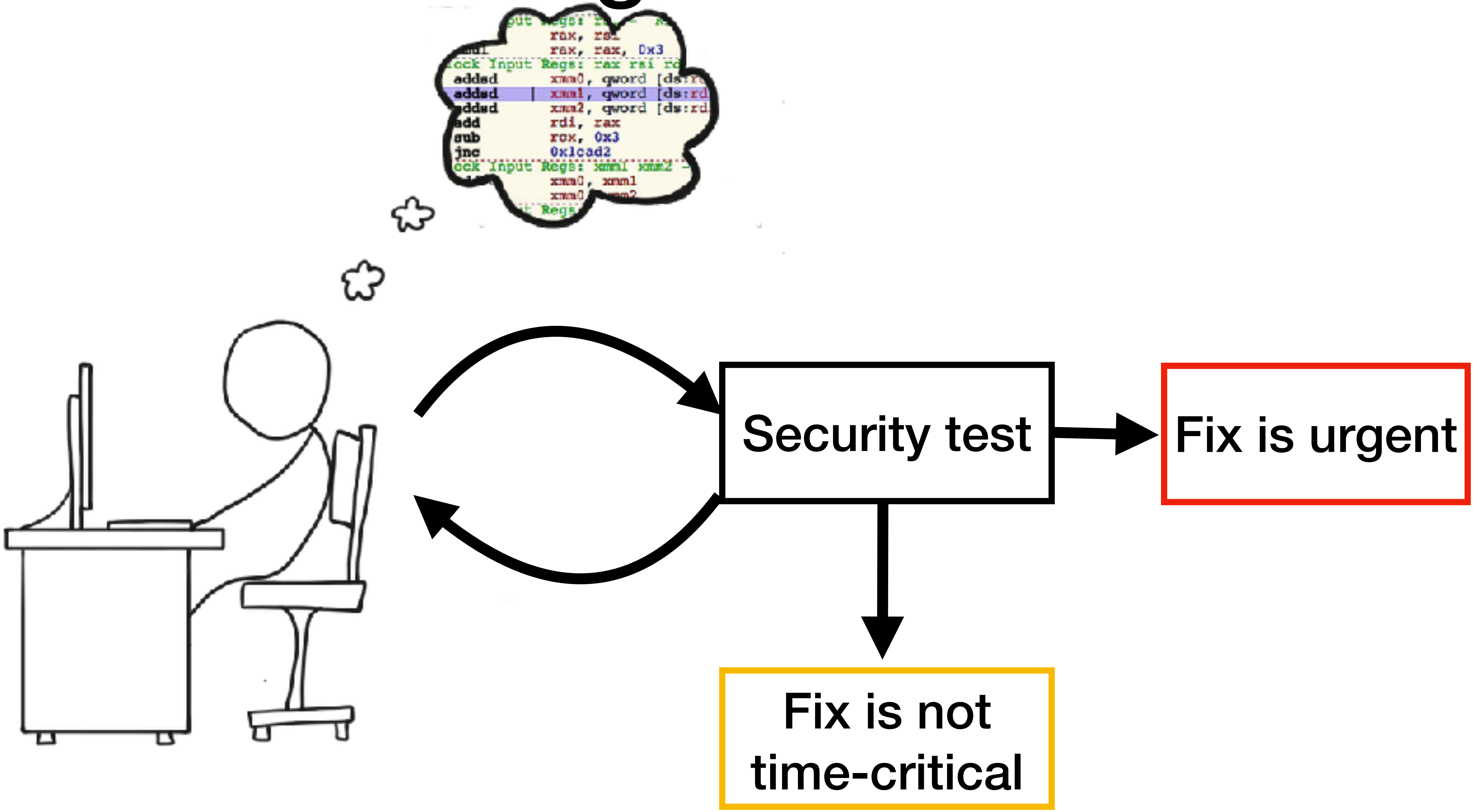
How do we determine if a fix is urgent?



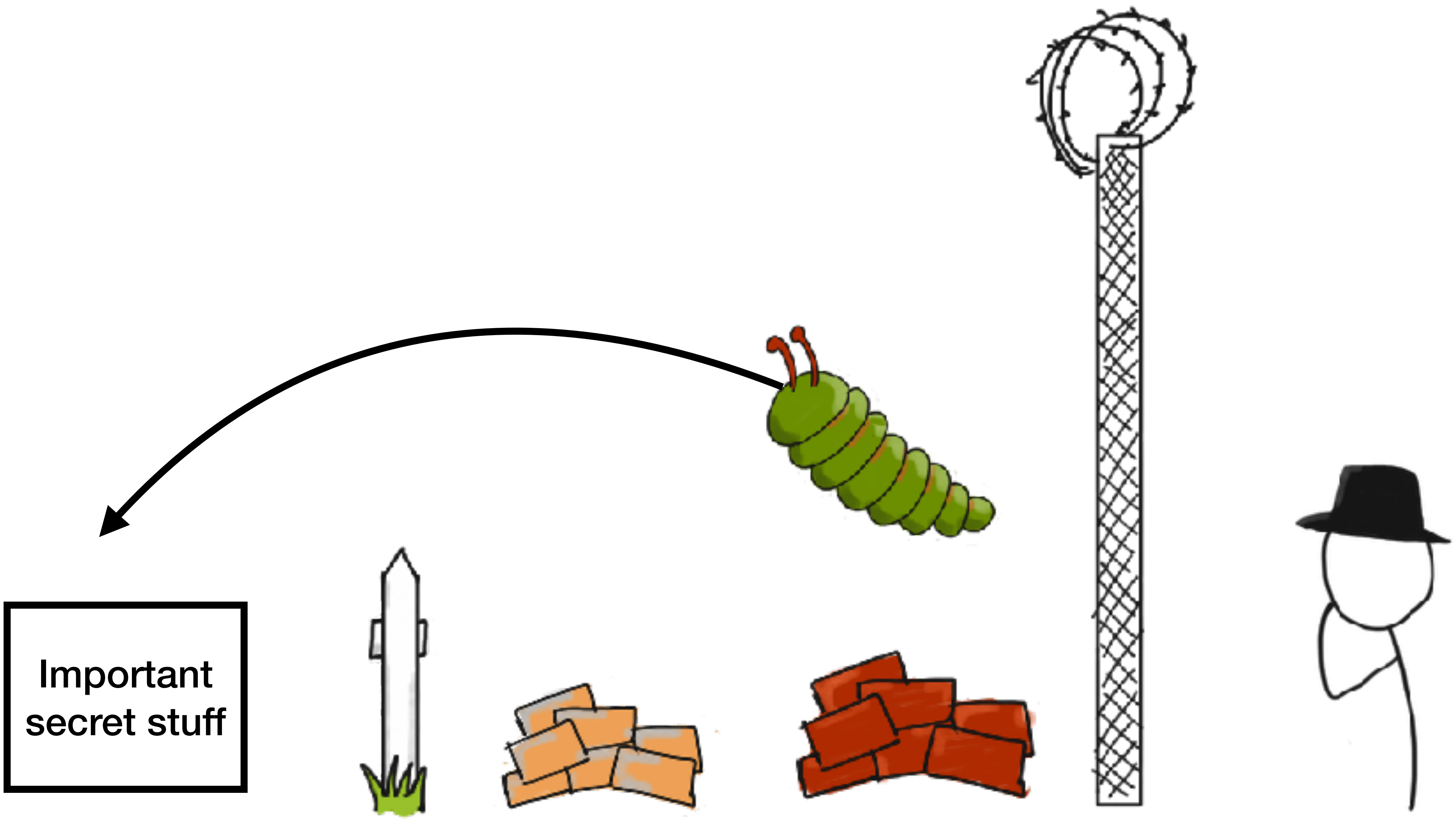
How do we determine if a fix is urgent?



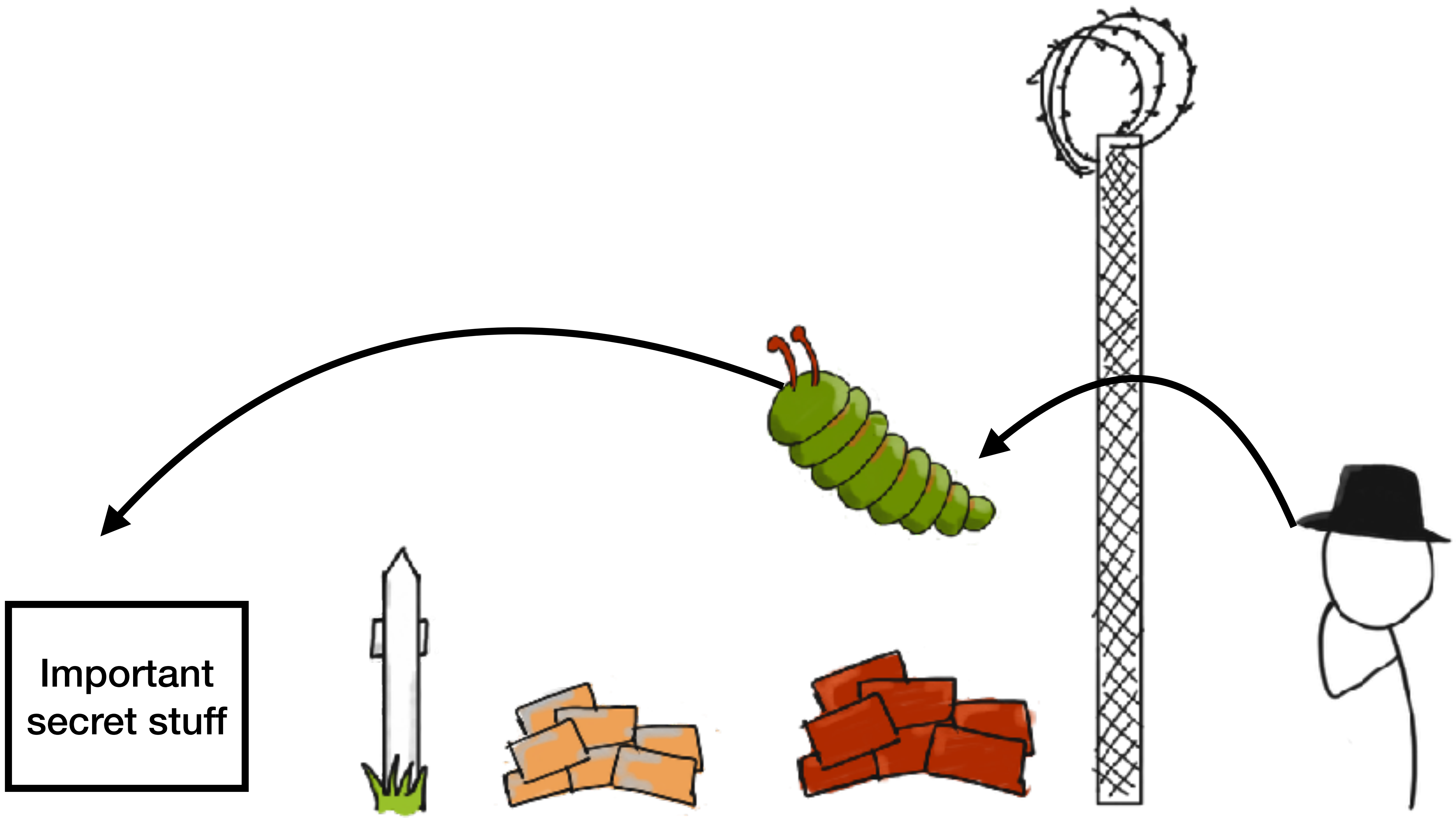
How do we determine if a fix is urgent?



Using model checking


Important
secret stuff

Security tests establish reachability of the bug



Reachability assertion

ISSUE DESCRIPTION
=================

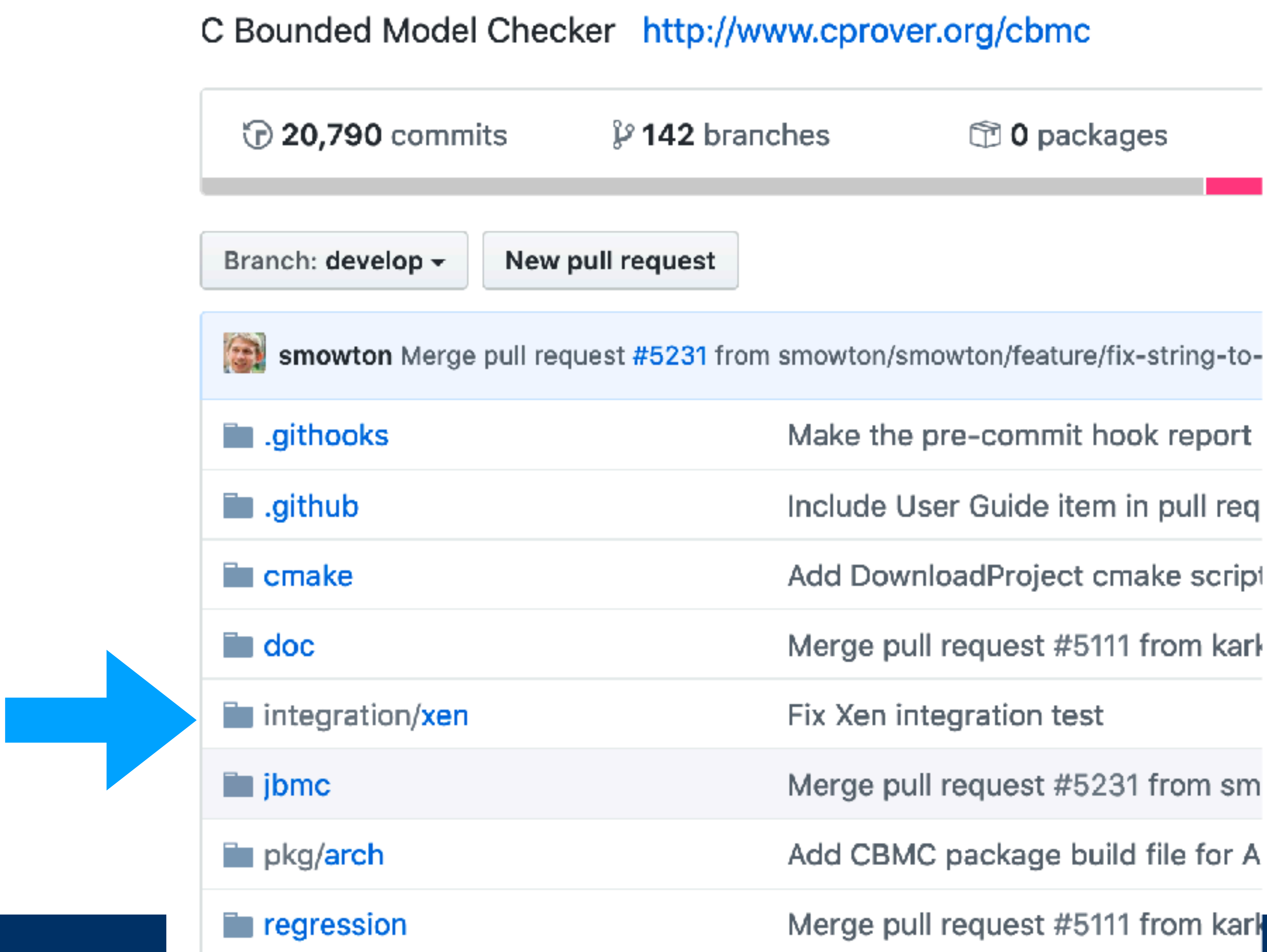
The x86 instruction CMPXCHG8B is supposed to ignore legacy operand size overrides; it only honors the REX.W override (making it CMPXCHG16B). So, the operand size is always 8 or 16.

When st [redacted] on emulat [redacted] was relied [redacted] eral, operan [redacted]

```
assert(op_bytes==8 || op_bytes==16);
```

As a result, if a guest uses a supposedly-ignored operand size prefix, a small amount of hypervisor stack data is leaked to the guests: a 96 bit leak to guests running in 64-bit mode; or, a 32 bit leak to other guests.

Can we use CBMC?



C Bounded Model Checker http://www.cprover.org/cbmc
20,790 commits
142 branches
0 packages
Branch: develop
New pull request
smowton Merge pull request #5231 from smowton/smowton/feature/fix-string-to-
.githooks
Make the pre-commit hook report
.github
Include User Guide item in pull req
cmake
Add DownloadProject cmake scrip
doc
Merge pull request #5111 from kar
integration/xen
Fix Xen integration test
jbmc
Merge pull request #5231 from sm
pkg/arch
Add CBMC package build file for A
regression
Merge pull request #5111 from kar

Can we use CBMC?

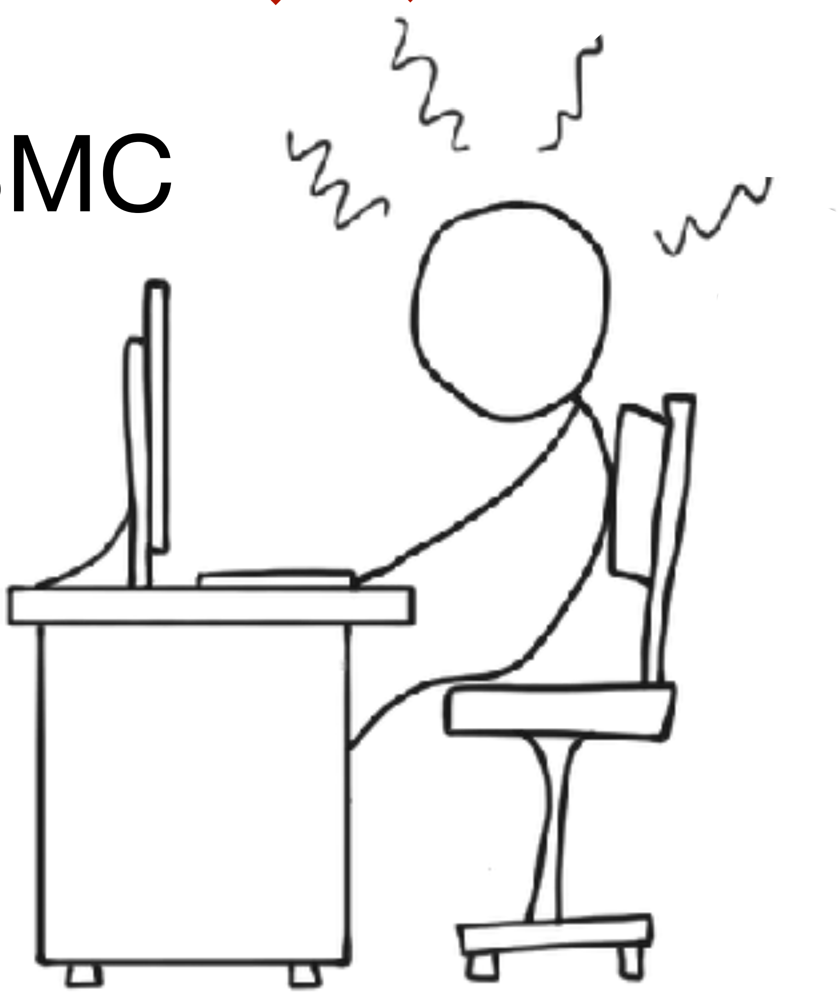
- CBMC
- Reachability slicer + CBMC
- Global init slicer + CBMC
- Full slicer + CBMC

Can we use CBMC?

- CBMC ✗
- Reachability slicer + CBMC
- Global init slicer + CBMC
- Full slicer + CBMC

Can we use CBMC?

- CBMC ✗
- Reachability slicer + CBMC ✗
- Global init slicer + CBMC
- Full slicer + CBMC



Can we use CBMC?

- CBMC ✗
- Reachability slicer + CBMC ✗
- Global init slicer + CBMC ✗
- Full slicer + CBMC

Can we use CBMC?

- CBMC ✗
- Reachability slicer + CBMC ✗
- Global init slicer + CBMC ✗
- Full slicer + CBMC ✗

Why is it hard?

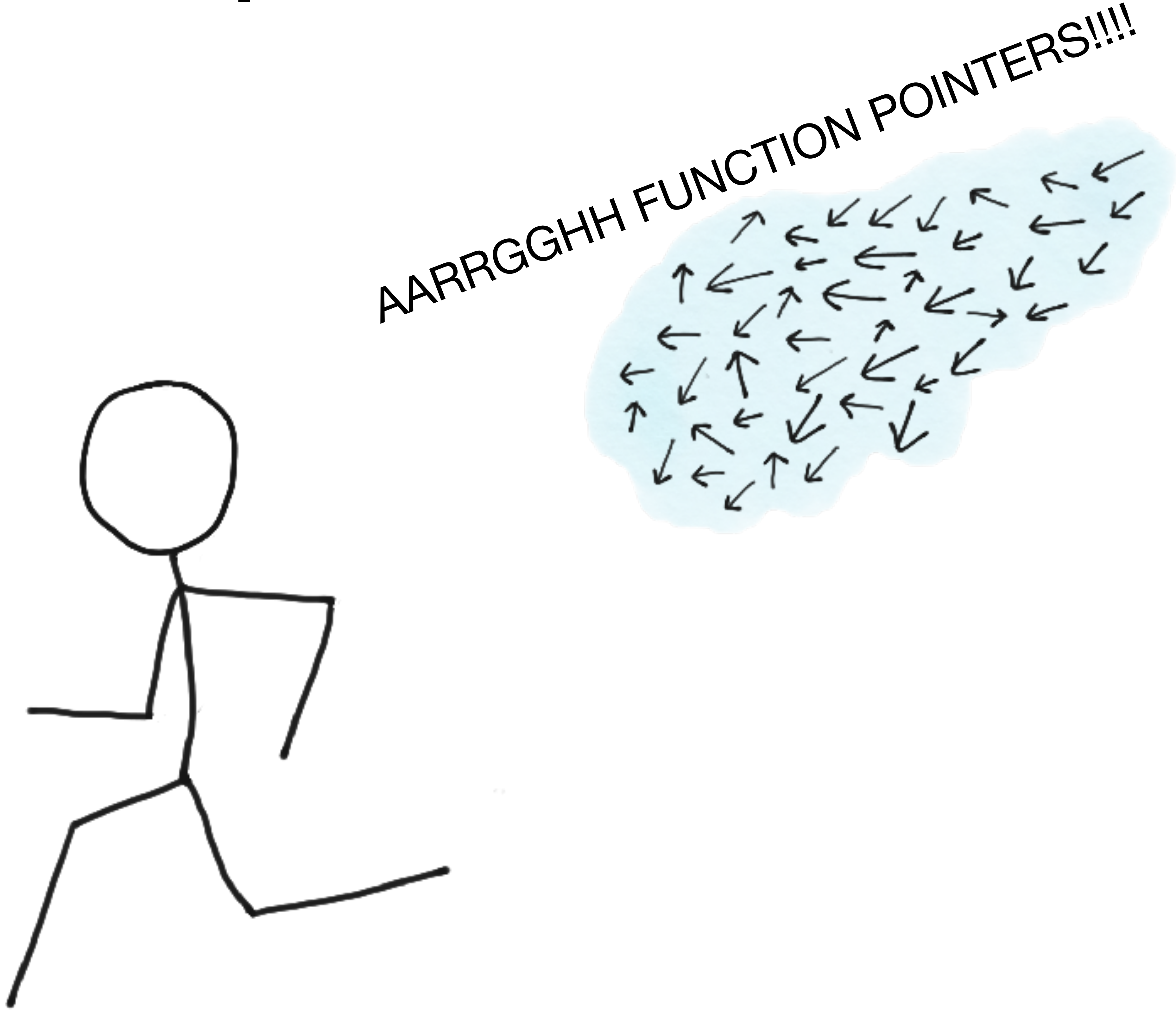
- Big(ish) code base, long CEX
- Function pointers everywhere
- Function pointers configured at boot and we can't analyse boot code
- Assembly code

Solution?

- Modelled assembly code by hand
- Alias analysis based function-pointer removal
- Aggressive program slicer
- Approximate removed code
- Spliced in code harnesses in order to start analysis mid-way through the code

Modelling assembly code

Function pointer removal

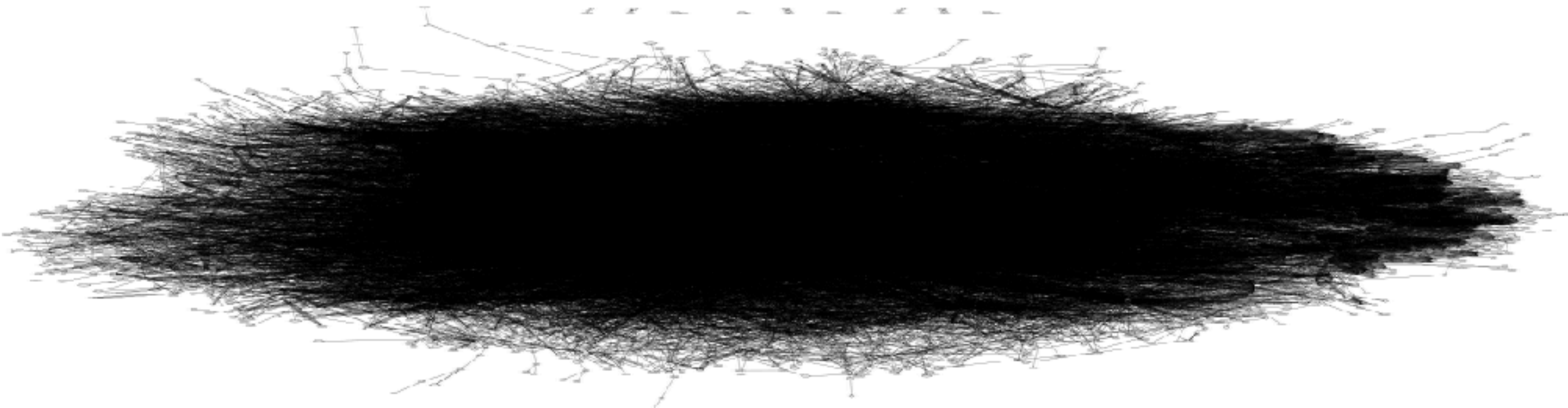


Solution?

- Modelled assembly code by hand
- Alias analysis based function-pointer removal
- Aggressive program slicer
- Approximate removed code
- Spliced in code harnesses in order to start analysis mid-way through the code

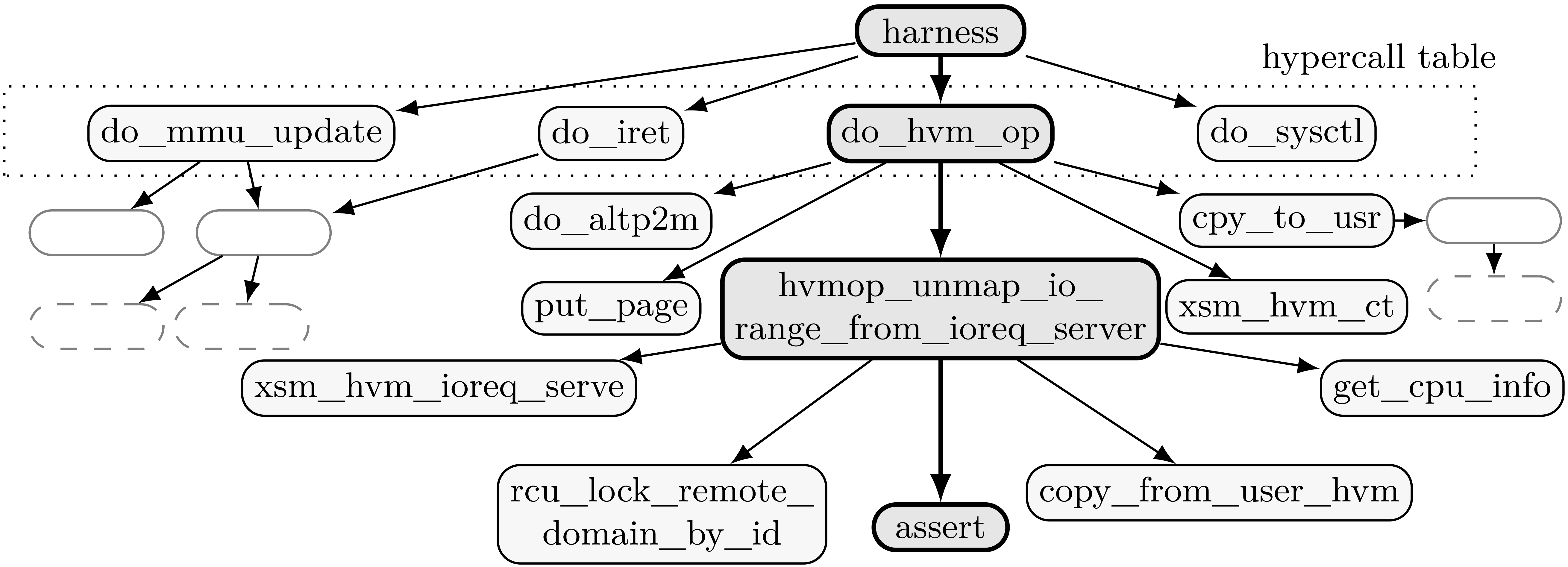
“Aggressive” slicer

- Analyses part of the code base
- Approximates the remaining code
- Tailored by engineer input



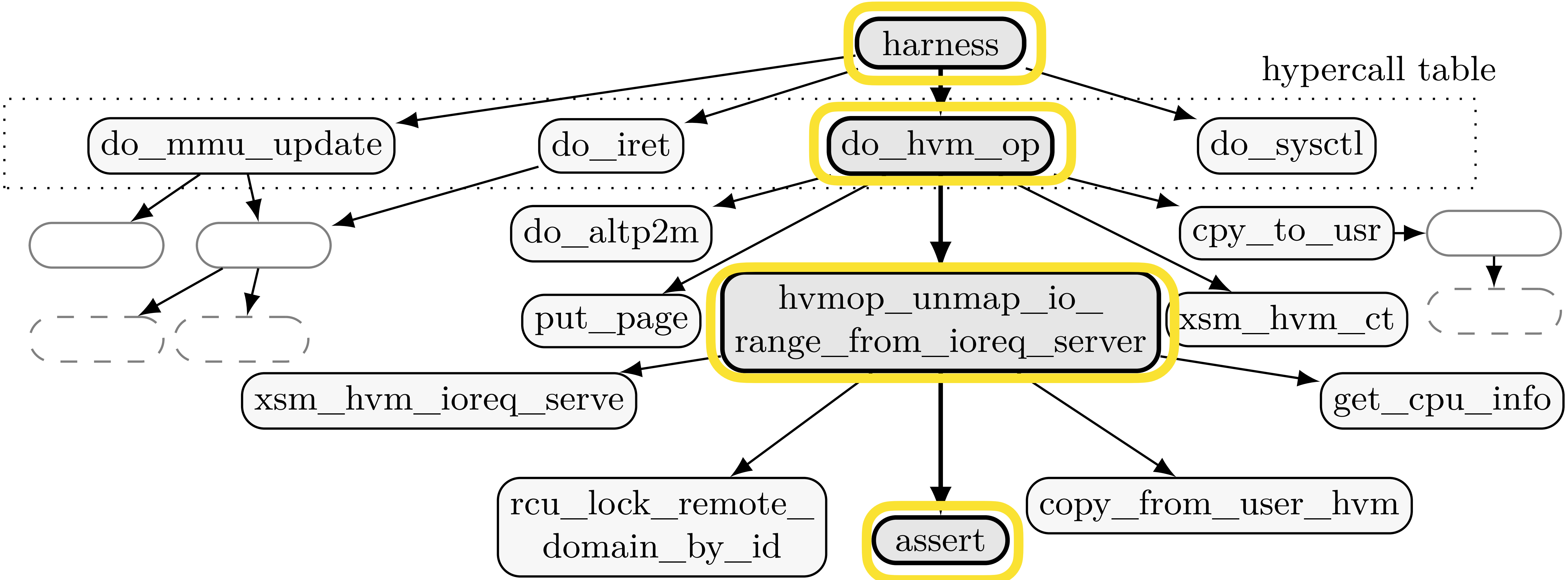
180,000+ function calls

“Aggressive” slicer



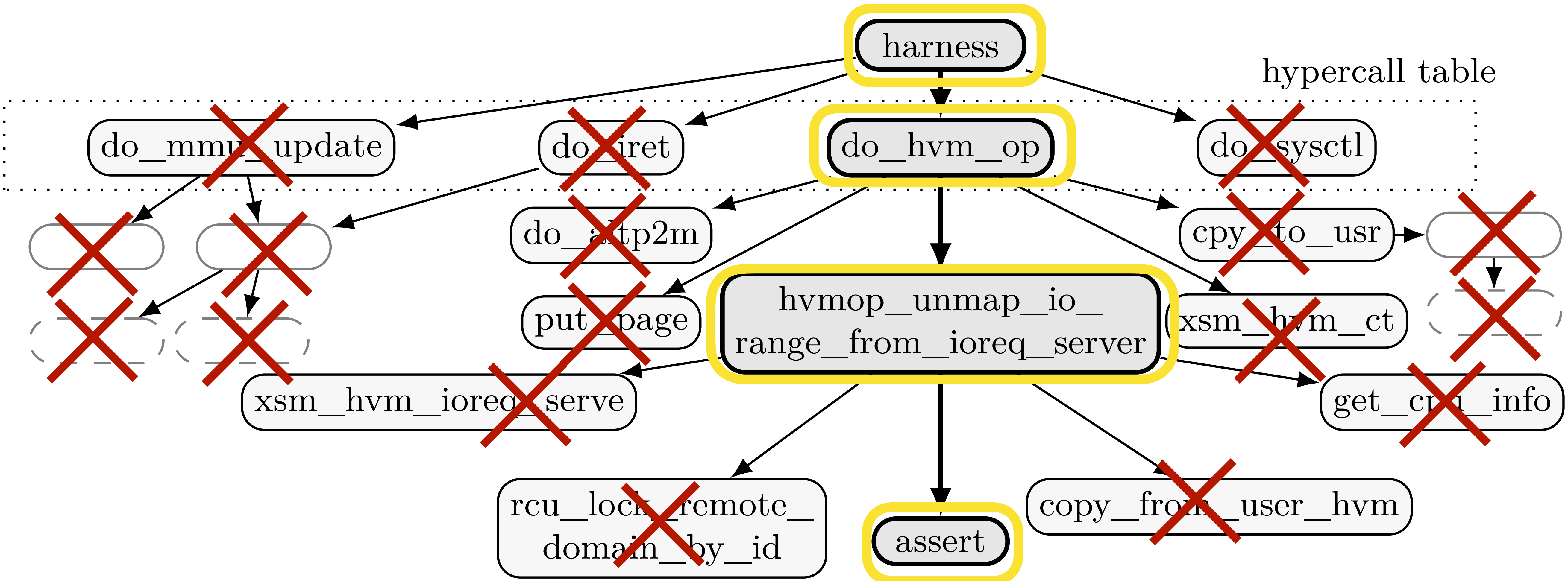
Construct call graph

“Aggressive” slicer



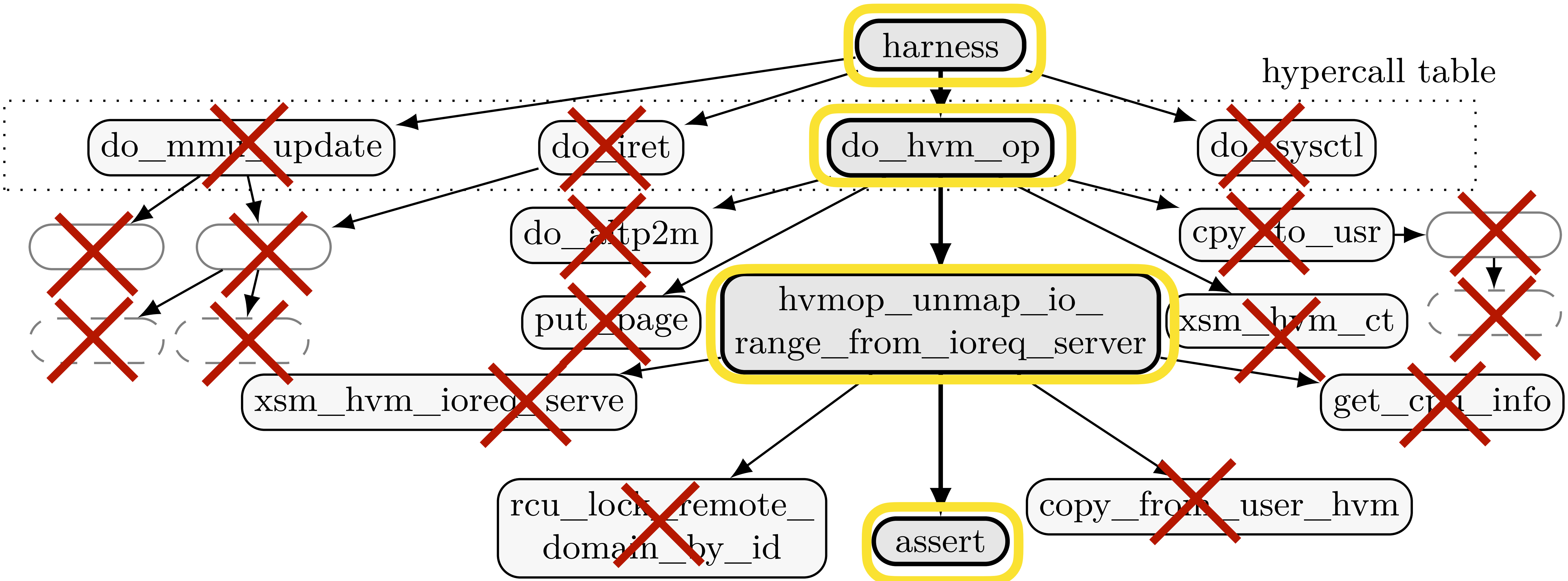
Find direct paths

“Aggressive” slicer



Mark functions not on direct paths to be havoc’d

“Aggressive” slicer



Havoc functions

Havoc-ing functions

```
int function_with_no_body(int *a, int *b);
```

Unknown return value

Arguments passed-by-pointer

Havoc-ing functions

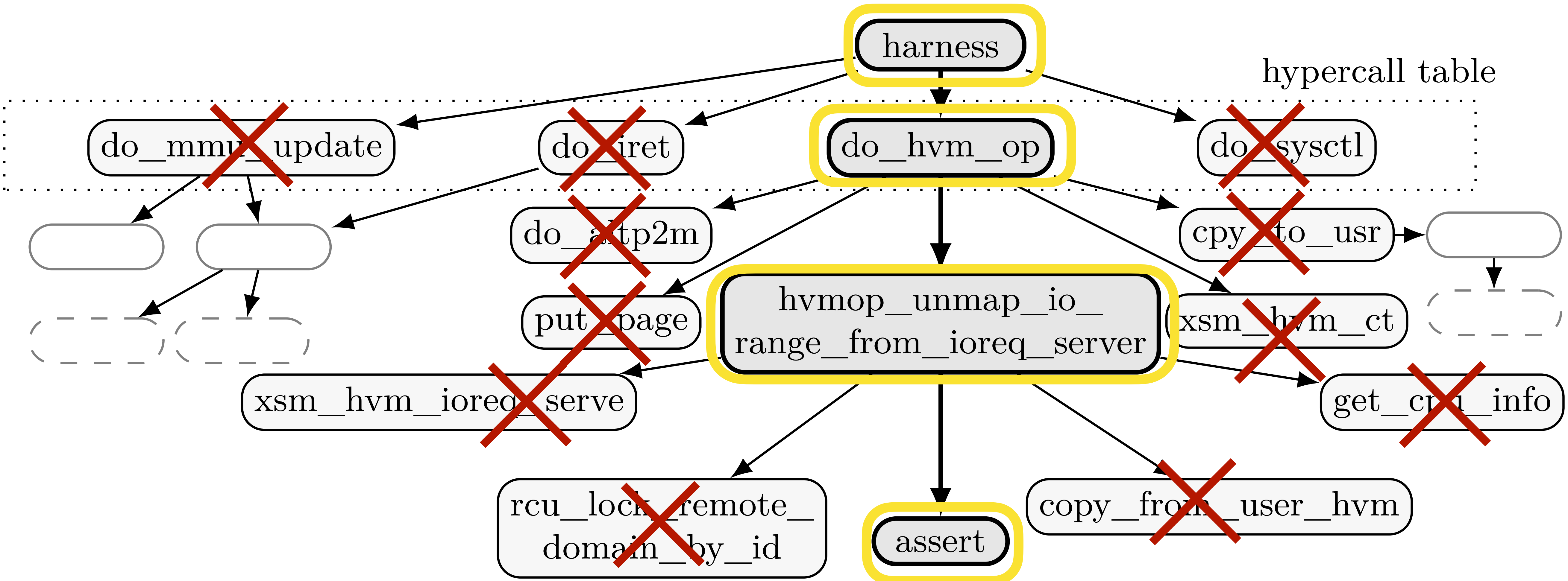
```
int function_with_no_body(int *a, int *b);
```

Unknown return value

Arguments passed-by-pointer

```
int function_with_no_body(int *a, int *b)
{
  int result = nondet_int();
  int a = nondet_int();
  int b = nondet_int();
  return result;
}
```

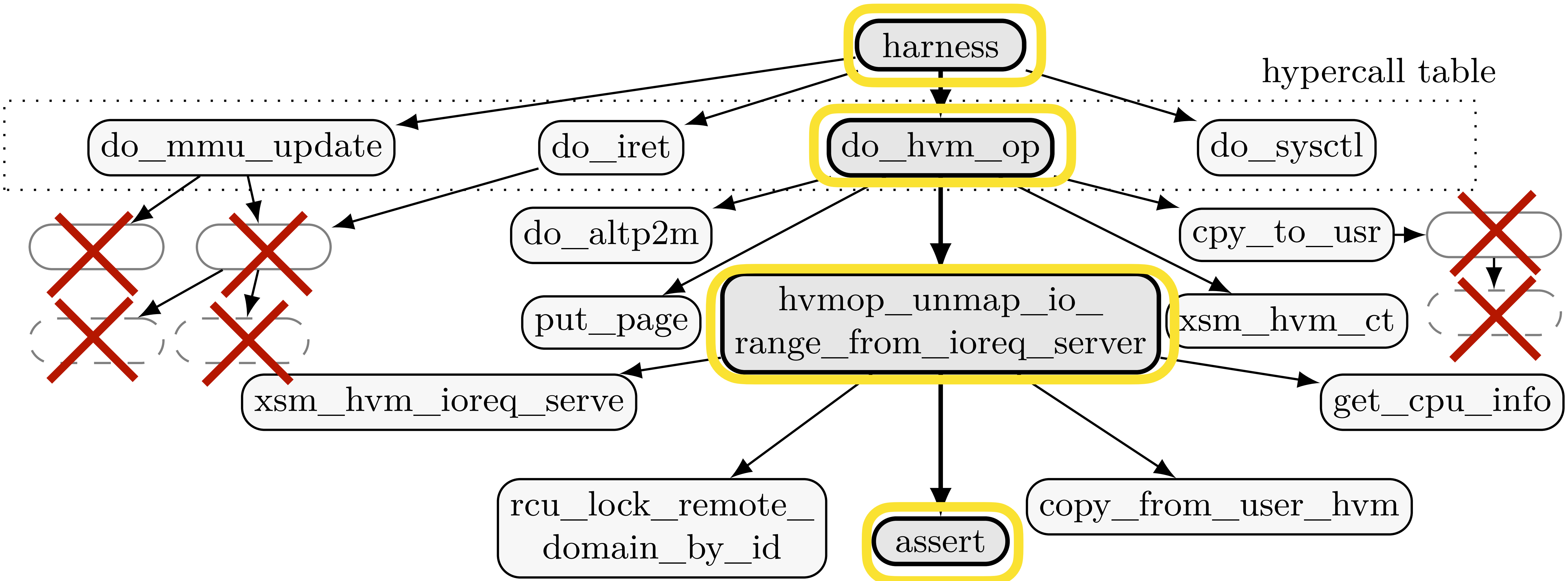
“Aggressive” slicer



Remove unreachable functions

“Aggressive” slicer configurations

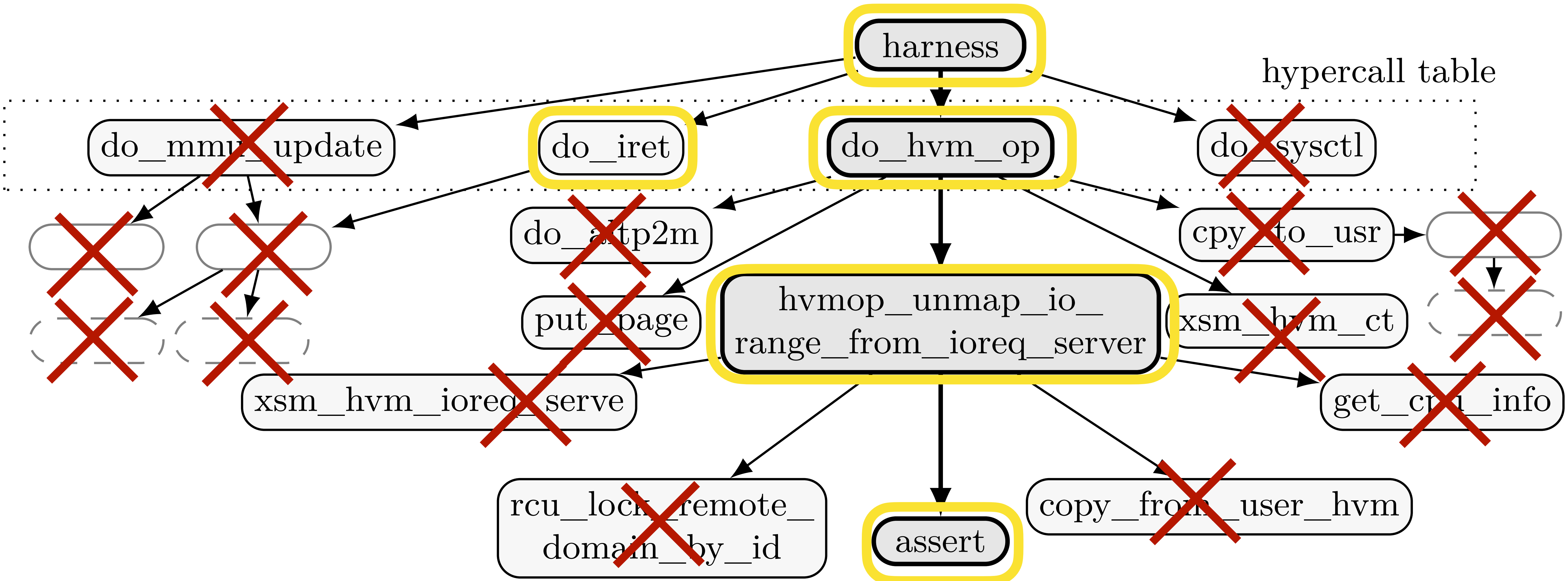
- Preserve all direct paths or shortest path
- Preserve functions N function calls away form preserved paths
- Preserve functions by name
- Remove specific functions



Havoc functions only more than 1 function call away from direct paths

“Aggressive” slicer configurations

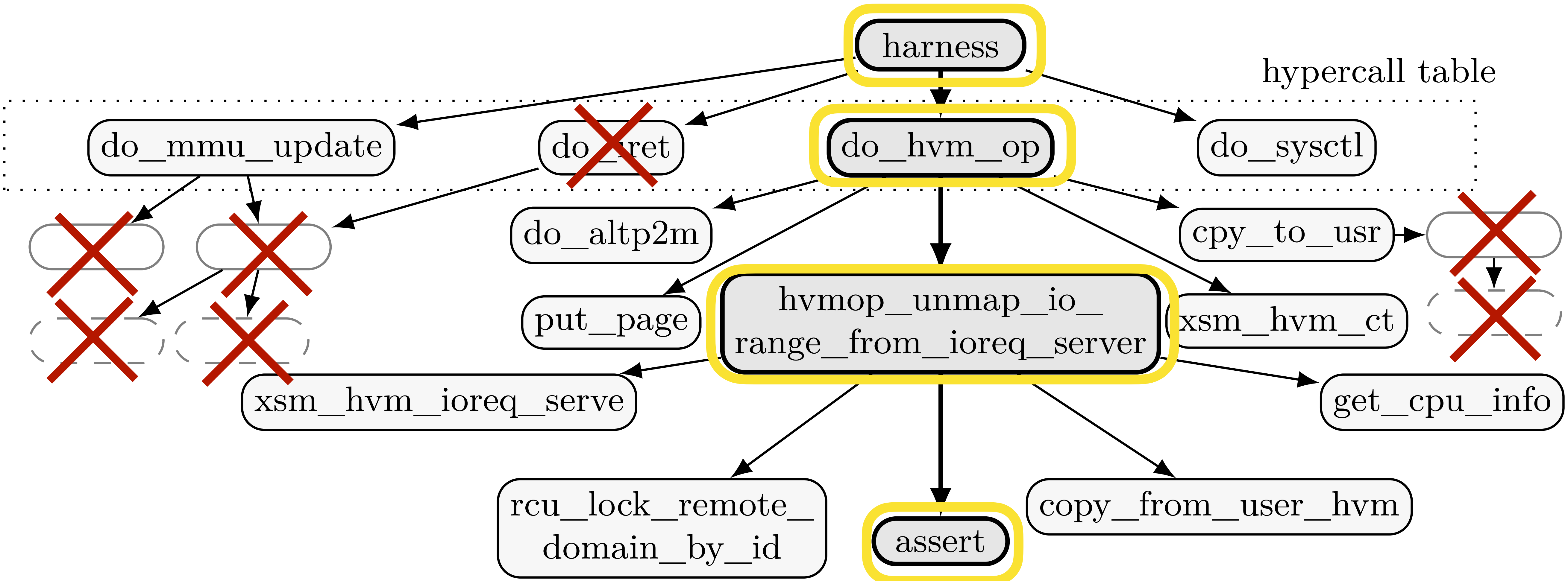
- Preserve all direct paths or shortest path
- Preserve functions N function calls away form preserved paths
- Preserve functions by name
- Remove specific functions



Do not havoc do_iret

“Aggressive” slicer configurations

- Preserve all direct paths or shortest path
- Preserve functions N function calls away form preserved paths
- Preserve functions by name
- Havoc specific functions



Havoc do_iret

Slicing algorithm:

Approximating_Slice (CFG g, node entry, node target, bool direct, int distance)

FP := remove_function_pointers(g)
CG := compute_call_graph(FP)
DP := get_direct_paths(CG, entry, target)
DP := shortest_path(DP) **if** $\neg$ direct **else** DP
mark_for_havoc $= \emptyset$
for node n **in** FP:
    if distance(FP, DP, n) $>$ distance:
        mark_for_havoc := mark_for_havoc $\cup \{n\}$
for node n **in** mark_for_havoc:
    havoc_object(n)

Figure 1. Approximating slicing is applied to input program represented by its control-flow graph g, and configurable in the entry- and target nodes, whether or not to consider all direct paths, and the maximum distance.

Slicing algorithm:

Approximating_Slice (CFG g, node entry, node target, bool direct, int distance)

```
FP := remove_function_pointers(g)
CG := compute_call_graph(FP)
DP := get_direct_paths(CG, entry, target)
DP := shortest_path(DP) if ¬ direct else DP
mark_for_havoc = ∅
for node n in FP:
    if distance(FP, DP, n) > distance:
        mark_for_havoc := mark_for_havoc ∪ {n}
for node n in mark_for_havoc:
    havoc_object(n)
```

Figure 1. Approximating slicing is applied to input program represented by its control-flow graph g, and configurable in the entry- and target nodes, whether or not to consider all direct paths, and the maximum distance.

Slicing algorithm:

Approximating_Slice (CFG g, node entry, node target, bool direct, int distance)

FP := remove_function_pointers(g)
CG := compute_call_graph(FP)
DP := get_direct_paths(CG, entry, target)
DP := shortest_path(DP) **if** $\neg$ direct **else** DP
mark_for_havoc $= \emptyset$
for node n **in** FP:
    if distance(FP, DP, n) $>$ distance:
        mark_for_havoc := mark_for_havoc $\cup \{n\}$
for node n **in** mark_for_havoc:
    havoc_object(n)

Figure 1. Approximating slicing is applied to input program represented by its control-flow graph g, and configurable in the entry- and target nodes, whether or not to consider all direct paths, and the maximum distance.

Slicing algorithm:

Approximating_Slice (CFG g, node entry, node target, bool direct, int distance)

FP := remove_function_pointers(g)
CG := compute_call_graph(FP)
DP := get_direct_paths(CG, entry, target)
DP := shortest_path(DP) **if** $\neg$ direct **else** DP
mark_for_havoc $= \emptyset$
for node n **in** FP:
  if distance(FP, DP, n) $>$ distance:
    mark_for_havoc := mark_for_havoc $\cup \{n\}$
for node n **in** mark_for_havoc:
  havoc_object(n)

Figure 1. Approximating slicing is applied to input program represented by its control-flow graph g, and configurable in the entry- and target nodes, whether or not to consider all direct paths, and the maximum distance.

Slicing algorithm:

Approximating_Slice (CFG g, node entry, node target, bool direct, int distance)

FP := remove_function_pointers(g)
CG := compute_call_graph(FP)
DP := get_direct_paths(CG, entry, target)
DP := shortest_path(DP) **if** $\neg$ direct **else** DP
mark_for_havoc $= \emptyset$
for node n **in** FP:
    if distance(FP, DP, n) $>$ distance:
        mark_for_havoc := mark_for_havoc $\cup \{n\}$
for node n **in** mark_for_havoc:
    havoc_object(n)

Figure 1. Approximating slicing is applied to input program represented by its control-flow graph g, and configurable in the entry- and target nodes, whether or not to consider all direct paths, and the maximum distance.

Starting mid-way through the code

Contains function pointers

```
int
x86_emulate(
    struct x86_emulate_ctxt *ctxt,
    const struct x86_emulate_ops  *ops)
{
```

Use a “harness” to approximate the environment

```c
static int harness_read(
    enum x86_segment seg,
    unsigned long offset,
    void *p_data,
    unsigned int bytes,
    struct x86_emulate_ctxt *ctxt)
{
  if(bytes==1)
    ((char *)p_data)[0]=nondet_char();
  else if(bytes==2)
    ((short *)p_data)[0]=nondet_short();
  else if(bytes==4)
    ((int *)p_data)[0]=nondet_int();
  else if(bytes==8)
    ((long long *)p_data)[0]=nondet_longlong();
  else if(bytes==10)
  {
  }
  else
    __CPROVER_assert(0, "read size");
}
```

Incorporate modelled functions

Make all pointers to data structures valid

```c
int main()
{
  struct cpu_user_regs harness_regs;
  struct x86_emulate_ctxt harness_ctxt;
  harness_ctxt.regs=&harness_regs;
  harness_ctxt.addr_size=64;
  x86_emulate(&harness_ctxt, &harness_ops);
}
```

```c
static const struct x86_emulate_ops harness_ops = {
    .read        = harness_read,
    .insn_fetch  = harness_read,
    .write       = harness_write,
    .cmpxchg     = harness_cmpxchg,
};
```

Make all function pointers valid

Hypercall table harness

```c
#define ARGS(x, n)                          \
    [__HYPERVISOR_ ## x ]={n, n}
#define COMP(x, n, c)                       \
    [__HYPERVISOR_ ## x ]={n, c}

const hypercall_args_t
  hypercall_args_table[NR_hypercalls] =
{
    ARGS(set_trap_table, 1),
    ARGS(mmu_update, 4),
    ARGS(set_gdt, 2),
  ...

#define HYPERCALL(x)                        \
    [ __HYPERVISOR_ ## x ] =                \
      { (hypercall_fn_t *) do_ ## x,        \
        (hypercall_fn_t *) do_ ## x }
#define COMPAT_CALL(x)                      \
    [ __HYPERVISOR_ ## x ] =                \
        {(hypercall_fn_t *) do_ ## x,       \
        (hypercall_fn_t *) compat_ ## x }
  ...

static const hypercall_table_t
    pv_hypercall_table[] = {
    COMPAT_CALL(set_trap_table),
    HYPERCALL(mmu_update),
    COMPAT_CALL(set_gdt),
  ...
```

```c
void do_hypercall()
{
  int nondet;
  switch(nondet)
  {
  case 1:
    XEN_GUEST_HANDLE (const_trap_info_t) traps1;
    do_set_trap_table(traps1);
    break;
  case 2:
    XEN_GUEST_HANDLE (mmu_update_t) ureqs2;
    unsigned int count2;
    XEN_GUEST_HANDLE (uint) pdone2;
    unsigned int foreigndom2;
    do_mmu_update(ureqs2, count2, pdone2, foreigndom2);
    break;
  case 3:
    XEN_GUEST_HANDLE (ulong) frame_list3;
    unsigned int entries3;
    do_set_gdt(frame_list3, entries3);
  ...
```

Can we use CBMC now?

Yes…

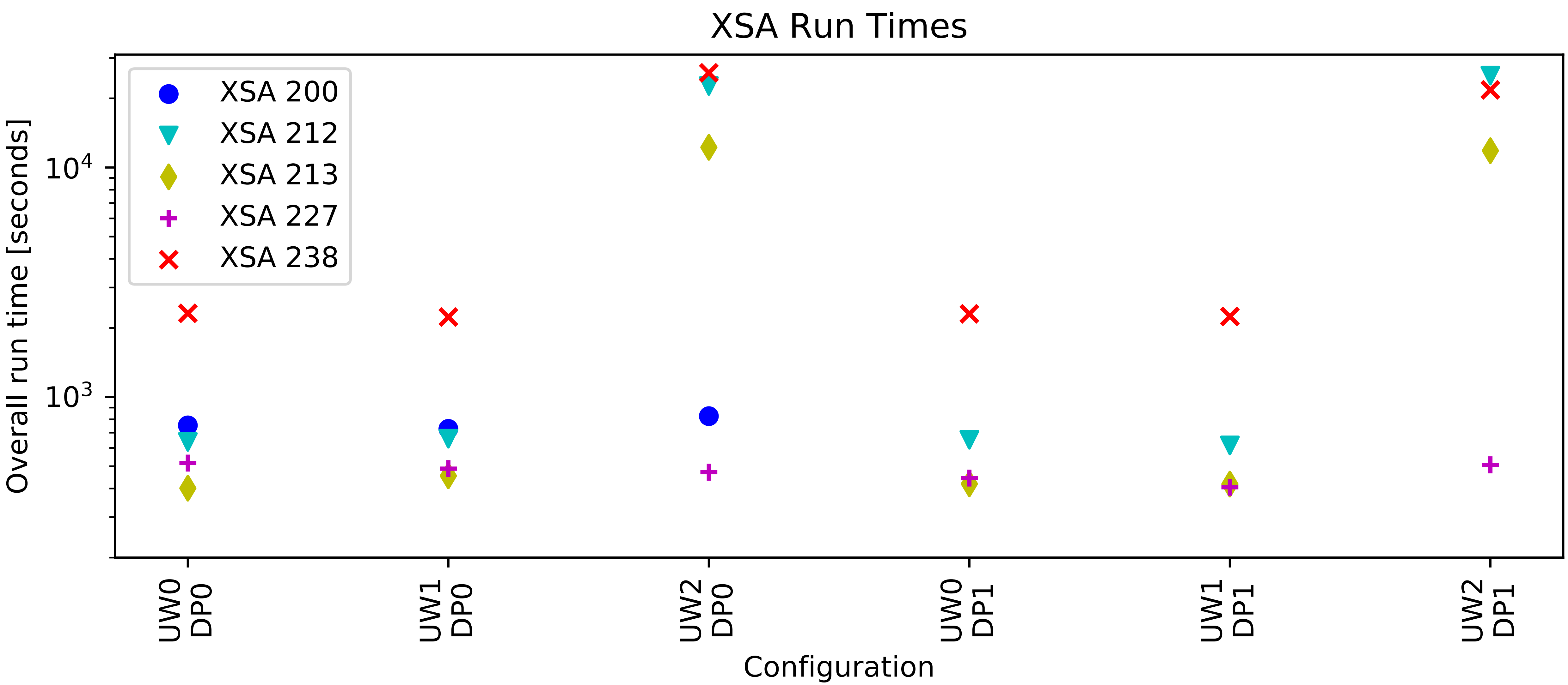


Figure 3. Run time of the overall approach for selected configurations that finish within 8 hours. We fixed the parameters to distance=2, and advanced function pointer removal as well as run full slicing after approximating slicing. Keeping all direct paths (DP1), as well as unwinding loops (UW) during search are altered.

But…

We may produce spurious traces if:

- Modelling is wrong,
- Havoc-ing over-approximates relevant behaviour
- Function pointer assignment is over-approximate

But…

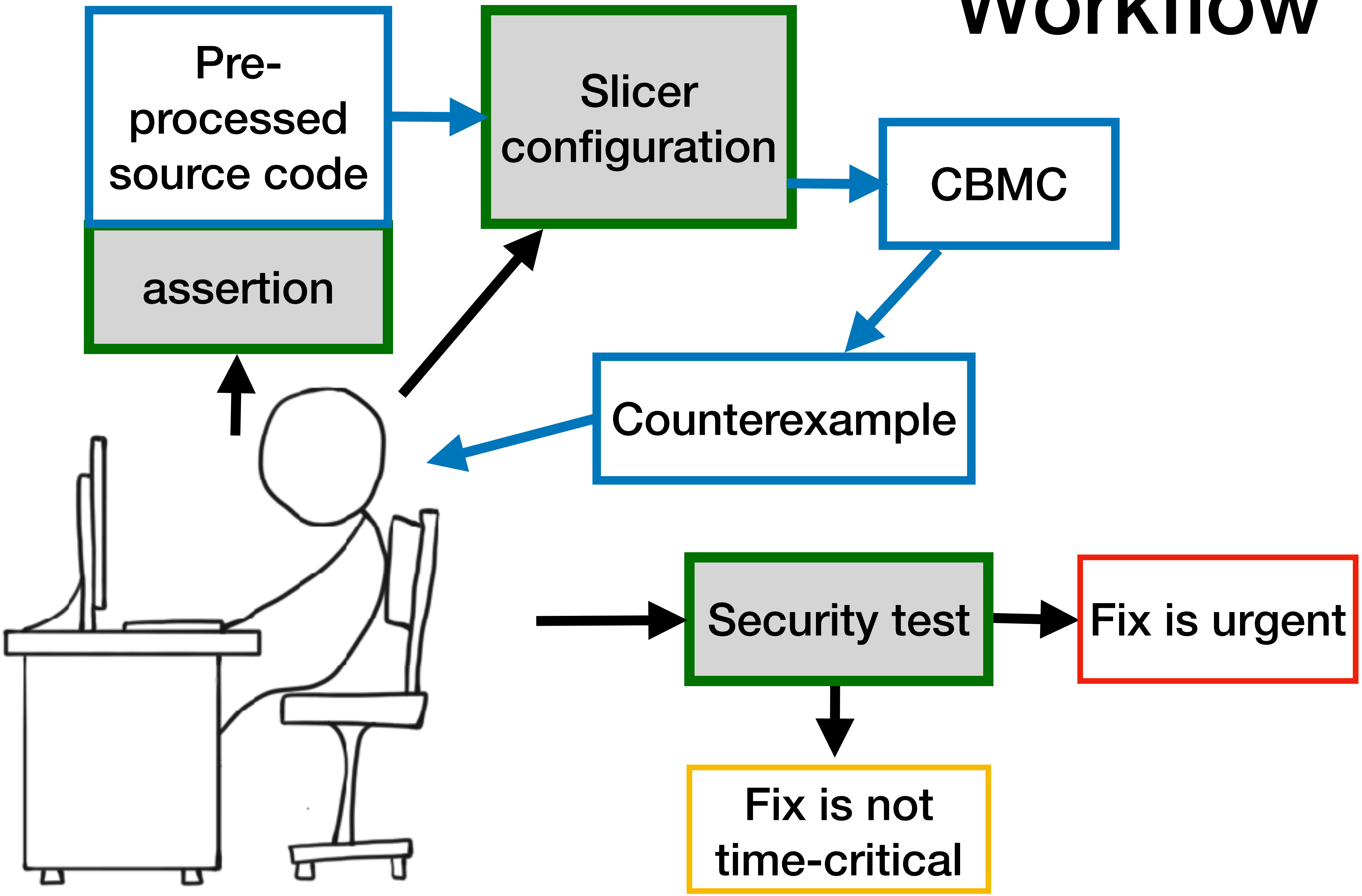
And may miss traces if

- Modelling is wrong,
- Havoc-ing under-approximates relevant behaviour (e.g., modifying globals)
- Not all direct paths are preserved

In practise

- We ran on 5 XSAs
- Ran multiple configurations in parallel using AWS Batch
- We found counterexamples for all 5 XSAs within an hour
- For 4/5 XSAs the counterexamples were useful for test generation

Workflow



Pre-processed source code
assertion
Slicer configuration
CBMC
Counterexample
Security test
Fix is urgent
Fix is not time-critical

Open problems

- Automatically verify counterexample traces
- Synthesise better function approximations
- Automatically generate harnesses

Conclusions

- Plenty of open challenges
- Not complete and not sound BUT still useful!
- We believe this is transferable to other code bases
- Developers get to sleep more

Conclusions

- Contact me: elizabeth.polgreen@ed.ac.uk
- Use our CBMC adaptations: github.com/diffblue/cbmc
- Run our experiments: github.com/nmanthey/xen/tree/FMCAD2020