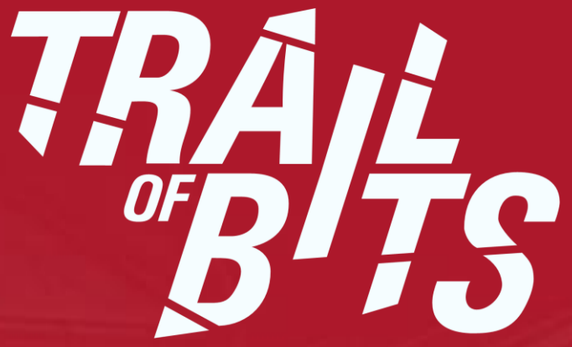


CYBERDYNE: Automatic bug-finding at scale

Peter Goodman

COUNTERMEASURE 2016


TRAIL
OF
BITS

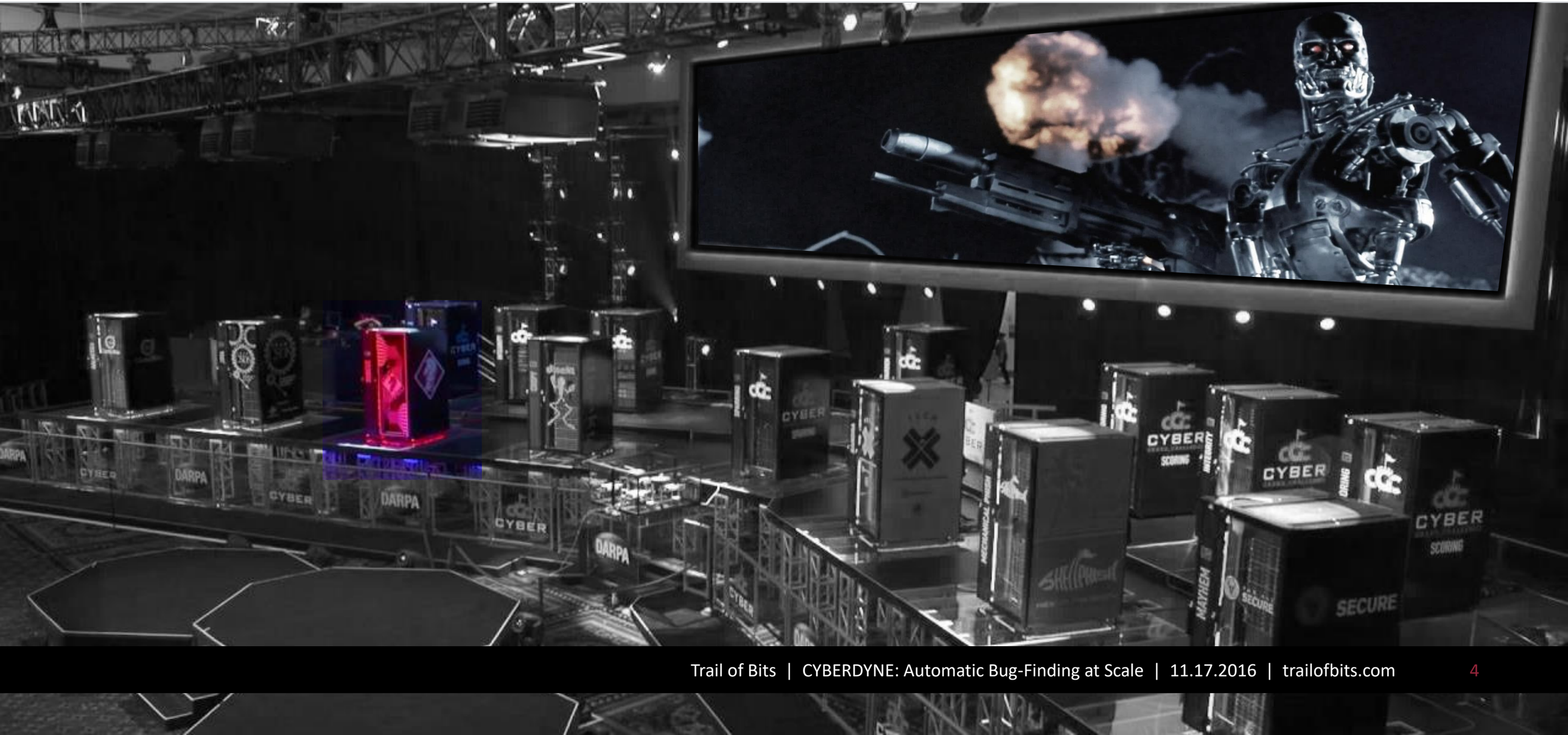
Cyberdyne (ex)terminates bugs

- Finds bug in binaries
- Combines different techniques
  - Coverage-guided fuzzing
  - Symbolic execution

Get to know the mind of the machine

- **Part 1: high level architecture**
  - How to coordinate bug-finding tools
- **Part 2: low level tools**
  - How do the bug-finding tools work?

History: Cyber Grand Challenge (1)



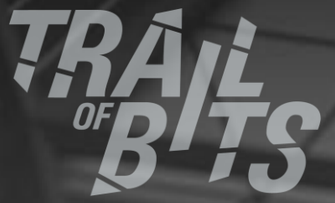
History: Cyber Grand Challenge (2)

- **Capture-the-flag (CTF) competition**
  - Goal: find and exploit bugs in binaries
  - Goal: patch binaries
- **Competitors were programs**
  - “Cyber Reasoning Systems” (CRS)

History: Cyber Grand Challenge (3)

- Shaped the design of Cyberdyne
- Distributed system
  - Runs on any number of nodes
- Automated system
  - No human intervention required

Part 1
Skeleton of a bug-finding system

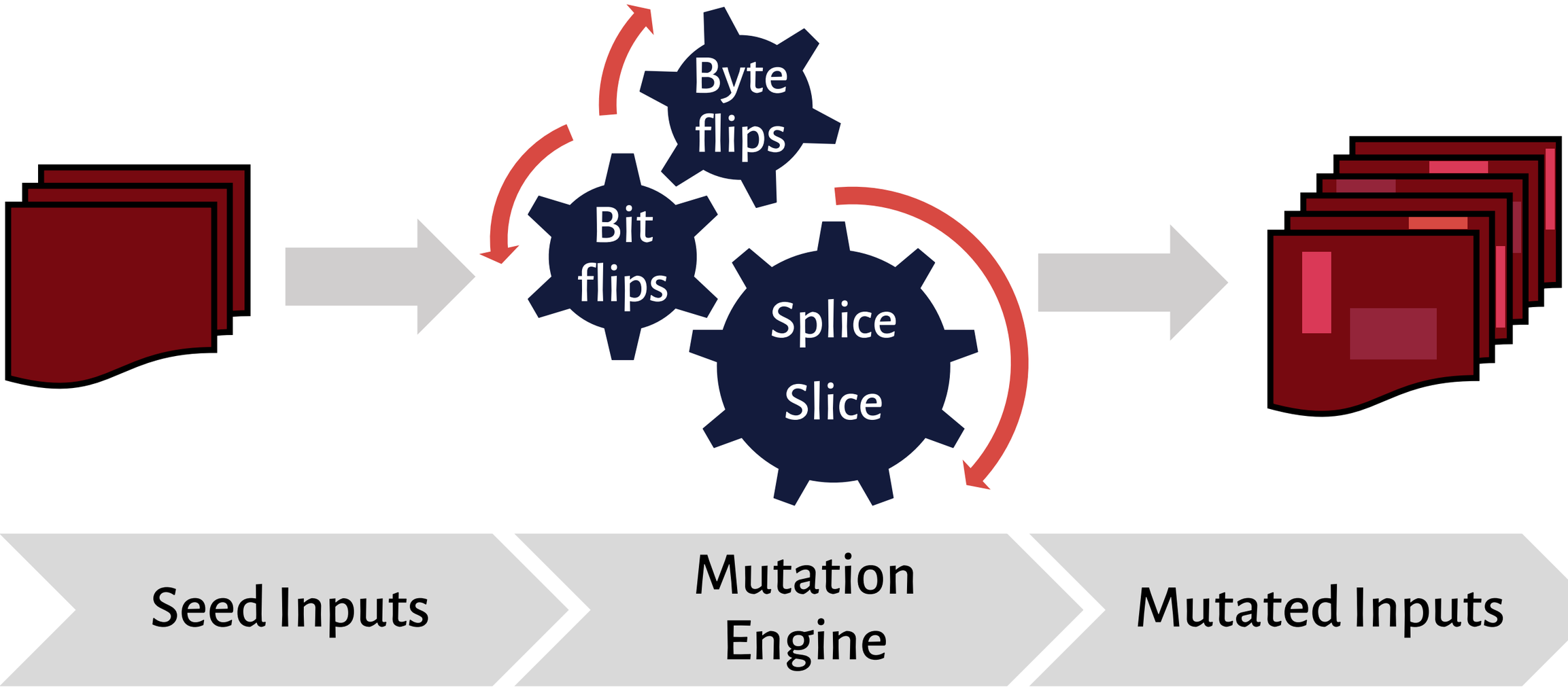


Ideally, a bug-finding system should…

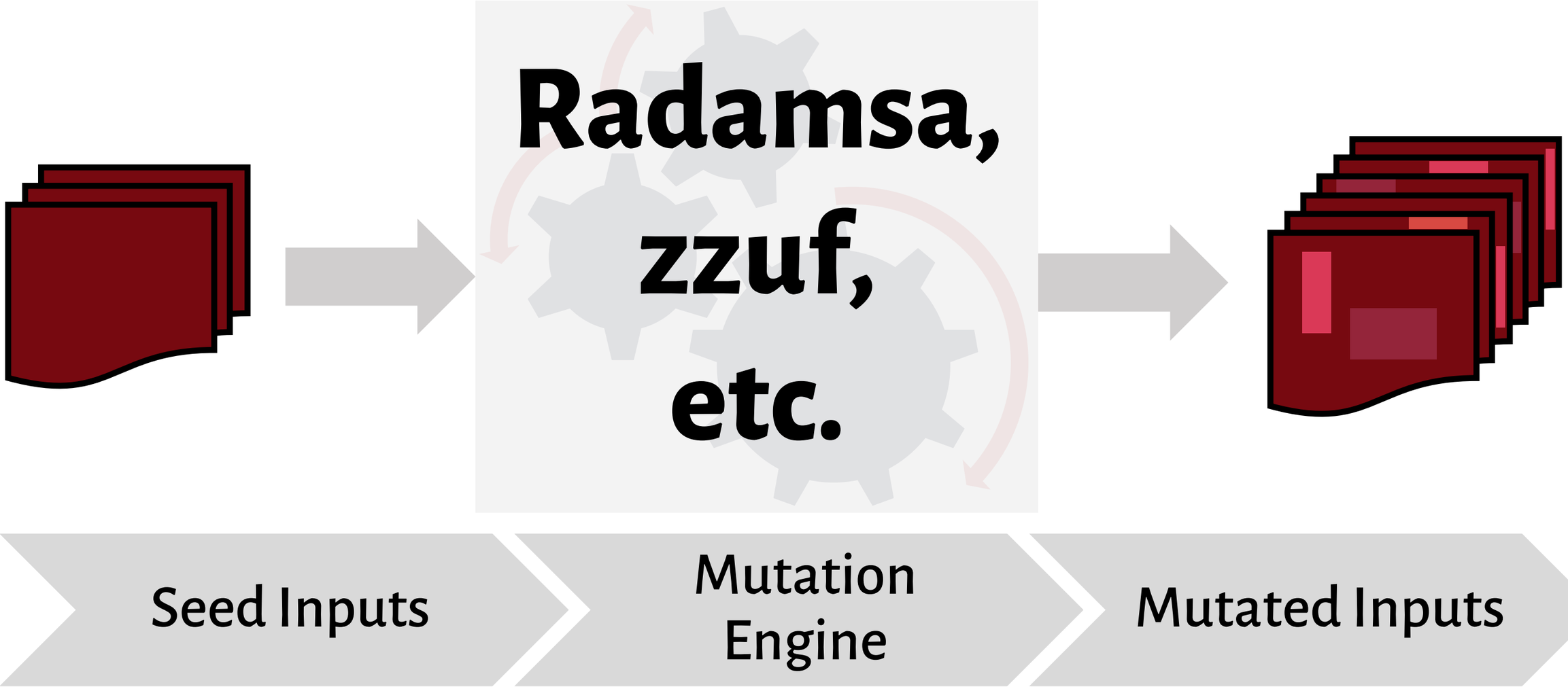
- Find bugs
  - Simple, right?
- Work on real programs
- Be easy to scale

When I grow up …

First kill: simple fuzzing (1)

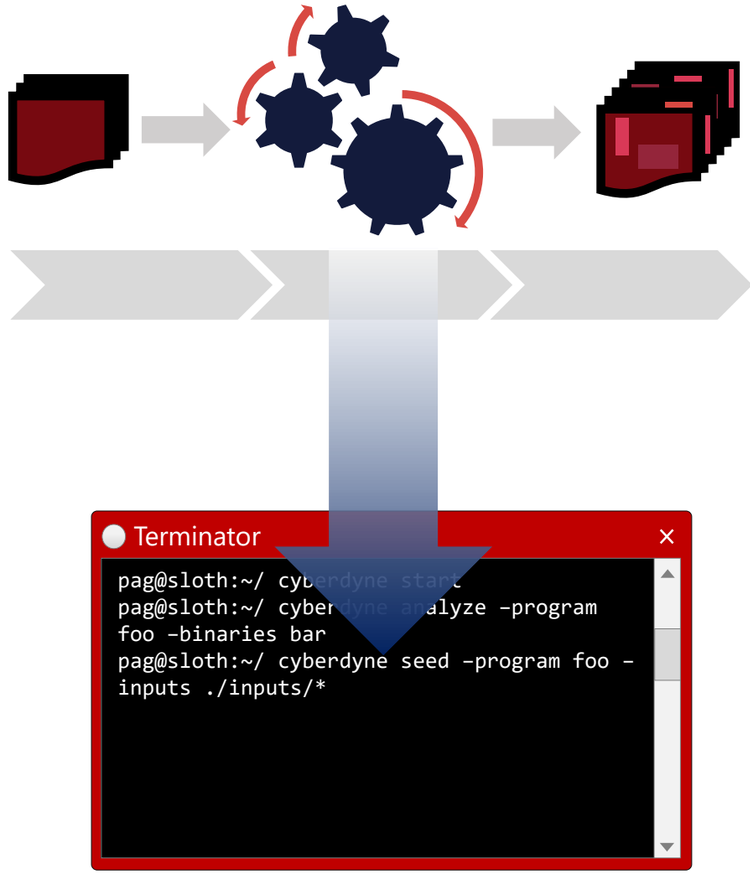


First kill: simple fuzzing (1)



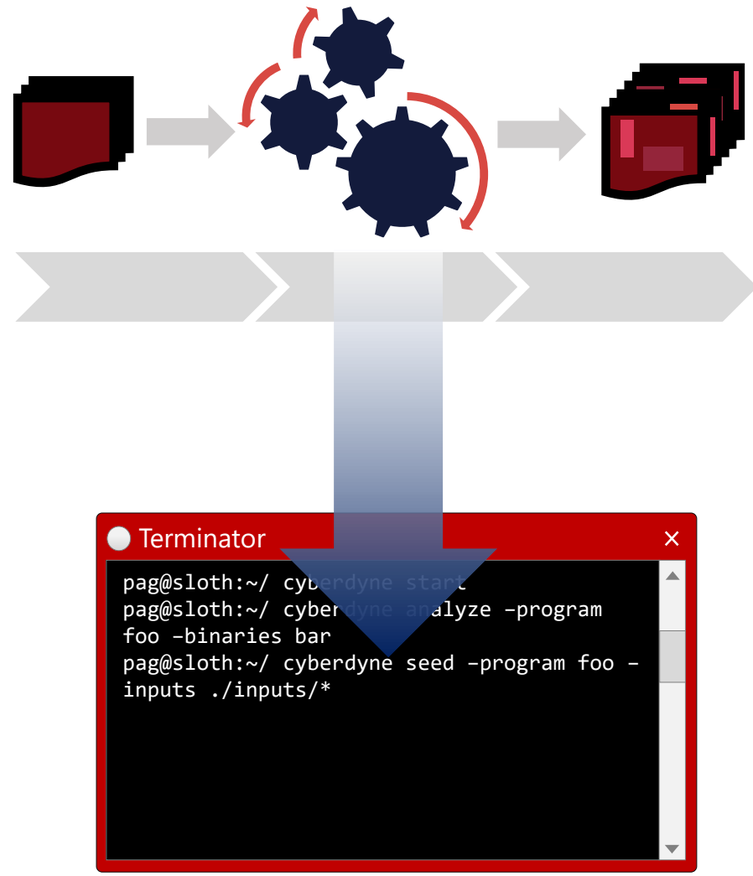
First kill: simple fuzzing (2)

- Mutate inputs
- ❑ Execute inputs
- ...
- ❑ Profit?
  - Find bugs!



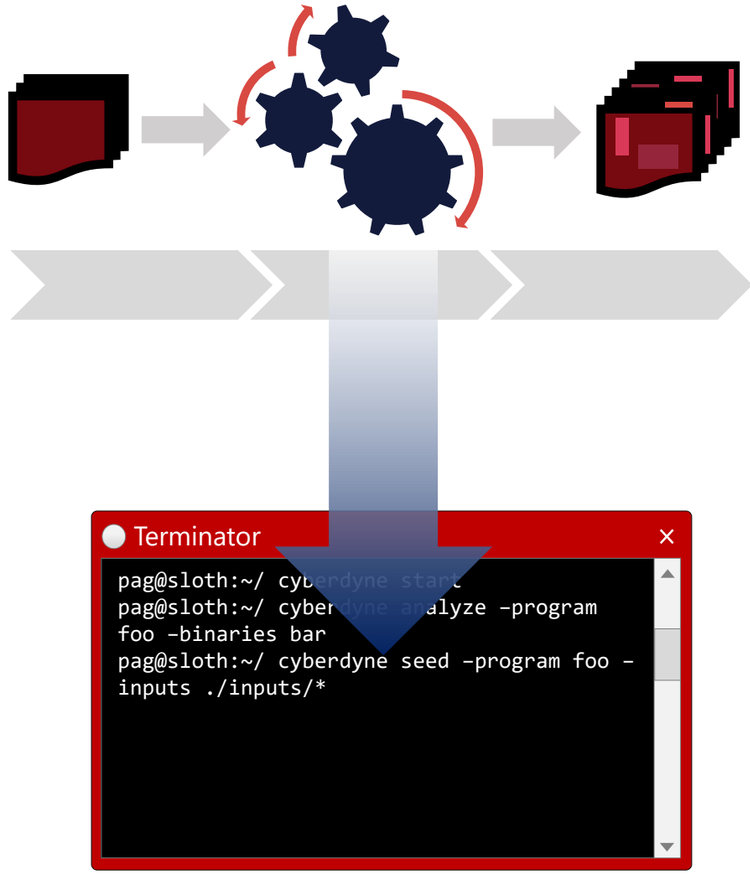
First kill: simple fuzzing (2)

- ☑ Mutate inputs
- 🕒 Execute inputs
- …
- ❑ Profit?
  - Find bugs!



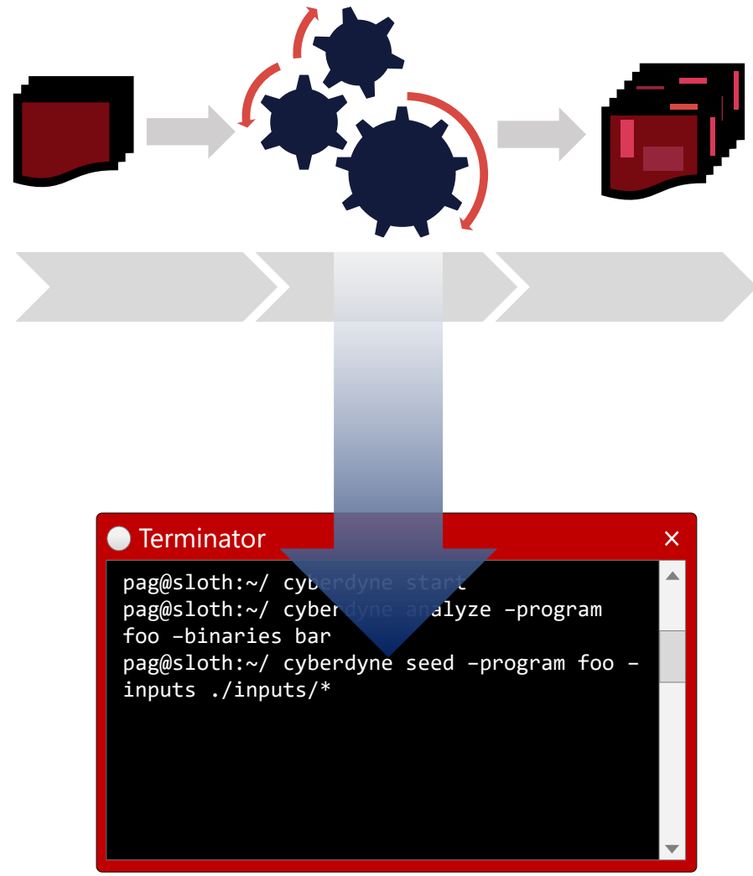
First kill: simple fuzzing (2)

- ☑ Mutate inputs
- ☑ Execute inputs
- …
-  Profit?
  - Find bugs!



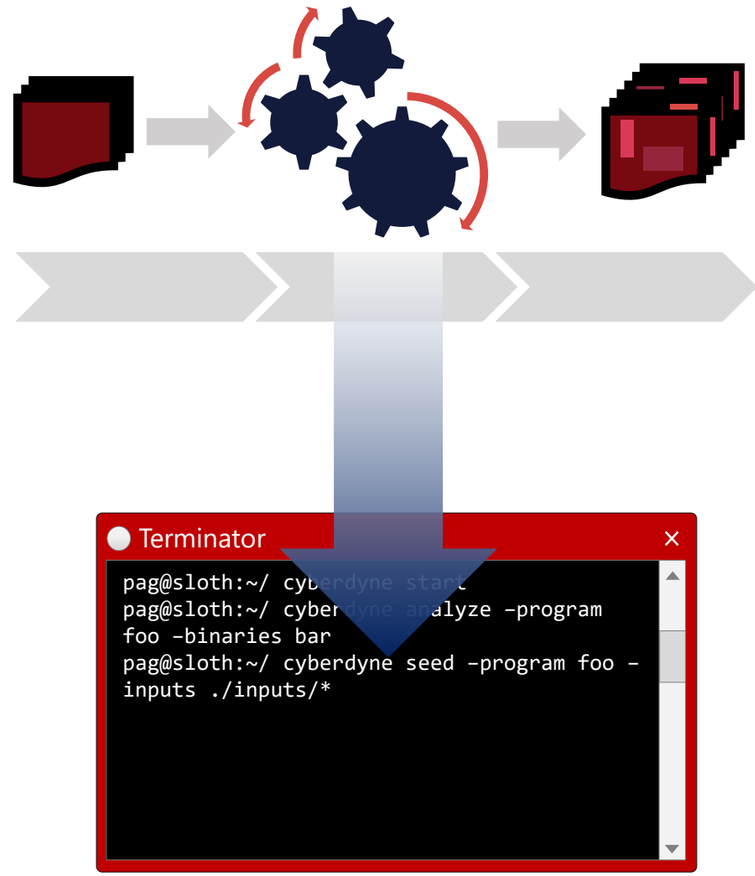
First kill: simple fuzzing (2)

- ☑ Mutate inputs
- ☑ Execute inputs
- …
- Profit?
  - Find bugs!
  - Right????



First kill: simple fuzzing (2)

- ☑ Mutate inputs
- ☑ Execute inputs
- …
- ☐ Risk of loss!
  - No bugs found
  - Lost cycles, time



Misfire: Check your targets

- **Searching for bugs takes time**
- **Need accountability**
  - Is it worth it to keep searching?
  - Is progress being made?
- **How do we measure progress?**

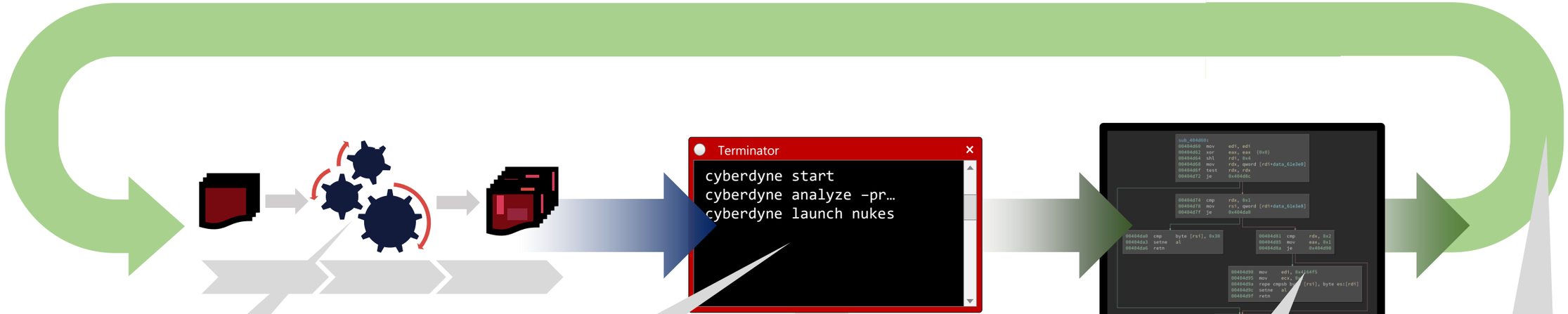
Reload: Track bug-finding progress

- **Idea: has something new happened?**
- **Track when new code is executed**
  - Code coverage: Instrument program to detect when new code is executed
  - Inputs that cover new code signal progress

Need more ammo

- **Eventually hit a "coverage ceiling"**
  - Decreasing marginal returns
- **Need heavier guns**
  - Coverage-guided fuzzing: re-seed with inputs that got new coverage (next)
  - Symbolic execution (later)

Coverage-guided mutational fuzzing (1)



Step 1 Mutate inputs

Step 2 Execute mutations

Crashes!

Step 3 Gets new Coverage?

Step 4 Re-seed mutator

Coverage-guided mutational fuzzing (1)

Step 1
Mutate inputs

Step 2
Execute mutations

Crashes!

Step 3
Gets new Coverage?

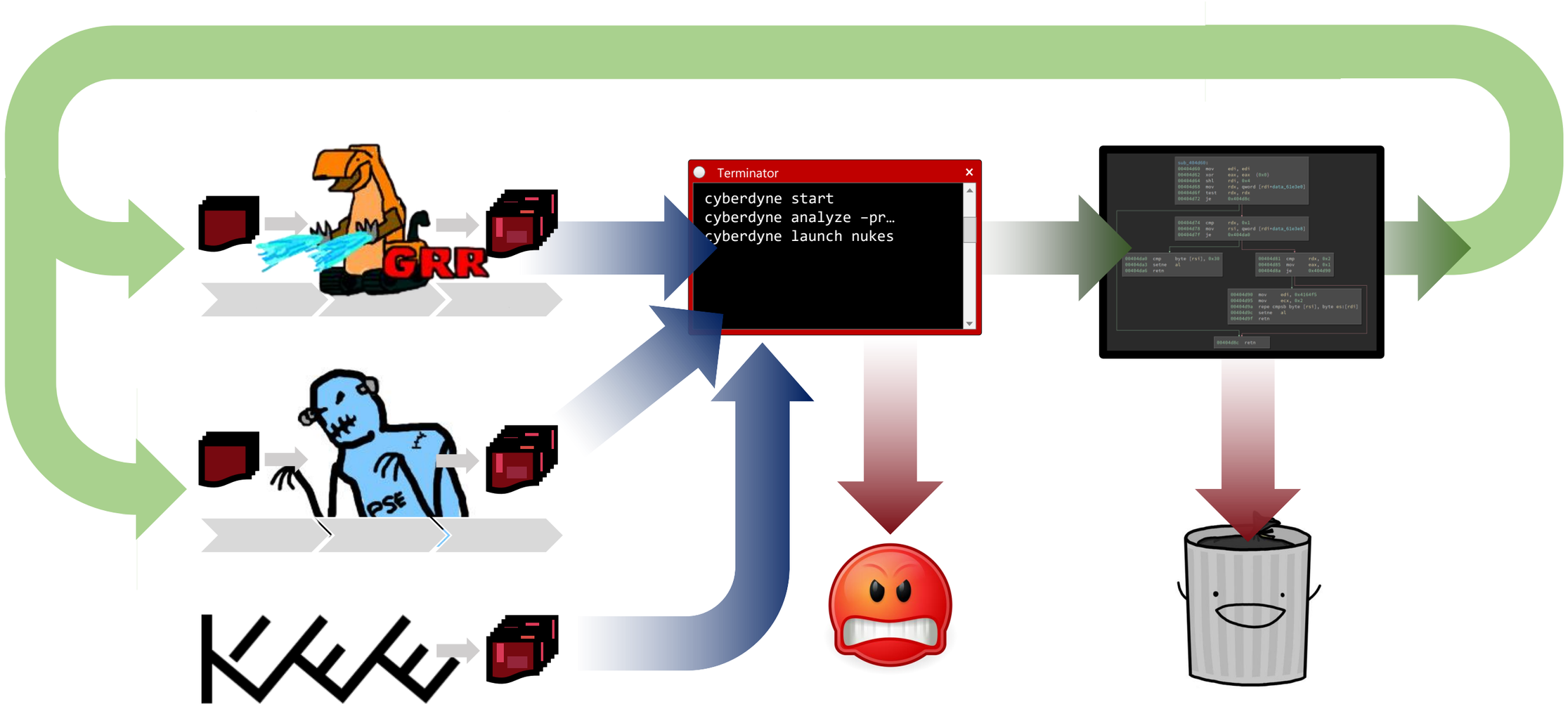
Step 4
Re-seed mutator

Coverage-guided mutational fuzzing (2)

- **Trivially parallelizable**
  - Run mutation engines concurrently
- **Scaling fuzzing in Cyberdyne**
  - Fuzzer service internalizes mutation, execution, code coverage
  - Runs many fuzzers, one mutator each

Look under the skin of Cyberdyne (1)

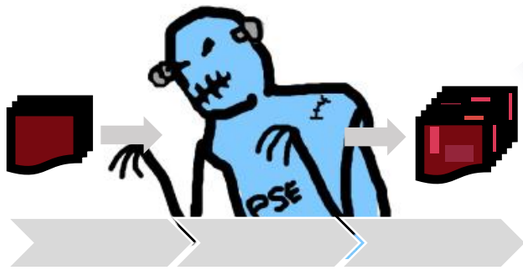
Look under the skin of Cyberdyne (2)



Look under the skin of Cyberdyne (3)

- **Fuzzer** (with GRR)
  - Mutates and executes inputs
  - Easy to scale

Look under the skin of Cyberdyne (4)



PySymEmu

- Coverage-guided binary symbolic executor
- Harder to scale

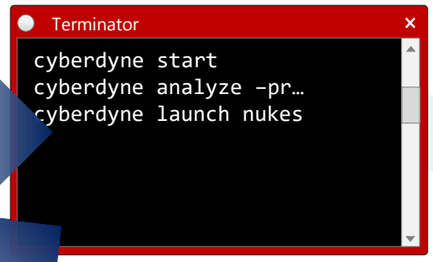
Look under the skin of Cyberdyne (5)

- **KLEE** (with McSema)
  - LLVM bitcode symbolic executor
  - Hard to use
  - Hard to scale

Look under the skin of Cyberdyne (6)

Oracle

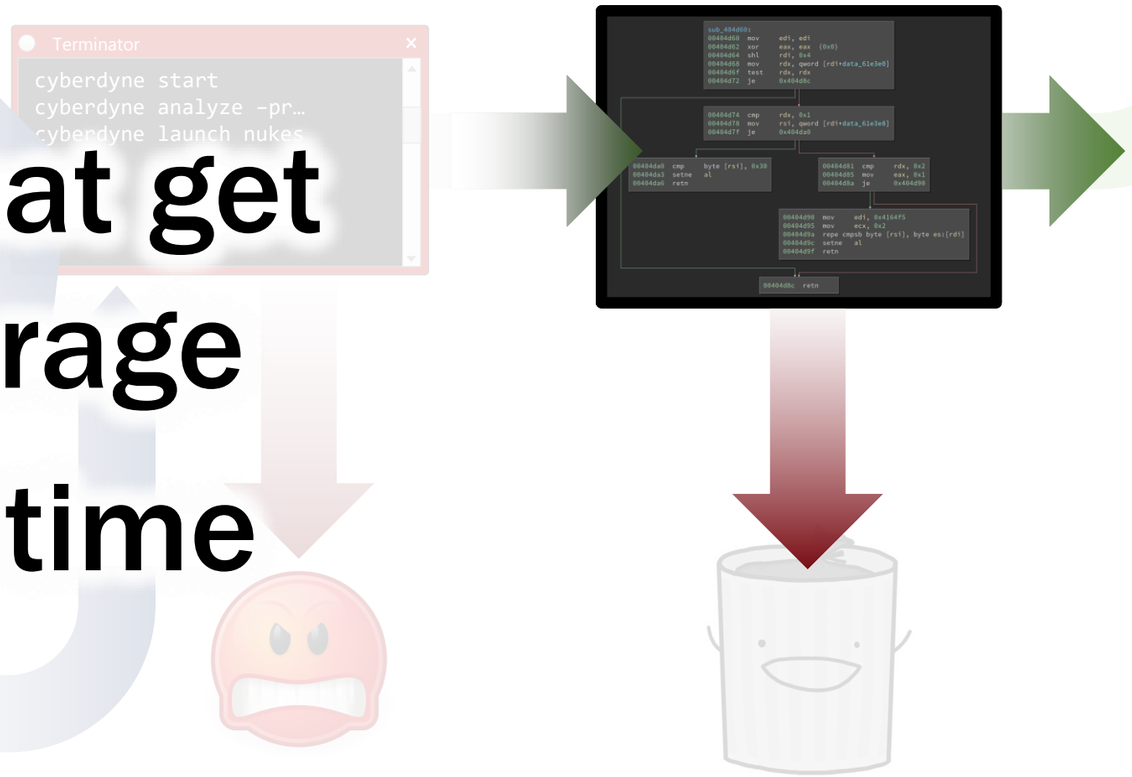
- Gatekeeper for minset
- Detects crashes
- Easy to scale



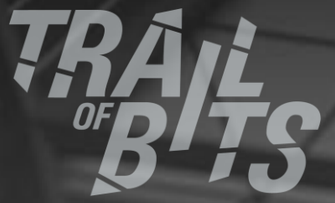
Look under the skin of Cyberdyne (7)

Minset

- Finds inputs that get new code coverage
- One input at a time
- Bottleneck?



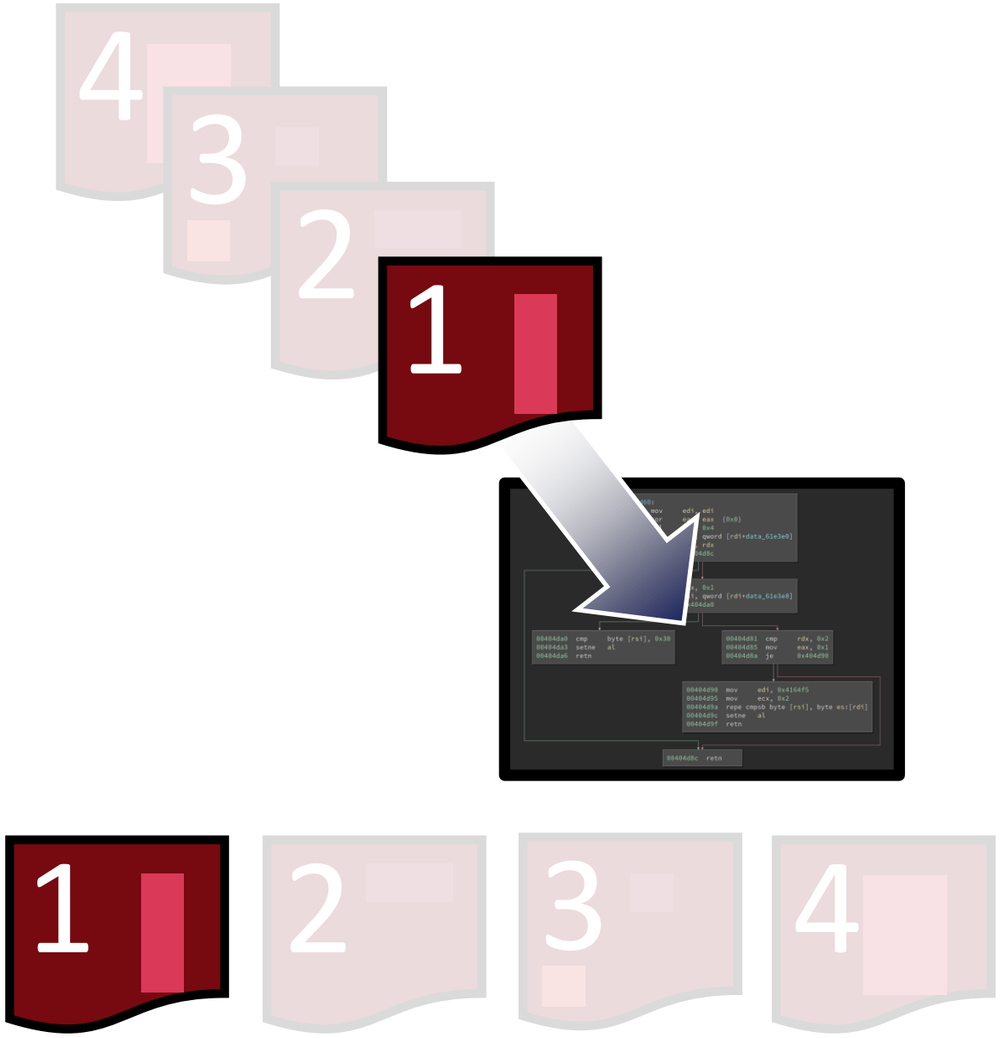
Part 2
The servos and the gears

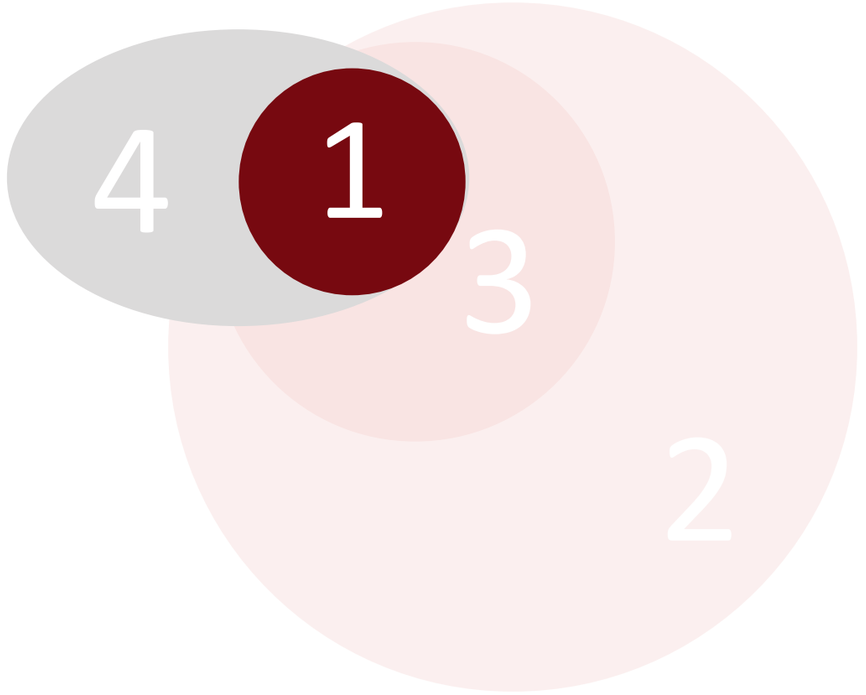


How it works: Minset (1)

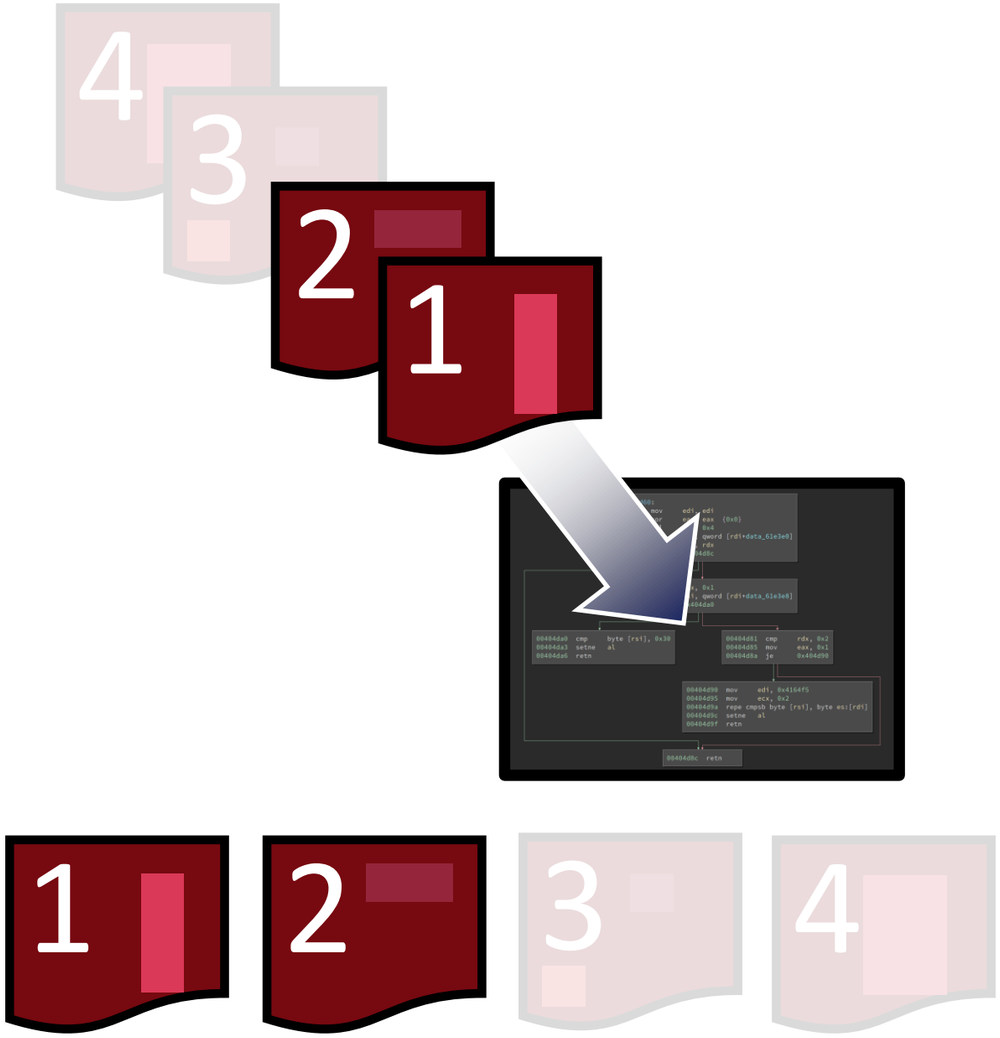
- **What is it?**
  - Minimum set of inputs that produce maximum code coverage
- **Why use it?**
  - Identify “interesting” inputs
  - Good candidates for exploration

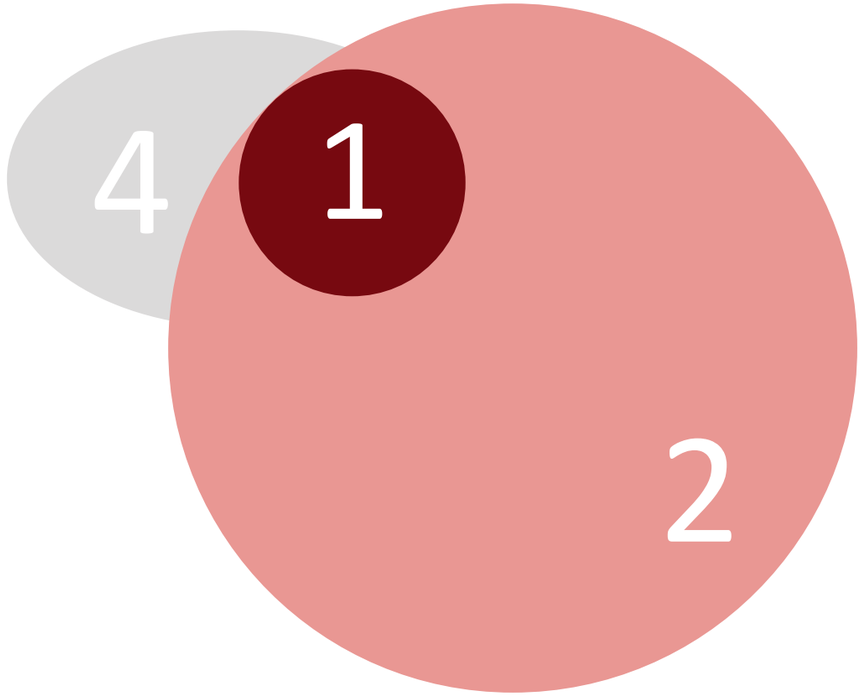
How it works: Minset (2)



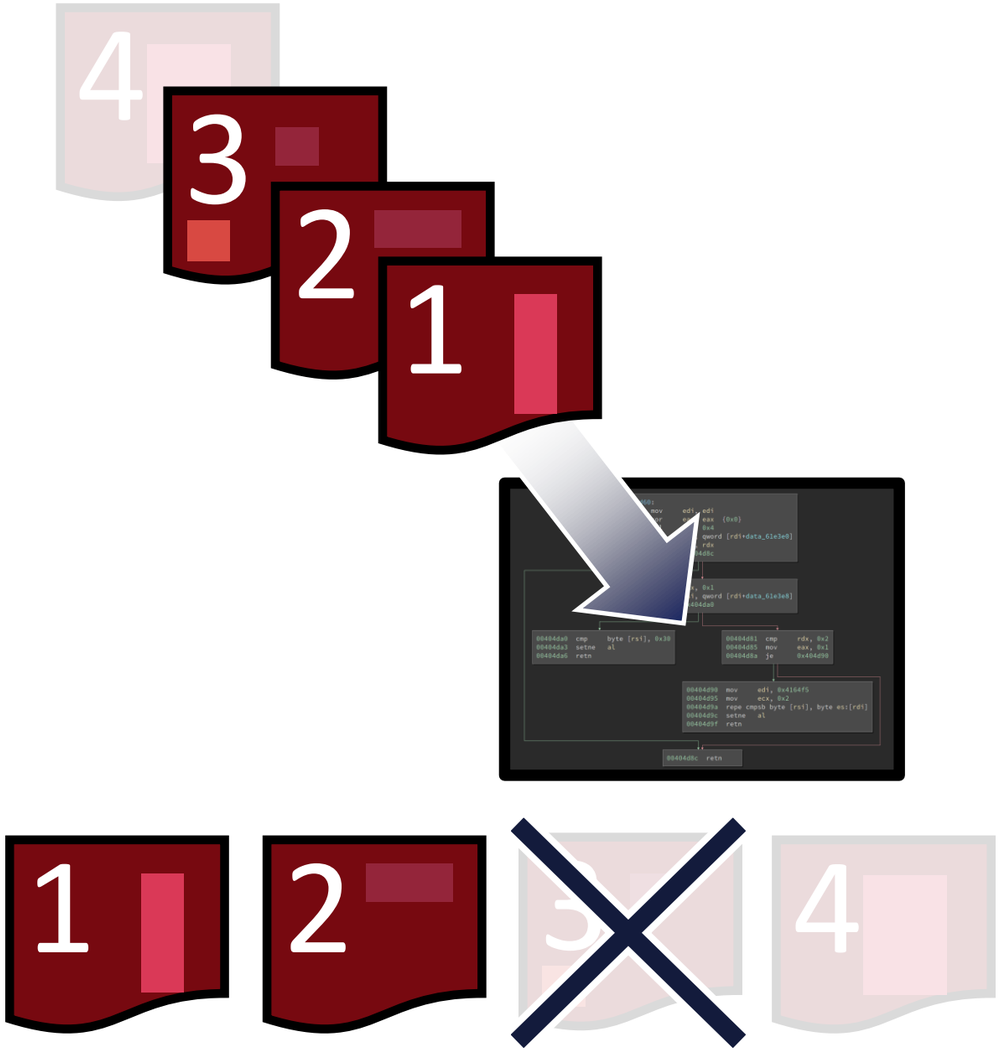


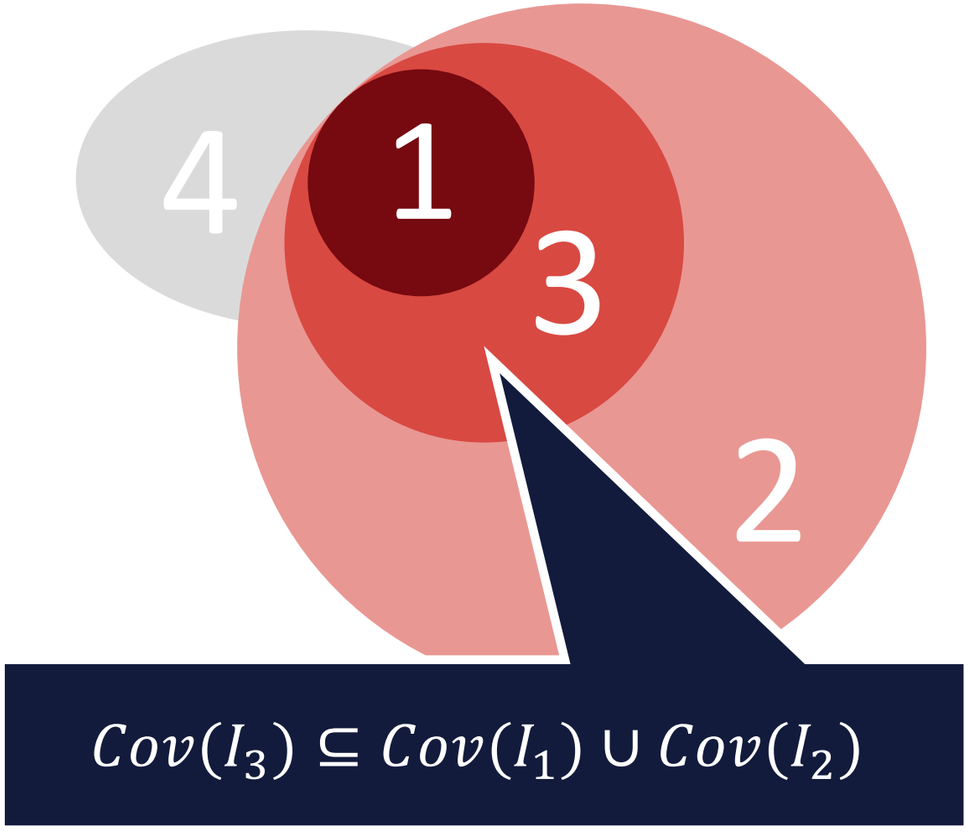
How it works: Minset (3)



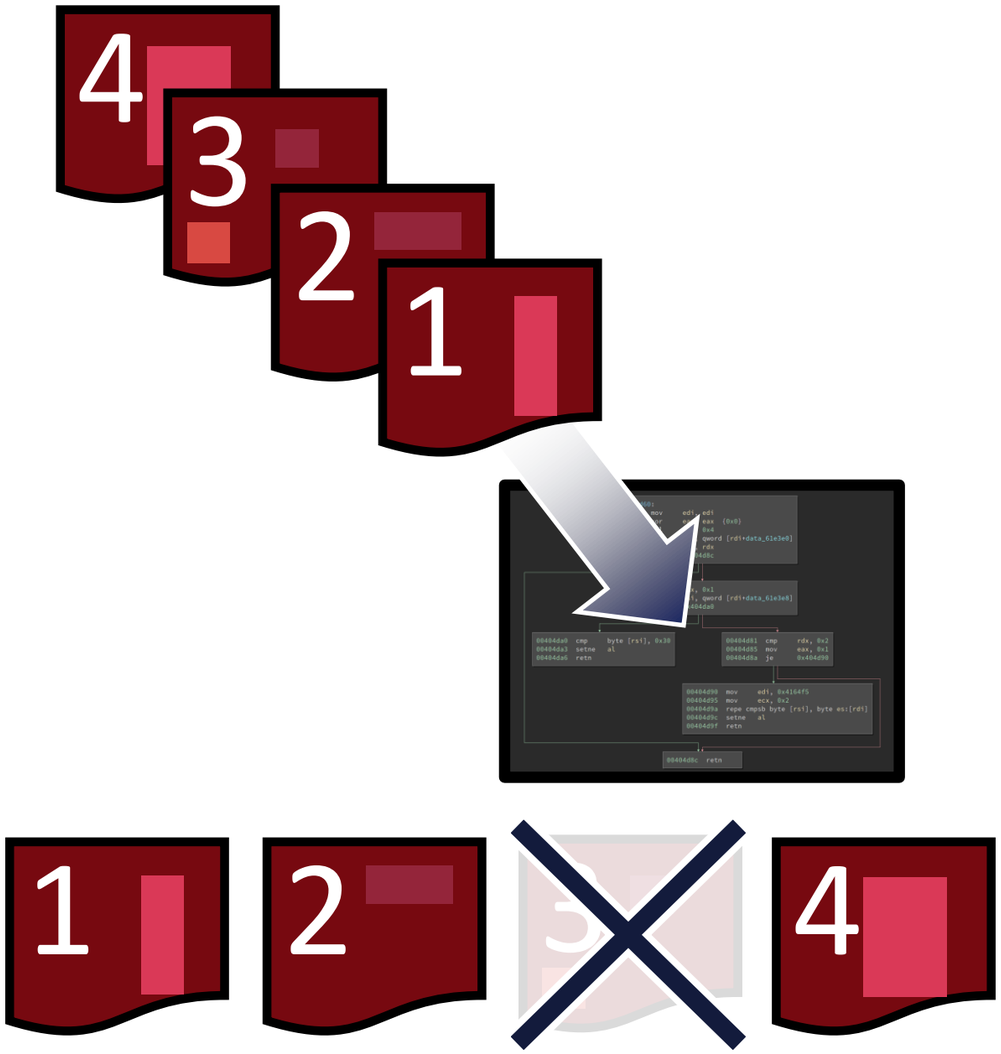


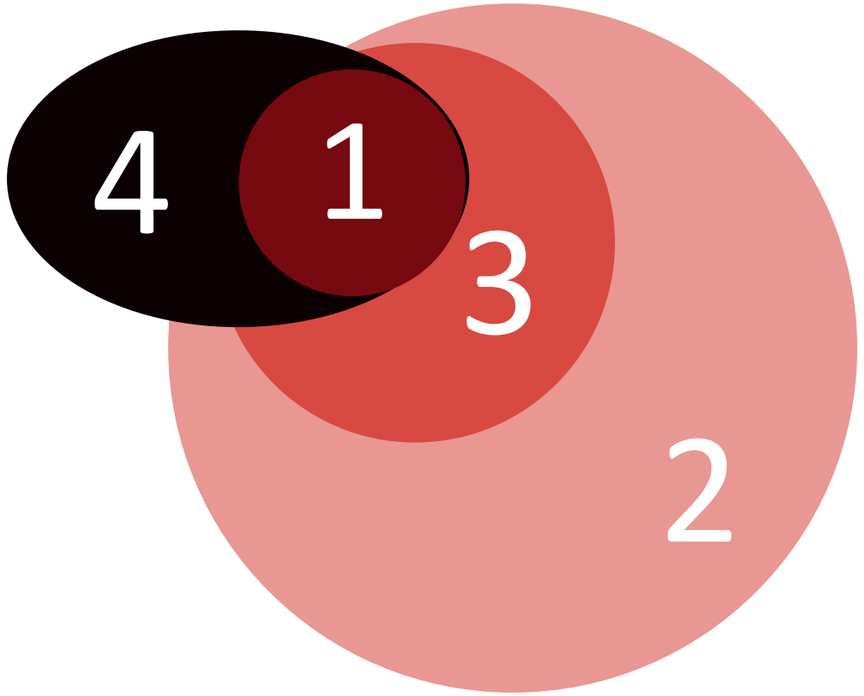
How it works: Minset (4)





How it works: Minset (5)

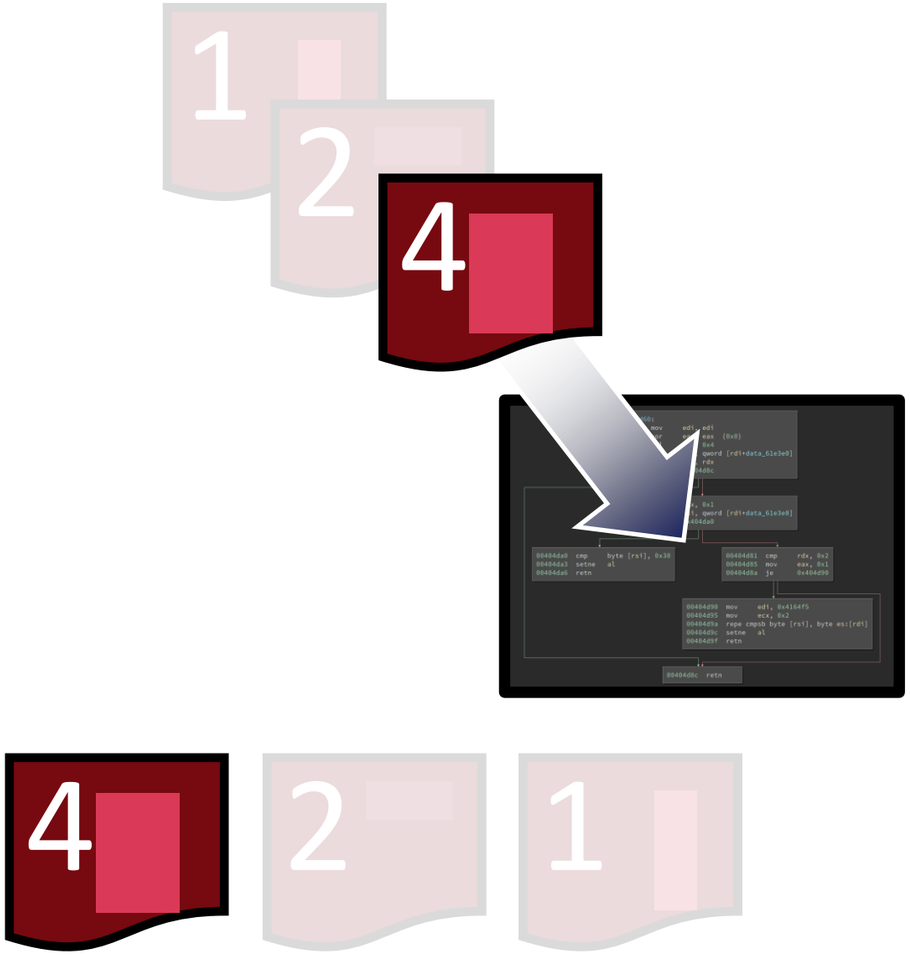


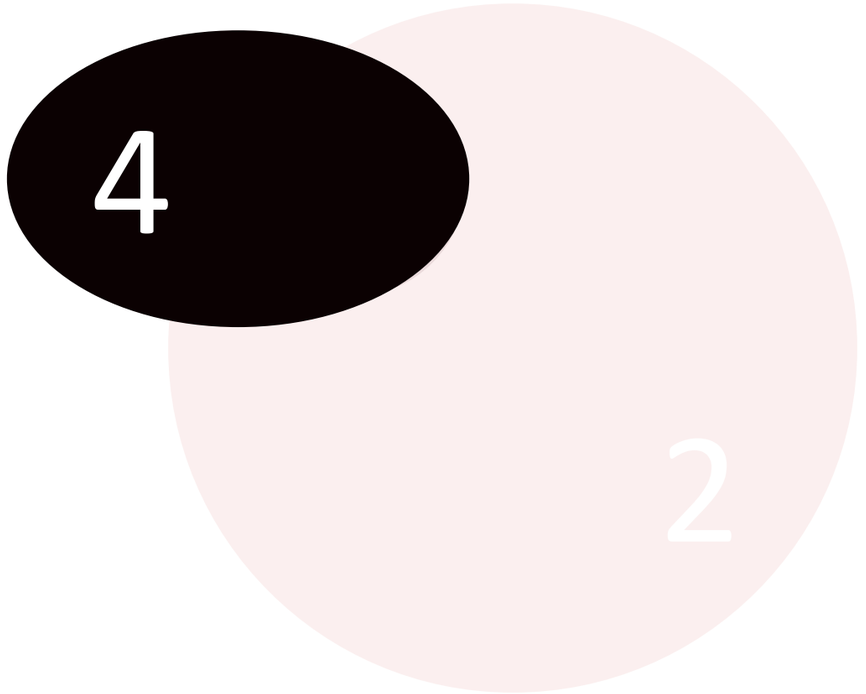


How it works: Minset (6)

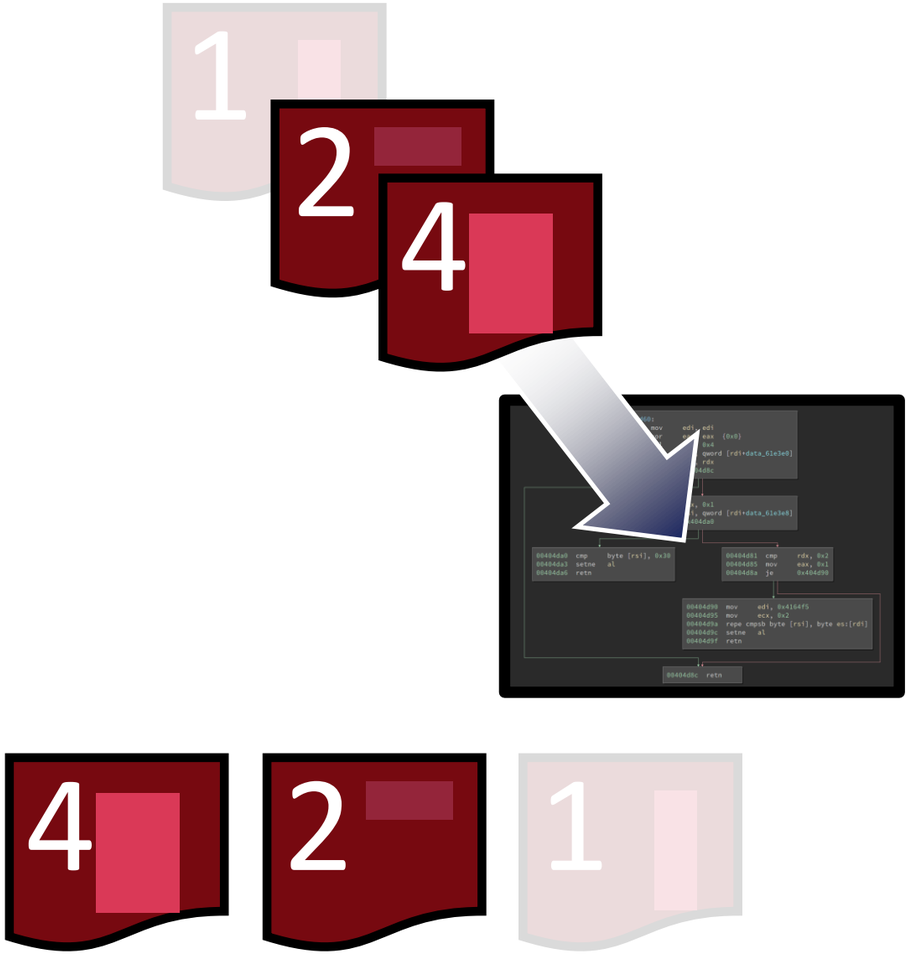
- **Redundancy within the Minset**
  - First input tested guaranteed entry
  - Newly added inputs tend to cover same code as old inputs
- **Idea: fold the minset**
  - Reconstruct it in reverse order

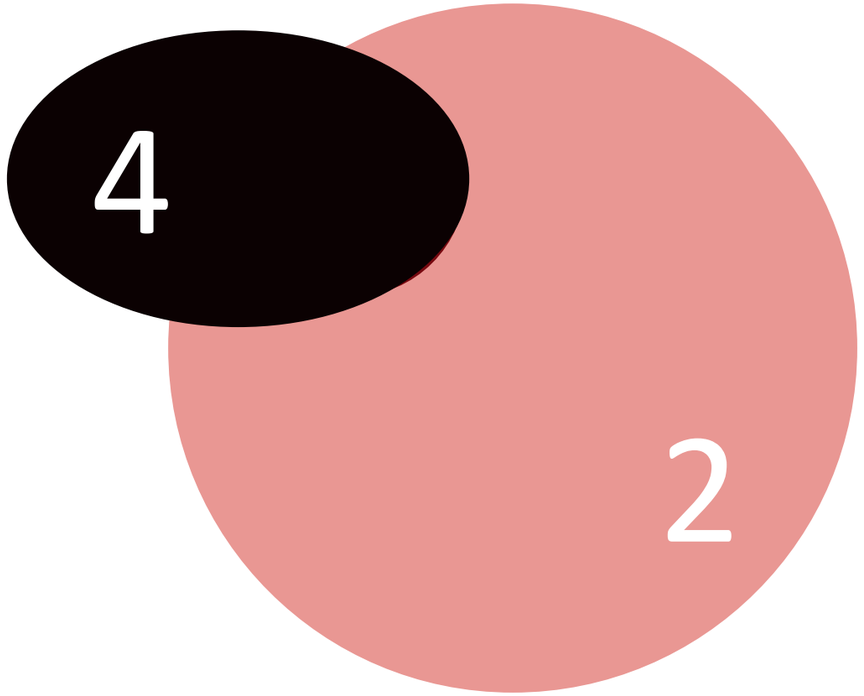
How it works: Minset (7)



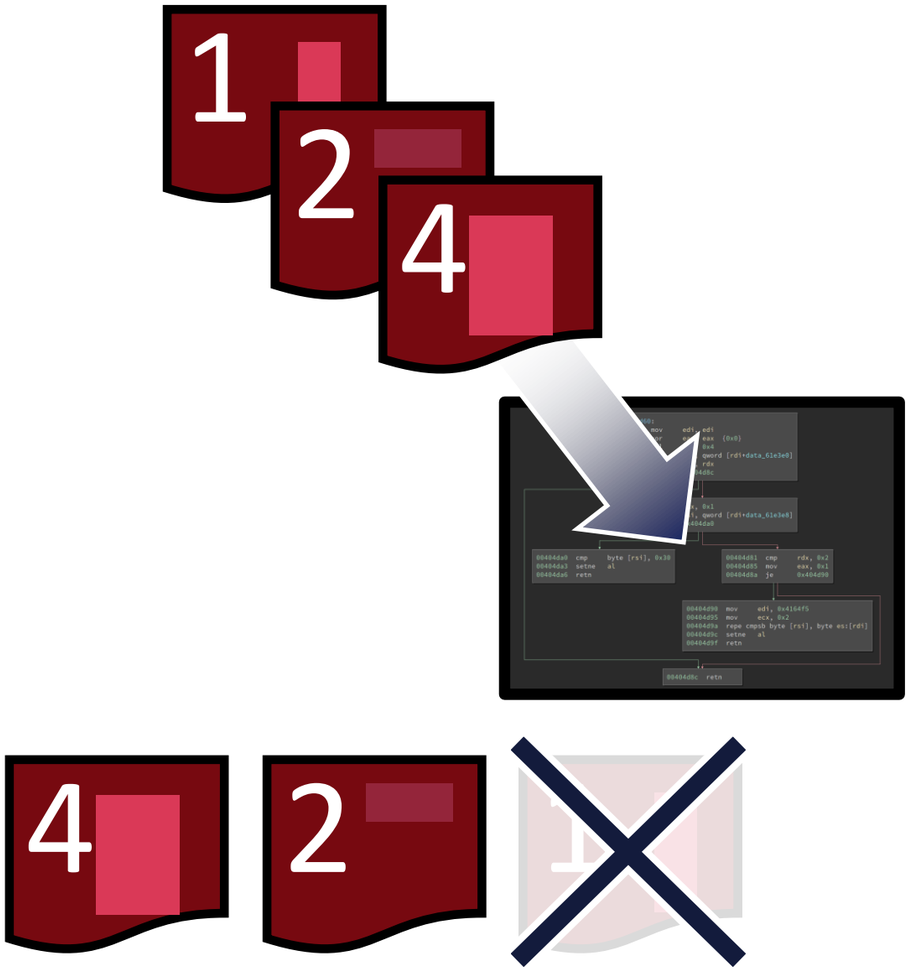


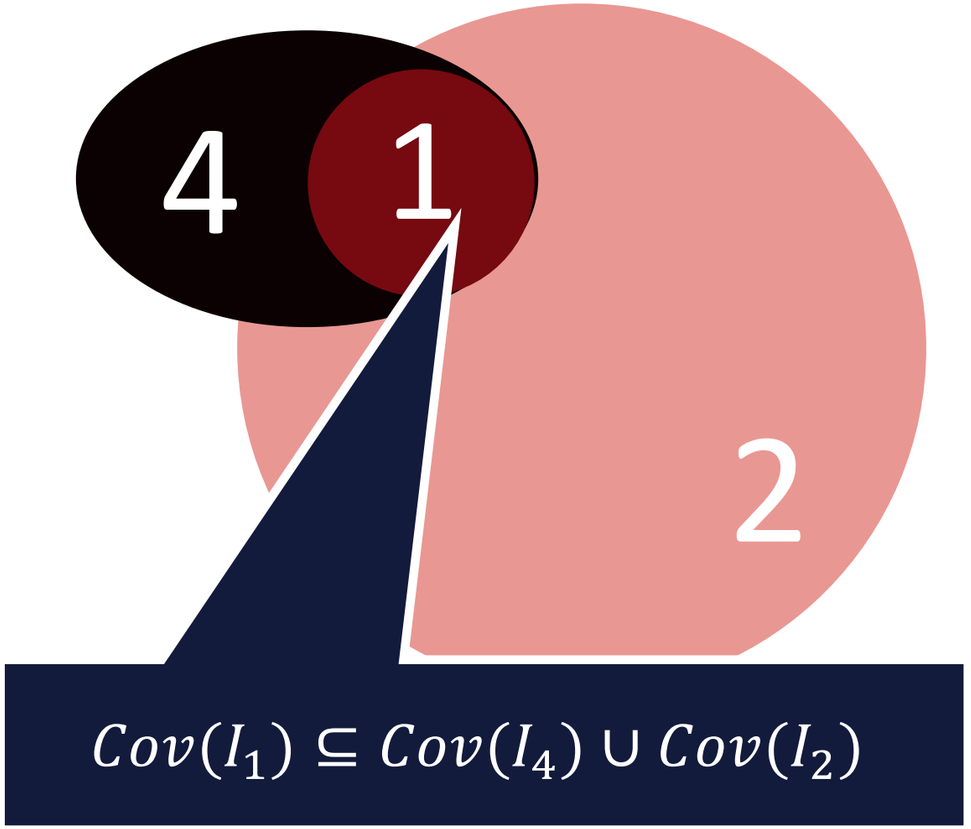
How it works: Minset (8)





How it works: Minset (9)





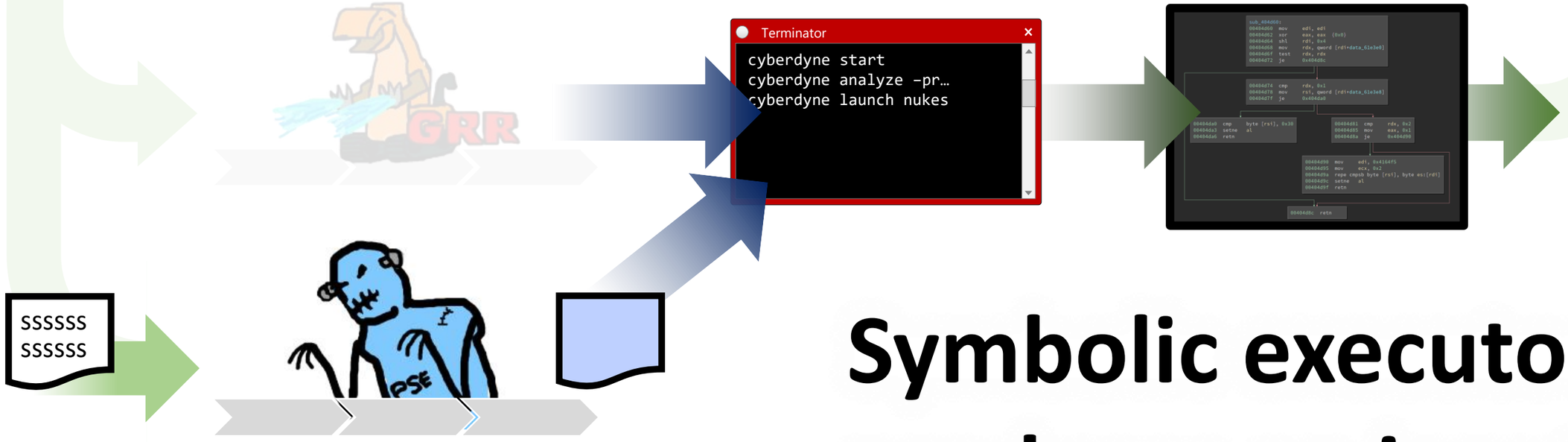
How it works: Minset (10)

- **Corpus distillation is fast and easy**
  - If bottleneck, map and reduce
- **What they don't tell you**
  - What you measure is important
  - Different metrics, different features
  - Fold to compose metrics/features

The gears don't fit

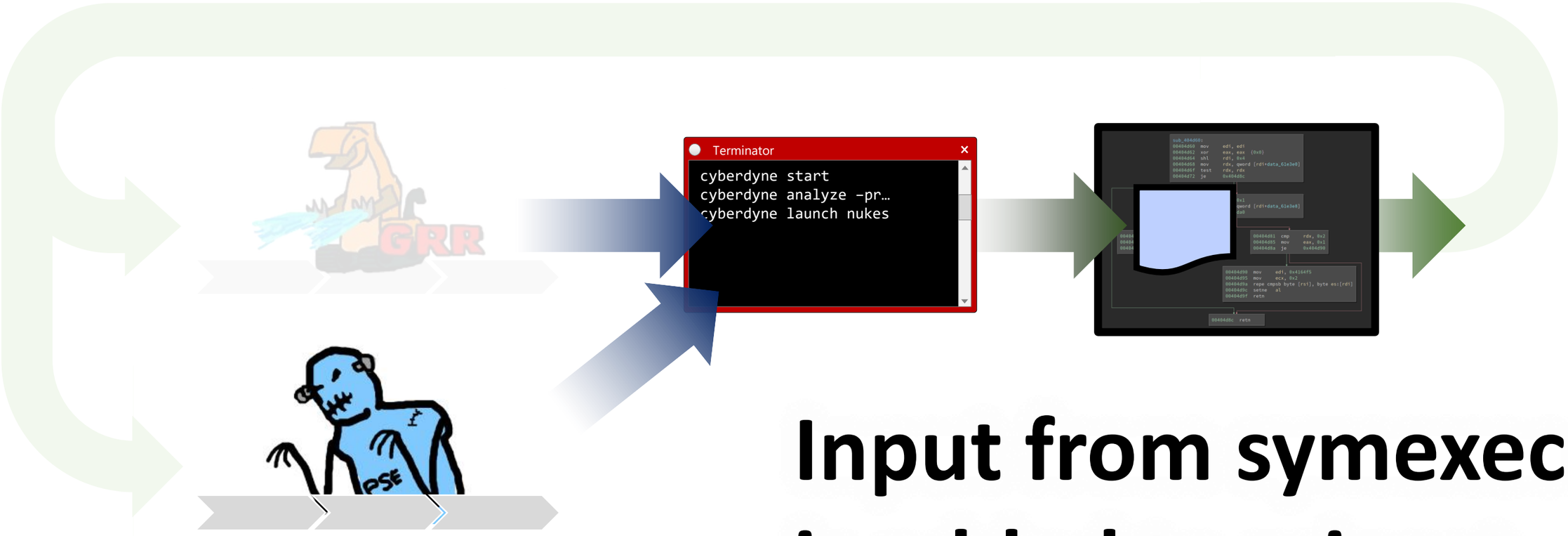
- **Minset is friendly**
  - Doesn't care who or what produced the inputs (e.g. fuzzer, symexec)
- **Challenge: cooperation**
  - Make two independent bug-finding tools coordinate to discover bugs

Cooperation among friends (1)



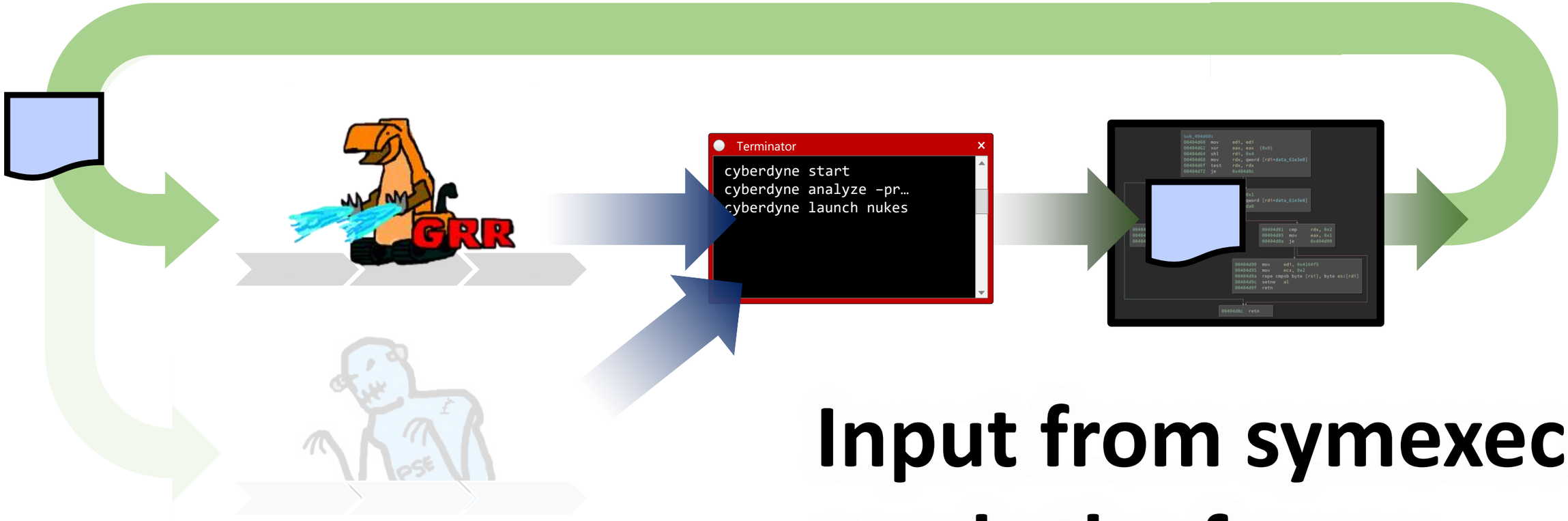
Symbolic executor produces an input

Cooperation among friends (2)



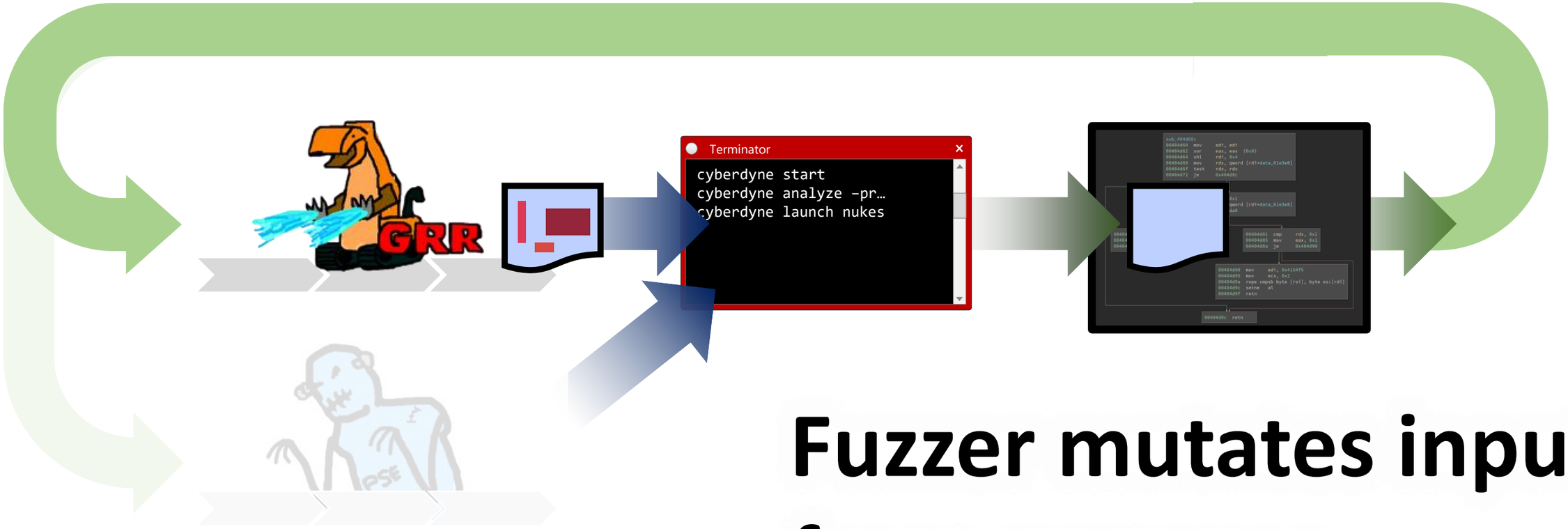
Input from symexec is added to minset

Cooperation among friends (3)



Input from symexec seeds the fuzzer

Cooperation among friends (4)

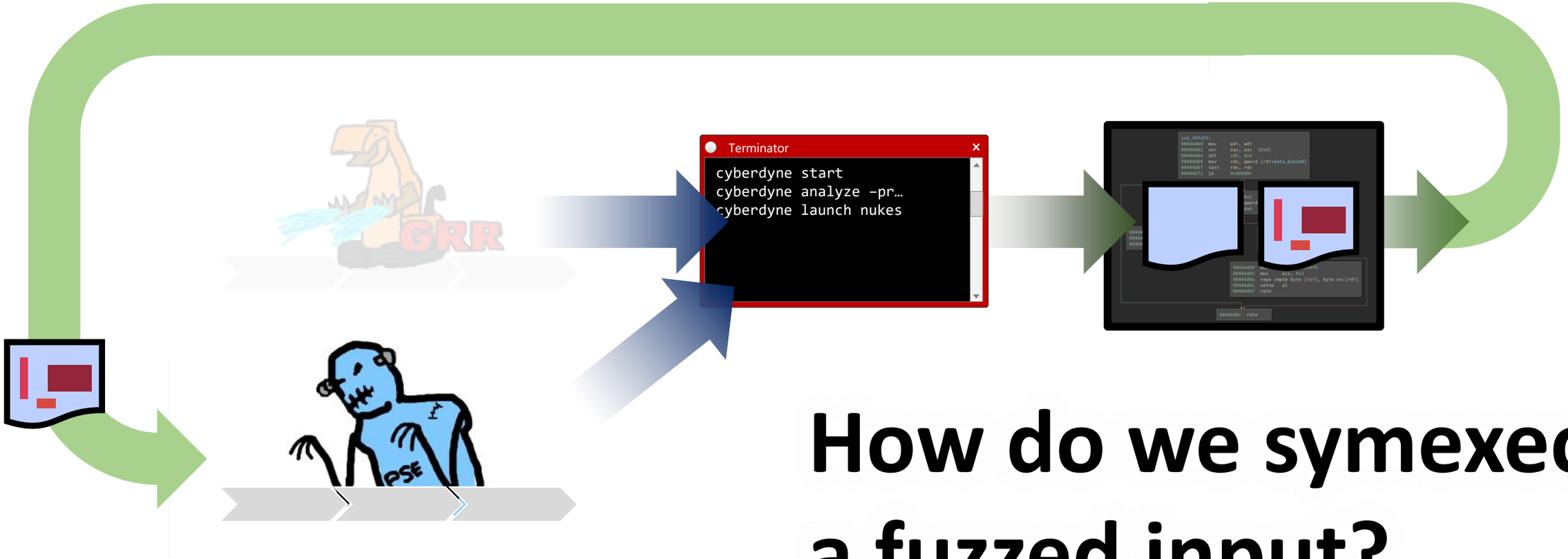


Fuzzer mutates input from symexec

Cooperation among friends (5)

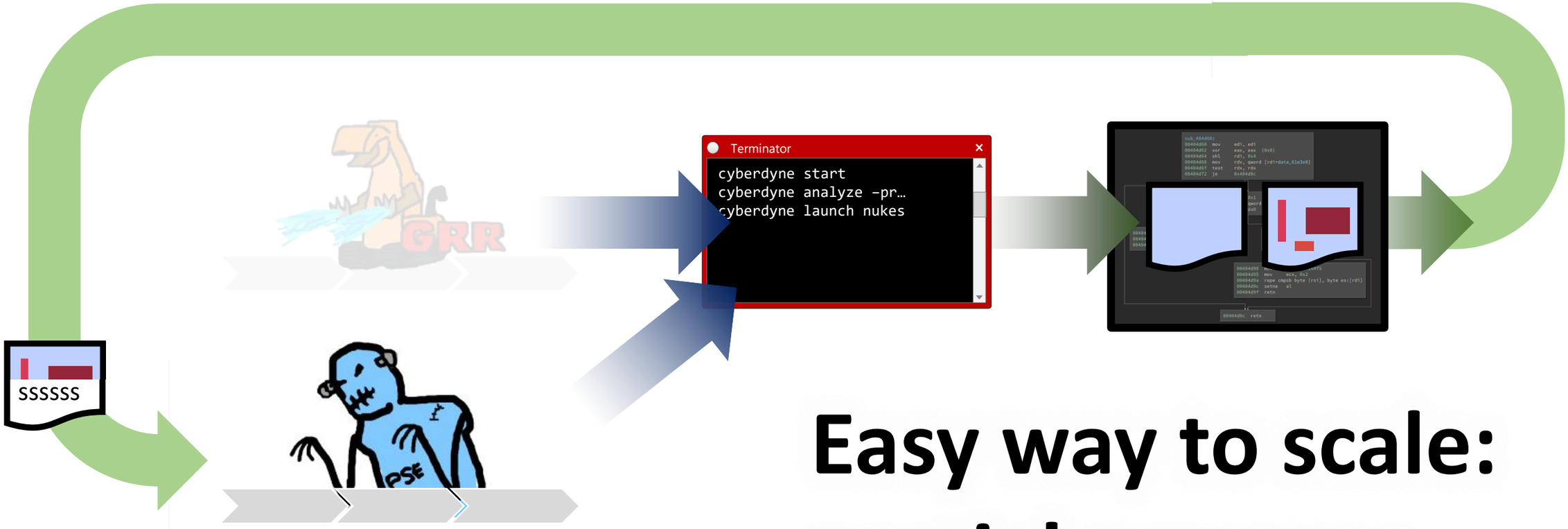
Mutated input is added to the minset

Cooperation among friends (6)



How do we symexec a fuzzed input?

Cooperation among friends (7)



Easy way to scale: partial symexec

Some friendships are a lot of work

- **Symbolic executors are monolithic**
  - Reason about all program paths
  - Somehow use theorem provers
  - Bugs fall out the other end…?
- **Challenge: make symexec cooperate in a scalable way**

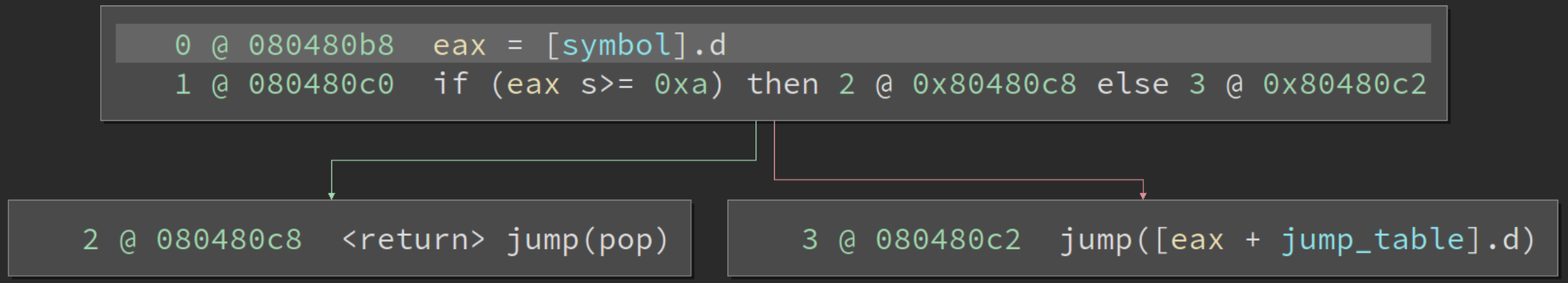
How it works: symbolic execution (1)

- All input bytes are “symbols”
- Fork execution when if-then-else branch depends on symbolic input
- Follow feasible branches, record tested constraints down each path

How it works: symbolic execution (2)

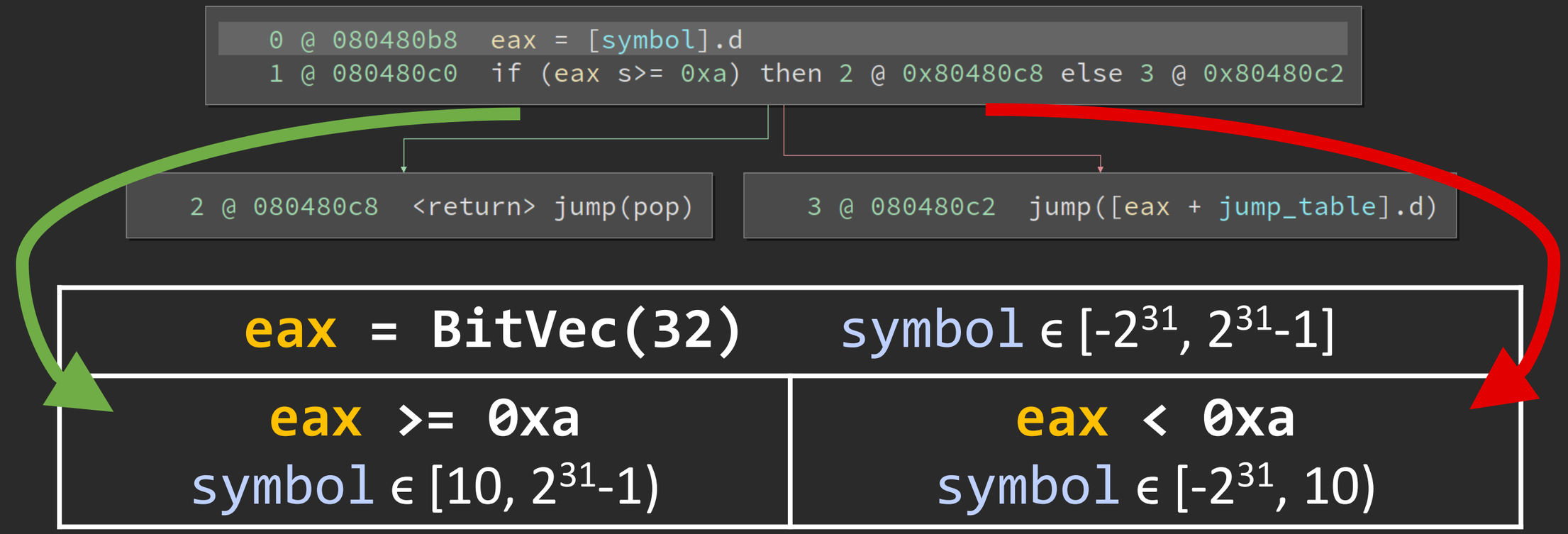
- **Special kind of CPU emulator**
  - Registers/memory can hold bytes, symbols, or symbolic expressions
  - Instructions emulated in software
  - Simulates operations of instructions to work with symbols and bytes

How it works: symbolic execution (3)

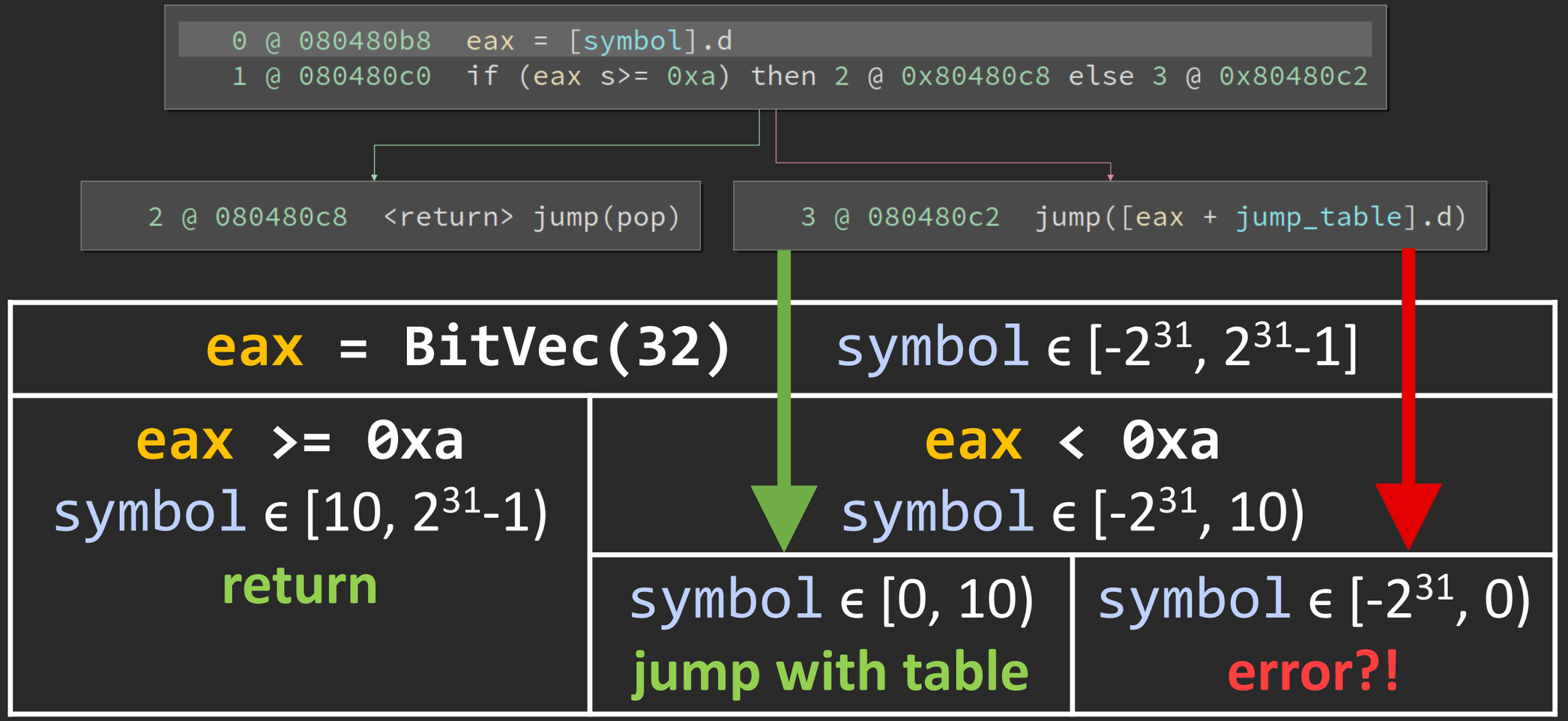


eax = BitVec(32) symbol $\in [-2^{31}, 2^{31}-1]$

How it works: symbolic execution (4)



How it works: symbolic execution (5)



There's too many of them!

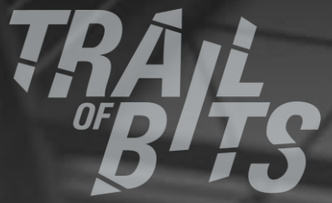
Symbolic execution is hard to scale

- **Symbolic executors fork *a lot*!**
  - Branches, loops, branches in loops
  - Takes too long to get deep into the program, only finds shallow bugs
  - Heuristics, like coverage-guided exploration, are band-aids

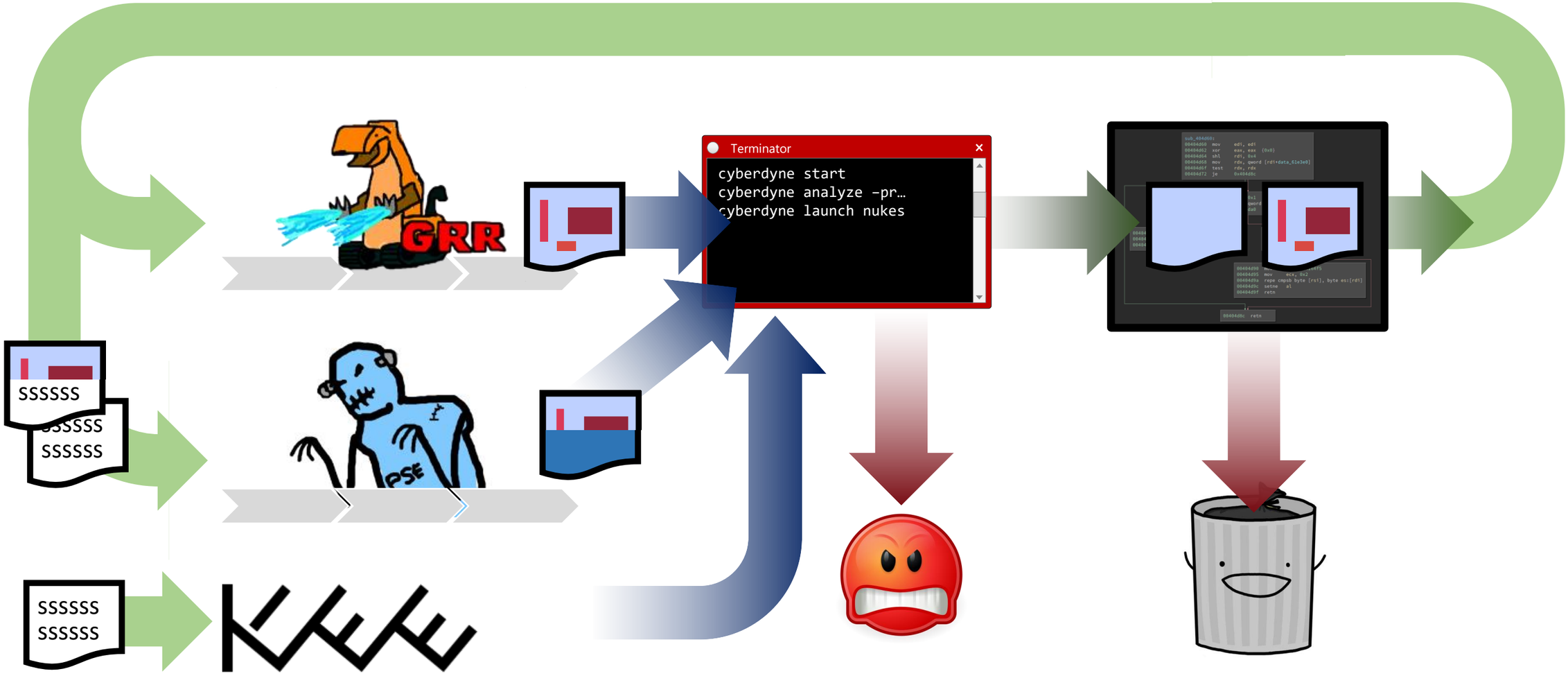
Easy way to scale symbolic execution

- **Partial symbolic execution**
  - Jump deep into a program using a concrete input prefix
- **Trivially parallelizable**
  - Run independent symbolic executors with different prefixes

End of days



Skeleton of a bug-finding system (1)



Skeleton of a bug-finding system (2)

- **Started with simple fuzzing**
  - Added accountability
- **Coverage-guided mutational fuzzing**
  - Sets groundwork for new tools
- **Going from there**
  - Minset as the mediator

The servos and the gears

- **Mediating with the minset**
  - Fuzzer cooperates with anything
  - Symbolic executors need a bit more massaging
- **The path to scalability**
  - Go for trivial parallelization

Cyberdyne kills bugs...now you can too!

Let’s chat

Peter Goodman

Senior Security Engineer

peter@trailofbits.com