Reversing faster.
Together.

Lars Haukli
lars@kodetracer.com

~~I am~~ I used to work as a reverse engineer

Malware sandbox technology (1998-)

Computability theory

Halting problem

Given description of program and input, decide if it finishes or runs forever

Alan Turing proved a general algorithm for all possible program-input pairs cannot exist (1936)

The Halting problem is undecidable

Sandbox evasion problem is likely a variation of the Halting problem

Given description of program and input,

Given description of program and input,

Given binary file, with unknown input

decide if it finishes or runs forever

~~decide if it finishes or runs forever~~

decide if it terminates early or runs intended behavior

The sandbox evasion problem is likely also undecidable

Interviewed malware and threat analysts about workflow and pain points

Relied on sandboxes, but often they produce no information

Manual reverse engineering is slow

Lots of false positives,
false negatives, and
too many files

Can we make reverse engineering faster?

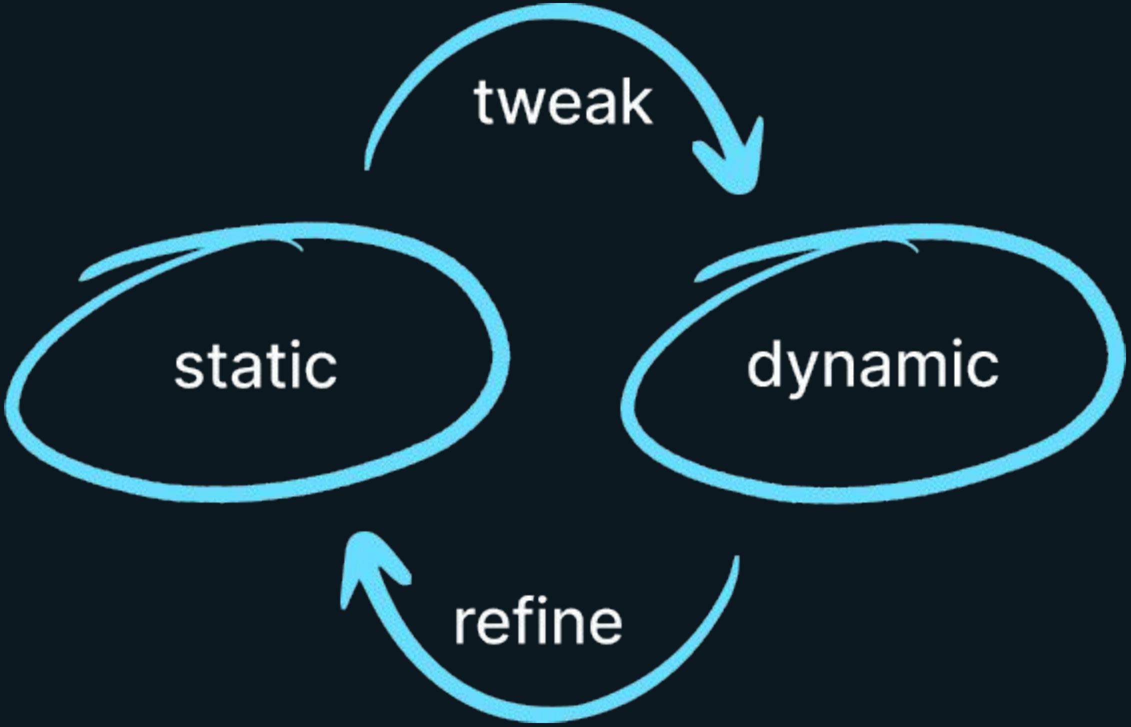
Iterative reverse engineering

Start with a quick static analysis

Tweak your dynamic analysis based on static code analysis

Refine your static analysis with events and traces from dynamic analysis

As you make new discoveries, iterate to improve your analysis


tweak
static
dynamic
refine

Static analysis based on r2

We want to disassemble and decompile all code

Faster analysis means improved UX

Disassembly and decompilation is CPU-bound

Multithreading should speed up the process significantly r2 is single-threaded, so (for now) we split the work into multiple r2 processes

We do autoanalysis single-threaded, but do as as little analysis as possible to decompile all code

Autoanalysis:

aa, aac, avrr, aar,
afva@@@F,
aaft, aanr

r2 interactive help:

?*~…

Once autoanalysis is completed, we save some meta information

Similar to projects, we generate r2 commands: afl*, f*, ax*

Multiple processes load meta information in new r2 processes and run in parallel

Rebasing is constantly required in our iterative reverse engineering approach

Currently we reanalyze the binary each time it’s loaded at a new address

We use Frida to generate events and traces

We also enable hypervisor debugging

Hypervisor debugging enables automatic load phase analysis

Hypervisor debugging also enables interactively debugging anything inside the VM

Enjoy the demo.

Thank you.

Join us on Discord via kodetracer.com

Lars Haukli
lars@kodetracer.com