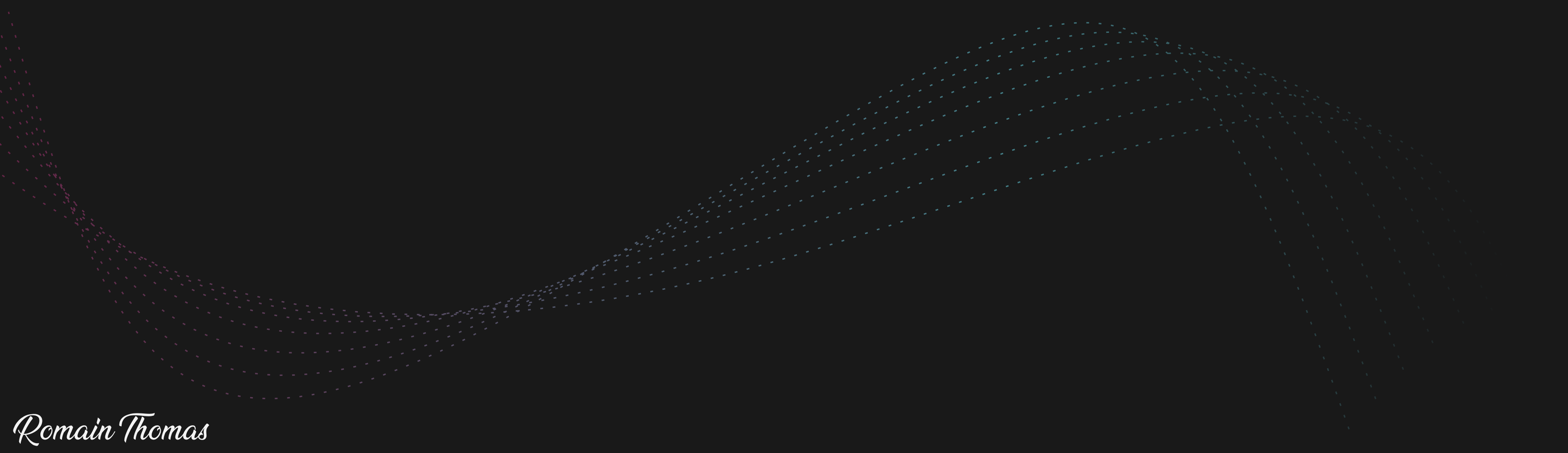
Rethinking Reverse Engineering for the AI Era

Calif - 2026

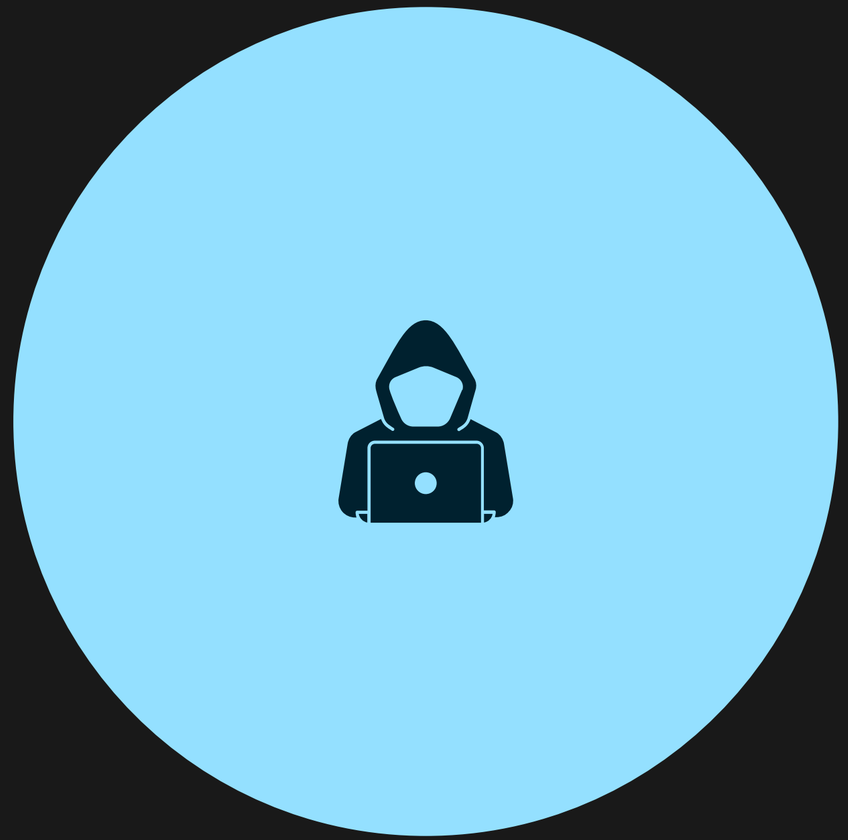


Romain Thomas

Introduction

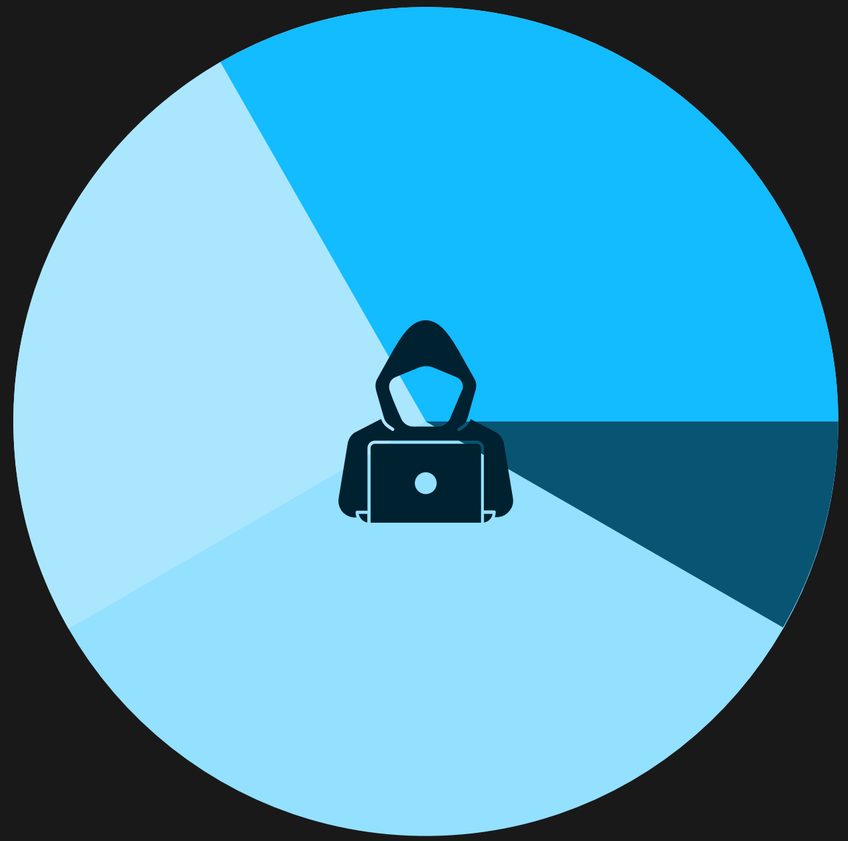
Reverse Engineering: Different Profiles

The guru: can break white-box ECDSA from objdump output alone.



Reverse Engineering: Different Profiles

The pragmatist: combines reverse engineering expertise with a custom tool stack.



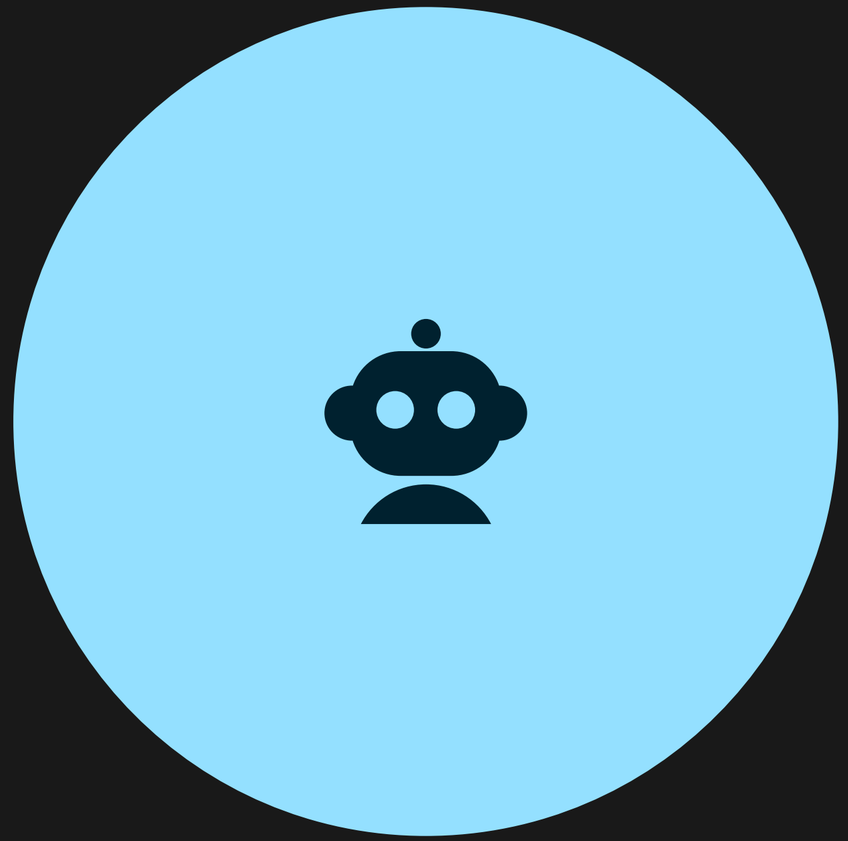
Reverse Engineering: Different Profiles

The decompiler user: relies on clean pseudocode to understand a binary.

Reverse Engineering: Different Models

Frontier models: can or could tackle complex reverse engineering tasks, assuming:

- An unlimited token budget
- No safety guardrail restrictions
- A clearly defined task



Reverse Engineering: Different Models

Medium models: can handle routine reverse engineering and analyze obfuscated code with a harness tailored to the task.

Reverse Engineering: Different Models

Tiny models: can analyze unprotected code or tackle specific tasks with fine-tuning.

The Challenge

The Challenge

16:45–17:00

Security Through Obscurity Is Dead and LLMs Killed It

Field Note · 15 min

Brendan Dolan-Gavitt · XBOW · Main Hall

Brendan Dolan-Gavitt · XBOW

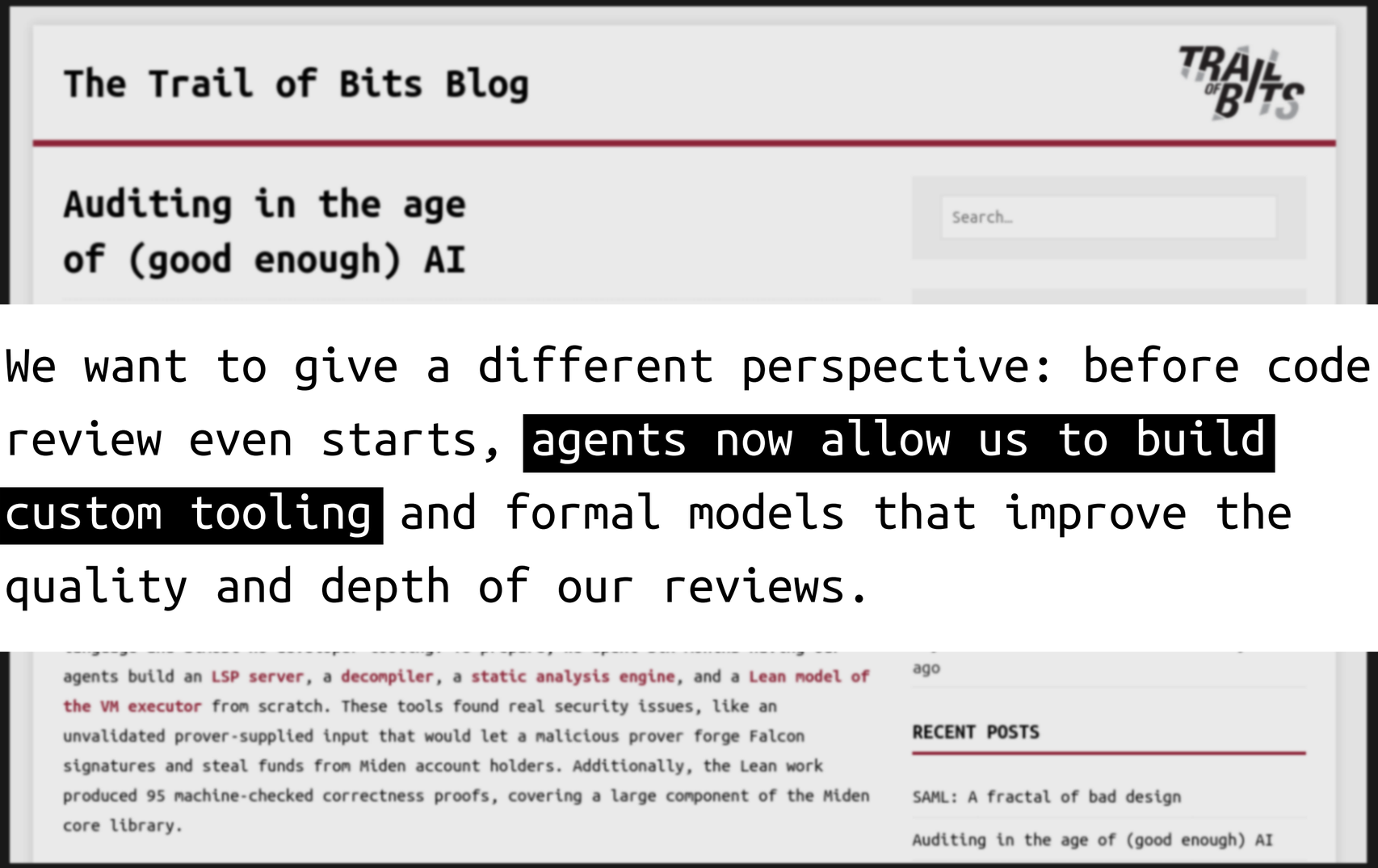
Two case studies in LLM agents ripping through obscurity: reverse-engineering the currency detectors in photocopiers and image editors to build an unphotocopiable cat, and auto-building a faithful QEMU emulator for a laser printer. His conclusion: obscurity now buys days, not years.

Seasoned hackers have always used "security through obscurity is not security" as a mantra to discourage building systems whose security depends on no one caring enough to look too closely at the details. But in practice these systems abound, and they often remain (publicly) unbroken because human expert time is

systems.

We conclude that the security afforded by obscure environments and algorithms can now only provide a few days of protection from anyone with curiosity and tokens to burn, and discuss the implications for anyone building robust systems.

The Challenge



We want to give a different perspective: before code review even starts, agents now allow us to build custom tooling and formal models that improve the quality and depth of our reviews.

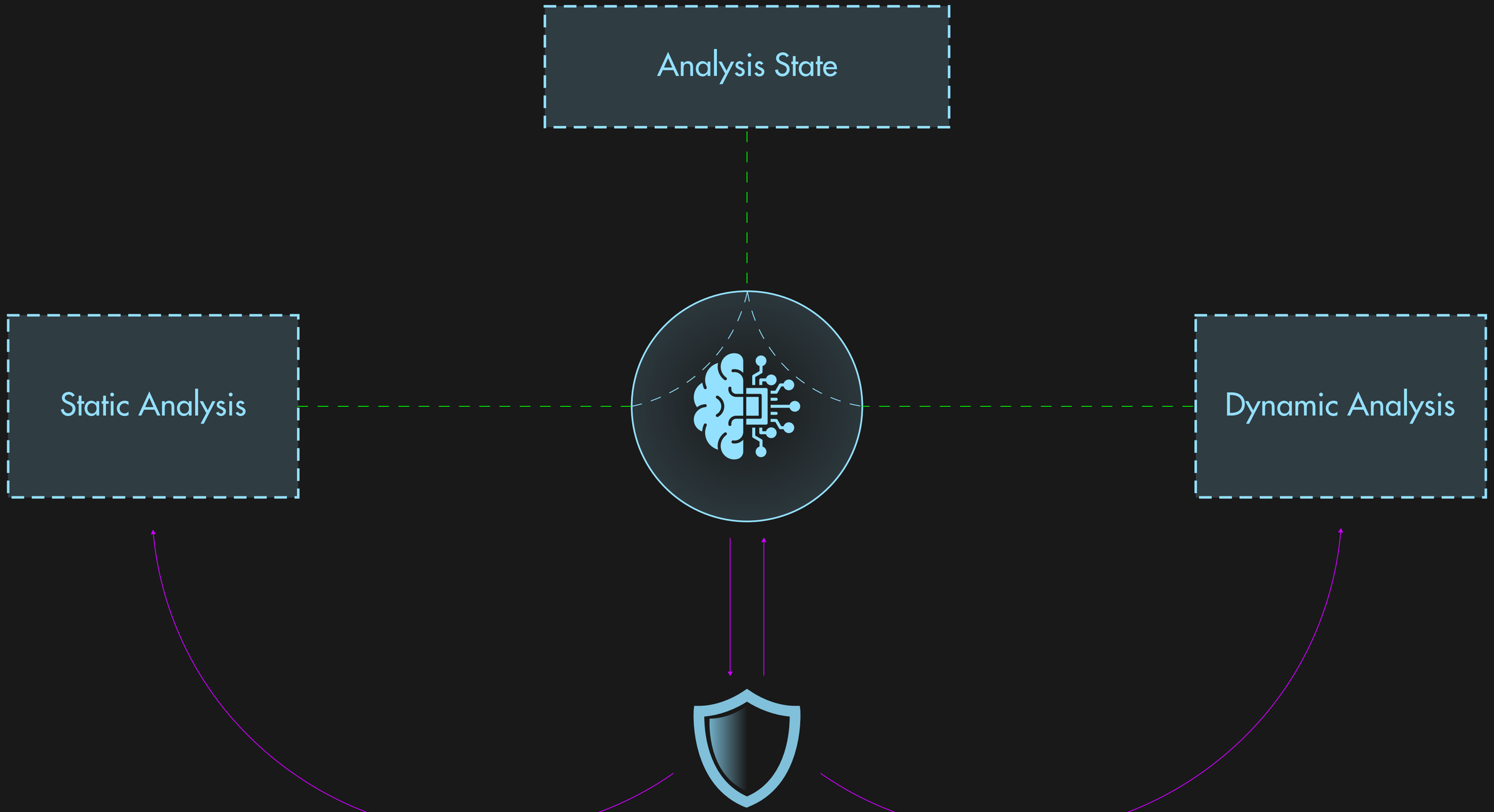
The Challenge

How can tiny models efficiently reverse engineer obfuscated native code?

The Challenge

How can tiny models efficiently reverse engineer obfuscated native code?

> Move from end-to-end frontier-model analysis to a focused tool stack for tiny models.


Analysis State
Static Analysis
Dynamic Analysis

Analysis State

Static Analysis

Dynamic Analysis

hex-rays

.bndb

BinExport

Analysis State

.gpr

.idb

Static Analysis

Dynamic Analysis

.bndb

BinExport

Analysis State

.gpr

.idb

Static Analysis

Dynamic Analysis

hex-rays

Intel Pin

FRIDA

QBDI

.bndb
BinExport

Analysis State

.gpr
.idb

Hooky

DynaMIR

MCStone

DWARF

Static Analysis

Symbi

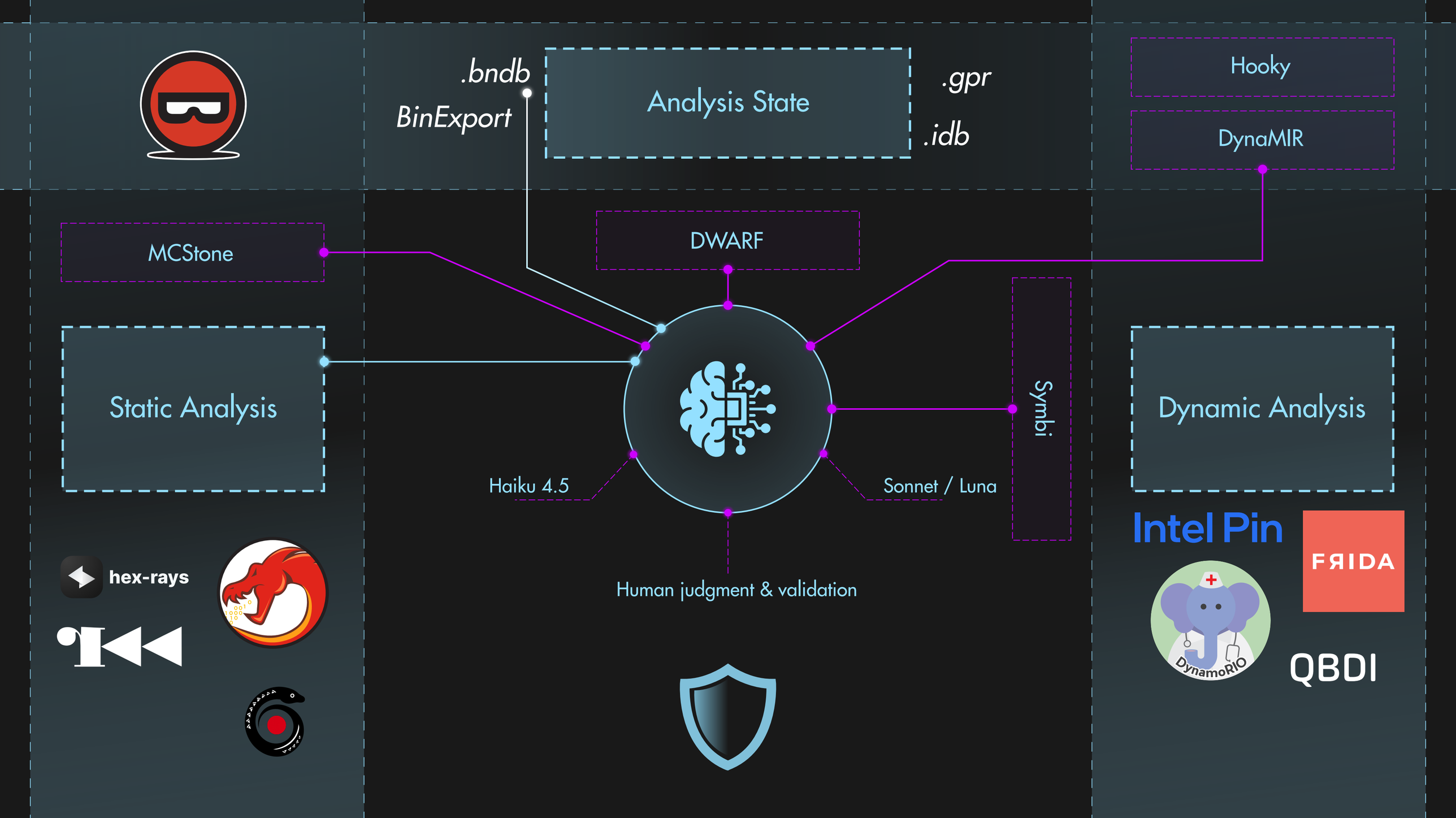
Dynamic Analysis

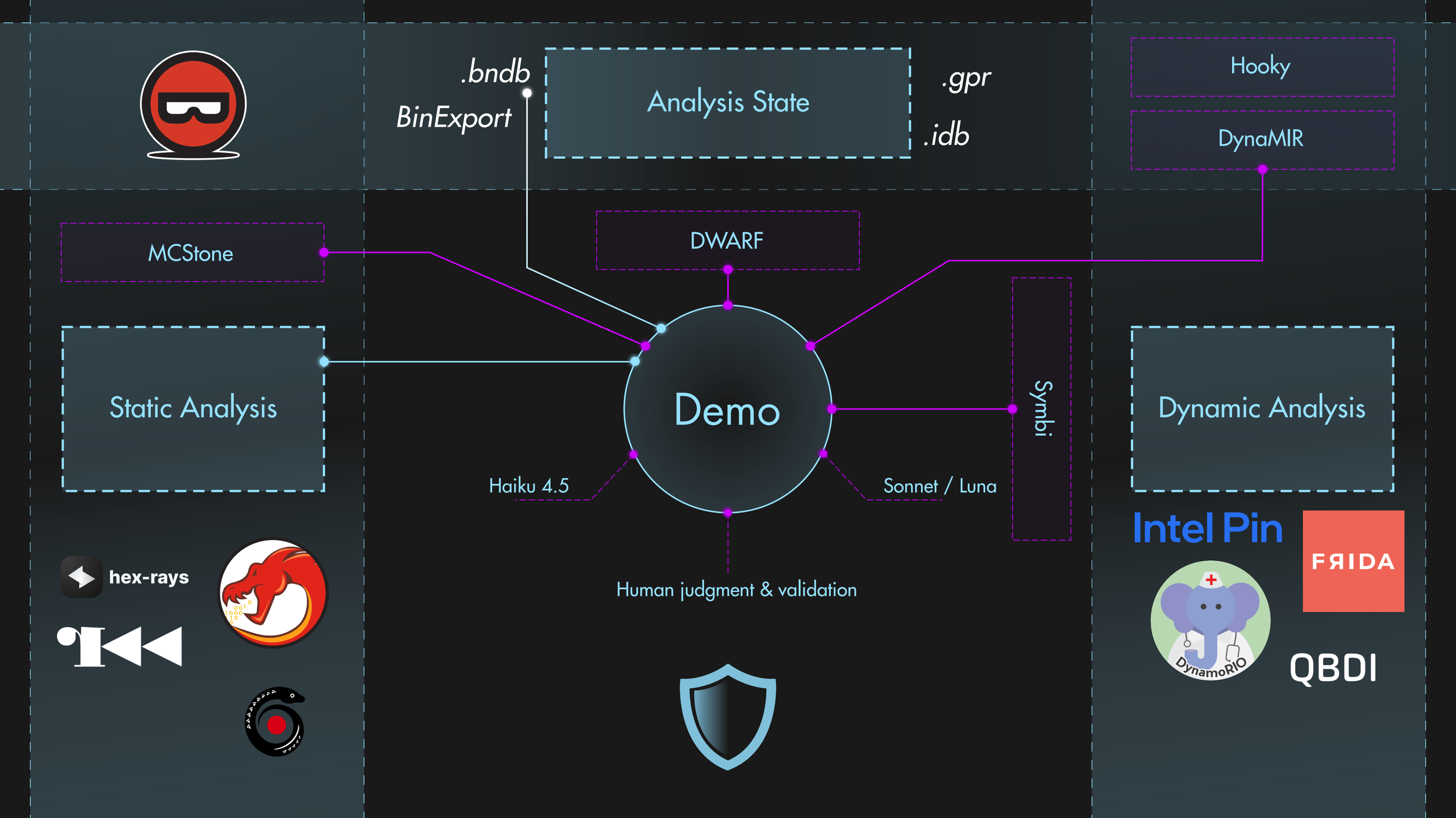
hex-rays

Intel Pin

FRIDA

QBDI


.bndb
BinExport
Analysis State
.gpr
.idb
Hooky
DynaMIR
MCStone
DWARF
Static Analysis
Symbi
Dynamic Analysis
Haiku 4.5
Sonnet / Luna
hex-rays
Intel Pin
FRIDA
DynamoRIO
QBDI
Human judgment & validation


.bndb
BinExport
Analysis State
.gpr
.idb
Hooky
DynaMIR
MCStone
DWARF
Static Analysis
Demo
Symbi
Dynamic Analysis
Haiku 4.5
Sonnet / Luna
Human judgment & validation
hex-rays
Intel Pin
FRIDA
DynamoRIO
QBDI

Experimentation

```
## Required analysis loop

Use Binary Ninja through its Python API and/or MCP for static analysis,
decompilation, cross-references, renaming, and type recovery. Use Symbi for
dynamic analysis, with DynaMIR, Hooky, MCStone, and LIEF as needed. Neither a static
decompilation nor an unexplained trace is a sufficient result.

1. Inventory and scope. Record each binary's path, hash/build ID, architecture,
   dependencies, and likely role. Start at libloader.so and follow initialization,
   JNI registration, library loading, and transfers into unpacked or generated
   code. Explain why each discovered library is included or excluded from scope.
2. Form hypotheses in Binary Ninja. Inspect entry points, constructors,
   exports, strings, callers, and callees. Identify the next unresolved behavior
   and the runtime observation that would confirm or refute it. Recover argument
   and return types, pointer targets, and structure layouts where the
   evidence supports them; account for the Android ARM64 ABI.
3. Export and connect DWARF. Save the .bndb, then export with
   lief-binaryninja-export-dwarf. Do not use Binary Ninja's built-in
   Export as DWARF plugin. It lacks required features.
   Deploy the matching DWARF to the device and attach it to the corresponding
   Symbi target before tracing. Check that loading succeeds and that a
   known function's name and signature appear correctly in the trace.
   Re-export and redeploy after changing names or types.
4. Run a focused experiment. Extend the         module to observe the chosen
   path, including constructors or newly loaded code when relevant. Capture the
   calls, arguments, results, memory changes, or syscalls needed to test the
   hypothesis. Record the command, build revision and local changes, device/app
   state, input action, output files, and whether the intended path completed.
   Preserve a baseline and record any patches, hooks, or execution changes that
   could alter the protection's behavior.
5. Reconcile and persist. Use the trace to revise the static interpretation.
   Apply supported names, prototypes, structures, and explanatory comments to
   the .bndb, update the report and function index, and refresh the DWARF.
   Repeat with the next unresolved protection path. A trace ending early or a
   branch not executing is a coverage gap, not proof that the behavior is absent.
```

Experiment setup

- Prompt: ~130 lines
- No skills
- 3 MCP servers:
  Git, filesystem, ripgrep
- Android obfuscator with no known online documentation
- Fully sandboxed (bubblewrap)

Claude Haiku 4.5

- Useful overview of the protections
- Time: ~20 min - Cost: $1.63
- Tokens: 2.6k input, 91.6k output
- Cache tokens: 8.1M read, 177.5k written

Claude Haiku 4.5

- Useful overview of the protections
- Time: ~20 min - Cost: $1.63
- Tokens: 2.6k input, 91.6k output
- Cache tokens: 8.1M read, 177.5k written

Claude Sonnet 5

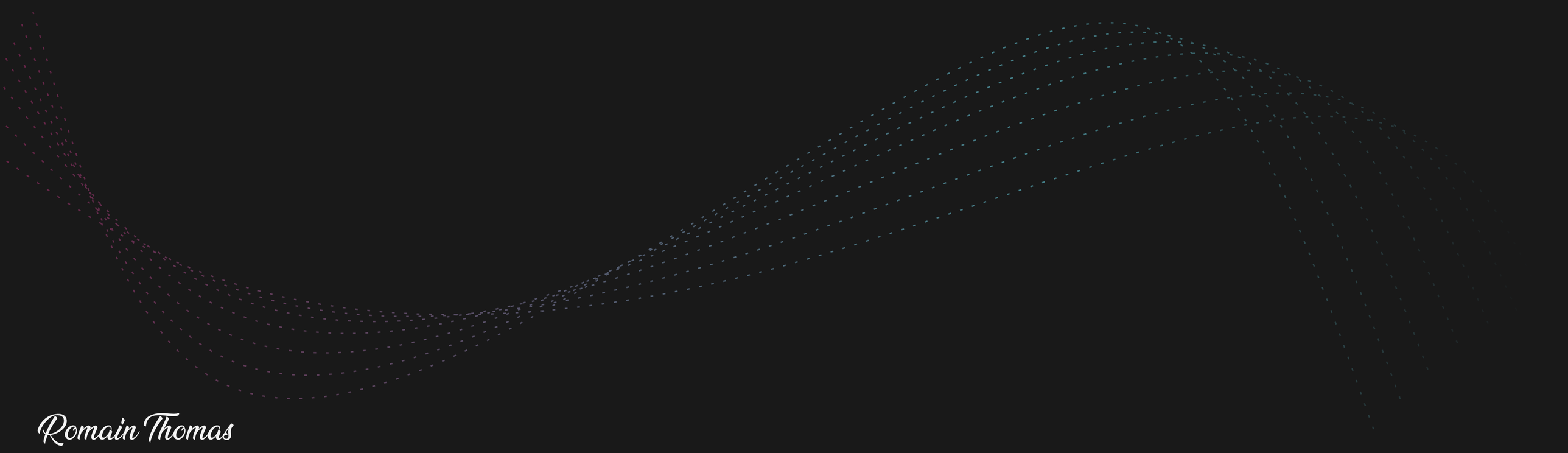
- Promising analysis of obfuscated protections
- Time: ~15 min, Cost: ~$16
- Tokens: ~1.3M

Work in progress…

Work in progress…

…and early results look promising.

Thank you for your attention



Romain Thomas