

# Malware Analysis and Generation: from IOCs to Malware and Malware to IOCs

Marcus Botacin<sup>1</sup>

<sup>1</sup>Assistant Professor  
Texas A&M University (TAMU), USA  
botacin@tamu.edu @MarcusBotacin  
marcusbotacin.github.io

# Whoami

## Background

- Assistant Professor @ TAMU (since 2022); CS PhD @ UFPR, Brazil (2021).
- **Malware** at all levels: ML-based detectors, sandboxes/tracers, HW-based detectors.
- 1st place (attacker *and* defender) ML malware evasion challenges (2019–2021).

## Relevant projects

- NSF SaTC: Hardware Performance Counters as next-gen AVs.
- LPS/BAH: *YARA Generator*.

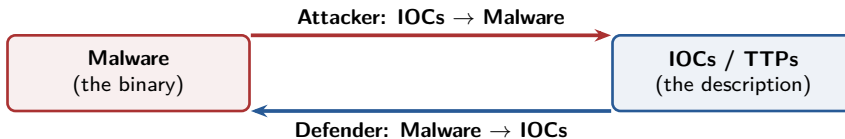
# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 The Hard Part: Report Interpretation
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

## Two directions, one object



- **Attacker** knows the intent, must *produce* the binary.
- **Defender** has the binary, must *recover* what matters.
- Same currency sits in the middle: the **IOCs**.

# The hypothesis

## Claim

On **both** sides, LLMs **solved the easy part** — writing code / driving tools. The hard part is **definition**: specifying the threat. And it is the *same* problem for attacker and defender.

- **Attacker:** spec the malware → code it → get past the model.
- **Defender:** read the threat → spec the rule → drive tools like an analyst.

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# What is an IOC? (a real 2026 example)

## Atomic / technical:

- File hashes (SHA-256)
- C2 domains / IPs
- Mutexes, registry keys
- Scheduled-task names, API-hash constants, build/campaign IDs

### PureRAT / CNB Bot config (REF1695)

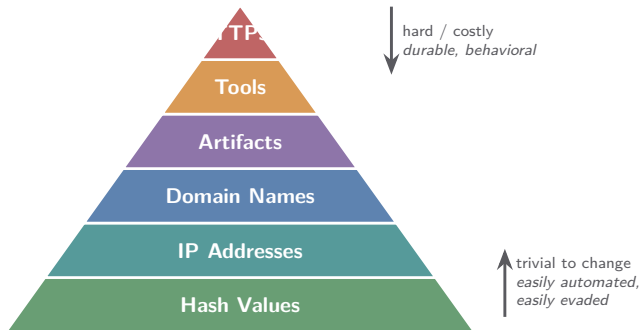
```
C2: windirautoupdates[.]top
Mutex: Aesthetics135
Task: HostDataProcess
Auth token: 0326GJSEC...
Build ID: 23-01-26
```

Both sides live or die by these: the attacker must *reproduce/avoid* them; the defender must *recover/detect* them.

---

Config example: Elastic Security Labs, *Fake Installers to Monero* (REF1695), 2026.

# Not all IOCs are equal: the Pyramid of Pain



- **Bottom** = atomic. Cheap for both to automate, cheap for attackers to change.
- **Top (TTPs)** = behavioral. The durable, *hard-to-specify* layer.

## The thesis, in one picture

LLMs commoditize the **bottom**. The **top** is where the real work — and this talk — lives.

Pyramid of Pain: D. J. Bianco, 2013 — <http://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html>

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# A shared vocabulary: MITRE ATT&CK and CISA reports

## MITRE ATT&CK

The common language for **TTPs** — tactics → techniques. Used by *both* red and blue teams.

## CISA advisories (#StopRansomware)

The canonical **threat report**: IOCs + TTPs mapped to ATT&CK, shared via STIX/TAXII.

## The key observation

The attacker's **spec** and the defender's **spec** are written in the *same language* — ATT&CK techniques.

e.g. the attacker's "building blocks"  $\approx$  ATT&CK techniques:

- AutoRun → T1547 (Persistence)
- DLL Injection → T1055
- Anti-VM / debugger → T1497

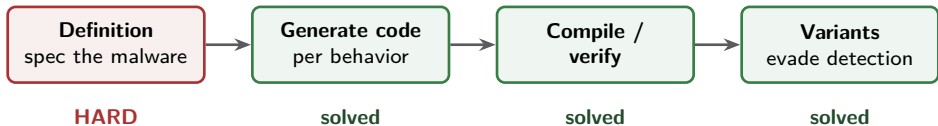
---

MITRE ATT&CK — <https://attack.mitre.org>; CISA #StopRansomware — <https://stopransomware.gov>.

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# The attacker pipeline (overview)



Counterintuitively, the **right** side is the solved part.

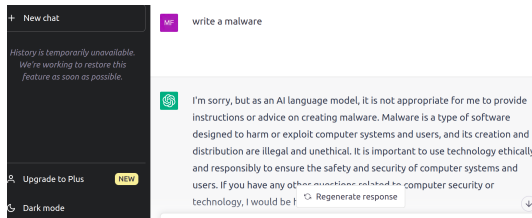
Why “Generate Code” is Solved

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOC → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOC
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

Why “Generate Code” is Solved

# (1) Jailbreaking — the guardrail is not a wall



- Direct refusals are routinely bypassed with prompt engineering / jailbreaks.
- Given the bypass, the model *will* write the code.
- No longer hypothetical: the first AI-orchestrated cyber-espionage campaign was reported in 2025.

---

Anthropic, *Disrupting the first reported AI-orchestrated cyber espionage campaign, 2025* — <https://www.anthropic.com/news/disrupting-AI-espionage>.

Why “Generate Code” is Solved

## (2) Abliteration — remove the guardrail entirely

### Idea

Instead of *bypassing* refusals, **ablate** them: identify and remove the refusal direction in the weights, or use an intentionally uncensored model.

- No guardrail ⇒ nothing to bypass.
- Open-weight models make this a practical attacker option.

So far: code generation *works*. Let us make it even easier, with structure.

---

e.g. Acosta-Bermejo et al., *Automated malware source code generation via uncensored LLMs*, Applied Sciences, 2025.

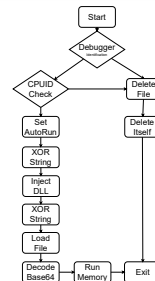
Why “Generate Code” is Solved

### (3) Building blocks — even a guardrailed model complies

#### GPTthreats-3 (my work)

A guardrailed model will *not* “write malware” — but it *will* write each innocuous **building block**. Decompose, then assemble.

- 10 behavioral blocks (AutoRun, injection, anti-VM, XOR, ...).
- Each block maps to an ATT&CK technique.
- Full sample assembled in ~2 minutes.



**Malware skeleton** — blocks generated by the LLM.

M. Botacin, *GPTthreats-3: Is Automatic Malware Generation a Threat?*, IEEE WOOT, 2023.

Why “Generate Code” is Solved

### (3) One block, many implementations $\Rightarrow$ variants

```
1  BOOL CPU_Identification(){
2      int CPUInfo[4];
3      __cpuid(CPUInfo, 0);
4      if (!strcmp((char *)&CPUInfo[1], "uneG"))
5          return 1;
```

Code: CPU identification via CPUID.

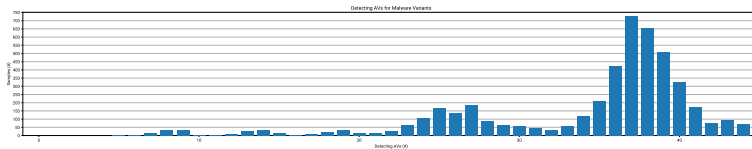
```
1  BOOL CPU_Identification() {
2      if (IsProcessorFeaturePresent(
3          PF_XMMI64_INSTRUCTIONS_AVAILABLE))
4          return TRUE;
```

Code: CPU identification via processor feature.

Same semantics, different APIs  $\Rightarrow$  evades API-based detectors.

Why “Generate Code” is Solved

## (3) Variants and the long tail



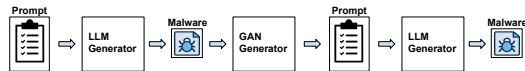
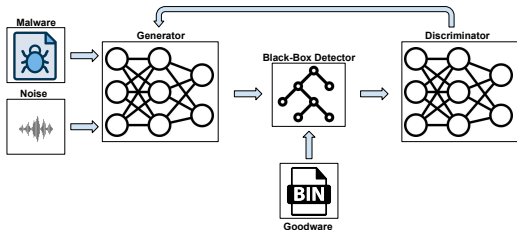
≈5K functional variants; detection spread **4–55 AVs**. Attacker needs *one* gap; defender must cover *all*.

---

M. Botacin, IEEE WOOT 2023. Two-peak curve = AV updates over a 2-week submission window.

Why “Generate Code” is Solved

## (4) Automating the block spec — GAN + LLM



LLM *realizes* them in a working binary.

GAN decides *which* functions.

GAN = automated spec generator (feature space); LLM bridges to the problem space (dead-code insertion / runtime linking).

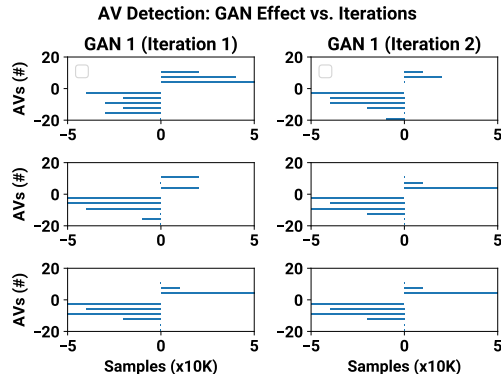
---

Dondapati, Gurram, Botacin, *When GANs meet LLMs*, ACM TAISAP, 2026.

Why “Generate Code” is Solved

## (4) It works — and at scale

- 3 GAN architectures; features  $K=150 \rightarrow 1500$ .
- Up to **99%** attack success rate.
- Evades up to **20** real AVs.
- Iterative loop keeps improving evasion.



**AV detection** (in/de)crease vs. GAN iterations.

Dondapati et al., ACM TAISAP, 2026.

Why “Generate Code” is Solved

## So what is left?

Code generation, variants, even evasion — **all solved.**

What is *not*?

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - **The Hard Part: Definition**
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# “Give me a ransomware” does not specify a malware

The naive request fails (GPThreats-3, Finding #8)

A generic prompt yields *prose about* malware, or trivial non-functional code.

“ransomware” leaves open:

- enumerate *which* files?
- which cipher, keyed how?
- persistence? C2? wiper fallback?

“keylogger” leaves open:

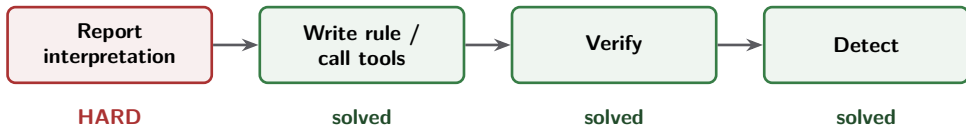
- hook-based, polling, or kernel?
- exfil channel?
- stealth / persistence?

The **name is not a spec**. The real work is turning intent into a precise behavioral spec (a set of ATT&CK techniques with parameters). **That** is the unsolved part.

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
  - The Hard Part: Report Interpretation
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# The defender pipeline (overview)



*Same move, mirrored: rule-writing and tool-calling are solved.  
**Report interpretation** is the hard part.*

Why Rule-Writing & Tool-Calling are Solved

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# Rule writing is solved (given the spec)

## Community tools

- DocYara — RAG over YARA best practices.
- NOVA — “YARA for prompts” (detect attacker LLM use).
- Report summarization, threat-actor suggestion.

## My group

- AutoPYara — automatic YARA via semi-supervised clustering.
- LPS/BAH *YARA Generator* (YaraGen grant).
- AutoStreamYara: Stream-based rule generation.

A YARA rule is the defender’s **spec unit** — the mirror of the attacker’s building block.

---

DocYara/NOVA: T. Roccia (SecurityBreak), 2025–2026. AutoPYara: N. Nguyen (TAMU MSc), 2025.

Why Rule-Writing & Tool-Calling are Solved

## Agentic tool-calling is solved

- **Agent** = LLM + tools + loop + goal; the LLM *drives* the tools (Ghidra-MCP, capa, YARA, FLOSS).
- Real labs: G DATA (REMnux + VT/Ghidra/x64dbg MCP, per-type skills); Elastic's Claude pipeline auto-extracted configs from *hundreds* of samples.
- Benchmarks: strong, cheap **triage**.

### Honest caveat

Triage is solved; deep **RE / static unpacking** still falls short — big gap between best and rest.

Given the spec, tools *get driven like an analyst*.

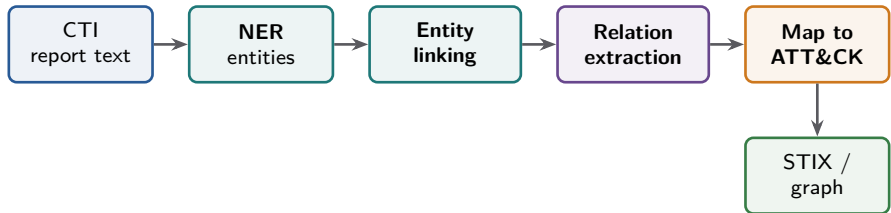
---

Blazytko (synthesis.to), 2026; Hahn/G DATA, 2026; Elastic REF1695, 2026; Malcat LLM benchmark, 2026.

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- The Hard Part: Report Interpretation
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# How it is done today: NER → linking → ATT&CK



This is the defender’s **definition** step: turning an unstructured report into structured TTPs — the mirror of the attacker’s “extract behaviors” step, run in the opposite direction.

... and the field says it is *not* solved

## SoK: Automated TTP Extraction from CTI Reports — *Are We There Yet?*

A systematization + unified re-evaluation of 40+ report-to-TTP approaches.

- ① A **performance ceiling** that current approaches cannot overcome.
- ② Traditional **NLP/NER beats** modern embedder- and LLM-based approaches in realistic settings.
- ③ The blocker: **ambiguity in the TTP ontology + lack of quality datasets**.

The defender's **definition** step is *empirically* the unsolved one — exactly like the attacker's.

---

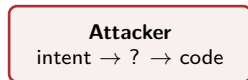
M. Büchel et al., *SoK: Automated TTP Extraction from CTI Reports – Are We There Yet?*, USENIX Security 2025, pp. 4621–4641.

# Agenda

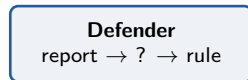
- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

## The Shared Diagnosis

Both sides: execution solved, *definition* is the frontier



missing: requirement layer



missing: interpretation layer

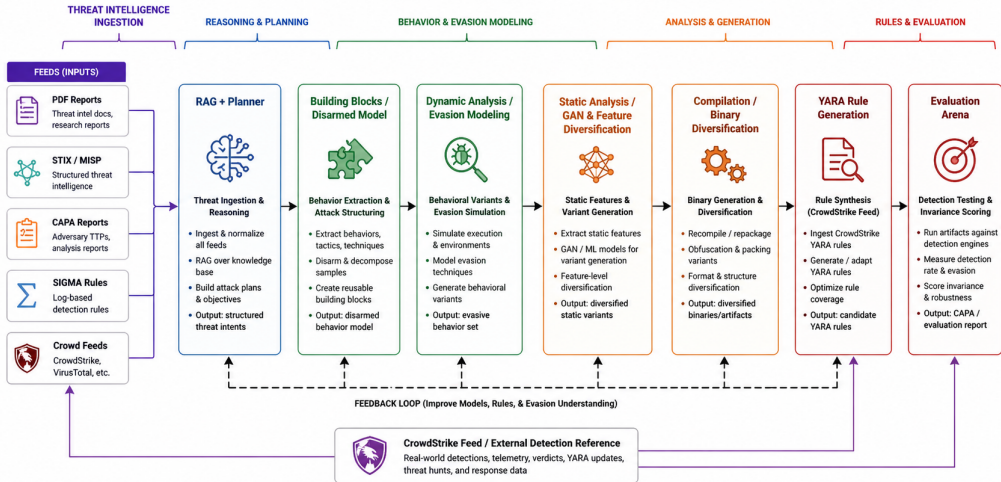
- My research adds the missing **definition layer** on each side.
- **Attack:** a requirement / interpretation / IOC layer *before* code generation.
- **Defense:** structured behavior extraction from reports (beyond naive NER).

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

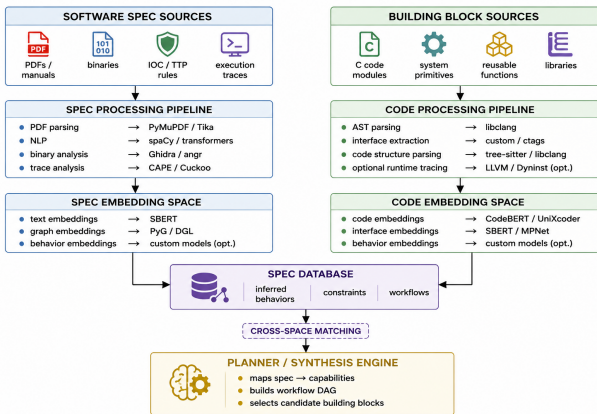
Attack: a Requirement Layer before Code Generation

# The new attacker architecture (event-driven agents)



Attack: a Requirement Layer before Code Generation

# Inside the requirement layer: planner + typed blocks



Blocks are **typed** by ATT&CK technique:

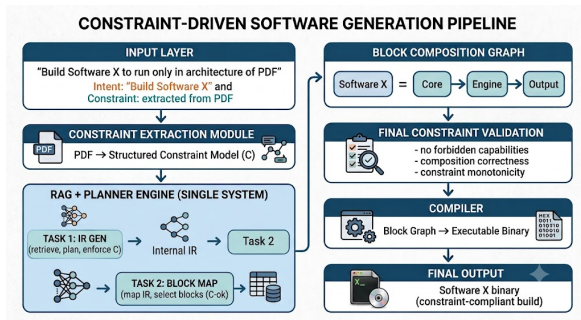
```

{
  "name": "launch_process",
  "implements": ["TTP-X"],
  "goal": "launch executable",
  "apis": ["CreateProcessA"],
  "effects": ["process_created"],
  "constraints": ["windows"]
}
    
```

Dual embedding (spec space ↔ code space)  
→ cross-space matching → synthesis.

Attack: a Requirement Layer before Code Generation

# Constraint-driven generation & open problems



## Open problems (generator):

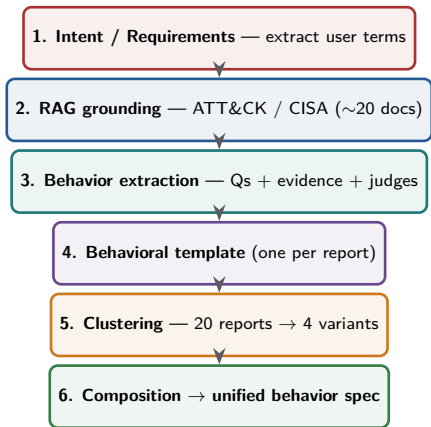
- **TTP coverage** — parse MITRE from samples; ask  $N$  models for  $N$  variants.
- **Skeleton validation** — is the “main” correct (all blocks reached, right order)?
- **Feed grounding** — raw IPs/URLs:  $V0$  assume →  $V1$  the agent queries (downloader? C2?).

Related: *Threat2Traffic* (OpenReview MrcEFZ5ckv); attack-planning + coding partitioning (arXiv:2501.16466).

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# My proposal: from report to a unified behavior spec



## The key idea

Don't ask an LLM for “the TTPs”.  
**Interrogate** each grounded document with structured questions, attach **evidence**, then **cluster** phrasings into behavioral variants.

Output: nodes = actions; edges = ordering + data flow. The spec that feeds synthesis.

# Why this beats naive extraction

| Challenge               | What goes wrong                    | Mechanism                     |
|-------------------------|------------------------------------|-------------------------------|
| WHAT/HOW entanglement   | implementation leaks into behavior | 2-stage grounding             |
| Implicit semantics      | behavior not explicitly stated     | inquiry / template            |
| Hallucination           | model invents constraints          | evidence + UNKNOWN + judges   |
| Semantic variation      | many phrasings, one behavior       | template clustering           |
| Behavioral polymorphism | genuinely different variants       | multiple templates            |
| Relational semantics    | ordering / dataflow matter         | explicit relation questions   |
| Evidence propagation    | constraints lost between stages    | provenance on template fields |

Directly targets the Büchel ceiling: the ambiguity *is* the problem — so make it explicit.

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# The symmetry

|                      | <b>Attacker</b> (IOCs→Malware)     | <b>Defender</b> (Malware→IOCs)      |
|----------------------|------------------------------------|-------------------------------------|
| Spec unit            | building block (ATT&CK technique)  | YARA / detection rule               |
| Spec source          | “what is a keylogger / ransomware” | threat report (CISA + ATT&CK)       |
| LLM role             | realize code from spec             | drive tools like an analyst         |
| The obstacle         | guardrails (bypass / ablation)     | trust (verification, human-in-loop) |
| Ground truth         | compile + sandbox                  | runnable script (not report)        |
| Scale trick          | GAN-guided feature spec            | agentic pipeline + MCP + skills     |
| <b>The hard part</b> | <b>definition</b>                  | <b>interpretation</b>               |

Both collapse to the same unsolved layer.

# The shared bottleneck

## One claim

The challenge is **understanding IOCs** — the *specification* of the threat — and it is common to attackers and defenders.

- LLMs commoditized the **bottom** of the pyramid (atomic IOCs, coding, tool-driving).
- They did *not* commoditize knowing **what a threat is** (the TTP layer) — the shared battleground.
- **Dual-use:** the same GAN+LLM generator is also a *defense* (adversarial retraining: AUT 78.5% → 89.5%).

---

Adversarial-retraining result: Dondapati et al., ACM TAISAP, 2026.

# Agenda

- 1 Introduction
  - The Hypothesis
  - IOC Background
  - The Shared Language
- 2 Attack: IOCs → Malware
  - The Attacker Pipeline
  - Why “Generate Code” is Solved
  - The Hard Part: Definition
- 3 Defense: Malware → IOCs
  - The Defender Pipeline
  - Why Rule-Writing & Tool-Calling are Solved
- 4 My Research: Moving It Forward
  - The Shared Diagnosis
  - Attack: a Requirement Layer before Code Generation
  - Defense: Structured Report Interpretation
- 5 Consolidation
  - The Symmetry
  - Closing Remarks

# Research directions

## Generator (attack)

TTP coverage guarantees; skeleton validation; grounding live feeds (IP/URL enrichment); malware from *general* definitions (constraint-driven synthesis).

## Interpretation (defense)

Break the report→TTP ceiling via better ontologies + datasets; structured, evidence-backed extraction; verifiable (script-based) agentic defense with the analyst on the verdict.

## Closing the loop

Detection-feedback-driven obfuscation (select O-LLVM flags from detection results).

---

Büchel et al., USENIX Security 2025; Botacin, WOOT 2023; Dondapati et al., TAISAP 2026.

# Takeaway

The arrow reverses, but the hard problem does not move.

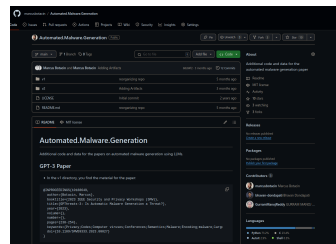
It is always the **IOC** — and the IOCs that matter  
live at the **top** of the pyramid.

# Thanks!

## Papers & code

- *GPThreats-3*, IEEE WOOT 2023.
- *When GANs meet LLMs*, ACM TAISAP 2026.
- Code: [github.com/marcusbotacin/Automated.Malware.Generation](https://github.com/marcusbotacin/Automated.Malware.Generation)

Acknowledgments: NSF CNS-2327427; LPS/BAH YARA Generator.



Questions? Comments?  
botacin@tamu.edu @MarcusBotacin  
[marcusbotacin.github.io](https://github.com/marcusbotacin/Automated.Malware.Generation)