Catch Me If You Can: Detector-Resistant Evasion via Semantics-Preserving Command Re-Realization

Muhammad Shoaib
University of Virginia

Hare Sudhan Muthusamy
Independent Researcher

Tareq Alkhatib
Fortinet

Wajih Ul Hassan
University of Virginia

***Abstract*—Red teams require evasion techniques to test Security Information and Event Management (SIEM) detection rules, yet existing approaches are (1) manual, (2) rely on string-level obfuscations (such as encoding schemes and quoting tricks) that are easily reversed by de-obfuscators, and (3) provide limited rule coverage. This leaves unexplored semantic-preserving evasions that achieve identical effects through different utilities, preventing assessment of whether rules detect attack intent or merely surface patterns. We present SPECTRA, an automated evasion generator that preserves attack effects while transforming command-line realization through functionally equivalent utilities and argument structures. By reasoning over semantic representations rather than syntactic patterns, SPECTRA automatically generates more durable and effective evasions. On Windows Sigma `process_creation` rules, SPECTRA achieves 72.9% rule coverage compared to 37.6% for AMIDES (the state-of-the-art method), with only 4.5% of evasions reversed by de-obfuscators versus 78.1% for AMIDES (17.4 times more resistant). When evaluated against the state-of-the-art evasion detector at its zero-false-positive operating point, SPECTRA achieves a detection rate of only 22.7% compared to 69.9% for AMIDES. SPECTRA also outperforms five general-purpose LLMs across metrics.**

1. Introduction

Red teams simulate realistic attacks to test whether an organization's security defenses can detect genuine threat [15, 26, 30]. A fundamental question drives these assessments: do Security Information and Event Management (SIEM) detection rules [1, 9, 34] capture the intent of malicious behavior, or do they merely match surface patterns that attackers can easily circumvent? SIEM systems monitor system events and raise alerts when observed activities satisfy rule predicates over fields such as `CommandLine` and `EventID` [7, 32, 36, 39]. Sigma is a widely used rule format for expressing such conditions over structured event records [32]. An attacker evades a rule when the malicious procedure achieves its intended effect but no longer satisfies the rule's detection conditions. Understanding the robustness of detection rules against evasion is essential for building resilient defenses.

Security researchers have long studied how rule-based detectors can be evaded by crafting obfuscated attack variants [46, 49, 66, 67]. These include encoding schemes (Base64, hex encoding), quoting tricks, character manipulations (caret insertions in CMD), and wrapper constructs. Recent work, notably AMIDES [66], systematically studied evasion techniques for Sigma rules [1] by manually analyzing and handcrafting evasion variants. The authors applied string-level transformations including CMD and PowerShell encodings, quoting tricks, and caret insertions while preserving the original command structure.

Existing manual approaches face critical limitations for red team assessments. More generally, turning procedure-level evasion into an automated systems problem raises three technical challenges.

1. Limited Scalability and Coverage. Manual evasion generation requires analysts to examine each rule, understand its semantics, identify triggering commands, and handcraft variants. This limits coverage: AMIDES generated evasions for only 37.6% of the 292-rule SIGMA benchmark [1], leaving 62.4% untested. Manual effort prevents continuous testing as rulesets evolve and makes full evaluation infeasible for organizations with hundreds or thousands of rules.

2. Brittle Evasions Vulnerable to Countermeasures. String-level obfuscations are easily reversed: 78.1% of AMIDES evasions are de-obfuscated by off-the-shelf tools [6, 29]. At the zero-false-positive operating point of the state-of-the-art evasion detector [66], 69.9% are flagged as suspicious. This brittleness stems from shallow syntactic manipulations (quotes, carets, encodings) that produce characteristic patterns distinguishable from benign commands; once normalized, original rule literals reappear and trigger detection.

3. High Collateral Alert Rates. When evaluated against the full ruleset, AMIDES evasions trigger numerous unintended alerts: 79 total collateral alerts across the benchmark, averaging multiple per evasion. String-level transformations introduce patterns matching unrelated rules, reducing practical utility for realistic attack simulations.

Taken together, these observations define the core technical problem addressed in this paper. A practical evasion system must recover a behavioral summary from a triggering command, identify detector literals that must not reappear, find alternative procedures that realize the same behavior under different utilities and argument contracts, and instantiate those procedures into Windows command lines without reintroducing the detector's cues.

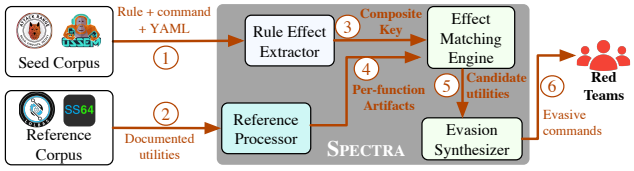


Figure 1: High level workflow of SPECTRA

To address these challenges, we introduce SPECTRA[1], the first fully automated, semantics-preserving evasion generator. Our key insight is that *robust security testing requires semantic equivalence over syntactic obfuscation: rather than disguising how an attack is written, we test whether rules detect what the attack accomplishes.* While prior approaches obfuscate strings (e.g., Base64 encoding), SPECTRA transforms procedures by replacing utilities, reshaping arguments, and changing command structure while preserving attack effects. This reveals whether detection rules are brittle pattern matchers or robust behavior detectors. At a high level, the task is to transform a seed command, that is, a command line that originally triggered the rule, into a new command line that preserves the seed's effects, avoids the rule literals matched by the detector, and remains legal for the replacement utility. Operationally, SPECTRA analyzes the triggering command and the space of documented utilities separately, matches them through shared behavioral structure, and then instantiates a compatible alternative into a concrete evasive command.

SPECTRA takes two inputs and produces validated evasive commands. In the evaluated setting, the *seed corpus* contains Windows Sigma `process_creation` rules, namely single-event rules over process-start records, paired with representative triggering commands from the source logs. The *reference corpus* contains documented system utilities from LOLBAS [20], a catalog of Windows binaries commonly repurposed by attackers, together with ss64 [35] and Microsoft documentation [25]. SPECTRA processes these inputs through four components as shown in Figure 1. The first two components analyze the rule side and the utility side separately, the third joins them through structured matching, and the fourth turns a compatible match into a concrete evasive command. Figure 3 instantiates the first two components on the running example, and Figure 4 continues with matching and synthesis.

1. Rule Effect Extractor. The Rule Effect Extractor extracts semantic effects from seed commands and identifies an avoid set. Given a rule and triggering command, it reasons over what the command accomplishes rather than how it is written. A BiLSTM-CRF tagger [53] segments normalized commands into `FLAG` and `VALUE` spans, assigns semantic types (`path`, `url`, `dll`), and a trained classifier maps flag-value-type tuples to primitive operations (*read*, *encode*, *write*). For example, `certutil.exe -encode file.bin encoded.txt` yields the effect sequence `read->base64_encode->write`. The processor builds an effect graph, a typed graph whose edges record which arguments enable which primitives, and extracts the avoid set, which records rule literals and operators (exact, contains, endswith, regex). To guard against correlated modeling errors, SPECTRA runs two extractor instances with different initializations and retains only commands where they agree on both the effect set and binding chains.

2. Reference Processor. In parallel, the Reference Processor analyzes documented utilities to discover which documented functions realize which effects. For each utility function, it maintains two independent views: normalized command examples (Example-View) and surrounding documentation prose (Anchor-View). The same consensus extractor from the Rule Effect Extractor analyzes examples to build effect graphs, while a separate text encoder predicts argument schemas from prose. A schema serves as the utility's argument contract: it specifies required versus optional flags, admissible value types per flag, flag arities, and positional templates. Consistency checks compare Anchor-derived schemas with Example-View patterns and cross-source agreement, retaining only schemas passing these tests. This grounds schemas in documented intent rather than any single model.

3. Effect Matching Engine. This component maps effect graphs to candidate utilities by indexing both seeds and reference functions using composite effect keys, retrieval keys that capture which primitives are present, their multiplicity, and coarse binding-chain structure. When queried with a seed effect graph, the index retrieves documented utilities that can realize the same effects with compatible argument patterns. The index enforces multiplicity agreement (primitives appearing multiple times must match), chain-shape agreement (binding patterns must align), and cross-binary preference when available, ensuring candidates preserve behavior without collapsing to trivial near-copies of the seed.

4. Evasion Synthesizer. The Evasion Synthesizer selects one candidate utility and synthesizes concrete arguments. It aligns the seed effect graph to each candidate's binding summary, a compact record of how that utility's arguments enable primitives, computing an alignment score measuring how many seed edges match candidate edges with compatible types, and a coverage score measuring what fraction of seed binding chains admit compatible realizations. Candidates with zero alignment or low coverage are filtered. Among remaining candidates, the generator applies lexicographic selection prioritizing effect graph alignment, binding chain coverage, exact key matches, and procedural diversity (different binaries, flags, and positional patterns). Once a utility is selected, the argument filler synthesizes concrete values guided by the binding summary while avoiding rule literals. For example, given the `certutil` seed with effects `{net.fetch, fs.write}`, the generator selects `curl.exe`, aligns URL and PATH arguments to the appropriate primitives, and fills the required command-line flags like `-L -s` and `-o` that realize the same download-and-write behavior without triggering the rule.

1. SPECTRA stands for Semantics PrEserving Command Transformations for Rule Avoidance.

We evaluate SPECTRA against manual evasions from AMIDES on the 292-rule Sigma `process_creation` benchmark [1]. SPECTRA generates evasions for 72.9% of target rules (vs. 37.6% for AMIDES), is 17.4× less likely to be undone by de-obfuscation (4.5% vs. 78.1%), causes 4.9× reduction in alerts per fully evaded rule, fully evades 1.92× more rules (208 vs. 108), and is 3.1× harder for a state-of-the-art evasion detector (22.7% vs. 69.9% detection). We further show that SPECTRA generalizes to newly published Sigma `process_creation` rules drawn from a newer Sigma revision. Moreover, SPECTRA also outperformed five general-purpose LLMs across all metrics.[2]

SPECTRA makes the following main contributions:

- *Automated procedure-level evasion:* First system to generate semantics-preserving evasions for Windows Sigma `process_creation` rules.
- *Effect graph abstraction:* Command representation that captures primitives and argument-to-effect bindings for effect-preserving transformation.
- *Cross-binary synthesis:* Diversity-biased generation that prefers distinct utilities, flags, and positional patterns over string-level variants.
- *Schema-constrained generation:* Automated synthesis and validation under learned schemas, binding constraints, and avoid-set enforcement.
- *Evaluation in the target setting:* Large-scale evaluation on Windows Sigma `process_creation` rules, including generalization to newly published rules from a newer Sigma revision.

2. Preliminaries

Core notation. We use four small notation families throughout the design. In the detector model, Φ denotes the trigger condition, $R = (spec, \Phi)$ a detection rule, and $\widehat{c}$ an evasive command emitted by SPECTRA.

The reusable core of SPECTRA reasons over a small set of recurring objects. On the rule side, the *avoid set* $\mathcal{A}_i$ records detector literals that must not reappear, and the *seed effect graph* G_i summarizes the behavior to preserve. On the reference side, the *reference effect graph* G_u summarizes documented behavior, the *binding summary* B_u records how arguments enable primitives, and the *argument schema* $\mathcal{S}_u$ records the allowed argument contract. The matching stage later introduces a composite effect key K, and the synthesis stage later introduces the compatibility measures α and β. We introduce these objects in that order below.

The evaluated setting is Windows Sigma `process_creation` rules rendered through a fixed PySigma/Azuma backend. A rule alerts when its trigger condition Φ evaluates true. We model command execution with a map $E(c)$ from command c to ordered log events under a Windows aware normalization N that applies Unicode NFKC, optional case folding, flag alias unification (`/f`, `-f`, `--flag`), quoted span preservation, path normalization to backslashes, and wrapper annotation. Under this model, an evasive command $\widehat{c}$ preserves the seed's effects while making the paired rule evaluate false, that is, $\Phi_R(E(\widehat{c})) = \text{FALSE}$. Unless stated otherwise, evasion is per rule; a *global evasion* E' is an attacker sequence satisfying $\Phi_R(E') = \text{FALSE}$ for all rules.

2. We include LLMs (used with default parameters) to contextualize SPECTRA's performance against widely available generation capabilities, not to claim optimality over potentially fine-tuned models. Specialized prompting strategies or task-specific training could improve LLM results; our goal is to demonstrate that purpose-built semantic reasoning outperforms both manual approaches and off-the-shelf general-purpose models.

Kind	Description with brief example
`FLAG`	Switch with no value; e.g., `flag="-O"`.
`PATH`	Filesystem path (file/dir; UNC allowed); e.g., `path="C:\textbackslashTemp\textbackslasha.exe"`.
`URL`	Network resource; e.g., `url="http://x/y"`.
`REG`	Registry key/val; e.g., `reg="HKCU\textbackslashSoft\textbackslashRun"`.
`NET`	Explicit socket target; e.g., `ip="10.0.0.5" port=53`.
`PAYLOAD`	Encoded/opaque blob; e.g., `b64="SAFgA..."`.
`TASK`	Scheduler semantics; e.g., `task="Backup" trigger="daily02{:}00"`.
`RAW`	Incidental literal required by some binaries; e.g., `txt="literal"`.

TABLE 1: CSL operand kinds.

This event-local rule-matching setting is distinct from detectors that reason over process trees, parent-child ancestry, or provenance graphs assembled across multiple events. Evaluating such systems would require a different benchmark and a different notion of semantic preservation, because the preserved object would no longer be a single command-line realization but a larger multi-event execution structure.

Command Statement Language (CSL). We next fix the command representation shared by extraction, matching, filling, and validation. We represent commands using a compact grammar written inline as `Statement ::= EXEC {ARG}`, where **EXEC** records the launcher (the normalized primary binary) and **ARG** is a multiset of typed operands, including flags and positionals. CSL assigns each operand a kind that we reuse throughout the design; Table 1 lists these kinds with examples. CSL is the common surface form used throughout the pipeline. The extractor uses it to identify typed arguments, the matching stage uses it to compare positional and binding-chain structure, and the synthesizer and validator use it to realize and check a concrete command line. In the running example, the `URL` and `PATH` arguments shown in Figure 3 are CSL operands that later reappear in the seed and reference effect graphs.

3. Design

SPECTRA combines a reusable core with setting-specific adapters. The reusable core operates over effect graphs, schema inference, effect matching, and constrained filling, while the evaluated adapter fixes the Windows normalization rules, primitive vocabulary, process-creation event schema, and Sigma backend behavior. The design follows the same order as the pipeline in Figure 1. At a high level, the pipeline answers four questions: what behavior and rule literals does

```
function EvasionPipeline(s, R)
  % indexing, retrieval, and evasion for seed s
  if ¬ValidSeed(s) then return ⊥
  if effect set I not built then
    R_h ← RCrawl(R)
    R_c ← NTok(R_h) % Example-View slice
    (G_r, S_r) ← RefGraph(R_c)
    I ← BuildIdx(G_r, S_r)
  (G_s, B_s, A, K_s, b_s, π_s) ← SeedIngCtx(s)
  if G_s = ⊥ then return ⊥
  U_c ← LookupIdx(I, K_s)
  if U_c = ∅ then return ⊥
  ĉ ← EvasionSyn(G_s, B_s, A, b_s, π_s, U_c)
  if ĉ = ⊥ then return ⊥
  return ĉ

function SeedIngCtx(s)
  (r, c_raw, y) ← UnpackSeed(s)
  c_norm ← Norm(c_raw)
  (c̃, A) ← SeedLink(c_norm, y)
  x_s ← NTok(c̃)
  (E_s, B_s) ← CmdExt(x_s)
  if FailCons(E_s, B_s) then return ⊥
  if |E_s| = 0 then return ⊥
  G_s ← SeedGraph(E_s, B_s)
  if Abstain(G_s) then return ⊥
  b_s ← PrimaryBin(x_s)
  π_s ← PosSig(x_s)
  K_s ← MakeKey(G_s, E_s, π_s, B_s)
  A ← AddAvoid(A, y)
  return (G_s, B_s, A, K_s, b_s, π_s)

function EvasionSyn(G_s, B_s, A, b_s, π_s, U_c)
  % filters, ranks, and fills a candidate utility
  U_adm ← ∅
  hasCross ← HasCrossBin(b_s, U_c)
  for each (u, B_u) ∈ U_c do
    if Adm(u, hasCross, G_s, B_u, π_s) then
      U_adm ← U_adm ∪ {(u, B_u)}
  if U_adm = ∅ then return ⊥
  (u*, B_u*) ← SelectBest(U_adm, G_s, π_s)
  for t = 1 to k do   % retry budget, e.g., k=3
    a ← FillArgs(u*, G_s, B_s, A, B_u*)
    ĉ ← AsmCmd(u*, a)
    e ← Sanity(ĉ, u*, G_s, A, B_u*)
    if e = OK then return ĉ
  return ⊥
```

Figure 2: SPECTRA Algorithm: summary of SPECTRA's end-to-end evasion-generation pipeline. Abbreviations: RCRAWL (reference-corpus crawler/cache), NTOK (normalization and tokenization), REFGRAPH (reference effect-graph builder), CMDEXT (consensus command extractor), BUILDIDX (effect matching engine), SEEDINGCTX (seed-ingest context linker), LOOKUPIDX (composite effect key lookup), FAILCONS (consensus-failure gate), MAKEKEY (composite effect key constructor), PRIMARYBIN (primary-binary extractor), POSSIG (positional-signature extractor), ADDAVOID (avoid-set constructor), EVASIONSYN (evasion synthesizer), HASCROSSBIN (cross-binary diversity predicate), ADM (admissibility predicate), and ASMCMD (command-line assembler).

the seed induce, what behavior and argument contracts do documented utilities expose, which utilities are compatible with the seed, and how can one be instantiated into a concrete evasive command. These questions correspond to the Rule Effect Extractor, Reference Processor, Effect Matching Engine, and Evasion Synthesizer, respectively.

The same few objects recur across these four components, using the notation introduced in Section 2. On the rule side, the pipeline carries forward an avoid set and a seed effect graph. On the reference side, it builds a reference effect graph, a binding summary, an argument schema, and a small quality annotation q_u used only for ranking. Matching then uses a composite effect key and two compatibility measures, and synthesis ranks the surviving candidates and realizes one of them as a concrete evasive command. Figure 3 instantiates the first two components on the running example, and Figure 4 continues with lookup, pairing, ranking, filling, and validation.

In this setting, SPECTRA starts with two inputs. **1) Seed Corpus:** The seed corpus is a collection $\mathcal{S} = \{s_i\}$, where each seed $s_i = \langle r_i, c_i, y_i \rangle$ contains a stable rule identifier r_i, a representative triggering command c_i, and the corresponding Sigma YAML block y_i, as shown in Figure 3. We build $\mathcal{S}$ from public Windows attack corpora including Mordor and related Security Datasets [10, 11, 21] and Splunk `attack_data` [33], then use the ATT&CK-to-events mapping [28] to join labeled process-creation events to Sigma rules. **2) Reference Corpus:** The reference corpus $\mathcal{R}$ is a collection of documented system utilities drawn from Microsoft docs [25], ss64 [35], and LOLBAS [20]; each reference item u provides a documented command template, argument descriptions, and example usages.

3.1. Rule Effect Extractor

The Rule Effect Extractor turns each seed into a structured description that the rest of the pipeline can use. Starting from the seed tuple (rule identifier, example command, and YAML), it normalizes the command, extracts the rule literals that must be avoided, and links the YAML to the normalized command to recover a concise behavioral view of what the command does. We assume that the underlying logs have not been tampered with. Securing log integrity is out of scope of this work, and deployments can rely on specialized systems [63, 71] to protect the logging pipeline. From this view, the extractor emits the seed-side objects used throughout the rest of Section 3. It builds a seed effect graph that captures the enabled actions and their argument bindings, records the primary binary and a typed positional signature, and derives a composite effect key that later drives retrieval from the effect matching engine. In this representation, the effect graph is the behavioral summary to preserve, the avoid set captures the rule literals that must not reappear, and the composite effect key is the retrieval handle passed to the matching stage.

3.1.1. Seed Ingest & Context Linker. We link the rule's declarative context to the normalized seed command so later stages can pair by effects and enforce schema. At this point the task is to place the triggering command and the rule YAML into a common representation before any reference-side matching occurs. We canonicalize the YAML y_i and compute the avoid set $\mathcal{A}_i = A(y_i)$ with operator and case provenance. When y_i must be backfilled, we enforce temporal consistency by selecting a Sigma revision contemporaneous with the logs that produced c_i (within $\pm\Delta$ days; default ± 14) and record commit hashes and timestamps. We then run UNPACKSEED→NORM→SEEDLINK→NTOK→CMDEXT and gate on FAILCONS. Surviving seeds yield a seed effect graph together with the primary binary and typed positional signature that later stages use for retrieval and alignment.

3.1.2. Paired Extractors and Orthogonal Oracle. A single extractor can make coupled mistakes on both seeds and references, which weakens effect matching and validation. We therefore train two extractor instances that share the same architecture and label guide but differ in their random initial-

izations and in the random permutations of training examples used during learning. Each extractor returns BIO spans, value types, flag arities, and an effect set with bindings. We keep a command only when *both* instances produce the same effect set with compatible bindings, following agreement-based reliability signals in prior work [43, 57, 72]. We accept the command if the two effect sets are identical and the binding chains show high overlap, and we choose this overlap threshold on a stratified development subset with oracle labels as described in §4 so that agreement conditioned precision stabilizes while preserving coverage for downstream pairing. Items that fail either the effect-set agreement or the binding overlap threshold abstain via FAILCONS (Figure 2, L22) and are not indexed or paired. To avoid correlated errors, we also evaluate a stratified subset with an orthogonal oracle that does not reuse learned extractor signals. The oracle checks PowerShell AST structure, `schtasks` XML or materialized arguments, and `reg.exe` grammar. We provide agreement conditioned precision with Wilson 95% confidence intervals [69]. Implementation details and full estimators are provided in Appendix A.5 and Section 5.6.

3.1.3. Seed Effect Graph Builder. Once the two extractors agree on an effect set and compatible bindings for a seed, we still need a structured object that can be carried into retrieval, pairing, and validation. We therefore build an *effect graph* G_i for each seed. Formally, G_i is a directed graph whose nodes are (i) CSL `ARG` nodes from §2, annotated with their value types and, when applicable, flag arity, and (ii) primitive nodes in $\mathcal{P}$. We add a directed edge from an argument node to a primitive node when that argument *enables* the primitive according to the learned association described in §4. The canonical key $k(\mathcal{E}_i)$ is the sorted primitive list for the inventory $\mathcal{E}_i$, obtained by ordering and deduplicating primitives under a fixed total order. Together with the binding chains $\mathcal{B}_i$, which record which argument-to-primitive edges contributed mass, the effect graph lets us later check that candidates preserve not only the effect set but also a compatible binding pattern, in line with prior graph-based representations of program and malware behavior [45, 56, 59].

These seed-side objects play distinct roles later in the pipeline. The sorted primitive inventory contributes to retrieval through the composite effect key, while the effect graph and binding chains are carried forward to compatibility checking and semantic validation. For the certutil seed in Figure 3, the seed effect graph G_i contains edges `ARG(URL)` $\rightarrow$ `net.fetch` and `ARG(PATH)` $\rightarrow$ `fs.write`, and the set-based component of the composite effect key is the sorted primitive list $k(\mathcal{E}_i)$ = `fs.write|net.fetch` with the associated chain sketch. We index a seed only if $|\mathcal{E}_i| \geq 1$ and $\max_{p \in \mathcal{P}} \pi_p \geq \lambda$; otherwise, for example under competing `FLAG`/`VALUE` segmentations or dominance by `opaque_blob`, the system abstains. Edges remain explainable because each comes from a scored `(flag,value,type)` tuple. Canonicalization and tokenization follow N, and auxiliary posteriors over $\mathcal{M}$ are retained as alignment features without enlarging $k(\cdot)$.

3.1.4. Text-based Calibration and Validation. We do not execute commands that SPECTRA generates. Instead, for each function u=(bin, fn) we supervise from two textual views: normalized example invocations E_u (Example-View) and surrounding prose A_u (Anchor-View). An example encoder and a text encoder predict posteriors over $\mathcal{P}$ and $\mathcal{M}$ from these views. Because the views are only weakly independent, we use their agreement as a stability signal in the co-/multi-view sense [43, 57] to calibrate thresholds via temperature scaling (§4) [48] and to gate acceptance together with the validator's schema, typing, arity, and rule-literal checks. Section 5 reports the resulting operational validity. We stress this setup in three ways. A *negative control* swaps A_u with $A_{u'}$ for $u' \neq u$ while keeping E_u fixed; a *paraphrase test* semantically perturbs A_u and E_u; and, when multiple public sources exist, we check cross-site consistency. We report precision on the agreement subset with Wilson 95% confidence intervals (CIs) [69] and use these estimates inside SANITY in Figure 2. Candidates that fail these checks either consume one of the bounded retries in EVASIONSYN or cause the system to abstain for that seed.

3.2. Reference Processor

Overview. SPECTRA's *Reference Processor* establishes how documented utilities realize the same primitives by running RCRAWL→NTOK→REFGRAPH→BUILDIDX (Figure 2, L5-8). This stage is the reference-side counterpart of the Rule Effect Extractor: it turns documentation into the same kinds of objects, but for candidate utilities rather than for seeds. In the running example, the right half of Figure 3 shows these objects.

3.2.1. Reference Corpus (Documented System Utilities). We build $\mathcal{R} = \{r_u\}$ by RCRAWL and NTOK and index by utility functions u = (bin, fn). For each function we keep two textual views: an *Example-View* E_u of preformatted command invocations and an *Anchor-View* A_u of the heading and surrounding prose. Sources are LOLBAS, SS64, and Microsoft Windows and PowerShell documentation [20, 25, 35]. We retain the binary, function label, and ordered list of extracted example command lines. All examples are normalized with the same map N defined in §2, and documented placeholders such as `<PATH>`, `<URL>`, and `<DLL>` are preserved as typed `VALUE`s under CSL. Downstream stages consume only artifacts derived from $\mathcal{R}$ using the same consensus extractor as for seeds, and Figure 3 shows a slice of this corpus for the running `certutil`→`curl` example.

3.2.2. Reference Crawler & Cache. We construct $\mathcal{R}$ by crawling and caching public LOLBAS, SS64, and Microsoft Windows and PowerShell documentation pages. For each utility function u = (bin, fn) we extract command examples and surrounding prose as E_u and A_u, and compute simple crawler-derived statistics for quality annotations q_u in pairing and ranking. Appendix A.2 gives the crawler implementation, caching strategy, and view construction details.

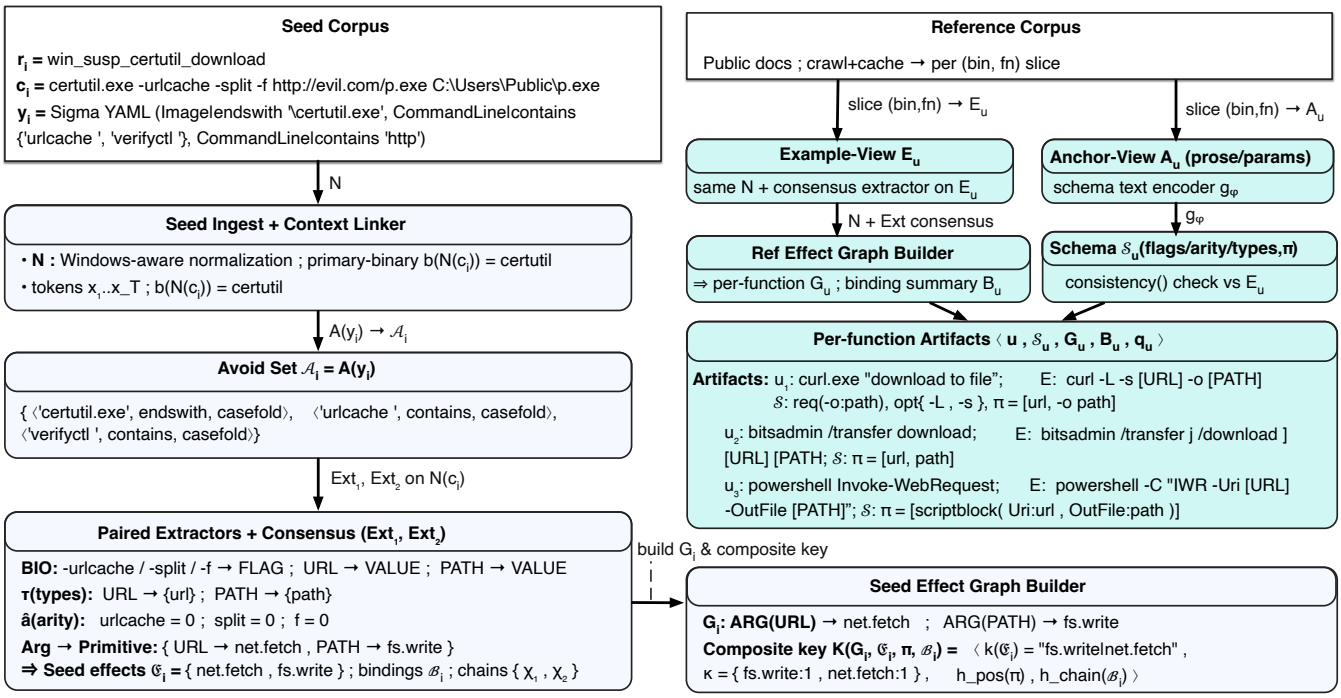


Figure 3: Running example: seed and reference processing for a Sigma rule detecting `certutil`-based downloads [4]. The seed effect graph G_i and reference effect graphs G_u are shown as the effect graph examples and are also referenced throughout §3.

3.2.3. Reference Effect Graph Builder. On the effect side, we use only the E_u. Each normalized example in E_u is passed through the *consensus extractor* (§3.1.2) to yield an example-level effect set and bindings. At the function level, the reference effect graph G_u summarizes which primitives are supported by the documented examples, while the binding summary B_u compresses how the arguments in those examples enable those primitives. These are the reference-side objects later consumed by matching, pairing, and filling. The function-level effect graph G_u is the union of example edges with support counts; edges supported only by prototype similarity are marked so confidence can propagate end to end. We compute the canonical function key $k(\mathcal{E}_u)$ by the same procedure used for seeds, retain the binding summary B_u, and form the composite effect key with MAKEKEY. Pairing later requires at least one compatible binding between the seed graph G_i and B_u, which is stricter than effect-set equality alone. For each (bin, fn) we retain the key, effect graph, normalized examples, and crawler-derived quality annotation q_u. Figure 3 shows these reference-side objects for the running example.

3.3. Effect Matching Engine

Overview. The previous stages give us comparable artifacts on the rule side and the reference side. The Effect Matching Engine maps the composite effect key from MAKEKEY to utility functions and their schemas and bindings, and is queried by LOOKUPIDX (Figure 2, L11). Its basic question is: given a seed command with a particular effect set and binding-chain structure, which documented utilities can realize the same effects with a compatible argument pattern?

At its core, this stage is a retrieval problem over shared intermediate objects rather than a direct rewrite of command strings. The composite effect key narrows search to procedures with the same primitive inventory and coarse structure, leaving fine-grained argument compatibility to the pairing stage. The matching stage therefore follows a coarse-to-fine pattern: exact key agreement first, then argument-level compatibility checks in pairing. Querying the index with a seed key retrieves procedures that match the seed under $\mathcal{P}$ and still carry enough structure for later argument validation against $\mathcal{S}_u$. In the running download example (Figure 4), the seed key for $\{\texttt{net.fetch}, \texttt{fs.write}\}$ retrieves `curl`, `bitsadmin`, and `PowerShell`. Multiplicity and chain-hash agreement then ensure that only procedures with matching effect counts and coarse binding structure are passed on to alignment.

3.3.1. Composite effect key (MAKEKEY). The composite effect key is the coarse retrieval handle used by the index; it is not yet the final compatibility decision. We write $K(G, E, \pi, \mathcal{B}) = \langle k(E), \kappa(E), h_{\text{pos}}(\pi), h_{\text{chain}}(\mathcal{B}) \rangle$, where $k(E)$ is the sorted primitive list, $\kappa(E) \in \{0, 1, 2\}^{|\mathcal{P}|}$ records primitive multiplicities (capped at 2), and $h_{\text{pos}}(\pi)$ and $h_{\text{chain}}(\mathcal{B})$ are 64-bit hashes of positional signatures and binding chains respectively.[3] The composite effect key prevents false matches. The pair $k(E)$ and $\kappa(E)$ records what behavior is present, including primitive multiplicities.

3. FNV-1a hash with fixed seed; no collisions observed.

The terms $h_{\text{pos}}(\pi)$ and $h_{\text{chain}}(\mathcal{B})$ record how arguments realize that behavior through positional structure and binding chains. Together, these four quantities let the index reject many obviously incompatible procedures before the pairing stage considers them in detail. We require exact matches on all key components: primitive inventory, multiplicity, positional hash, and binding hash must agree. This conservative policy ensures retrieved candidates already match the seed's effect identity and structure.

3.3.2. Argument schema per (bin, fn). For each utility function $u = (\text{bin}, \text{fn})$ we derive an argument schema $\mathcal{S}_u$ that summarizes accepted flags, requiredness, admissible value types, arities, and positional layout under CSL. We estimate it from the Anchor-View A_u so that effect extraction and schema learning use disjoint views: examples say what a utility does, whereas prose says how it should be invoked. A text encoder predicts the candidate flag set, admissible value types, positional signature, and flag arities, and the consistency stage then checks these predictions against E_u and, when available, cross-source agreement. The final schema combines the Anchor-based predictions with the binding summary B_u derived from E_u and serves as the contract the synthesizer must satisfy.

Schema consistency. Each Anchor-derived schema passes through a consistency stage that compares it with the normalized examples E_u and, when available, with cross-source agreement. This stage adjusts or rejects flags, arities, value types, and positional templates that conflict with strong evidence and marks a schema as consistent only when all checks pass. Appendix A.7 gives the exact estimators, thresholds, and edit rules.

Index contents and query. Each index entry stores the utility identifier together with its schema $\mathcal{S}_u$, its quality annotations q_u, and metadata that pairing and later analysis will consult. We query the index with the seed's composite effect key. In the configuration used for all results in this paper, LOOKUPIDX returns only utilities whose composite effect key exactly matches that seed key. In other words, the set key $k(E)$, the multiplicity sketch $\kappa(E)$, the positional hash $h_{\text{pos}}(\pi)$, and the binding chain hash $h_{\text{chain}}(\mathcal{B})$ must all agree. If no such bucket exists, the query returns an empty candidate set and EVASIONPIPELINE abstains for that seed (Figure 2, L11-12).

Consensus gating. We include a seed or function entry in $\mathcal{I}$ only when Ext_1 and Ext_2 agree on the effect set and the binding-chain overlap reaches at least τ_{bind} (§3.1.2); items that fail either test are not indexed. For audit purposes we record, for each item, the agreement bit and the binding-overlap score. On a stratified subset $\mathcal{S}^\star$ we also run the orthogonal oracle ORACLE (PowerShell AST, `schtasks` XML, `reg.exe` grammar) and require that the consensus assignment satisfy the corresponding function-specific constraints. This is the same setup used to define agreement-conditioned precision (ACP) in §3.1.2, and §5 provides the resulting ACP values.

3.4. Evasion Synthesizer

Given effect-compatible utilities from the Effect Matching Engine, the Evasion Synthesizer turns one compatible procedure into a concrete evasive command. It does so in three steps: pairing, selection, and filling with validation. Figure 4 should be read left to right as lookup, compatibility testing, ranking, argument filling, and validation. The seed effect graph and avoid set fix what must be preserved or excluded, while the candidate binding summary and schema specify how the chosen utility can legally realize that behavior.

Using the seed-side objects and the candidates returned by exact key lookup, the generator selects utilities, synthesizes arguments, and validates the resulting command lines. Figure 4 shows $J(u)$ and the filler for the download example. We record each output as a pair $\langle c_i, \widehat{c} \rangle$ so downstream analyses can reason about literal reuse and novelty.

Inputs and contract. By construction of the Effect Matching Engine, each candidate u already matches the seed effect set. The generator consumes the candidate schema $\mathcal{S}_u$ and binding summary B_u and may emit one or more Windows command lines whose first token is the candidate binary, that satisfy the schema, respect the seed avoid set $\mathcal{A}_i$ and seed graph G_i, pass the validator gates in §4, and preserve the extractor-grounded semantics. Thus generation is driven by the candidate schema, the binding summary, and the avoid set, not by edits to the target behavior itself. In the running example, the filler instantiates $\mathcal{S}_{u^\star}$ with the seed URL and PATH to yield `curl.exe-L-s[URL]-o[PATH]` while avoiding all literals in $\mathcal{A}_i$. Formally, each evasive command $\widehat{c}$ must satisfy $\Phi_r\big(E(\widehat{c})\big) = \text{FALSE}$ for the paired rule r; we place no requirement on $\Phi_{R'}\big(E(\widehat{c})\big)$ for any $R' \neq r$ during generation and report cross-rule outcomes in §5.

3.4.1. Candidate Pairing. Within EVASIONSYN (Figure 2, L31-45), retrieved candidates match by effect but may still be incompatible at the argument level. We therefore align the seed effect graph G_i to each function's binding summary B_u and discard utilities whose arguments cannot realize the seed bindings. Pairing asks whether the candidate exposes argument slots that can realize the same primitive-to-argument structure as the seed. We compute an effect graph alignment score, written $\alpha(G_i, B_u) \in [0, 1]$, by solving a maximum bipartite matching problem over edges. One side contains seed edges and the other candidate edges. An edge pair is eligible only when the primitives match and the argument endpoints are schema compatible: positionals must unify in type and relative order, and flags must admit the observed value types and arity. A learned scorer weights the eligible pairs, low-compatibility pairs are ignored, and α is the fraction of seed edges covered by the resulting maximum matching. Candidates with vanishing alignment are discarded because they offer no compatible way to realize the seed bindings even when effect sets match.

We also measure binding coverage, written $\beta(G_i, B_u) \in [0, 1]$, as the fraction of seed binding chains that admit type-compatible realizations in B_u under the same matching. As

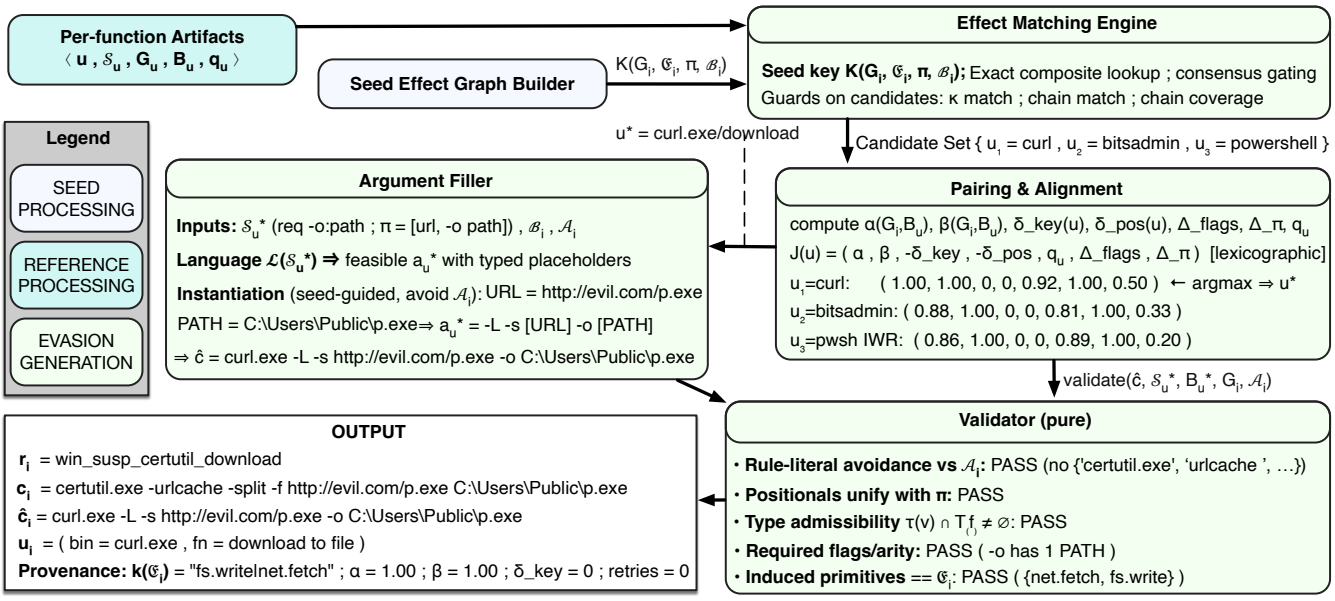


Figure 4: Running example continued: index lookup, pairing, and evasion generation for the seed in Figure 3.

Figure 4 shows, α and β separate candidates that merely share the primitive inventory from candidates whose argument structure can realize the seed behavior. Candidates below fixed alignment or coverage levels are discarded. Appendix A.6 and §4 give the matching details and thresholds.

3.4.2. Diversity Bias. To encourage structural change, EVASIONSYN applies diversity guards only after semantic compatibility has been established through pairing. For each candidate function, it measures whether the binary changes, how different the flag set is, and how different the positional pattern is from the seed. If at least one cross-binary option exists, only cross-binary candidates are kept. In all cases we also require enough flag novelty, enough positional novelty, and enough alignment and binding coverage before a candidate is passed to the selector. Appendix A.6 gives the exact statistics and thresholds, and §4 gives the fixed values used in our experiments. Configuration H in Table 5 ablates these guards as a group and shows that disabling them yields only a small coverage gain while increasing brittleness and collateral alerting, so we retain them in the default configuration.

Lexicographic selection. After alignment and diversity filtering we obtain a small admissible set of utilities. The generator ranks these candidates lexicographically: it first maximizes effect graph alignment, then binding chain coverage, then consults the fixed retrieval-tier fields shown in Figure 4, and only then prefers higher-quality and more diverse candidates. We denote this score vector by $J(u)$. In the exact composite-lookup configuration used in this paper, the retrieval-tier fields are zero for all surviving candidates, so they do not affect the ordering. The ranking therefore refines a semantically admissible set rather than repairing incompatible candidates. This constraint-first policy avoids learned mixing weights and keeps selection reproducible. Appendix A.6 gives the exact definition of $J(u)$. In the download example in Figure 4, this ranking selects `curl` over `bitsadmin` and `PowerShell` because it achieves perfect alignment with an exact key match. The same ordering induces a stable top-L prefix, so the system can emit multiple evasive commands per seed.

3.4.3. Argument Filler. Once u^* is chosen, FILLARGS instantiates $\mathcal{S}_{u^*}$ using the seed binding chains $\mathcal{B}_i$ and the avoid set $\mathcal{A}_i$, and can be invoked multiple times to produce distinct variants for the same seed and utility. The output is a surface string of arguments a_{u^*} that, when prefixed with the binary from u^*, yields the evasive command line $\widehat{c}$. We retain the original seed c_i so that validation can compare the pair $\langle c_i, \widehat{c} \rangle$. In practice we work with a small set of operand kinds such as flags, paths, URLs, registry items, network endpoints, payloads, scheduler items, and literals (Table 1). We write $\mathcal{L}(\mathcal{S}_{u^*})$ for the set of argument strings permitted by the schema $\mathcal{S}_{u^*}$. The filler constructs an argument sequence that respects the required and optional flags in $\mathcal{S}_{u^*}$ and follows the positional template π. A candidate argument string from that set is feasible only if every slot can be instantiated with a schema-admissible value and if its tokens do not match any element of $\mathcal{A}_i$ under the recorded operator semantics (exact, contains, endswith, regex) and case policy.

Feasibility alone is not enough. The resulting command must also induce the same effect set under the consensus extractor and remain aligned with the seed binding chains within tolerance. Instantiation is further biased by the candidate binding summary B_{u^*}, which helps choose among optional flags and positionals when several schema-legal realizations are available. A generative component proposes argument strings similar to canonicalized examples in E_{u^*} while honoring the seed-side bindings and avoid set. We assemble the final command line with ASMCMD, check it

with SANITY, and allow a small retry budget k (Figure 2, L39-45); otherwise the system abstains for that seed.

4. Implementation

Extractor and training. We implement the extractor by annotating Windows command-lines with BIO spans for `FLAG` and `VALUE` and enforce disjoint splits by binary family so that seed- and reference-derived examples live in separate folds. The extractor returns span annotations, a value-typing distribution τ for each `VALUE`, an arity label $\widehat{a}_f$ for each `FLAG`, and a mapping from `(flag,value,type)` tuples to primitives in $\mathcal{P}$ (§3.1.2). Two independently initialized instances (Ext_1, Ext_2) run in parallel, and downstream stages only consume examples where the effect sets agree and binding-chain overlap is high. We measure overlap as the Jaccard similarity between the two sets of binding chains and treat it as high when this similarity exceeds a fixed threshold chosen by a small grid search on a stratified development subset with orthogonal oracle labels; we select the smallest value at which agreement-conditioned precision plateaued while retaining enough examples for pairing (0.8 in our experiments). For flag arity we use a hierarchical backoff estimator that interpolates across binary, function, and flag contexts with Laplace smoothing and a contribution from the current command; unseen flags fall back to a prior derived from reference statistics, and the validator enforces the inferred $\widehat{a}_f$. Configuration L in Table 5, which disables negative controls and abstention, raises coverage but reduces confidence and de-obfuscation, consistent with keeping this conservative consensus gate in the main configuration.

Indexing and retrieval. We build an effect matching engine keyed by the composite effect key $K(G, E, \pi, \mathcal{B})$ from §3.3, which extends a pure set key with multiplicity, positional, and binding-chain sketches. Each entry stores a function schema $\mathcal{S}_u$ and a binding summary B_u inferred under the text-grounded supervision protocol. Retrieval is a pure hash lookup: LOOKUPIDX returns only utilities whose composite effect key exactly matches the seed key; if no such bucket exists, the query yields an empty candidate set and the pipeline abstains for that seed. Candidates are forwarded to pairing only when the consensus gate from §3.1.2 holds.

Selection, filling, and thresholds. From the retrieved set we form an admissible subset with the diversity guards described in §3.4.2, then compute an effect-graph alignment score $\alpha(G_i, B_u)$ and a binding-chain coverage score $\beta(G_i, B_u)$ as in §3.4.1. We set the edge-compatibility pruning threshold ε and the acceptance levels for (α, β) once on a validation subset and keep them fixed for all experiments. We then select the function $u^\star$ by a lexicographic policy over (α, β) and auxiliary quality terms, synthesize an argument string a_u that satisfies $\mathcal{S}_u$ while respecting seed-side bindings and the avoid set $\mathcal{A}_i$, and pass the resulting command-line to the validator, which enforces schema conformance, value typing, arity, and the avoid-set check. The diversity predicate ADM (§3.4.2) uses fixed thresholds $(\tau_f, \tau_\pi) = (0.4, 0.3)$ for flag and positional novelty, chosen once by a small grid search on a validation subset and then frozen. Configurations H and M in Table 5, which disable the diversity guards or remove pruning, increase coverage but also raise collateral alerts or reduce the fraction of fully evaded rules, which supports treating these fixed settings as conservative defaults rather than tuning them per workload.

Calibration specifics. We set the primitive decision threshold λ and the mixture weights in the argument-to-primitive scoring head by maximizing a precision-weighted F_β score with $\beta < 1$ on a validation subset where both encoders predict $p \in \mathcal{P}$ above their individual decision thresholds, and we keep these settings fixed for all experiments. We then calibrate both encoder heads with temperature scaling on the same validation subset, defined solely by document-level agreement, and use the resulting calibrated scores in all components that consume encoder outputs. Calibration adjusts only the decision temperatures for the extractor heads; the consensus, alignment, diversity, and pruning thresholds remain fixed once chosen. Configuration C in Table 5, which disables temperature scaling, shows the expected pattern of slightly higher coverage and lower confidence and is consistent with calibration acting as a single global adjustment rather than a tuned knob on the evaluation set.

5. Evaluation

We evaluate SPECTRA along nine research questions: (1) evasion generation coverage; (2) robustness of generated evasions under deobfuscation tools; (3) target rule and collateral evasion rates; (4) operational validity on Windows; (5) overall detection performance; (6) semantic equivalence of generated evasions; (7) component ablations; (8) generalization to newly published Sigma `process_creation` rules; and (9) comparisons against LLM-based baselines.

These questions serve different purposes and are presented in that order. We begin by asking whether a system produces plausible Windows command lines at all, then test robustness against normalization, cross-rule alerting, and a learned evasion detector. We next examine whether the resulting commands remain operationally valid and semantically equivalent, and conclude with ablations and temporal generalization to newer Sigma `process_creation` rules.

We adopt the 292 Sigma rules released with the AMIDES study, which provide evasions for Windows `process_creation` rules [1]. AMIDES evaluates and reports its results on this fixed set, and to our knowledge it is the only publicly described end to end system that generates rule evasive Windows command lines directly from SIEM rules, so we treat it as the state of the art for this task. There is a broader line of work on command obfuscation that produces evasive Windows command lines and PowerShell scripts without consuming SIEM rules, including argument obfuscators and script level transformers [2, 16, 18]. AMIDES already evaluates rule robustness under obfuscation styles drawn from this space. However, these frameworks do not take SIEM rules as input and are not designed to provide coverage estimates over a fixed rule set, so we do

TABLE 2: Evaluation on Sigma rules. **Generation & conformance**: *Gen.* = rules with any candidate, *Cov. (%)* = generation coverage. **Conformance**: *Valid* = Windows single-line conformant candidates, *Conf. (%)* = conformance rate. **De-obfuscation**: *Total* = candidates tested, *De-obf.* = de-obfuscated, *De-obf. (%)* = de-obfuscation rate. **Alerting across ruleset**: H_{tot} = total alerts, R_{TB} *(%)* = target-rule bypass rate, $R_{CB(count)}$ and R_{CB} *(%)* = fully evaded (collaterally bypassed) rules, H_{avg} = alerts per fully evaded rule ($H_{\text{tot}}/R_{\text{CB}}$).

	Generation & conformance				De-obfuscation			Alerting across ruleset				
System	**Gen.**	**Cov. (%)**	**Valid**	**Conf. (%)**	**Total**	**De-obf.**	**De-obf. (%)**	$\mathbf{H_{tot}}$	$\mathbf{R_{TB}}$ **(%)**	$\mathbf{R_{CB}}$	$\mathbf{R_{CB}}$ **(%)**	$\mathbf{H_{avg}}$
AMIDES	110	37.6	103	93.6	512	400	78.1	79	100.0	108	37.0	0.7315
SPECTRA	213	72.9	207	97.2	804	36	4.5	31	100.0	208	71.2	0.1490

not treat them as direct baselines in our evaluation. This benchmark tests within-family temporal generalization for Windows Sigma `process_creation` rules, not cross-category or cross-language generalization.

5.1. Evasion Generation Coverage

We begin with coverage and conformance because the later metrics are only meaningful for rules that receive plausible Windows command lines. Evasion coverage is the share of rules in the 292-rule set for which a system produces *any* candidate string (*Gen.* in Table 2). *Windows conformance rate* is the share of all returned candidates that pass a minimal single-line Windows command check, computed as *Valid* divided by *Valid* + *Invalid* in Table 2. The conformance check has three deterministic checks: single line, balanced quoting and legal escaping under Windows parsers, and Windows-targeted context. These checks follow Microsoft's quoting and parsing guidance for PowerShell [23, 24] and the Windows command shell [22].

These checks are intentionally minimal and were added primarily because general-purpose LLMs often emit malformed or non-Windows strings. Since we provide the same coverage and conformance numbers for LLMs in §B.1, we apply the identical check to SPECTRA and AMIDES for parity. The checks are not a substitute for execution or semantic verification. They act as a guardrail so hallucinated outputs do not inflate downstream metrics. Minimal correctness suffices here because §5.4 later measures operational validity on Windows through actual execution. The goal at §5.1 is only to certify that a candidate is a plausible Windows command line that can be fairly tested in subsequent experiments. Table 2 shows that SPECTRA produces at least one evasion for 72.9% of the rules, while AMIDES does so for 37.6%. Among the candidates each system generates, the share that fails the Windows single-line conformance check is 6.4% for AMIDES and 2.8% for SPECTRA. This approach makes later improvements attributable to semantic changes rather than formatting.

5.2. Robustness to De-obfuscation

We next ask whether generated evasions survive commodity de-obfuscation tools [6, 29, 38] or collapse back into recognizable rule triggers. We define *brittleness* as the fraction of generated candidates that are de-obfuscated by these commodity normalization or decoding tools; a higher fraction means the evasion collapses back to its intent and thus offers little real protection. AMIDES candidates are de-obfuscated in 78.1% of cases (400/512) (see Table 2), whereas SPECTRA candidates are de-obfuscated in only 4.5% (36/804). In defender terms, if a candidate collapses under commodity normalization, the detector effectively sees the original intent and the evasion loses value. SPECTRA alters procedure and argument structure across binaries and functions rather than relying on string-layer tricks, so there is little for these tools to undo, yielding a much lower brittleness rate.

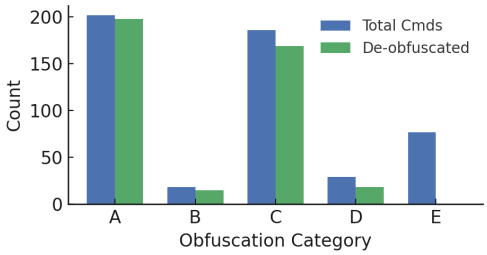
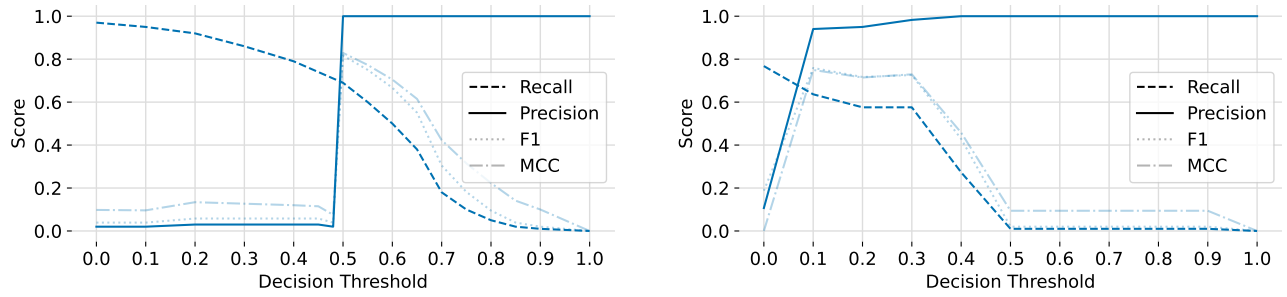


Figure 5: De-obfuscation outcomes for AMIDES' evasive commands. Categories: A = Plain CMD tricks, B = Path normalisation only, C = PowerShell with heavy decoding, D = Mixed CMD and PowerShell -enc, E = Already legible (no obfuscation).

Why is AMIDES *easily de-obfuscated?* A category-level analysis provided in Figure 5 explains the 78.1% deobfuscation outcome. Plain CMD tricks are the largest category, and 198 of 202 are restored by quote, caret, and environment variable normalization. PowerShell-heavy encodings contribute 169 recoveries out of 186, which is precisely where common decoders unwind `-enc`/base64 payloads and string-operator abuse. Path normalisation is smaller but mostly recoverable, mixed CMD and PowerShell wrappers are partially undone, and already legible lines category naturally yield no de-obfuscations. In short, AMIDES leans on string and encoding transformations that the aforementioned open-source normalizers are built to reverse.

5.3. Collateral Alerting Across Ruleset

We next ask whether suppressing the target rule merely shifts alerting to other rules in the same ruleset. For each candidate we simulate Sysmon events and evaluate against all 292 rules. We calculate: total alerts across all candidates (H_{tot}); *target-rule bypass rate* R_{TB} (%), the fraction of target rules for which the evasion suppresses the rule that produced the seed; *fully evaded rules* R_{CB}, the count of rules for which there exists at least one candidate that triggers no alerts by



(a) AMIDES's detection results. D.R. = 69.9%. (b) SPECTRA's detection results. D.R. = 22.7%.

Figure 6: Outcomes of evasion detection component of AMIDES (SOTA adaptive evasion detection) against each system's evasions. D.R. = Detection Rate. D.R. is calculated at the point where there are no FPs, as per AMIDES's protocol.

any rule in the set; and *alerts per fully evaded rule* $H_{\text{avg}} = H_{\text{tot}}/R_{\text{CB}}$. We report R_{CB} as both a count and a percentage of the 292 rules in Table 2. All alerts are computed with Azuma, a PySigma-based matching tool that renders Sigma rules under a fixed canonical normalization.

Both systems achieve a 100% target-rule bypass rate. The difference is collateral alerting. SPECTRA reduces total alerts from 79 to 31 (60.8% fewer) and lowers alerts per fully evaded rule from 0.7315 to 0.1490. The number of fully evaded rules rises from 108 (37.0%) to 208 (71.2%). This shows that SPECTRA not only defeats the intended detectors but also avoids spillover alarms by other rules, which is the stronger notion of evasion in practice (Table 2).

A natural concern is that AMIDES uses handcrafted techniques on the same 292 rules it is evaluated on, which could overfit to that dataset. Unlike AMIDES, SPECTRA *is fully automated*: it trains a shared extractor once, builds effect graphs and schemas from cached documentation, pairs by composite effect keys, and fills arguments under a fixed validator, with no per-rule tuning or manual editing. Fixed model parameters and decision thresholds limit rule-specific leakage and keep the evaluation protocol consistent across runs.

5.4. Operational Validity

This subsection asks whether candidates that pass the earlier string-level checks also execute as valid Windows command lines under an automated harness.

Error classification. We distinguish syntax or usage errors that require changing the invocation from environment-dependent failures. A command is counted as *Not valid* only when the process exits with an error code that Windows and the relevant tool families reserve for invalid parameters or malformed command lines: 2, 87, 1639, -2147024809 (`0x80070057`), or -196608 (`0xFFFD0000`). For MSI, we additionally treat 1619 and 1620 as syntax related only when the accompanying error text matches known usage or parse patterns; otherwise they are treated as environmental. All remaining failures, including missing files, access denied, offline resources, or absent binaries, are labeled *inconclusive* and excluded from the denominator. We install required utilities and replay commands non-interactively. A candidate is **Valid** if it executes without triggering one of the syntax or usage codes above or an equivalent parse error, and **Not valid** only when such an error occurs. Failures that can be resolved only by changing the runtime environment are never penalized as invalid command lines.

TABLE 3: Operational validity under the penalized exit-code policy. Abbreviations: **Inv.** = invalid count; **VPR** = validity per rule (fraction of rules for which all tested evasions were valid); **VPE** = validity per evasion (fraction of individual evasions that were valid); **Codes**× = penalized exit codes with multiplicities. We denote error codes by symbols: e_1=`-2147024809`, e_2=`-196608`, e_3=`2`, e_4=`87`.

System	Inv.	VPR	VPE	Error code frequency
AMIDES	26	0.8968	0.9486	$\{e_1{\times}1,\ e_2{\times}15,\ e_3{\times}3,\ e_4{\times}7\}$
SPECTRA	10	0.9603	0.9875	$\{e_1{\times}5,\ e_3{\times}2,\ e_4{\times}3\}$

Under these conditions, AMIDES attains strong operational validity (0.8968 per rule; 0.9486 per evasion; 26 invalid cases). Despite being fully automated, SPECTRA surpasses the operational validity of AMIDES's manual, expert-curated evasions, reaching 0.9603 validity per rule and 0.9875 validity per evasion with only 10 invalid cases. Under the same, fully automated and reproducible testbed conditions as AMIDES, SPECTRA yields evasions with higher operational validity. Table 3 summarizes these results.

5.5. Evasion detection performance

This subsection asks whether a learned detector designed to recover evasions still flags SPECTRA's variants. AMIDES provides an SOTA detector that learns from rules versus benigns and recovers variants that remain near rule signatures [66]. No other compared system includes an evasion detector, so we reuse the released AMIDES detector and measure, for each system, how often its evasions are still flagged. Figure 6 shows the precision, recall, F1, and MCC curves for AMIDES and SPECTRA as we vary the detector threshold. At the zero false positive operating point reported by AMIDES, the detector flags 69.9% of AMIDES evasions but only 22.7% of SPECTRA evasions. The curves are consistently higher for AMIDES, the distribution the detector was tuned to, and degrade on SPECTRA, where recall drops quickly once we move away from very low thresholds. AMIDES's detector uses vector embeddings with

Family	n	k	Rate	CI_L	CI_U
Overall	260	242	93.1	89.3	95.6
Registry (`reg`)	26	25	96.2	81.1	99.3
Scheduled tasks	24	22	91.7	74.2	97.7
Services (`sc`)	20	18	90.0	69.9	97.2
Startup links (.lnk)	18	17	94.4	74.2	99.0
Copy/Move	30	29	96.7	83.3	99.4
NTFS ADS	16	15	93.8	71.7	98.9
DLL load (`rundll32`)	22	20	90.9	72.2	97.5
WSH (`wscript/cscript`)	15	14	93.3	70.2	98.8
PowerShell (proc swap)	28	26	92.9	77.4	98.0
WMI subscription	12	11	91.7	64.6	98.5
COM reg (CLSID)	14	13	92.9	68.5	98.7
BITS (`bitsadmin`)	10	9	90.0	59.6	98.2
MSHTA	8	8	100.0	67.6	100.0
.NET hosts (`msbuild/installutil`)	17	15	88.2	65.7	96.7

TABLE 4: Extractor grounded semantic equivalence by family. Column headers: n is the number of seed variant pairs; k is the number of pairs with $[\![\hat{c}]\!]_{\mathcal{E}} = [\![s]\!]_{\mathcal{E}}$; *Rate* is $100 \times k/n$ (percent); CI_L and CI_U are the lower and upper bounds of the Wilson 95% confidence interval for *Rate* (percent).

similarity scoring and tends to fire when there is substantial lexical overlap with a rule or with training exemplars, for example long shared substrings or paths. SPECTRA changes procedure and argument structure across binaries and functions while preserving the underlying effect, which reduces lexical overlap and leaves a rule-near similarity detector with far fewer hits. This pattern is consistent with § 5.1: commodity string normalizers undo AMIDES in 78.1% of cases but affect only 4.5% of SPECTRA evasions.

5.6. Semantic Equivalence

This subsection defines the paper's semantic-preservation criterion and measures how often generated variants satisfy it. We evaluate semantic equivalence under SPECTRA's canonical effect vocabulary without per-pair human labels, and use a consensus extractor in place of manual judgment. This formalizes the semantic-preservation objective used throughout the paper: a seed and variant are semantically equivalent when they induce the same primitive multiset and compatible binding chains under the consensus extractor. For each normalized surface command c (§2) we write $[\![c]\!]_{\mathcal{E}}$ for the primitive multiset and binding chains returned by the two extractor instances (§3.1.2). We retain $[\![c]\!]_{\mathcal{E}}$ only when the instances agree on the primitive multiset and their binding chains have high overlap; otherwise the command abstains and is excluded from effect-level comparison. On a stratified subset with orthogonal oracle labels, this consensus filter attains an agreement-conditioned precision (ACP) of 95.2%, so retained items usually have correct primitives and bindings, and the remaining 4.8% reflect residual extractor errors that we treat as measurement noise. At the scale of our evaluation (804 generated commands), full manual semantic labeling is infeasible, and designing a complete ground-truth semantics for hundreds of Windows utilities is an open problem beyond the scope of this work. We therefore explicitly scope our claims to this extractor-grounded abstraction and use it consistently throughout the paper.

For each seed and variant we then test whether $[\![\hat{c}]\!]_{\mathcal{E}} = [\![s]\!]_{\mathcal{E}}$ and report results by LOLBAS-style family. Table 4 summarizes the number of seed-variant pairs n, the number k that satisfy extractor-grounded equivalence, the resulting rate $100 \times k/n$, and Wilson confidence intervals. The overall rate is 93.1% with narrow family-specific intervals, which indicates that the generated variants usually preserve the seed's intended effect under this extractor-defined notion of semantic equivalence.

TABLE 5: Ablation matrix (one change at a time). Columns match the unified eval: Cov/Conf in %, De-obf in %; H_{tot} total alerts; H_{avg} alerts per fully evaded rule; R_{CB} fully-evaded rules (% of 292).

Configuration	Cov (%)	Conf (%)	De-obf (%)	H_{tot}	H_{avg}	R_{CB} (%)
Full SPECTRA (baseline)	72.9	97.2	4.50	31	0.149	71.2
A: Calibration & Validation off (§ 3.1.4)	58.7	88.4	12.9	64	0.421	52.1
B: Near-match lattice off ($\lvert\Delta\rvert \leq 1$)	61.3	95.3	6.38	47	0.267	60.3
C: Calibration off (no temp scaling)	69.4	94.1	7.82	39	0.201	66.4
D: Typing coercions off ($\mathcal{T}$ collapses)	68.1	93.7	8.16	41	0.217	64.7
E: Relaxed $N(\cdot)$ (alias/Unicode less strict)	66.2	91.5	9.47	44	0.240	62.7
F: Backfill off (no ± 14 d consistency)	71.8	96.1	5.21	34	0.169	68.8
G: Arity smoothing/backoff off	67.9	92.6	8.73	42	0.225	64.0
H: Diversity guards off ($\Delta_{\text{bin}}, \Delta_{\text{flags}}, \Delta_{\pi}$)	74.0	97.0	6.95	45	0.243	63.4
I: Wrappers/control ops allowed	75.6	95.2	10.4	53	0.308	58.9
J: Weak avoid-set fidelity ($\mathcal{A}_i$ relax)	72.5	96.0	6.27	58	0.354	56.2
K: Stochastic decode (no determinism)	72.2	96.6	5.03	33	0.161	70.2
L: Negative controls/abstention off	78.9	89.3	11.7	62	0.392	54.1
M: No pruning (K=40, ε=0)	73.1	97.1	4.62	32	0.155	70.9

5.7. Ablations of SPECTRA's components

This subsection asks which components are responsible for the gains in coverage, robustness, and low collateral alerting. We ablate one component at a time and re-run the full evaluation. Bold marks the best value per column (higher is better for Cov, Conf, R_{CB}; lower is better for De-obf, H_{tot}, H_{avg}). Coverage gains without quality controls tend to increase brittleness and collateral alerts, while the full configuration balances all objectives.

The goal is not raw coverage but *strong evasions*: low brittleness, low collateral alerting, and many fully-evaded rules. The full SPECTRA row is Pareto-efficient on these objectives: it attains the **lowest** De-obf (4.50%), **lowest** H_{tot} (31) and H_{avg} (0.149), and the **highest** R_{CB} (71.2%) while keeping coverage high (72.9%) and conformance best-in-class (97.2%).

5.8. Generalization to Newly Published Sigma `process_creation` Rules

We close by asking whether the same frozen pipeline transfers to newer Sigma `process_creation` rules from the same detector family. This experiment is intentionally narrow: it tests whether the same pipeline transfers to newly published Sigma `process_creation` rules without

retuning. We randomly sampled ten `process_creation` rules from the most recent Sigma commit [19], treated them as unseen seeds, and ran the full SPECTRA pipeline with the same thresholds and validators used in the main evaluation. SPECTRA produced evasions for eight of the ten rules. All eight passed the conformance checks from §5.1, none were flagged as deobfuscations under §5.2, all bypassed their target rules without additional collateral alerts under §5.3, seven of the eight passed the operational validity check from §5.4, and none of those seven were detected by the AMIDES evasion detector from §5.5. The two failures follow the same semantics-driven failure modes as in the main evaluation. For one PowerShell rule on suspicious partially encoded invocations (rule id `536e2947-3729-478c-9903-745aaffe60d2`), the consensus extractor abstains because the obfuscated payload does not yield a stable effect graph. For the Windows Defender definition removal rule on `MpCmdRun.exe` with `-RemoveDefinitions -All` (rule id `9719a8aa-401c-41af-8108-ced7ec9cd75c`), the avoid set and documented schema leave no effect-compatible replacement utility. These results support only within-family generalization claims.

6. Related work

Rule evasion detection. AMIDES [66] provides state-of-the-art evasion detection to compare events against SIEM rules and benign traffic to surface rule bypasses. Its learned similarity scoring works best for near-miss variants; SPECTRA's cross-binary, semantics-preserving transformations move far from surface patterns while preserving effects. Our evaluation shows SPECTRA evasions remain difficult to detect while AMIDES' own evasions are frequently caught. Network IDS evasion work targets adversarial traffic [40] rather than process-creation semantics. Practitioner techniques demonstrate signature bypasses [13, 37, 65] via renaming, script edits, or shellcode alterations, but target different inputs (network features, byte patterns) and do not automatically generate effect-equivalent evasions.

Living-off-the-land detectors. LOLBins detectors [44, 62] tokenize commands and train classifiers over lexical features and n-grams. They are detectors, not generators, using lexicon-centric features that neither model cross-utility equivalence nor enforce schema legality. Under our threat model (semantics fixed, implementation variable), name/regex-driven detection fails because effect-preserving transformations swap utilities and reshape arguments beyond lexical decision boundaries. These systems require predefined classes and analyst effort [66], whereas SPECTRA operates fully automatically.

Provenance-Based Detection and Investigation. On the detection side, provenance-based and graph-based systems [41, 51, 52, 58, 60, 70] reason over process trees, ancestry relations, contextual correlations, and provenance or causal event graphs rather than over a single command-line realization. They are therefore related to our setting, but they are not directly comparable baselines for the present benchmark, which is built around event-local SIEM rule matching over fields such as `CommandLine`, `Image`, and `ParentImage`. On the investigation side, provenance-based investigation systems [42, 50, 54, 55, 64, 68] that reconstruct multi-event execution histories are likewise orthogonal, because they support post hoc analysis over richer event structure rather than single-event rule matching. Our results therefore do not speak to the effectiveness of either class of system.

De-obfuscation and canonicalization. Tools like Invoke-DOSfuscation [17], Revoke-Obfuscation [31], CMD-DeObfuscator [6], and PowerDecode [29] collapse superficial encodings, quoting, and aliases back to the same underlying utility/flag pattern. SPECTRA instead changes the procedure (different utilities and typed arguments) while preserving effects. String-level de-obfuscation is thus orthogonal to our evasions: even after normalization, the realized procedure differs, so rule predicates tied to specific binaries/flags no longer fire.

Pyramid of Pain. Bianco's "Pyramid of Pain" [3] places tools/TTPs at the top as the hardest defensive signals, above atomic indicators (hashes, IPs, strings). SPECTRA targets this top tier by rewriting procedures (swapping utilities and arguments) while holding behavior constant. Prior work [2, 16, 18] and AMIDES [66] target lower tiers where renamings and regex tweaks suffice. Our procedure-level, effect-preserving transformations target the hard-to-change tool/TTP tier defenders prioritize, where our evaluation shows SOTA defenses fail most often.

7. Limitations and Future Work

Scope of the Evaluated Setting. These results concern rule-based SIEM detection that matches event-local fields such as `CommandLine`, `Image`, `ParentImage`, and `EventID`. We do not evaluate detectors that score multi-event structures such as process trees, parent-child ancestry chains, or provenance and causal graphs built across multiple events. Our findings should therefore be interpreted as evidence about command-line-centric rule matching, not as a claim about all detector families.

Defensive Strategies. Our findings highlight the limits of string-based rules and point toward defenses that model behavior rather than surface form. Promising directions include behavior-based detection over causal event relationships [52, 60], effect-level signatures that recognize equivalent procedures across utilities, contextual analysis over ancestry, users, and time [70], and hybrid rule-plus-learning systems over semantic representations such as effect graphs [41]. Designing and evaluating such defenses is beyond our scope, but SPECTRA can serve as a testbed for measuring their robustness against procedurally diverse attacks.

Extension to Other Sigma Rule Categories. SPECTRA currently focuses on Sigma `process_creation` events. Extending to Sigma file, registry, and network rules would

require adapter changes rather than changes to the core retrieval and synthesis pipeline. Concretely, the primitive vocabulary, normalization rules, event schema, and reference corpus would need to be expanded to cover operations such as file creation and modification, registry writes and deletes, and network actions such as DNS queries or HTTP posts. The same effect-graph, schema-inference, effect-matching, and constrained-filling machinery would then operate over those new adapters.

Extension to Other Rule Languages and Linux Command Lines. Supporting other rule languages such as Elastic or Splunk, or moving from Windows to Linux command lines, would similarly require adapter work at the renderer and backend boundary, field mapping layer, normalization stage, primitive vocabulary, and reference corpus. For Linux in particular, the normalization layer would need to model shell-specific syntax such as pipes, redirection, and command substitution, while the reference side would need to draw from sources such as `man` pages and GTFOBins. We leave these adapter implementations and their evaluation to future work and do not claim them as validated settings in this paper.

BiLSTM-CRF over Transformers. As explained in Appendix A, span detection uses a BiLSTM-CRF tagger for named entity recognition [53]. Our inputs are short Windows commandlines (a few to a few dozen tokens), so a bidirectional LSTM captures relevant local context while the CRF layer enforces coherent span boundaries. We trained a comparable two-layer transformer encoder on the same annotated corpus and evaluated both on the held-out validation split (Appendix A.3). The transformer did not improve span F1 or downstream agreement conditioned precision (ACP, 95.2% in our setup), but incurred higher training and inference cost. We therefore adopt the simpler and more stable BiLSTM-CRF architecture.

Availability of seed corpus and reference corpus. Both corpora are built from public sources. The seed corpus combines the same 292 Windows `process_creation` Sigma rules used in the public benchmark with representative commands from Mordor, Splunk, and related open attack traces [10, 11, 21, 33]; reproduction requires only those datasets and the pinned Sigma commits from §5. The reference corpus is crawled from LOLBAS, SS64, and Microsoft Windows and PowerShell documentation [20, 25, 35]. For each documented binary and function we collect example command lines and surrounding prose, then normalize and index them into effect graphs, making the corpus publicly reproducible and broad for documented Windows command-line behavior.

Scope: why no wrapper level evasions?. Our design restricts generation to single-utility procedures and does not synthesize shell wrappers or control operators such as `cmd /c`, `powershell -Command`, `&&`, `||`, or pipes. This keeps extractor-grounded equivalence independent of shell-specific parsing and canonicalization while focusing the search space on utility-level changes. Wrapper-based evasions are therefore orthogonal and left to future work.

8. Conclusion

SPECTRA reframes SIEM evasion as automated semantics-preserving transformation rather than manual rule reverse-engineering. By matching procedures through effects instead of surface patterns, the system achieves broader coverage, stronger resilience, and better generalization within the evaluated setting than handcrafted baselines. Our work establishes that documentation-driven synthesis can systematically generate operationally valid evasions at scale.

9. Ethics Statement

Stakeholders. Relevant stakeholders include defenders who maintain or deploy rule-based detections, such as SOC analysts, detection engineers, rule authors, and SIEM vendors; organizations whose security depends on those detections; red teams and auditors conducting authorized security assessments; maintainers of public rule repositories; and the broader security research community.

Potential for misuse. SPECTRA automates a capability that previously required significant manual expertise, and automation meaningfully lowers the barrier for both attackers and defenders. An attacker with access to the tool could generate evasive commands at scale against any Sigma `process_creation` ruleset they possess. Organizations that deploy widely available rule repositories such as public Sigma rules with little or no customization would be most directly exposed; these same organizations stand to gain the most from SPECTRA as a diagnostic tool, since they are least likely to have the resources for manual rule auditing. We weighed this risk against two mitigating factors. First, effective misuse still requires knowledge of which rules a target organization has deployed, what logging configuration is in place, and how normalization is applied; these details are typically site-specific and not publicly visible, creating disproportionate uncertainty for an attacker relative to a defender auditing their own rules. Second, sophisticated adversaries are known to perform procedure-level evasion through manual analysis, and prior work has likewise demonstrated such evasions against security rules [47, 66]. SPECTRA surfaces these weaknesses systematically so that defenders can address them before they are exploited. On balance, we believe the defensive benefit of identifying brittle rules at scale justifies the incremental risk, particularly given the dissemination safeguards described below.

Defensive value. We believe the primary benefit of this work is defensive. SPECTRA helps identify rules that rely heavily on brittle literals rather than capturing the underlying malicious behavior, supports more systematic robustness evaluation than manual testing alone, and highlights directions for improving rule design and moving toward more semantically grounded detection.

Research safeguards. Our evaluation was conducted only with public rules, public documentation, and public attack traces. We deliberately restricted all testing to isolated lab

environments and public rule repositories, and did not attempt evasion against any production SIEM deployment. Operational validity checks were performed only in isolated lab settings using controlled execution.

Dissemination safeguards. In line with prior work on rule-evasion evaluation [66], we present the system as a robustness-evaluation tool for authorized security testing and plan to scope artifact release in a way that preserves scientific value. Rather than broadly releasing the full artifact, we will provide access through a request form (https://dartlab.org/spectra/) and review requests case by case based on the requester's stated use before granting access. We will require requesters to affirm institutional affiliation, describe the intended use case, and confirm that testing will be conducted only against systems they are authorized to evaluate.

10. LLM Usage

Claude (Anthropic), ChatGPT (OpenAI), and GitHub Copilot were used during the preparation of this manuscript for editing, grammar checking, code development, and technical writing assistance. No passages were copied without full author review and revision, and all substantive ideas, system designs, algorithms, experimental methodologies, and conclusions are the product and responsibility of the authors. The core contributions of this work, the SPECTRA architecture, consensus extraction mechanism, effect matching engine, semantic validation framework, and all experimental results, were conceived, designed, implemented, and evaluated entirely by the authors without AI assistance.

11. Acknowledgment

We thank the anonymous reviewers and our shepherd for their valuable feedback. We also thank Briana Nabriat and Matthew Beck for their help with AMIDES' experiments. This material is based upon work supported by the National Science Foundation (NSF) under Award Nos. 2339483 and 2530655, and by the Commonwealth Cyber Initiative (CCI).

References

[1] Main sigma rule repository. https://github.com/SigmaHQ/sigma/tree/12054544bbac415438b2207c08bd92633a51b116.

[2] ArgFuscator. https://argfuscator.net/.

[3] Bianco's Pyramid of Pain. https://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html.

[4] Suspicious download via certutil.exe. https://github.com/SigmaHQ/sigma/blob/master/rules/windows/process_creation/proc_creation_win_certutil_download.yml.

[5] Claude. https://www.anthropic.com/claude.

[6] CMD De-Obfuscator. https://github.com/bobbystacksmash/CMD-DeObfuscator.

[7] Siem correlation rules. https://cymulate.com/cybersecurity-glossary/siem-correlation-rules/.

[8] Deepseek-v3. https://www.deepseek.com/.

[9] Elastic rules. https://github.com/elastic/detection-rules.

[10] Windows evtx samples. https://github.com/sbousseaden/EVTX-ATTACK-SAMPLES/, .

[11] Evtx to mitre att&ck. https://github.com/mdecrevoisier/EVTX-to-MITRE-Attack, .

[12] Gemini models. https://ai.google.dev/gemini-api.

[13] Outsmarting the Watchdog: How can Adversaries evade Sigma Rule Detection during a Kerberos Golden Ticket Attack? https://lolcads.github.io/posts/2025/01/evadingsigma/.

[14] Gpt-4.1. https://platform.openai.com/docs/models#gpt-4-1.

[15] Ibm: What is red teaming? http://ibm.com/think/topics/red-teaming.

[16] Invoke-ArgFuscator. https://github.com/wietze/Invoke-ArgFuscator, .

[17] Invoke-DOSfuscation. https://github.com/danielbohannon/Invoke-DOSfuscation, .

[18] Invoke-Obfuscation. https://github.com/danielbohannon/Invoke-Obfuscation, .

[19] Main sigma rule repository: Most recent commit. https://github.com/SigmaHQ/sigma/tree/3a20687cadd9d3f05cecf2ecc46c22a6d4996b01.

[20] LOLBAS Project: Living off the land binaries, scripts and libraries. https://lolbas-project.github.io/.

[21] Mordor. https://github.com/UraSecTeam/mordor.

[22] Command-line syntax key. https://learn.microsoft.com/windows-server/administration/windows-commands/command-line-syntax-key, .

[23] Powershell about_parsing documentation. https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_Parsing, .

[24] Powershell about_quoting_rules documentation. https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_Quoting_Rules, .

[25] Windows commands. https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands.

[26] Nist: Technical guide to information security testing and assessment. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-115.pdf.

[27] o4-mini. https://platform.openai.com/docs/models#o-series.

[28] Att&ck techniques to security events. https://ossemproject.com/dm/mitre_attack/attack_techniques_to_events.html.

[29] PowerDecode. https://github.com/Malandrone/PowerDecode.

[30] Rapid: What is red teaming? https://www.rapid7.com/fundamentals/what-is-a-red-team/.

[31] Revoke-Obfuscation. https://github.com/danielbohannon/Revoke-Obfuscation.

[32] Sigma conditions. https://sigmahq.io/docs/basics/conditions.html.

[33] Splunk att&ck data. https://github.com/splunk/attack_data, .

[34] Splunk security content. https://github.com/splunk/security_content, .

[35] Ss64 command line reference. https://ss64.com/.

[36] Sysmon v15.15 documentation. https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon.

[37] Bypass YARA Rule f0b627fc for CobaltStrike to Evade EDRs. https://wafflesexploits.github.io/posts/Bypass-YARA-Rule-Windows_Trojan_CobaltStrike_f0b627fc-to-Evade-EDRs/.

[38] String.normalize method. https://learn.microsoft.com/en-us/dotnet/api/system.string.normalize?view=net-10.0.

[39] B. A. Alahmadi, L. Axon, and I. Martinovic. 99% false positives: A qualitative study of SOC analysts' perspectives on security alarms. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2783–2800, 2022.

[40] S. Alahmed, Q. Alasad, M. M. Hammood, J.-S. Yuan, and M. Alawad. Mitigation of black-box attacks on intrusion detection systems-based ml. *Computers*, 11(7):115, 2022.

[41] U. Aslam, E. Batool, S. N. Ahsan, and A. Sultan. Hybrid network intrusion detection system using machine learning classification and rule based learning system. *International Journal of Grid and Distributed Computing*, 10(2):51–62, 2017.

[42] A. Bates and W. U. Hassan. Can data provenance put an end to the data breach? *IEEE Security & Privacy*, 17, 2019.

[43] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.

[44] T. Boros, A. Cotaie, A. Stan, K. Vikramjeet, V. Malik, and J. Davidson. Machine learning and feature engineering for detecting living off the land attacks. In *IoTBDS*, pages 133–140, 2022.

[45] Y. Ding, X. Xia, S. Chen, and Y. Li. A malware detection method based on family behavior graph. *Computers & Security*, 73:73–86, 2018.

[46] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 59–68, 2006.

[47] M. Graeber and L. Christensen. Subverting sysmon: Application of a formalized security product evasion methodology. https://specterops.io/wp-content/uploads/sites/3/2022/06/Subverting_Sysmon.pdf, 2018. Prepared for Black Hat USA 2018.

[48] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pages 1321–1330. PMLR, 2017.

[49] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and {End-to-End} protocol semantics. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[50] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates. NoDoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed System Security (NDSS)*, 2019.

[51] W. U. Hassan, A. Bates, and D. Marino. Tactical provenance analysis for endpoint detection and response systems. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.

[52] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium*, 2017.

[53] Z. Huang, W. Xu, and K. Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[54] M. A. Inam, W. U. Hassan, A. Ahad, A. Bates, R. Tahir, T. Xu, and F. Zaffar. Forensic analysis of configuration-based attacks. In *NDSS*, 2022.

[55] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[56] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun. Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning. *Computers & Security*, 122:102872, 2022.

[57] P. S. Liang, D. Klein, and M. Jordan. Agreement-based learning. *Advances in Neural Information Processing Systems*, 20, 2007.

[58] Q. Liu, M. Shoaib, M. U. Rehman, K. Bao, V. Hagenmeyer, and W. U. Hassan. Accurate and scalable detection and investigation of cyber persistence threats. *arXiv preprint arXiv:2407.18832*, 2024.

[59] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008.

[60] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan. HOLMES: Real-time apt detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] S. Muzammil, R. Reddy, V. Kamalakrishnan, H. Ahmadi, and W. U. Hassan. Towards small language models for security query generation in soc workflows, 2026. URL https://arxiv.org/abs/2512.06660.

[62] T. Ongun, J. W. Stokes, J. B. Or, K. Tian, F. Tajaddodianfar, J. Neil, C. Seifert, A. Oprea, and J. C. Platt. Living-off-the-land command detection using active learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 442–455, 2021.

[63] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *NDSS*, 2020.

[64] M. Shoaib, A. Suh, and W. U. Hassan. Principled and automated approach for investigating ar/vr attacks. In *USENIX Security Symposium*, 2025.

[65] A. V. Turukmane, G. Khekare, N. Shelke, G. Sakarkar, and S. Buchade. Evasion techniques in cybersecurity: An in-depth analysis. In *2024 International Conference on Artificial Intelligence and Quantum Computation-Based Sensor Application (ICAIQSA)*, pages 1–9. IEEE, 2024.

[66] R. Uetz, M. Herzog, L. Hackländer, S. Schwarz, and M. Henze. You cannot escape me: Detecting evasions of SIEM rules in enterprise networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5179–5196, 2024.

[67] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[68] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter. Fear and logging in the internet of things. In *Network and Distributed System Security (NDSS)*, 2018.

[69] E. B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158): 209–212, 1927.

[70] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

[71] R. Zhao, M. Shoaib, V. T. Hoang, and W. U. Hassan. Rethinking tamper-evident logging: A high-performance, co-designed auditing system. In *ACM Conference on Computer and Communications Security (CCS)*, 2025.

[72] Z.-H. Zhou and M. Li. Tri-training: Exploiting unlabeled data using three classifiers. *IEEE Transactions on knowledge and Data Engineering*, 17(11):1529–1541, 2005.

Appendix A. Extractor Architecture and Calibration

A.1. SeqTagger: BiLSTM-CRF

Span detection uses a standard BiLSTM-CRF sequence tagger for named entity recognition [53] over the label set $\mathcal{Y} = \{\texttt{O}, \texttt{B-FLAG}, \texttt{I-FLAG}, \texttt{B-VALUE}, \texttt{I-VALUE}\}$. A bidirectional LSTM encodes the token sequence, and a CRF layer enforces globally coherent label assignments. Token representations combine a character-level CNN with token embeddings and simple indicator features. In preliminary comparisons with a two layer transformer encoder of similar capacity, we did not see consistent gains in post validation acceptance, so we retain the BiLSTM-CRF tagger for all experiments.

A.2. Reference crawler and cache

We build the reference corpus $\mathcal{R}$ by crawling public indexes, as described in §3, and caching per-binary pages and function anchors in a content-addressed store with HTTP freshness metadata. Subsequent runs use cached bytes, and entries are invalidated when the page digest changes or on

explicit refresh, so repeated experiments see a stable corpus. From each cached page we derive two textual views for each utility function $u = (\text{bin}, \text{fn})$: the Example View E_u, which contains preformatted command invocations, and the Anchor View A_u, which contains the heading and surrounding prose. We normalize examples with N from §2, preserving placeholders such as `<PATH>` and `<URL>` as typed `VALUE`s so that differences between seeds and references reflect procedural alternatives rather than tokenization artifacts. For each function we compute statistics such as the number of distinct examples and the fraction that parse via the extractor; these quality annotations q_u are exposed to pairing and ranking and are used only as tie breakers. After schema construction and effect-graph building, each documented function is represented by the per-function artifact tuple $\langle u, \mathcal{S}_u, G_u, B_u, q_u \rangle$ shown in Figure 3. Here u identifies the binary and function, $\mathcal{S}_u$ is the final argument schema, G_u is the reference effect graph, B_u is the binding summary, and q_u is the crawler-derived quality annotation. This tuple is the unit stored in the reference index and later consumed by lookup, pairing, ranking, and filling.

A.3. Training corpus and metrics

We annotate Windows command lines with BIO spans for `FLAG` and `VALUE`. The label guide treats flags as switch initial tokens and values as argument spans, preserving quoted multi token values; ambiguous cases such as fused switches are resolved by annotator consensus. We report span level F1 and Cohen's κ for agreement. Splits are disjoint by binary family across training, validation, and test, and seed versus reference examples live in separate folds to reduce leakage. The tagger is trained on the training split, checkpoints are selected by validation F1, and the test split is held fixed. Hyperparameters and random seeds are frozen. The extractor is trained on this annotated corpus, which induces a canonical vocabulary of 1,638 primitive effects, with Krippendorff's $\alpha = 0.86$ for `FLAG`/`VALUE`. Posteriors are temperature calibrated (ECE below 3%) via temperature scaling on the validation subset.

A.4. Seed helpers: primary binary and avoid sets

Given the token alphabet Σ and normalization map $N : \Sigma^* \to \Sigma^*$, we define two deterministic helpers per normalized command line. The primary binary map $b : \Sigma^* \to \Sigma^*$ returns the first non-wrapper token corresponding to an executable image, with common suffixes stripped; if none exists, the seed is treated as malformed. The avoid set extractor $A(\cdot)$ parses the Sigma YAML detection block y_i into a structured set $\mathcal{A}_i = \{\langle t, op, cs \rangle\}$ recording rule literals, operator semantics, and case policy. $\mathcal{A}_i$ is matched over the normalized token stream $N(c_i)$ and is later used by the validator.

A.5. Orthogonal oracle implementation

To bound correlated modeling errors, we evaluate a stratified subset $\mathcal{S}^\star$ with an orthogonal parser ORACLE that does not reuse the extractor's learned signals. For `powershell.exe` we parse the PowerShell AST and check side conditions on process launches, scriptblock literals, and encoded commands. For scheduled tasks created with `schtasks /create` we inspect the XML or the materialized task arguments and verify the presence and typing of the task name, schedule, and action. For `reg.exe` we check hive, key, and value grammar together with write semantics. The oracle emits predicate constraints over primitives, and we retain only commands whose consensus assignment satisfies all applicable constraints for $c \in \mathcal{S}^\star$. From this subset we compute agreement conditioned precision (ACP) as the fraction of commands on which the extractors agree on a non null primitive multiset and the oracle also accepts that assignment, and we report Wilson 95% confidence intervals as summarized in §5.6.

A.6. Pairing, diversity, and selection details

Let $G_i = (U_i, V_i, E_i)$ be the seed effect graph and B_u the binding summary for function u. We build a bipartite graph whose left nodes are seed edges $e_i \in E_i$ and whose right nodes are candidate reference edges summarized by B_u. A pair (e_i, e_u) is eligible only when the primitive labels match and the argument endpoints are schema compatible. Eligible pairs receive a learned compatibility score $p(e_i, e_u) \in [0, 1]$ based on local token features, value types, and primitive identity, and pairs below a fixed threshold are pruned. A maximum weight matching $M^\star$ then yields the effect graph alignment score $\alpha(G_i, B_u) = |M^\star|/|E_i|$. Given the seed chains $\mathcal{B}_i$, we define binding coverage $\beta(G_i, B_u)$ as the fraction of chains that admit a type compatible realization in B_u under $M^\star$. For candidates that survive these filters, argument filling is then carried out by the lightweight generative component under the chosen schema and avoid set rather than by manual construction. Candidates with low α or β are discarded. For diversity, each candidate u has flag novelty $\Delta_{\text{flags}}(u) = 1 - |F_u \cap F_{\text{seed}}|/|F_u \cup F_{\text{seed}}|$, positional novelty $\Delta_\pi(u) = \text{ed}(\pi_u, \pi_{\text{seed}})/\max\{|\pi_u|, |\pi_{\text{seed}}|\}$, and a cross binary indicator $\Delta_{\text{bin}}(u) = \mathbb{1}[\text{bin}(u) \neq \text{bin}_{\text{seed}}]$. The admissible set $\mathcal{U}_{\text{adm}}$ keeps only candidates that satisfy the fixed diversity thresholds (τ_f, τ_π), and, when any cross binary option exists, also requires $\Delta_{\text{bin}}(u) = 1$. We rank the surviving candidates with:

$$J(u) = (\alpha(G_i, B_u), \beta(G_i, B_u), -\delta_{\text{key}}(u), -\delta_{\text{pos}}(u), q_u, \Delta_{\text{flags}}(u), \Delta_\pi(u)),$$

where $\delta_{\text{key}}(u)$ and $\delta_{\text{pos}}(u)$ record retrieval-tier bookkeeping. In the exact composite-lookup configuration used in this paper, both are zero for all surviving candidates. We then select $u^\star$ as the lexicographic maximizer over $u \in \mathcal{U}_{\text{adm}}$, which prioritizes effect alignment, binding coverage, retrieval tier, utility quality, and diversity in that order.

Once $u^\star$ is selected, the filler ranges over $\mathcal{L}(\mathcal{S}_{u^\star})$, the set of argument strings licensed by the selected schema. Required flags, optional flags, admissible arities, and the

TABLE 6: LLM evaluation on 292 Sigma process creation rules. Columns match Table 2.

	Generation & conformance				De-obfuscation			Alerting across ruleset				
System	**Gen.**	**Cov. (%)**	**Valid**	**Conf. (%)**	**Total**	**De-obf.**	**De-obf. (%)**	$\mathbf{H_{tot}}$	$\mathbf{R_{TB}}$ **(%)**	$\mathbf{R_{CB}}$	$\mathbf{R_{CB}}$ **(%)**	$\mathbf{H_{avg}}$
DeepSeek-V3-0324	177	60.6	23	13.0	177	59	33.3	336	55.4	98	33.6	3.4286
Claude-3.7-sonnet	292	100.0	212	72.6	292	61	20.9	1514	34.2	100	34.2	15.1400
Gemini-flash-2.0	292	100.0	139	47.6	292	88	30.1	1582	35.3	103	35.3	15.3592
GPT-4.1	292	100.0	198	67.8	292	59	20.2	951	31.8	93	31.8	10.2258
o4-mini	267	91.4	202	75.7	267	74	27.7	1086	52.1	139	47.6	7.8129
SPECTRA	213	72.9	207	97.2	804	36	4.5	31	100.0	208	71.2	0.1490

TABLE 7: Token usage for LLM evasion generation over 292 rules. Columns: **In** = input tokens; **Out** = output tokens; **Out/In** = output to input ratio; **Out share** = Out divided by Total. Totals include provider reported reasoning tokens when available.

Model	**In**	**Out**	**Out/In**	**Out share**
o4-mini	204,193	696,008	3.408579	0.440509
Gemini-flash-2.0	215,905	35,984	0.166666	0.142857
Claude-3.7-sonnet	233,444	16,461	0.070514	0.065869
DeepSeek-V3-0324	204,731	11,438	0.055868	0.052912
GPT-4.1	204,485	7,317	0.035783	0.034546

positional template determine membership in this language. The seed binding chains $\mathcal{B}_i$ and avoid set $\mathcal{A}_i$ then constrain how typed placeholders are instantiated, while the candidate binding summary $B_{u^\star}$ and canonicalized examples in $E_{u^\star}$ bias the choice among schema-legal realizations. These biases affect only which realization is chosen, not the target effect set or the validator checks applied afterward.

A.7. Schema consistency details

Given a utility function $u = (\text{bin}, \text{fn})$ with Example-View E_u and an initial Anchor-only schema $\widehat{\mathcal{S}}_u = \langle\{(f, \widehat{a}_f, T_f)\}, \pi, \text{Req}, \text{Opt}, B_u\rangle$, we validate and edit $\widehat{\mathcal{S}}_u$ using only statistics from E_u and cross-source agreement, without hand curation. For each flag f, we compute the support s_f with a Wilson 95% confidence interval $\text{CI}(s_f)$ [69], an arity histogram $\hat{p}(a \mid f)$ over $a \in \{0, 1, N\}$ with Laplace smoothing α_0=1, and the dominant typed positional template $\tilde{\pi}$. When mirrored sources exist, we require agreement at least τ_{agree} and use a support threshold τ_{supp} to decide whether a flag is sufficiently represented.

We then apply deterministic edits. A flag in Req whose upper confidence bound $\text{CI}_{\text{upper}}(s_f)$ falls below τ_{supp} and whose agreement is below τ_{agree} is demoted, while a flag whose lower bound $\text{CI}_{\text{lower}}(s_f)$ exceeds τ_{supp} and whose agreement meets τ_{agree} is promoted to Opt.

If the Anchor-derived arity $\widehat{a}_f$ is inconsistent with the empirical distribution $\hat{p}(a \mid f)$, we reset $\widehat{a}_f$ to $\arg\max_a \hat{p}(a \mid f)$. For each flag f we add the dominant observed value type from E_u to T_f and prune types with zero or negligible support, with a small-count tolerance for rare but plausible types. Finally, we require π to unify with the dominant typed positional template $\tilde{\pi}$; if unification fails, we reject the schema for u.

If none of these checks triggers rejection, we set $\text{consistency}(u)$=1 and take the edited schema as $\mathcal{S}_u$; otherwise, we set $\text{consistency}(u)$=0 and exclude u from schema-based generation and validation. Thus the consistency decision is a deterministic function of E_u and cross-source agreement, separate from the Anchor-only predictions produced by $g_\phi(A_u)$.

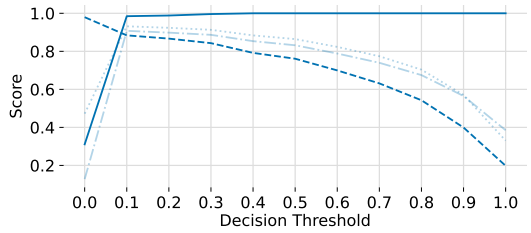
Appendix B. Additional Experiments

B.1. LLMs Performance Comparison

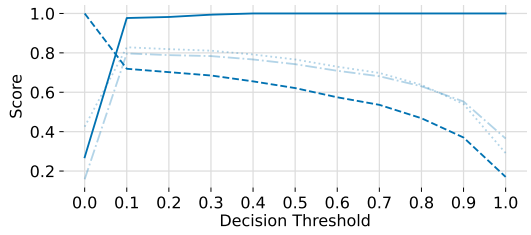
We benchmark five widely-used general-purpose LLMs (DeepSeek-V3-0324 [8], Claude-3.7-sonnet [5], Gemini-flash-2.0 [12], GPT-4.1 [14], and o4-mini [27]) using their latest public versions (as of August 2025) with default parameters and identical zero-shot prompts. These models represent state-of-the-art instruction-following capabilities and are commonly applied to security tasks, but are not specialized for evasion generation. We include them to contextualize SPECTRA's performance against widely available tools, not to claim optimality over specialized models; recent work on small language models for SOC query generation suggests that task-specific training can materially change performance in security workflows [61]. Our goal is to demonstrate that purpose-built semantic reasoning outperforms both expert manual crafting and off-the-shelf general-purpose generation.

LLM cost analysis. Table 7 shows that o4-mini dominates token usage, with 696k output tokens, an Out/In ratio of 3.41, and 44% of its total tokens spent on output. The remaining models generate much shorter responses: their Out/In ratios lie between 0.036 and 0.17 and their output shares between about 3.5% and 14%. In a token priced setting, this pattern means that o4-mini is the main cost driver among the LLM baselines, while the other models are comparatively cheaper per rule under the same prompting protocol. Notably, this higher token budget does not close the performance gap to SPECTRA, which remains stronger on conformance, alerting, and fully evaded rules.

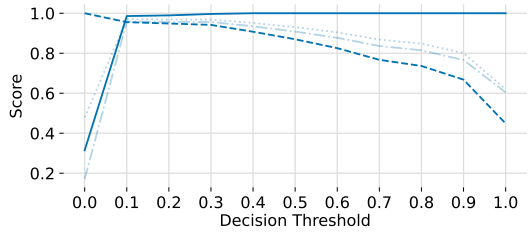
Coverage. As shown in Table 6, raw coverage is high for several LLMs. Claude-3.7-sonnet, Gemini-flash-2.0, and GPT-4.1 return a string for every rule, and o4-mini reaches 91.4%. This behavior is expected from instruction-following models that attempt an answer for nearly any input. §5.1 showed that raw coverage is not a proxy for useful evasions: SPECTRA achieved lower raw coverage than LLMs, yet later metrics favored SPECTRA decisively.



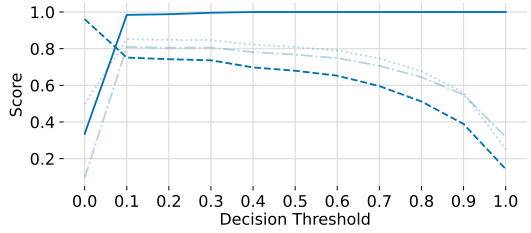
(a) Claude-3.7-sonnet



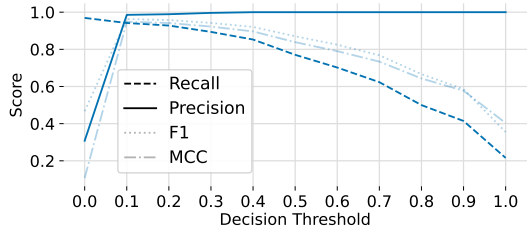
(b) DeepSeek-V3-0324



(c) Gemini-flash-2.0



(d) GPT-4.1



(e) o4-mini

Figure 7: Outcomes of the evasion detection component of AMIDES (state-of-the-art adaptive evasion detection) against each LLM's evasions.

Conformance. Once we apply the minimal Windows command-line conformance filter from §5.1, the picture changes. The strongest LLMs retain between roughly half and three quarters of their generations as valid commands, with o4-mini and Claude-3.7-sonnet at the high end and Gemini-flash-2.0 and GPT-4.1 lower, and DeepSeek-V3-0324 dropping to 13.0%. On the same test, SPECTRA keeps 97.2% of its outputs. Much of the LLM "coverage" is therefore carried by malformed or off-platform strings that would not function as concrete evasions, whereas SPECTRA stays close to ceiling on syntactic and platform conformance.

De-obfuscation with off-the-shelf tools. Under the §5.2 protocol with CMD-DeObfuscator and PowerDecode, LLM outputs are de-obfuscated in 20-33% of cases. This is consistent with the §5.2 finding that surface tricks and shallow encodings are quickly undone by public tools. In the same setting, SPECTRA is de-obfuscated only 4.5% of the time despite generating many more variants per rule. LLM evasions are therefore markedly more brittle than SPECTRA.

Alerting against the ruleset. All LLMs generate substantial alert volume: Gemini-flash-2.0 and Claude-3.7-sonnet produce more than 1500 alerts in total, GPT-4.1 and o4-mini roughly 950-1100, and even DeepSeek-V3-0324 produces several hundred. Their H_{avg} values range from 3.4 to 15.4 alerts per fully evaded rule. In contrast, SPECTRA yields 31 total alerts, 208 fully evaded rules, and $H_{\text{avg}} = 0.149$. Even the best LLM on this metric, o4-mini, raises over an order of magnitude more alerts per fully evaded rule than SPECTRA, and none of the LLMs approach SPECTRA's 71.2% fully-evaded-rule rate. This gap indicates that LLM-generated evasions frequently continue to satisfy rule predicates or trigger collateral rules, while SPECTRA achieves lower alert volume and higher true cross-benchmark evasion.

Differences across LLM baselines. o4-mini is the strongest LLM baseline on downstream metrics, with the highest conformance (75.7%), the most fully evaded rules (139), and the lowest H_{avg} (7.81), but it still remains well behind SPECTRA on both alerting and true evasion. Claude-3.7-sonnet and Gemini-flash-2.0 saturate raw coverage yet generate the heaviest alert volume, while DeepSeek-V3-0324's 60.6% raw coverage collapses to 13.0% after conformance filtering, indicating that instruction following helps completion more than true behavior-preserving evasion.

Evasion detection outcomes. Figure 7 summarizes detection by the AMIDES evasion detector. LLM graphs show higher recall when precision becomes 1.0 than SPECTRA, meaning the detector recovers a larger fraction of LLM outputs. SPECTRA achieved a miss rate higher than all LLM baselines. Together with the de-obfuscation and alerting results, the detector outcomes reinforce the same pattern observed in §5: high LLM raw coverage does not translate into strong evasions.

Across the LLM-only tables, three properties recur. First, raw coverage is high because models tend to answer every prompt. Second, a large share of those answers fail minimal Windows conformance, which removes a significant fraction before any substantive evaluation. Third, among the conforming subset, many instances are either undone by commodity de-obfuscation tools or still raise alerts across the ruleset, and they are more likely to be caught by the SOTA evasion detector than SPECTRA. Earlier experiments showed that SPECTRA meets the conformance at near-perfect rates, resists de-obfuscation at 4.5%, minimizes alerting, and is hardest for the SOTA detector to recover. The LLM results provide context: general-purpose LLMs can generate evasion command-lines, but these do not yield strong evasions under the same tests as SPECTRA.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper presents SPECTRA, an automated framework for generating semantics-preserving evasions for Security Information and Event Management (SIEM) detection rules. Rather than relying on syntactic obfuscation techniques such as encoding or quoting tricks, SPECTRA models the intended effects of commands and synthesizes alternative command realizations that preserve those effects while avoiding specified detection patterns.

The system is evaluated on the Sigma process_creation rule set and compared against AMIDES as well as several general-purpose LLM baselines. Across this evaluation, SPECTRA achieves substantially higher rule coverage and substantially lower de-obfuscation rates than AMIDES while also outperforming the LLM baselines. The committee found the core idea compelling and the empirical evaluation convincing within the scope considered, and therefore recommended acceptance despite the concerns outlined below.

C.2. Scientific Contributions

- Provides a New Data Set For Public Use.
- Creates a New Tool to Enable Future Science.
- Addresses a Long-Known Issue.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

C.3. Reasons for Acceptance

1) The paper addresses an important problem: evaluating whether SIEM detection rules capture true attack intent or merely surface-level command patterns.
2) The central idea—effect-preserving command re-realization rather than string-level obfuscation—was viewed as novel relative to prior work such as AMIDES.
3) The evaluation is strong within scope, covering 292 Sigma process_creation rules and comparisons against both AMIDES and several LLM baselines.
4) The system automates a process that prior work handled manually, and several reviewers noted that components of the framework may prove useful beyond the specific SIEM evasion use case.