Universal Theory of Exploitation:

Synthesizing Weird Machine Programs from Guarded Behavior Models

Adam Doupé Zachary Smith

Arizona State University

Technical Report August 2026

Abstract

Exploitation frequently requires a sequence of attacker-triggerable behaviors whose effects establish the guards of later behaviors. Existing exploit-generation systems can derive such behaviors from concrete program semantics, but carrying the resulting execution detail into chain composition makes the composition query account for state that no modeled guard or goal observes. We present the Universal Theory of Exploitation (UTE), a system that instead projects supplied behavior models onto guarded transitions over sparse, model-selected state. UTE partially lowers supported Weird Machine Instructions (WMIS), also accepts explicit native Python action models, and encodes one bounded composition problem in Z3. A satisfying model selects an internal plan; the JSON path serializes that plan as a Weird Machine Program (WMP) artifact, whereas the native Python path exposes selected invocations as `FlatAction` records. A common state and operation contract allows admitted clauses from distinct exploitation theories to participate in the same query and permits an earlier effect to satisfy a later guard. We use *universal* to denote this common compositional interface, not completeness over vulnerabilities, targets, or theories. This report defines the model and its interfaces, then supports the correspondence with an auditable trace and focused semantic checks rather than an empirical evaluation.

1 Introduction

Many exploits are programs rather than isolated corruptions. A leak can disclose the address needed by a later write; an allocation can arrange the object consumed by a later use; a file operation can supply the value exposed by a subsequent read. Each setup behavior changes the conditions under which the next behavior is useful. The weird-machine view captures this structure by treating attacker-accessible target behavior as an instruction vocabulary and the exploit as a program over that vocabulary [2, 7].

Discovering and validating those behaviors requires concrete execution semantics. Source- and binary-level exploit-generation systems use program semantics to find paths, reason about machine state, and construct inputs [1, 3]. Those semantics become a liability, however, if the later composition query retains every discovered path and every concrete state distinction. The planner must then distinguish facts even when no supplied guard, effect, or goal observes them. This mismatch makes composition pay for discovery detail that does not decide composition.

Prior systems reduce that burden by choosing abstractions suited to their exploit domain. Code-reuse systems compose gadgets, data-oriented systems connect semantic data flows, and heap or kernel systems reason about abstract operations and vulnerability capabilities [4, 8, 11, 12, 17]. These precedents establish that abstraction is necessary, but their composition units remain tied to particular domains. The open representation problem is to give supported behaviors from several exploitation theories a common state and operation contract.

UTE starts from that representation boundary. It collects state symbols from the supplied model, successfully lowered actions, the goal, and the active theories, and it reasons only over the resulting instance. The construction is sparse by design: concrete memory or target state does not enter merely because it existed during behavior discovery. This projection reduces the modeled state by construction, but it neither proves that the projection is minimal nor establishes a performance advantage.

UTE accepts WMI JSON through partial lowering or explicit native Python action models through direct instance construction, then encodes one bounded composition query in Z3 [5]. A satisfying model selects an internal plan. The JSON path can serialize that plan as a WMP interchange artifact, whereas the native Python path exposes selected invocations as user-visible `FlatAction` records. Figure 1 places these distinct inputs and outputs between the external tasks that produce behavior models and materialize or replay the selected plan.

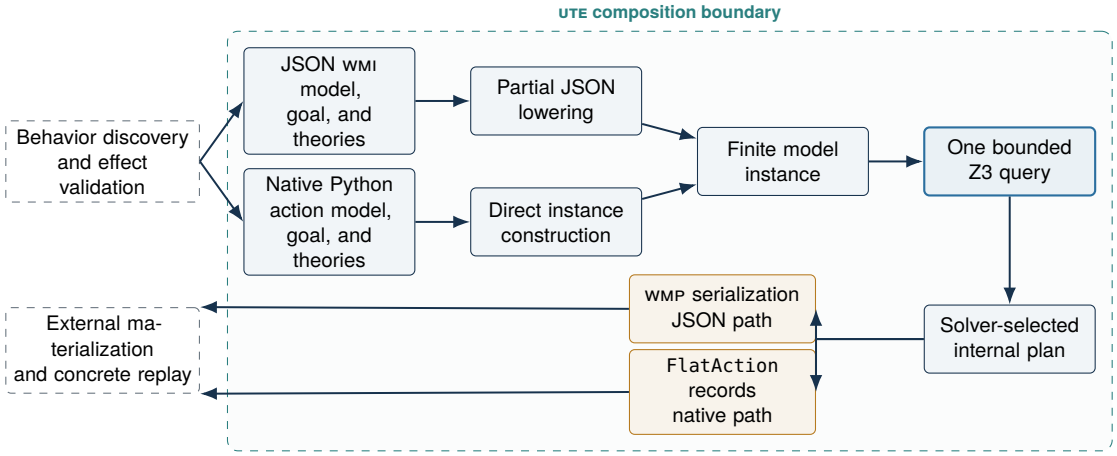

Figure 1: UTE separates composition from behavior discovery and concrete validation. Partial JSON lowering and native Python construction are distinct paths into a finite instance; WMP serialization and `FlatAction` exposure are corresponding projections of the selected internal plan, not target-execution evidence.

This report makes four contributions. First, it defines guarded composition over model-selected state and ordered effects. Second, it describes a JSON WMI interchange ontology whose declarations and constraints use SMT-LIB syntax. Third, it presents the bounded Z3 encoding and its plan projections. Fourth, it identifies Python and C integration paths that leave concrete execution with the consumer. We call the contract universal because supported operations share this interface; we do not claim coverage of every theory or effect. The evidence is an auditable worked trace and focused correspondence checks, not measurements of scalability, coverage, or exploit success.

2 The Chain-Synthesis Problem

UTE begins after a behavior producer has identified a concrete behavior, chosen the state that represents it, described its guards and effects, and supplied trigger metadata. The composition problem therefore asks whether the supplied models admit an ordered chain; it does not ask where those models came from. Behavior discovery, independent validation of modeled effects, and replay of the resulting trigger sequence remain outside the composition boundary. This division assigns fidelity to the producer and composition to UTE.

We use the report-local `prepare/commit` fixture to make that problem concrete. It is a minimal abstraction of the setup-and-use pattern from section 1, not a model of a particular target. The fixture retains only `phase`, which records progress through the supplied behaviors, and `target_word`, which carries the state dependency between them. Each WMI contains one JSON operation with two write effects; lowering represents those effects as two ordered internal operations behind one incoming guard. Each WMI also includes a neutral trigger description.

The fixture starts at the action boundary $\bar{S}_0 = (phase = 0, target\_word = 0)$. Its goal asks for a boundary state satisfying $phase = 2 \wedge target\_word = \texttt{0x42424242}$. Neither variable attempts to reproduce the target's complete execution state. They are precisely the two modeled facts needed to decide whether the supplied setup can enable the supplied use.

At $\bar{S}_0$, the incoming guard of `prepare` requires $phase = 0$ and $target\_word = 0$, so the action is available. Its first operation writes `0x41414141` to `target_word`; its second writes 1 to `phase`. Executing both ordered operations establishes the bridge state $\bar{S}_1 = (phase = 1, target\_word = \texttt{0x41414141})$.

The incoming guard of `commit` observes the same interned locations. It requires $phase = 1$, $target\_word = \texttt{0x41414141}$, and $target\_word \neq \texttt{0x42424242}$. The action is therefore unavailable at $\bar{S}_0$ but available at $\bar{S}_1$. Its ordered writes replace `target_word` with `0x42424242` and publish $phase = 2$, which establishes the goal at $\bar{S}_2$. Figure 2 shows the complete handoff.

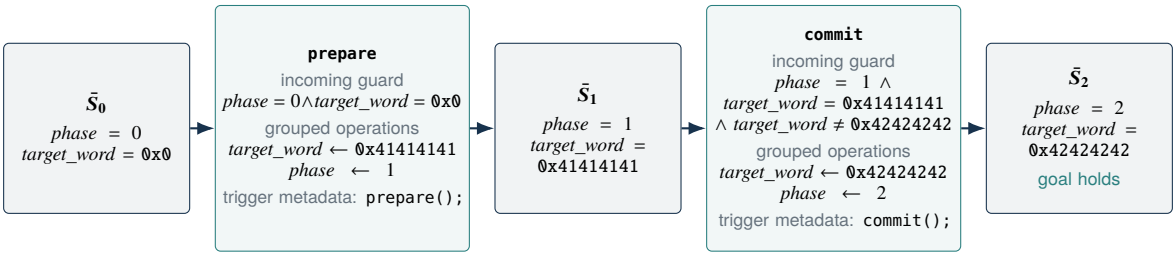

Figure 2: The continuous running trace. Guards are evaluated on the incoming action-boundary state; operations execute in the displayed order. Trigger strings are metadata and were not executed by this trace.

We reproduced the trace against the recorded source snapshot rather than inferring it from the fixture text alone. As summarized in table 1, the parser retained both raw `prepare` conjuncts and all three raw `commit` conjuncts, strict conversion skipped none, and subgoal orchestration reported no deduced subgoals. The direct query was unsatisfiable at one action and satisfiable at two, where the model selected exactly `prepare` followed by `commit`.

Table 1: Auditable evidence for the running trace. Counts use the order raw JSON, parsed syntax tree, and lowered conjuncts.

Check	Observation	Claim supported
Fixture identity	`prepare.json`: `233ff51f636fe414c0ebc6b8227867f8 5dd1dbbe9981fa95b5c433aa7ad24793` `commit.json`: `cf58694e374d489af0a1e40535bd2d6d f282d27807e923d0f184efa4adac45cb`	The checked bytes are the report-local fixtures.
Guard accounting	`prepare`: 2/2/2; `commit`: 3/3/3	The printed conjuncts survive parsing and lowering.
Strict conversion	Accepted; zero skipped guards	No guard in this fixture is silently discarded.
Subgoals	No subgoals deduced	The reported bounds are direct-query results.
Bound $k = 1$	UNSAT; no WMP written	No one-action plan exists in this constructed instance.
Bound $k = 2$	SAT; minimum reported bound 2	A two-action plan exists in this constructed instance.
Selected order	`prepare`, then `commit`	The model contains the state-dependent handoff in figure 2.
Scope limit	No target replay, workload sample, or timing measurement	The observations do not establish trigger validity, coverage, or scalability.

The trace exposes six representation requirements: shared state identity, incoming guards, grouped ordered operations, an explicit goal, selected order, and a trigger handoff. Separate checks in section 3 establish operation ordering, argument reuse, and cross-domain composition. Grouping effects in one WMI does not prove that a concrete trigger performs them atomically.

3 An Operational Model of Composition

3.1 Projected State and Lowered Actions

For a constructed instance I, let Σ_I contain exactly the state symbols instantiated by the active models and theories, and let $\mathcal{S}_I = \mathrm{Val}(\Sigma_I)$ be their valuation space. These symbols can include tracked memory values—including generated stack locations when an initial stack is supplied—and, when enabled, known-constant status, permissions, registers, allocator state, file contents, and descriptor state. A solver state $S_t \in \mathcal{S}_I$ assigns these components at one operation time. Thus, Σ_I is a sparse, model-selected state signature, not a symbolic copy of every byte in the concrete process.

We represent a supplied WMI as $w = (n, \Delta, P, E, \tau)$. The name n identifies the behavior; Δ contains its declarations; P is its precondition list; E is its ordered effect list; and τ is trigger metadata. A producer asserts that this behavior is

attacker-triggerable and chooses the abstraction recorded by P and E. The tuple therefore describes a supplied transition, while τ neither executes that transition nor proves that a target accepts it.

Lowering converts this external tuple into the actions admitted by a particular construction context. We model it as the partial, set-valued function

$$L_\Gamma : \mathcal{W} \rightharpoonup \mathcal{P}_{\text{fin}}(\mathcal{A}_0).$$

The function is partial because a supplied construct can fail conversion, and it is set-valued because a WMI can be filtered, normally produce one uninstantiated action, or produce alternatives. A singleton result is common, but building singleton behavior into the model would hide both unsupported inputs and alternative lowerings.

Formal core.

$$w = (n, \Delta, P, E, \tau) \in \mathcal{W}, \qquad w \in \operatorname{dom}(L_\Gamma) \Longrightarrow L_\Gamma(w) \in \mathcal{P}_{\text{fin}}(\mathcal{A}_0),$$
$$a = \big(n_a, g_a, \langle o_1, \ldots, o_m \rangle, c_a\big) \in \mathcal{A}_0, \qquad B_a^I\big(\beta_i, \eta_{i,1}, \ldots, \eta_{i,\ell}\big).$$

Here $\mathcal{W}$ is the set of supplied WMIs and Γ is the lowering context. The notation $\mathcal{P}_{\text{fin}}(\mathcal{A}_0)$ denotes finite sets of uninstantiated internal actions. An action a has name n_a, incoming guard g_a, m ordered operations o_j, and call cap c_a. For instance I, ℓ is the padded operation width, β_i is the vector of action-level choices at action slot i, and $\eta_{i,j}$ is the vector of choices for its jth operation slot. The finite relation B_a^I holds exactly for combinations admitted by the instance-specific argument domains and binding map.

The internal vectors β_i and $\eta_{i,j}$ are solver choices for a constructed instance. By contrast, JSON `(ARG n)` introduces a trigger-level variable α_n^w in the supplied WMI. Lowering may use that variable to construct or approximate an internal action, and optional trigger synthesis may later assign it a consumer-facing value. There is no general identity $\alpha_n^w = \beta_{i,n}$, nor a general one-to-one mapping from JSON arguments to operation arguments $\eta_{i,j}$.

3.2 Operation Time and Guarded Dispatch

The distinction between action boundaries and operation time preserves ordered effects. For action bound k, let ℓ be the maximum operation width after shorter actions are padded with NOPs. For action slot $i \in \{1, \ldots, k\}$ and operation slot $j \in \{1, \ldots, \ell\}$, operation time is $t(i, j) = (i - 1)\ell + j$. State S_t follows operation slot t, whereas $\bar{S}_i = S_{i\ell}$ is only an alias for the state after action slot i. The incoming guard for slot i observes $\bar{S}_{i-1}$, while an operation can observe the state left by an earlier operation in the same action.

The generic false-branch frame covers only state components that receive cached inertia in the merged construction. Let $\Sigma_I^{\text{frame}} \subseteq \Sigma_I$ contain its memory, known-constant status, register, and permission components. The action-plan manager in `Solver._build_merged_apm_mm_cm` handles the operation label separately, while that construction supplies the frame for these state components. Independently registered file, descriptor, and allocator theory state lies outside Σ_I^{frame} and remains governed by theory-specific clauses.

For a false operation-local condition, the main dispatcher selects the NOP operation label and applies cached inertia without discarding unconditional requirements. For a source operation o, operation-slot label O_t, and arguments η_t, its central shape is

$$\begin{aligned} D_o^I(t, O_t, \eta_t) = \; & \operatorname{Req}_o^I(S_{t-1}, \eta_t) \\ & \wedge \operatorname{ite}\Big(p_o(S_{t-1}, \eta_t), \big(O_t = \operatorname{label}(o) \wedge T_o^I(S_{t-1}, \eta_t, S_t)\big), \big(O_t = \text{NOP} \wedge \operatorname{Stutter}_{\Sigma_I^{\text{frame}}}(S_{t-1}, S_t)\big)\Big) \\ & \wedge \operatorname{Aux}_o^I . \end{aligned}$$

Here p_o is the local condition, T_o^I combines the applicable state updates, and the false branch both selects the NOP label and equates the components in Σ_I^{frame}. Requirements such as dereference validity remain outside the conditional branch, and auxiliary clauses can restrict either outcome.

Focused checks in table 2 establish the operational consequences of this dispatcher rather than assuming them from the notation. Forward intra-action order is satisfiable and the reversed order is not; a general false-condition write selects NOP and lets the next operation execute; and a false conditional dereference still requires a nonnull pointer. A false-conditioned `FileCreate` also selects NOP, but file existence changes because that theory state lies outside the cached frame. Separately, the implementation treats a direct, non-dereferenced, non-partial `Write` with exactly one destination, one value other than `ANY`, and one offset as fixed. Its auxiliary effect hint does not follow the local condition, so a specialized false-condition fixed write becomes inconsistent instead of stuttering. The displayed clause therefore scopes the generic stutter to cached framed state and remains subject to both explicit exclusions.

Table 2: Focused operational microchecks. Each row uses an independent model instance and records a semantic witness, not an exhaustive test.

Concern	Minimal fixture	Observation	Consequence
Ordered effects	Write *phase* ← 1, then conditionally write *result* ← 2 when *phase* = 1; reverse the two operations for the control.	Forward SAT; reverse UNSAT.	Later operations observe earlier operation-time state.
General false condition	A false conditional write can select `target` or `decoy`; an unconditional second write sets `marker`.	Timeline `Nop`, `Write`; both conditioned locations remain zero.	The false branch selects NOP, stutters over the checked memory state, and the grouped action continues.
Theory-state frame	A false-conditioned `FileCreate` is paired with a goal requiring the modeled file to exist.	SAT; operation labels are `Nop`, `Nop`, while existence is false, true, true.	NOP selection does not frame independently registered file-theory state.
Unconditional requirement	Repeat a false conditional write through a pointer, once nonnull and once null.	Nonnull SAT; null UNSAT.	Dereference validity applies even when the conditioned effect stutters.
Fixed-write hint	A false, single-destination, single-value, full-cell write precedes the marker write.	UNSAT, not the otherwise expected stutter.	The current effect hint is action-guarded but not condition-guarded; this case is excluded from the general claim.

A separate argument check addresses a different kind of sequencing. `PreviousArgument` makes a later operation reuse a solver choice made for an earlier operation in the same action. In the check, both operations select the same `left` location and the one-action query is satisfiable. This result establishes one shared internal binding. `PreviousArgument` does not read memory and is not the mechanism that connects `prepare` to `commit`.

3.3 Relational Composition and Goals

Cross-WMI composition follows from state identity. When lowering interns the same modeled location for two supplied behaviors, an effect of the first action and a guard of the second refer to the same time-indexed state component. If the earlier action establishes a predicate over that component, the predicate can enable the later action. No special cross-instruction argument channel is required; the handoff is an ordinary fact in Σ_I.

Let $R_a \subseteq \mathcal{S}_I \times \mathcal{S}_I$ be the action-boundary relation admitted by action a, including its incoming guard, one admissible binding, its ordered operation transitions, and applicable auxiliary constraints. The running plan is conventional relational composition:

$$(\bar{S}_0, \bar{S}_2) \in R_{commit} \circ R_{prepare} \iff \exists \bar{S}_1 . (\bar{S}_0, \bar{S}_1) \in R_{prepare} \wedge (\bar{S}_1, \bar{S}_2) \in R_{commit}. \tag{1}$$

The witness $\bar{S}_1$ contains both bridge facts, so it satisfies the entire incoming guard of `commit`. The relation hides intermediate operation-time states but does not erase their order.

Goals are predicates over labeled checkpoints rather than only final boundary states. Let $\lambda_t = (O_t, \eta_t)$ contain the operation label and its selected arguments. A goal has the form $\gamma(S_{t-1}, \lambda_t, S_t)$ and is tested at a configured checkpoint set C. State-only goals ignore λ_t; operation-sensitive goals can inspect it. In a focused check, one action changes a tracked value $0 \to 1 \to 2$. The goal $value = 1$ is SAT at operation-end checkpoints and UNSAT at action-end checkpoints, confirming that checkpoint choice changes the bounded claim.

We call this contract universal because every supported operation contributes constraints over one shared time-indexed instance. The contract allows a memory clause, a file clause, or a theory-specific clause to constrain the same plan without forcing those clauses through one uniform manager API. Universal therefore describes compositional interoperability among admitted operations. It does not assert that all theories, effects, targets, or vulnerabilities are implemented.

3.4 A Mixed-Domain Witness

The focused trace in table 3 supplies one cross-domain witness. An unknown constant `0xc0ffeebabe` begins in modeled file state. The selected `file-to-memory` action applies `CopyFileToMem`, moving the value into a tracked buffer without changing its known-constant status. The selected `memory-read` action then applies `Read`, which changes that status from false to true. Both actions occur in one direct bounded query with subgoal orchestration and minimization disabled.

Table 3: Reproduced file-to-memory handoff and change in known-constant status. The action sequence is `file-to-memory`, then `memory-read`; NOP slots are padding at operation width $\ell = 2$.

State	Preceding operation	Tracked buffer	Known-constant status
$S_0 = \bar{S}_0$	Initial state	`0x0`	False
S_1	`CopyFileToMem`	`0xc0ffeebabe`	False
$S_2 = \bar{S}_1$	`Nop`	`0xc0ffeebabe`	False
S_3	`Read`	`0xc0ffeebabe`	True
$S_4 = \bar{S}_2$	`Nop`	`0xc0ffeebabe`	True

Scope. This witness establishes composition among these admitted file, memory, and known-constant status constraints. It does not measure theory coverage or execute a concrete file operation.

The mixed-domain witness establishes one instance of the interoperability claimed above: state produced through one admitted domain satisfies a goal expressed through another in the same query. It neither catalogs supported constructs nor measures their behavior over target workloads. Those broader questions require evidence that this trace does not provide.

4 The JSON WMI Ontology

Independent behavior producers and composition tools require stable names for exploitation locations, arguments, guards, effects, object regions, triggers, and programs. We call this vocabulary the WMI ontology in the engineering sense: it defines an interchange model and a shared interpretation for its terms. It is not an OWL/RDF ontology, and the implementation does not enforce it through a JSON Schema. Because the ontology is an interchange contract rather than a schema, parser acceptance establishes syntax only; it does not establish model validity or trigger realizability.

The canonical `prepare` object makes the vocabulary concrete. Its one JSON `operation` contains two write effects and two conjunctive guards; lowering represents the admitted effects as ordered internal operations. The object also retains the neutral trigger description `prepare();`. Listing 1 abbreviates it; the complete, report-local fixture is `examples/prepare.json`. The paired `commit` fixture uses the same location names, which allows the two independently supplied descriptions to share state.

Listing 1: Abbreviated canonical `prepare` WMI. The omitted effect fields are unchanged in the complete report-local fixture.

```
{
  "name": "prepare",
  "operation": {
    "variables": [
      "(declare-const phase (_ BitVec 64))",
      "(declare-const target_word (_ BitVec 32))"
    ],
    "preconditions": [
      "(= phase #x00)",
      "(= target_word #x00000000)"
    ],
    "effects": [
      {"type": "write", "id": "initialize_target_word",
       "arguments": {"destinations": ["target_word"],
                     "values": ["#x41414141"],
                     "max_length": "4"}},
      {"type": "write", "id": "publish_prepared_phase",
       "arguments": {"destinations": ["phase"],
                     "values": ["#x01"], "max_length": "8"}}
    ]
  },
  "trigger": "prepare();"
}
```

Declarations enter the shared location environment; convertible state preconditions become the incoming action guard when lowering succeeds; and admitted effect objects become an ordered internal-operation sequence. By contrast, the trigger

is retained as metadata for a later consumer. A JSON field therefore does not become executable merely because the parser accepted it, nor does a solver-selected action establish that the recorded trigger realizes the modeled transition.

Location expressions preserve more structure than a flat address name. A bare symbol denotes an interned location, while `GLOBAL` identifies a named or addressed root and `FIELD_ADDR` and `ARRAY_ADDR` derive locations with object provenance. `DEREF` denotes the pointee reached by loading through a modeled pointer. Finally, `(ARG n)` names the WMI producer's trigger-level variable α_n^w; lowering can approximate such a value with the finite-domain `ANY` sentinel, and optional trigger synthesis can solve it separately, but it has no general identity with an internal action or operation binding.

In particular, the pointer cell `p` and the pointee expression `(DEREF 64 p)` denote different locations. Treating a write through `p` as a write to `p` would destroy the dependency that a later guard is intended to observe. The declared `BitVec` and pointer widths document producer types and guide parsing, but current scalar solver values are integer-backed; a successful lowering does not establish bit-precise equivalence to the producer's source semantics.

Structured addresses also support bounded spatial properties without reconstructing the target's complete address space. Consider a root `region` with the canonical declaration `"base":"region", "size":"0x80"` and the direct destination `(ARRAY_ADDR (ARRAY_ADDR region 2 64) 9 4)`. The nested expression retains the originating object even though it denotes a derived address.

Let b be the assigned address of `region`. The destination in this example is $b + 2 \cdot 64 + 9 \cdot 4 = b + 164$. A four-byte direct write therefore covers $[b + 164, b + 168)$, which lies outside the declared half-open object range $[b, b + 128)$. This establishes an out-of-bounds write range in the abstract model; it does not establish an arbitrary write, a changed concrete byte, or a realizable target trigger.

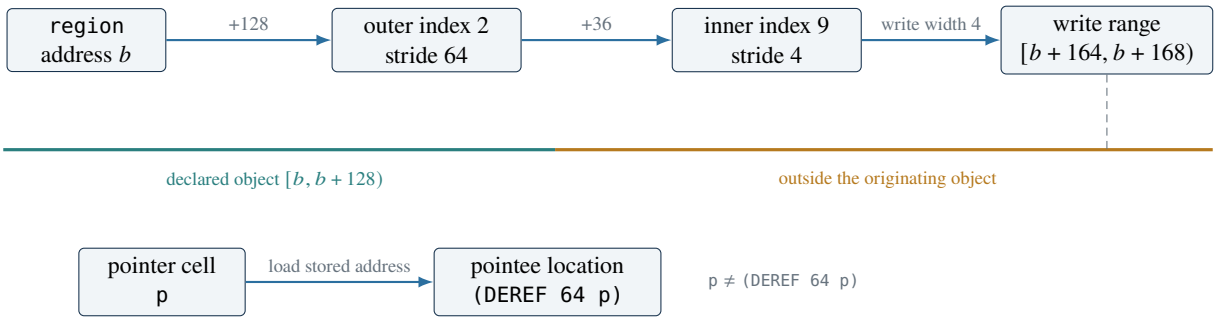

Figure 3: Structured location identity and range provenance. Nested address construction retains the root object, while dereference separates a pointer cell from its pointee.

The interface consequently has several trust layers. Table 4 separates accepted syntax, canonical interchange, current lowering, model validity, and external trigger validity. It also records which WMP fields appear by default and which require optional trigger synthesis. This separation prevents a parser result, a satisfying model, and a concrete exploit step from being treated as interchangeable evidence.

Table 4: JSON constructs, lowering behavior, and the claim supported at each layer.

Construct or layer	Canonical interchange status	Current treatment	Supported claim
`name`, `operation`, `trigger`	Required WMI fields	Names the behavior, contains its model, and retains trigger text	Structural WMI acceptance only
`variables` with bare names or `GLOBAL`	Required declaration list	Interns root locations in the shared environment	Stable abstract identity, not concrete liveness
`DEREF`, `FIELD_ADDR`, `ARRAY_ADDR`	Accepted expression forms	Constructs pointee or derived locations; supported derived forms retain provenance	Modeled address relationship for successfully lowered forms
`(ARG n)`	Accepted trigger-level reference	May be approximated during lowering; raw constraints remain available to optional trigger synthesis	Trigger-variable identity α_n^w, not an internal binding
`preconditions`	Optional conjunctive list	Convertible conjuncts form the incoming action guard; strict mode rejects skipped conjuncts	Guard retention, not producer-semantic validity
`effects[]`	Ordered typed objects	The supported effect list normally lowers to one action containing ordered operations; allocator filtering may remove effects, selected capabilities may yield alternative actions, and unsupported effects become NOPs	Semantics only for the admitted lowered operations
`memory_regions`	Optional canonical `base`/`size` entries	Registers a static half-open extent for supported spatial goals	Abstract direct-write range, not allocation lifetime
WMP `definitions`, `chains[].wmi`, and `human_readable`	Emitted by default	Preserves used WMI definitions and selected order	Interchange projection of a plan admitted by the constructed instance
WMP `trigger`, `bindings`, and `solved_args`	Optional; emitted with trigger synthesis	Reconstructs consumer-oriented trigger data from a selected step and raw WMI constraints	Candidate materialization data, not a lossless internal binding or replay result

Lowering is therefore intentionally partial. The ordinary case produces one uninstantiated action, while allocator selection can filter effects or the entire WMI, and selected capability descriptions can produce alternative actions. An unsupported effect is represented as a NOP rather than silently acquiring semantics. In ordinary non-strict conversion, argument-only, mixed, or unsupported guard conjuncts can remain in the source definition without constraining composition state; strict conversion rejects any skipped conjunct. These cases explain why the ontology preserves the supplied guarded model even when the constructed instance admits only a projection of it.

Strict-mode acceptance and structural guard accounting are necessary representation checks, not proofs of complete effect semantics or concrete trigger fidelity. They determine which clauses may enter the finite constructed instance; they do not strengthen any subsequent satisfiability result beyond that instance.

5 Bounded Composition with Z3

The ontology preserves supplied guarded models; synthesis operates only on the finite instance constructed from models that lower successfully or are supplied directly. For a constructed instance I, UTE collects the admitted actions, active theories, goal, represented locations, constants, and finite internal argument domains before creating any plan slot. This ordering is consequential: it fixes the state vocabulary and choice universe to which every later satisfiability result refers.

Each action slot i chooses an action label A_i and an internal binding vector β_i. If ℓ is the padded operation width, its operation slots choose labels and argument vectors $\lambda_t = (O_t, \eta_t)$ at $t = (i-1)\ell + j$. Both running WMIs contain two effects, so `prepare` and `commit` occupy two action boundaries but four operation times, as shown in figure 4.

The encoding imposes *suffix compaction* on those action slots. If slot i selects NOP, every later slot must also select NOP. This constraint removes assignments that differ only in the placement of trailing padding while preserving the selected non-NOP prefix.

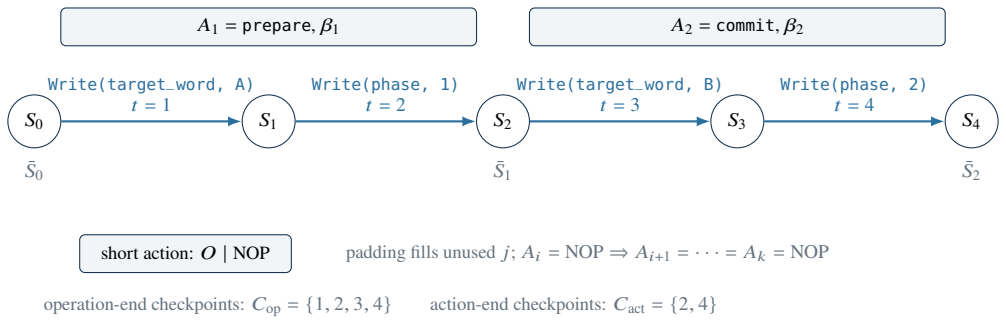

Figure 4: Action slots and operation time for the running plan, with A = `0x41414141` and B = `0x42424242`. Padding gives every action width ℓ, while suffix compaction removes permutations that differ only by trailing NOP actions.

NOP padding, finite internal argument domains, per-action call bounds, and instance-specific binding constraints make this plan space finite. The ANY sentinel denotes a member of those finite domains, not a mathematically unbounded value; attacker-selected write payloads occupy a bounded scratch representation sized from the admitted writes. These are implementation domains chosen for one query, not claims about all values that a concrete target may process.

The bounded Z3 encoding follows the broader planning-as-satisfiability pattern [5, 14]. It does not ask one generic manager to interpret every operation. Instead, one action-selected branch combines action-plan label and argument constraints with tracked-memory, known-constant, register, and permission clauses. For an operation with a false local condition, unconditional requirements remain active; the action-plan clauses enforce $O_t = \text{NOP}$, while cached inertia equates S_{t-1} and S_t on Σ_I^{frame}. Independently registered theory managers add separate state constraints, so neither the NOP label nor equality on Σ_I^{frame} establishes stuttering for every theory state. Table 5 records this boundary.

Table 5: Constraint routes for representative operations in the constructed Z3 instance.

Route	Constraint role	Evidence and scope
Plan selection	Chooses A_i, O_t, β_i, and η_t from finite domains; enforces padding, suffix compaction, and call bounds	Running trace and finite-choice check
Dispatcher branch	Conjoins incoming action guards and unconditional requirements with an active/NOP choice; the false branch sets $O_t = \text{NOP}$ and equates S_{t-1} and S_t on Σ_I^{frame}	Core-memory false-condition check passes; dereference requirements remain active
Tracked memory	Applies `Write`, `CopyFileToMem`, and `CopyRegToMem` updates and frames represented locations	Running trace, mixed-domain chain, and tracked-location inertia check
Known-constant status	Records values exposed by `Read`, `FileRead`, and descriptor reads	Mixed-domain file-to-memory-to-read check
File and descriptor theory	Applies operation-specific file and descriptor updates in the active branch; adds inertia separately, guarded by selected actions rather than local operation conditions	`CopyFileToMem` witness passes; false-conditioned `FileCreate` exposes a frame limitation
Allocator theory	Adds allocator-specific allocation, release, and heap-state constraints	Source correspondence only; no heap witness in this report
Auxiliary clauses	Add scratch domains, direct-write hints, gap defaults, and search-pruning constraints	The fixed full-cell write hint exposes a separate false-condition limitation

Focused checks distinguish three local-condition outcomes. In the core-memory witness, a false conditional multi-destination `Write` enforces $O_t = \text{NOP}$; the tracked target and decoy retain their values, and the next `Write` executes. This scoped core-memory check passes. A fixed, single-destination full-cell `Write` instead becomes UNSAT because its auxiliary effect hint is guarded by action selection but not by the local condition. In the file-theory witness, a false-conditioned `FileCreate` selects NOP while file existence changes from false to true and remains true. The fixed-write and file-theory observations delimit the core-memory result: one prevents an admitted assignment, whereas the other exhibits change outside Σ_I^{frame}.

Tracked-memory framing is correspondingly sparse. The merged memory clauses constrain values at locations collected into I and apply each modeled write over reachable tracked cells. Their frame rules preserve a checked unwritten location across operation times, as the correspondence microcheck confirms. This observational inertia does not cover independently managed theory state, equate complete concrete memory arrays, or prove an omitted byte irrelevant to target execution.

For a fixed instance I and action bound k, UTE constructs the bounded formula in equation (2). The checkpoint set C selects the operation ends or action ends at which the labeled goal $\gamma(S_{t-1}, \lambda_t, S_t)$ may hold. Auxiliary clauses remain explicit because independently added theory-state frames, search pruning, scratch values, and specialized write hints can rule out assignments that the central dispatcher clauses alone admit.

$$
\begin{aligned}
\Phi_{I,k} &= \mathit{Init}_I \wedge \mathit{Plan}_{I,k} \wedge \mathit{Trans}_{I,k} \wedge \mathit{Aux}_{I,k} \wedge \mathit{Goal}_{\gamma,C}, \\
\mathit{Init}_I &: \text{ initial represented state and active-theory state,} \\
\mathit{Plan}_{I,k} &: \text{ finite labels, bindings, padding, and call bounds,} \\
\mathit{Trans}_{I,k} &: \text{ guarded labels and active updates, with false-branch equality on } \Sigma_I^{\text{frame}}, \\
\mathit{Aux}_{I,k} &: \text{ independent theory frames, pruning, scratch, and hint clauses,} \\
\mathit{Goal}_{\gamma,C} &: \bigvee_{t \in C} \gamma(S_{t-1}, \lambda_t, S_t).
\end{aligned}
\tag{2}
$$

A satisfying assignment to $\Phi_{I,k}$ witnesses a plan admitted by this lowering, finite universe, active-theory set, auxiliary clauses, checkpoint policy, and bound. An unsatisfiable result excludes only plans within the same constructed instance. In particular, it neither proves a target unexploitable nor excludes a plan that requires an omitted state component, unsupported

effect, larger domain, different checkpoint, or greater action bound.

UTE can orchestrate several direct queries around $\Phi_{I,k}$. A probe at effective bound b constrains slots after b to NOP; progressive probing increases b and releases those slots while suffix compaction keeps any remaining NOPs trailing. Optional subgoal deduction can solve intermediate obligations before the parent query. For the running fixtures, deduction reports no subgoals: the one-action probe is UNSAT and the first satisfying probe at two actions selects `prepare` followed by `commit`. These observations establish the trace at the recorded bounds, not an asymptotic result.

From a satisfying Z3 model, UTE first extracts the solver-selected internal plan. It then constructs user-visible `FlatAction` records containing an action name, grouped source-operation summaries, selected action arguments, a handler, an action time, the originating action, and subgoal provenance. A `FlatAction` is therefore a convenient record for one selected invocation, not a lossless trace of every operation slot; in particular, source operations can remain in the summary when a local condition selected a solver NOP.

Default WMP serialization makes a different projection. It emits the shared WMI definitions and ordered references, including a human-readable annotation, but no concrete `bindings`. Optional trigger synthesis separately solves raw `(ARG n)` constraints and may add instantiated trigger text, semantic `bindings`, and positional `solved_args`. Listing 2 shows the exact field distinction exercised by the binding microcheck; those optional fields are consumer-oriented and do not reconstruct the internal vectors β_i and η_t.

Listing 2: Default and trigger-synthesized WMP step excerpts for the same selected WMI.

```
Default step:
{
  "wmi": "binding",
  "human_readable": "..."
}

With trigger synthesis:
{
  "wmi": "binding",
  "trigger": "invoke(1, 0x401000, 6);",
  "bindings": {
    "location": "0x401000",
    "length": 8,
    "data": "\\x06\\x00\\x00\\x00\\x00\\x00\\x00\\x00"
  },
  "solved_args": {"0": 1, "1": 4198400, "2": 6},
  "human_readable": "..."
}
```

We call a formal clause implementation-faithful only when it names a production symbol and a focused executable check produces the expected observation at the recorded source snapshot. This criterion is deliberately narrower than source similarity: it connects a statement about initialization, transition, extraction, or projection to behavior that the reproduction harness can audit. Clauses supported only by inspection remain labeled as such in the correspondence record.

That record also turns discrepancies into exclusions instead of hiding them behind the formal model. The scoped dispatcher claim is precise: its false branch enforces O_t = NOP and equality on Σ_I^{frame}. A fixed-write hint can make that otherwise-framed branch UNSAT, while independently managed theory state can change under the NOP label. Separately, a `FlatAction` can retain a source operation after the solver selected NOP, WMI lowering is partial, and trigger-binding serialization is lossy with respect to internal bindings. These limits bound the projections of the solver-selected plan; neither projection is a completed target exploit.

6 Interfaces and Plan Materialization

UTE exposes distinct handoffs rather than one end-to-end execution pipeline. The JSON frontend projects a solver-selected plan into a WMP artifact for an external consumer, whereas the native Python path exposes `FlatAction` records to callable handlers or the C generator. Table 6 compares these paths at the point where the internal plan leaves the solver. In each case, the consumer—not the satisfiability result—is responsible for concrete target success.

Table 6: Interfaces from supplied behavior models to consumer-visible artifacts.

Interface	Input and projection	Binding semantics	Consumer responsibility
JSON/WMP	WMI JSON lowers into a constructed instance; serialization emits shared definitions and ordered WMI references	Default steps contain no concrete binding; optional fields assign trigger-level `ARG`s and semantic step data	Interpret triggers, perform target I/O, and validate each modeled effect
Native Python	Explicit model objects produce an internal plan and `FlatAction` records	Finite internal choices map to positional handler arguments; fixed operation data need not become action arguments	Encode protocol actions or dispatch callable handlers; check runtime outcomes
Annotated C	A solved plan, `FlatAction` records, C handler excerpts, and a shell file produce one C translation unit	`ARGS` substitutes selected action arguments; `RETURNS` names a captured value for later reuse	Compile and execute for the target; compare observed effects with the model

A WMP preserves the source definitions used by the plan and their solver-selected order, but UTE currently provides no general WMP runner. An external consumer must interpret each definition's trigger metadata and decide how it affects a target session. When trigger synthesis is enabled, that consumer may also use the optional fields shown in listing 2; their presence supplies candidate concrete text and values, not evidence that the trigger executed or implemented the modeled effect. Current JSON synthesis emits one selected chain rather than enumerating alternatives.

The native Python path removes that serialization boundary by constructing the model directly. Listing 3 uses the running `phase/target_word` state and explicit `MemoryLocation`, `Action`, `ArgumentMap`, `Goal`, `Binary`, and `Solver` objects inside a `SolverScope`. The commit value has two finite candidates, and its explicit argument map delivers the solver-selected candidate to one callable handler. Full scalar-cell writes keep this API example focused on composition rather than JSON width conversion.

An `ArgumentMap` connects a finite solver choice to a positional action argument; it does not equate that argument with a JSON `(ARG n)`. In the listing, operation zero's `value` becomes handler position zero, and the goal forces the selected value to `0x42424242`. `solve()` populates the internal solution and `FlatAction` sequence, `get_flat_action_sequence()` returns invocation records, and `compose()` dispatches available handlers. `run()` is the checked convenience path that performs solving followed by composition.

Model construction remains ordinary Python. A producer can use functions, comprehensions, or generators to construct a finite list of `Action` objects, materialize that list, and pass it explicitly through `Solver(actions=...)`. These are producer-side factories, not a second planning interface: the solver still selects from the resulting finite action set.

Application code can also treat a `FlatAction` list as an intermediate representation. A consumer may iterate the records, encode each `name` and `action_args` for a protocol, and maintain a reusable connection across steps. This loop is an extension pattern, not a UTE chain-generator API: the solver generates the plan, the list is its user-visible projection, and the application defines how projected records become concrete input.

For C-backed handlers, the checked entry point is `ute.codegen.generate_c`. It reads `ARGS`, `RETURNS`, and `CALL` directives from handler excerpts, substitutes selected arguments, and can reuse a captured return value in a later call. When `ARGS` is present it takes precedence over `CALL`; `RETURNS` augments that argument-driven call. The focused check behind listing 4 emitted the three calls in order and passed a C11 syntax check, but did not compile for or execute against a target. This annotation-aware generator is distinct from the package-level legacy `generate_c` helper.

Listing 3: Checked native Python composition path. The explicit handler receives the value selected for `commit`.

```python
import ute
with ute.SolverScope():
    phase = ute.MemoryLocation("phase", initial_value=ute.Constant(0))
    target_word = ute.MemoryLocation(
        "target_word", initial_value=ute.Constant(0))
    def commit_handler(value):
        return value                # perform target I/O here
    prepare = ute.Action(
        "prepare",
        operations=[
            ute.Write([target_word], [0x41414141], max_len=8),
            ute.Write([phase], [1], max_len=8)],
        precondition=ute.ConditionalAnd(
            ute.Equals(phase, 0), ute.Equals(target_word, 0)))
    commit_args = ute.ArgumentMap()
    commit_args.add_mapping(0, "value", 0)
    commit = ute.Action(
        "commit",
        operations=[
            ute.Write([target_word], [0x42424242, 0x43434343],
                      max_len=8),
            ute.Write([phase], [2], max_len=8)],
        arg_map=commit_args,
        precondition=ute.ConditionalAnd(
            ute.Equals(phase, 1),
            ute.Equals(target_word, 0x41414141),
            ute.NotEquals(target_word, 0x42424242)),
        handler=commit_handler)
    goal = ute.Goal(ute.ConditionalAnd(
        ute.Equals(phase, 2),
        ute.Equals(target_word, 0x42424242)))
    binary = ute.Binary(state_vars=[phase, target_word])
    solver = ute.Solver(
        binary, goal=goal, max_actions=2, actions=[prepare, commit])
    solver.args.subgoals = False
    solver.args.minimize = False
    solver.solve()
    flat = solver.get_flat_action_sequence()
    returns = solver.compose()      # dispatches commit_handler
    # On a fresh Solver, run() performs solve() then compose().
```

Listing 4: Python consumer and generator calls with C handler and output excerpts. Ellipses abbreviate application-specific bodies.

```
# Python: a FlatAction consumer supplied by the application.
for entry in flat:
    emit_protocol(entry.name, entry.action_args)

/* C (shell.c): handler excerpts consumed by generate_c. */
// ARGS: {0}
// RETURNS: unsigned long leaked, {0}
unsigned long leak(unsigned long value) { return value; }

// ARGS: {0}, {1}
// CALL: should_not_appear();
void use_value(unsigned long address, unsigned long value) { ... }

// CALL: finish();
void finish(void) { }

# Python: solver and flat denote the checked C-backed fixture.
ute.codegen.generate_c(solver.solution, "shell.c", "generated.c",
                       flat_action_sequence=flat)

/* C (generated.c): emitted call sequence. */
unsigned long leaked_0 = leak(0x1234);
use_value(leaked_0, 0x42);
finish();
```

Compatibility surfaces require narrower claims. The core `@action` decorator creates an `Action`, appends it to a module-level registry, and returns the decorated function unchanged; the native `Solver` does not automatically consume that registry. A caller can retrieve the actions with `get_registered_actions()` and pass them explicitly through `actions=`; the decorated function becomes the native handler only when `handler=` names it. The `@goal` decorator records a function, but this report checks only an explicit `Goal` object and does not claim automatic goal conversion or discovery. The compatibility `chain()` surface likewise remains limited and is not a substitute for the checked native `Solver(binary, actions=actions)` path.

Subgoal generation is a separate extension point. A `SubgoalGenerator` registered for a conditional type with `register_generator()` can produce prerequisite or alternative subgoal nodes when subgoal orchestration is enabled. The direct Python witnesses in this report set `solver.args.subgoals=False`, so registered subgoal generators do not contribute to those results.

The analogy to pwntools is therefore useful but narrow [19]. A callable handler may close over a reusable tube or protocol object, allowing the selected `FlatAction` sequence to drive stateful target I/O in familiar exploit-development code. UTE supplies the model-level step selection and finite arguments; the handler supplies encoding, transport, observations, error handling, and replay. Keeping this boundary explicit prevents target interaction from being mistaken for a solver transition.

7 Evidence, Trust, and Limitations

The evidence in this report has four layers: the reproduced JSON trace, focused mixed-domain and Python API checks, correspondence between the formal clauses and production symbols, and direct source inspection. Each layer answers a semantic question about one constructed instance or code path. None samples representative workloads or measures runtime, scaling, WMI coverage, prevalence, or concrete exploit success. The evidence therefore supports the mechanisms and boundaries stated in this report, not an empirical comparison.

The principal trust boundary precedes the solver. A WMI producer chooses the represented state, guard, effects, action grouping, and trigger description. An over-approximate model can admit a plan whose effects cannot occur in the target, whereas an under-approximate model can omit a feasible plan. Successful parsing, lowering, and satisfiability establish consistency within the supplied abstraction; they do not establish that the abstraction faithfully describes the target.

Three validation layers must consequently remain distinct. Parser checks establish that accepted syntax produces the expected internal clauses, independent effect validation establishes that a modeled transition matches an observed target transition, and concrete replay establishes that a materialized trigger realizes the selected step in context. The report-local verifier supplies the first layer and checks selected engine semantics. Effect validation and trigger replay require target-specific

evidence from the WMI producer or consumer and remain outside this report.

Every SAT or UNSAT result is also relative to the constructed instance. That instance contains sparse, explicitly tracked locations, integer-backed scalar values, Boolean status components, finite argument domains, a finite action bound, and a selected goal-checkpoint policy. Theory-specific rules, effect hints, user constraints, and optional orchestration can further restrict the query. Changing any of these choices changes the obligation; an UNSAT result at one bound, for example, does not exclude a longer plan or a transition omitted by lowering.

The current implementation introduces narrower boundaries in addition to those inherent modeling choices. Lowering is partial and some supported constructs are approximate. For an operation-local false condition, the dispatcher selects the NOP operation label and applies inertia to represented memory, known-constant status, and active register and permission state, but it does not frame independently registered theory state. In the focused `FileCreate` probe, the query was SAT and the operation label was NOP even though modeled file existence changed from false to true. Separately, a fixed, single-destination full-cell conditional write can conflict with an effect hint when its local condition is false.

Projection introduces a different limitation. A `FlatAction` record can retain a grouped source-operation summary when the corresponding selected internal operation is NOP. The record is therefore useful at action granularity but is not an operation-accurate solver trace.

Several convenience paths are also incomplete. General WMP execution and a JSON-to-C path are absent. The decorator and chain-generator surfaces do not constitute the checked native Z3 path, and the JSON frontend emits one selected chain. Table 7 separates these implementation gaps from fundamental boundaries.

Table 7: Trust boundaries and current gaps. "Witness" denotes a focused report-local check; "inspection" denotes a claim bounded by the recorded production source.

Boundary or gap	Kind	Evidence level	Consequence for use
Producer-selected state, guards, effects, and grouping	Fundamental	Assumption	A satisfying plan can inherit over- or under-approximation in the supplied WMI.
Sparse tracked state and integer/Boolean domains	Fundamental	Construction and inspection	Results concern represented state, not every byte or target fact.
Finite arguments, action bound, and checkpoint policy	Fundamental	Witness and inspection	SAT and UNSAT are conditional on the selected domains, horizon, and goal timing.
Effect validation and concrete trigger replay	Fundamental	External obligation	Model consistency does not establish target feasibility or realizability.
Partial or approximate lowering	Implementation	Parser witness and inspection	Unsupported clauses are not entitled to the formal transition claim.
False local condition on theory-managed state	Implementation	Limitation witness	The checked `FileCreate` instance selected NOP while file existence changed; local-condition stuttering is therefore not a global theory-state guarantee.
False local condition on a fixed full-cell write	Implementation	Limitation witness	This distinct specialized case becomes UNSAT because its action-guarded effect hint is not guarded by the operation-local condition.
`FlatAction` operation extraction	Implementation	Inspection	Grouped source-operation summaries are lossy and need not identify a solver-selected NOP at operation granularity.
WMP execution and JSON-to-C integration	Implementation	Inspection	A WMP is an interchange artifact that requires a separate consumer.
Decorator and chain-generator compatibility surfaces	Implementation	Inspection	They are not substitutes for the checked native `Solver(binary, actions=actions)` path.
Single-chain JSON projection	Implementation	Trace and inspection	The JSON frontend records one selected chain rather than enumerating alternatives.

Users should therefore treat the internal plan, the WMP artifact, and the `FlatAction` sequence as distinct artifacts. The internal plan records model-selected actions and operations. The WMP serializes an interchange projection of selected WMI order and definitions. On the checked direct path, each `FlatAction` record exposes one selected action invocation, its positional arguments, handler, action time, and provenance, together with lossy grouped source-operation summaries. None of these artifacts establishes target execution. A consumer must validate the modeled effects and replay the corresponding

trigger against the target. When observed behavior diverges, the appropriate response is to refine the WMI or model boundary and solve the revised obligation.

8 Related Work

The label *automatic exploit generation* spans systems with materially different inputs and obligations. We therefore compare prior work by the artifact it consumes, who discovers exploitable behavior, the unit and state it composes, the artifact it produces, and how that artifact is validated. These axes place UTE by representation boundary rather than by the word *automatic*. Tables 8 and 9 summarize the categories; the following discussion preserves distinctions among the representative systems within them.

Table 8: Inputs, composition units, and products of related categories. Entries are characteristic of the cited representatives, not claims that every system in a category implements the same pipeline.

Category	Representative input and discovery	Composition unit and retained state	Product and validation
Program-level AEG	Source, binary, or observed crash; program analysis discovers paths [1, 3, 18]	Path constraints and concrete machine or program state	Exploit input or autonomous action, checked against the target
Code- and data-reuse synthesis	Binary plus payload specification or reuse objective [6, 12, 16, 17]	Semantic gadgets, basic blocks, or intermediate-language fragments	Reuse chain or payload, checked with target semantics
Data-oriented synthesis	Binary plus data-flow objective, syscall objective, or attack program [11, 13, 15]	Data flows, target operations, and target-specific state	Data-oriented program with target-specific checking
Heap and kernel specialization	Vulnerability, allocator interface, or proof of concept [4, 8–10, 20–22]	Capabilities, heap operations, vulnerability paths, or layouts	Primitive, layout, or exploit candidate with domain-specific validation
Weird-machine foundations	Attacker-accessible target semantics [2, 7]	Instructions and programs induced by unintended computation	Semantic account of exploitability
Bounded logical planning	Initial state, transitions, goal, and horizon [5, 14]	Time-indexed logical state and selected actions	Satisfying model, checked by the logical encoding
Exploit-construction library	Analyst-written exploit logic and target interface [19]	Procedural I/O and packing primitives	Executable interaction script, validated by use

Table 9: Relation of each category to the scoped contribution of UTE.

Category	Relation to UTE
Program-level AEG	Can discover and validate behaviors that UTE assumes are supplied.
Code- and data-reuse synthesis	Establishes semantic summaries and solver-assisted chaining as prior art.
Data-oriented synthesis	Establishes abstract, multi-step attack construction in one domain.
Heap and kernel specialization	Supplies close domain precedents and potential producer and consumer roles.
Weird-machine foundations	Motivates the instruction and program vocabulary used by UTE.
Bounded logical planning	Provides the planning pattern and solver substrate.
Exploit-construction library	Complements model selection with concrete materialization.

AEG and MAYHEM place concrete program semantics inside exploit construction. AEG uses program analysis and symbolic reasoning to find exploitable paths and construct an input, while MAYHEM applies hybrid symbolic execution directly to binaries and reasons about exploitability along explored states [1, 3]. Their retained execution detail supplies fidelity that UTE deliberately does not reconstruct. The approaches are therefore complementary: program-level analysis can produce and check behavior models, whereas UTE composes models already supplied at the WMI boundary.

Mechanical Phish broadens this program-level setting into an autonomous vulnerability-discovery, triage, exploitation, and patching pipeline [18]. Its exploit-construction component begins from a crashing input, symbolically traces that crash, and applies techniques for selected crash classes. That published account describes a crash-local workflow, whereas UTE begins from producer-encoded behavior models that may include setup effects. This comparison neither equates the systems' internal state representations nor claims that Mechanical Phish generally cannot recover preparatory behavior.

Code- and data-reuse systems already construct chains from semantic summaries and logical constraints. Q describes useful computation through gadget semantics, BOPC maps a data-only payload to candidate basic blocks and connects them, SGC synthesizes code-reuse chains from semantic gadgets, and p-code synthesis compiles an intermediate program into

reuse behavior [6, 12, 16, 17]. These systems differ in attack model and output, but together preclude a novelty claim for semantic summarization or solver-assisted chain construction alone.

Data-oriented exploit generation provides equally direct precedents for abstraction over multiple steps. FlowStitch connects target data flows to realize a data-oriented objective; Einstein constructs practical data-only attacks around security-sensitive effects; and DOPPLER synthesizes a target-level realization of a data-oriented attack program [11, 13, 15]. Their representations retain the target facts required by their respective attack models. UTE adopts the same general lesson—compose over a purpose-built abstraction—while placing admitted operations behind a shared WMI contract.

Capability and heap reasoning are the closest domain-specific precedents to that contract. KOOBE derives and composes capabilities for kernel out-of-bounds writes, while HeapHopper and ArcHeap reason over sequences of abstract heap operations to expose allocator behaviors and exploitation primitives [4, 8, 22]. They show that an exploit question can be stated over capabilities or operations rather than a complete machine trace. Their abstractions remain tailored to kernel writes or heap behavior; UTE instead supplies one interface for clauses admitted from several theories.

Other systems combine behavior discovery with the concrete work needed to realize it. Revery advances from a proof of concept toward an exploit, FUZE explores kernel use-after-free contexts and evaluates candidate operations, SHRIKE searches for heap layouts, and Gollum uses modular greybox reasoning for interpreter heap overflows [9, 10, 20, 21]. These systems illustrate producer and consumer responsibilities that UTE leaves external. The analogy identifies compatible roles, not implemented integrations or equivalence among their techniques.

The weird-machine literature supplies the semantic interpretation, and bounded planning supplies the logical construction. Weird-machine accounts treat attacker-accessible behavior as an instruction set whose composition forms an exploit program [2, 7]. Planning as satisfiability encodes a finite transition horizon as a logical formula, while Z3 provides the SMT substrate used here [5, 14]. UTE applies these established ideas to supplied WMI transitions; it does not introduce either the weird-machine interpretation or satisfiability-based planning.

Pwntools occupies the complementary materialization boundary. It provides reusable procedural facilities for packing values, transporting bytes, and interacting with a target, but it does not select a WMI plan from a guarded transition model [19]. A UTE handler can close over such an interaction object after the solver has selected an action and its arguments. Across the categories above, the scoped distinction of UTE is therefore a common contract for supplied WMIs and an explicit separation between model-level composition and target-specific materialization.

9 Conclusion

The running `prepare/commit` dependency captures the representation problem at the center of this report: an earlier behavior establishes `phase` and `target_word` facts that a later behavior requires. Concrete target semantics remain necessary to discover those behaviors and to test whether their triggers realize the stated effects. Once a producer supplies faithful guarded transitions, however, shared abstract state allows the composition query to connect the dependency without retaining a complete target execution model.

UTE realizes that separation through a sequence of explicit boundaries. The JSON ontology serializes a WMI vocabulary; partial lowering converts supported constructs into internal actions; and the bounded Z3 encoding selects an internal plan admitted by the constructed instance. The WMP artifact serializes selected WMI order and definitions as an interchange projection. On the native Python path, the separate `FlatAction` sequence exposes selected action invocations, positional arguments, handlers, times, and provenance to a consumer, with grouped source-operation summaries that are not operation-accurate traces. Neither projection is execution evidence, so materialization and replay remain separate responsibilities.

The report-local witnesses show that admitted file, memory, and known-constant status clauses can participate in one guarded composition contract, including a file-to-memory step whose effect enables a later memory read. This is the operative meaning of *universal*: supported theories share an interface for state, guards, effects, and ordering. The evidence does not establish complete theory coverage, WMI fidelity, trigger realizability, concrete exploit success, or empirical scaling. Those obligations require additional producers, consumers, and measurements beyond the bounded Z3 encoding described here.

A Formal–Implementation Correspondence

We bind the formal claims to source revision `2eb3cfb5e3f26baad0e80b2e404f6c10efc9ee27`, the implementation-file hashes in Appendix B, WMI parser version 0.8.1, WMI specification version 0.3.0, Z3 version 4.13.0, and CPython 3.12.3. The verifier found no tracked modifications. Table 10 distinguishes executed report-local checks from source-only observations and maps each claim to its production symbols.

Table 10: Correspondence between report claims and the recorded implementation.

Claim or clause	Production route and focused check	Scoped expectation	Recorded observation	Disposition
Partial JSON lowering	`JSONWMIParser.parse_json_file`; `WMIToUTEConverter.convert_wmi_to_action`; parser probe and strict CLI	The two fixtures parse, convert, and retain their admitted writes.	Strict conversion accepted both fixtures; `prepare` and `commit` each yielded two effects. Inspection shows that other constructs can be filtered, approximated, or rejected.	PASS, scoped
Guard retention	`WMIToUTEConverter.convert_preconditions`; parser probe	Raw, parsed, and lowered guard counts agree, with no skipped conjunct.	`prepare`: 2/2/2; `commit`: 3/3/3; both reported zero skipped guards.	PASS
Structured locations and ranges	`SMTToUTEConverter._get_array_location`; `WMIToUTEConverter._register_memory_regions`; `_bounded_region_for`; `_OutOfBoundsWriteCondition.z3ify`; source inspection	Constant array offsets can be flattened to a root location, declared region sizes can be attached to that root, and the direct-write range condition can compare a selected write interval with that extent.	The inspected routes accumulate constant index–stride offsets, attach `region_size`, and test nondereferenced `Write` or `CopyMem` intervals. No report-local query witnesses this route.	INSPECTED
Finite internal bindings	`ArgumentMap`; `ActionPlanManager.action_argument_constraints`; `operation_argument_constraints`; `microcheck_finite_choice`	One action chooses one location and value from finite domains.	Domains {left,right} and {42,99} produced the selected pair (left,42).	PASS
Reused internal binding	`PreviousArgument`; `ArgumentMap`; `microcheck_previous_argument`	A later operation reuses the earlier operation's selected location argument.	Both operation mappings referenced the same action argument and selected `left`.	PASS
JSON `ARG` separation	`convert_preconditions`; `synthesize_trigger`; `chain_to_steps`; `microcheck_wmp_trigger_binding`	Trigger-level `(ARG n)` values remain distinct from general internal action bindings.	The definition retained parameterized `ARG`s; optional synthesis produced positional `solved_args` and semantic `bindings`, while the default step contained neither.	PASS, scoped
Ordered operation time	`ActionPlanManager.get_global_operation_index`; `ActionPlanManager.z3_rules`; forward and reverse intra-action checks	A later operation observes an earlier operation's effect, but not the reverse order.	The write-then-guard fixture was SAT; the guard-then-write fixture was UNSAT.	PASS
Local-condition dispatch for represented core state	`Solver._build_merged_apm_mm_cm`, nested `action_op_rules`; `operation.precondition.z3ify`; `operation.requirements`; nested `_c_if`; `ActionPlanManager.z3_rules`; `microcheck_operation_condition_stutter`	In the checked memory instance, a false local condition selects the NOP operation label, preserves represented memory, and permits the next grouped operation.	Operation labels were NOP then Write; target and decoy memory remained unchanged, and the later marker write occurred. Source inspection shows that the cached NOP state clauses cover only memory, known-constant, and active register and permission state.	PASS, scoped
Unconditional operation requirements	Nested `Solver._build_merged_apm_mm_cm.action_op_rules`; `operation.requirements`; conditional dereference checks	A requirement appended outside the local `_c_if` remains active even when the effect condition is false.	The nonnull control was SAT, selected NOP, and preserved the checked memory; the otherwise identical null-pointer instance was UNSAT.	PASS

Continued on next page

Table 10: Correspondence between report claims and the recorded implementation (continued).

Claim or clause	Production route and focused check	Scoped expectation	Recorded observation	Disposition
Local-condition frame gap for theory state	Nested `Solver._build_merged_apm_mm_cm.action_op_rules`; `FileCreate.z3_rule`; `FileSystemManager.add_z3_rules`; `TheoryManager.add_inertia_rules`; `microcheck_theory_state_condition_limitation`	A global stutter interpretation would require a false-conditioned `FileCreate` to preserve file existence.	The local NOP branch omitted the file effect, but action-level theory inertia did not frame the selected action's file state. The query was SAT with labels [NOP,NOP] while file existence followed [false,true,true].	PARTIAL
Fixed-write effect hint	`Solver._add_write_effect_hints_at`; `microcheck_fixed_write_condition_limitation`	Without the auxiliary hint, the false local condition would select NOP and preserve represented memory under the dispatcher relation.	The specialized fixed single-destination, full-cell instance was UNSAT because the effect hint is action-guarded but not condition-guarded.	PARTIAL
Goal checkpoints	`Goal.all_z3_rules`; `Solver._add_goal_constraints`; operation-end and action-end checks	A transient value can satisfy an operation-end goal but not an action-end goal after a second overwrite.	For the timeline $0 \to 1 \to 2$, operation-end was SAT and action-end was UNSAT.	PASS
Direct finite bounds	`Solver.solve_iterative`; JSON CLI trace	Bound one excludes the running dependency; bound two admits `prepare`, then `commit`.	The one-action query was UNSAT and wrote no WMP; the first SAT bound was two with exactly that order and no deduced subgoals.	PASS
Cross-domain composition	`CopyFileToMem.new_mem_val`; `MemoryManager.z3_rules`; `Read.get_read_values`; `ConstantManager.z3_rules`; `microcheck_mixed_domain_chain`	One direct bounded query can pass a file value through tracked memory into known-constant state.	The selected actions were file-to-memory then memory-read, with operation slots [CopyFileToMem,NOP,Read,NOP]; the buffer changed from zero to `0xc0ffeebabe`, and the value's known-constant status followed [false,false,false,true,true].	PASS, scoped
Tracked-location inertia	`MemoryManager._z3_rules_standard`; `microcheck_tracked_location_inertia`	An explicitly tracked, unwritten location retains its value across operation slots.	The unwritten location remained `0x63` throughout; the written location followed $0 \to 1 \to 1$.	PASS, scoped
Permission state	`Solver._classify_operations`; nested `action_op_rules`; `PermissionManager.z3_toggle_rules`; `PermissionManager.z3_inertia`; source inspection	When permission-changing operations and toggled groups are active, the merged dispatcher can apply their updates and frame other toggled permissions.	Inspection finds permission operations classified by `permission_updates`, with toggle rules on the selected route and inertia otherwise. No report-local query exercises this state.	INSPECTED
Register state	`Solver._extract_registers`; `Solver._classify_operations`; nested `action_op_rules`; `RegisterManager.z3_rules`; `RegisterManager.z3_inertia`; source inspection	Active register operations can update selected register state while other registers persist.	Inspection finds register extraction, operation classification, per-register writes, and inertia in the merged route. No report-local query exercises register state.	INSPECTED
Allocator theory state	`BaseHeapManager.add_z3_rules`; `TheoryManager.add_inertia_rules`; `ActionPlanManager.z3_rules`; source inspection	A selected allocator manager can register theory-state persistence, while allocator operations contribute their own transition clauses through the action-plan route.	The inspected base manager delegates persistence to common theory inertia, and operation clauses enter through `ActionPlanManager.z3_rules`. No report-local query instantiates an allocator manager.	INSPECTED
Stack-relative locations	`Solver._extract_registers`; `StackManager.resolve_stack_locs`; `StackLoc.resolve`; source inspection	When stack configuration is active, a stack placeholder can resolve to the prior register value plus its modeled offset before operation constraints are built.	Inspection finds stack-driven register activation and argument replacement through `StackLoc.resolve`. No report-local query exercises this route.	INSPECTED

Continued on next page

Table 10: Correspondence between report claims and the recorded implementation (continued).

Claim or clause	Production route and focused check	Scoped expectation	Recorded observation	Disposition
Python subgoal generators	`register_generator`; `deduce_subgoal_tree`; `Solver.deduce_and_solve_subgoals`; source inspection	Condition-specific generators can be registered and invoked by optional subgoal orchestration before the parent solve.	Inspection finds registration by condition type and DAG construction through registered generators. The direct report witnesses disable subgoals and therefore do not exercise this extension path.	INSPECTED
WMP projection	`chain_to_steps`; `build_wmp_output`; WMP binding-mode check and JSON trace	Definitions preserve parameterized triggers and steps preserve selected WMI order; concrete fields are optional.	Default steps contained only `wmi` and `human_readable`; trigger synthesis added the exact trigger, semantic bindings, and positional solved arguments. No target replay occurred.	PASS, scoped
`FlatAction` projection	`Solver.get_flat_action_sequence`; `Solver._make_flat_action`; handler check and source inspection	On the checked direct path, each record exposes `name`, `steps`, `action_args`, `handler`, `at`, `action`, and `subgoal_reason` for one selected action invocation.	The check preserved handler identity and positional arguments. The `steps` field contains grouped source-operation summaries and can retain a source operation when the selected internal operation is NOP.	PARTIAL
Python handler dispatch	`Solver.compose`; `Solver.run`; `microcheck_handler_dispatch`	`run()` solves, then invokes the callable with selected positional arguments and returns collected handler results.	The handler was called once with (left,42), the argument map was {location:0, value:1}, and `run()` returned `[42]`.	PASS
Annotated C generation	`ute.codegen.generate_c`; `microcheck_native_c_codegen`	`ARGS`, `RETURNS`, and `CALL` directives produce the expected calls from solved records.	Generation emitted `unsigned long leaked_0 = leak(0x1234);`, `use_value(leaked_0, 0x42);`, and `finish();`; `cc -std=c11` accepted the synthetic output. It was not executed against a target.	PASS, scoped
C entry-point separation	Import-identity check in `microcheck_native_c_codegen`; source inspection	The annotation-aware entry point is identified without equating distinct helpers or inventing a JSON-to-C route.	`ute.codegen.generate_c` was callable; package-level `ute.generate_c` was a distinct implementation, and the checked JSON path had no integrated C output.	PASS
WMP execution boundary	`build_wmp_output`; source inspection of JSON and native interfaces	WMP serialization is not described as native handler dispatch or concrete execution.	The checked path writes the artifact but exposes no general WMP deserializer/runner; native `compose()` consumes `FlatAction` records instead.	INSPECTED
JSON output multiplicity	JSON frontend synthesis and output branch; running trace and source inspection	The report claims one selected JSON chain, not alternative-chain enumeration.	The trace contained one chain; the current JSON path remains single-chain when alternative-chain output is requested.	INSPECTED

We use *implementation-faithful* only for rows whose scoped expectation passed at this snapshot. PARTIAL rows record a concrete discrepancy and narrow the corresponding main-text relation; INSPECTED rows report source structure without implying that the report executed the route. Even a passing row establishes correspondence only within its constructed instance, not WMI fidelity or target behavior. This disposition rule prevents the formal notation from asserting stronger semantics than the implementation evidence supports.

B Reproduction

The complete report-local fixtures are `examples/prepare.json` and `examples/commit.json`. They use only the abstract names `phase` and `target_word` and contain no case-identifying metadata. The verifier checks their digests and the exact declarations, guards, ordered writes, triggers, and goal shown below.

Listing 5: Canonical transition signature checked by the verifier.

```
variables:
  (declare-const phase (_ BitVec 64))
  (declare-const target_word (_ BitVec 32))

prepare:
  guards:  (= phase #x00)
           (= target_word #x00000000)
  effects: write(target_word, #x41414141, max_length=4)
           write(phase,       #x01,       max_length=8)
  trigger: prepare();

commit:
  guards:  (= phase #x01)
           (= target_word #x41414141)
           (not (= target_word #x42424242))
  effects: write(target_word, #x42424242, max_length=4)
           write(phase,       #x02,       max_length=8)
  trigger: commit();

goal:
  (AND (EQUAL phase #x02)
       (EQUAL target_word #x42424242))
```

Listing 6: Recorded environment and fixture provenance.

```
source revision:
  2eb3cfb5e3f26baad0e80b2e404f6c10efc9ee27
versions:
  parser=0.8.1  WMI-spec=0.3.0  Z3=4.13.0  CPython=3.12.3
fixture SHA-256:
  prepare.json=233ff51f636fe414c0ebc6b8227867f85dd1dbbe9981fa95b5c433aa7ad24793
  commit.json=cf58694e374d489af0a1e40535bd2d6df282d27807e923d0f184efa4adac45cb
```

The Git revision identifies tracked source history, and the verifier records whether tracked files differ from that revision. The separate manifest below supplements that provenance with hashes for selected implementation files used by the correspondence checks; it is not presented as an exhaustive manifest of every imported module.

Listing 7: Selected implementation-file hashes checked by the verifier.

```
implementation SHA-256:
  action_model=25fe5657c649c6c45193a2e29516efea086b723e87464a308cbdb45101d77214
  annotated_codegen=b63366e70ae54e204749985aee9191455acfe24dd0fd7a58cd6fc4aaa106f000
  package_level_codegen=2619c65f4eb300f7406cce2c99343c7533448bbc2d3ca8a6861099a79b5d7ff0
  wmi_parser=200662b36a801a80bc53b3280793f150c8c87ab7bb7b826046b631dc7e99d861
  z3_constant_manager=661d090d42754bfd39cbc28a148cde21db7a21a5c69adbce53810ffe447036da
  z3_file_operations=9ddccbdcd9a60a2703430300a1fabca6f2c9c9b526df5565ee6d2a84884461b6
  z3_goal_model=40f8e603cc66ffc5758baa9268445ef144370d75cd869acad7f7b7e18f56bfcf
  z3_solver=6ffb1a8d07b40a174ca384d50672bd54109eb7f5be507e7f4c62761c0c53d4e0
```

Running `scripts/verify_examples.py` launches each stateful Python microcheck in an isolated process. For the JSON trace, the script accounts for every running-example guard, accepts strict conversion, reports one-action UNSAT and two-action SAT in the order `prepare`, `commit`, and records that no subgoals were deduced.

Focused semantic queries check ordered operation time, goal checkpoints, tracked-location inertia, and one file-to-memory-to-known-constant chain with subgoal orchestration and minimization disabled. The operation-local checks separate three observations: a false-conditioned multi-destination write selects NOP and preserves the checked memory locations; a false-conditioned `FileCreate` is SAT with NOP labels while file existence changes; and a fixed single-destination full-cell write is UNSAT because its effect hint is not condition-guarded.

Interface checks cover finite and reused internal arguments, default and synthesized WMP fields, exact Python handler dispatch, and annotation-aware C emission. The C check compiles its synthetic output for syntax but does not execute it, and the WMP check serializes an artifact without invoking a target. A successful verifier run ends with status `PASS` and can be invoked from the report root as follows.

Listing 8: Report-local reproduction command.

```
PYTHONDONTWRITEBYTECODE=1 ./scripts/verify_examples.py
```

These checks reproduce an abstract state-transition trace and focused clauses of the recorded implementation. They do not execute either trigger against a target, independently establish WMI effect fidelity, run a general WMP consumer, or execute the generated C. The constructed instances do not sample target workloads and provide no measurement of runtime scaling, theory coverage, or exploit success. Reproduction should therefore be read as evidence for the scoped correspondence in Table 10, not as evidence that a selected candidate realizes a concrete exploit.

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS 2011. The Internet Society, 2011. URL https://www.ndss-symposium.org/ndss2011/aeg-automatic-exploit-generation/.

[2] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. Exploit programming: From buffer overflows to "weird machines" and theory of computation. *;login:*, 36(6):13–21, Dec. 2011. URL https://www.usenix.org/publications/login/december-2011-volume-36-number-6/exploit-programming-buffer-overflows-weird.

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing MAYHEM on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012. doi: 10.1109/SP.2012.31.

[4] W. Chen, X. Zou, G. Li, and Z. Qian. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *29th USENIX Security Symposium*, USENIX Security 20, pages 1093–1110. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng.

[5] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[6] M. DenHoed and T. Melham. Synthesis of code-reuse attacks from p-code programs. In *34th USENIX Security Symposium*, USENIX Security 25, pages 395–411. USENIX Association, 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/denhoed.

[7] T. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2020. doi: 10.1109/TETC.2017.2785299.

[8] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. HeapHopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium*, USENIX Security 18, pages 99–116. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/eckert.

[9] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium*, USENIX Security 18, pages 763–779. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/heelan.

[10] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1689–1706. ACM, 2019. doi: 10.1145/3319535.3354224.

[11] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, USENIX Security 15, pages 177–192. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu.

[12] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882. ACM, 2018. doi: 10.1145/3243734.3243739.

[13] B. Johannesmeyer, A. Slowinska, H. Bos, and C. Giuffrida. Practical data-only attack generation. In *33rd USENIX Security Symposium*, USENIX Security 24, pages 1401–1418. USENIX Association, 2024. URL https://www.usenix.org/conference/usenixsecurity24/presentation/johannesmeyer.

[14] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992. URL https://www.cs.cornell.edu/selman/papers/pdf/92.ecai.satplan.pdf.

[15] Y. Ling, G. Rajiv, K. Gopinathan, and I. Sergey. Sound and efficient generation of data-oriented exploits via programming language synthesis. In *34th USENIX Security Symposium*, USENIX Security 25, pages 413–429. USENIX Association, 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/ling.

[16] M. Schloegel, T. Blazytko, J. Basler, F. Hemmer, and T. Holz. Towards automating code-reuse attacks using synthesized gadget chains. In *Computer Security – ESORICS 2021*, volume 12972 of *Lecture Notes in Computer Science*, pages 218–239. Springer, 2021. doi: 10.1007/978-3-030-88418-5_11.

[17] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *20th USENIX Security Symposium*, USENIX Security 11. USENIX Association, 2011. URL https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy.

[18] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, N. Stephens, R. Wang, and G. Vigna. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy*, 16(2):12–22, 2018. doi: 10.1109/MSP.2018.1870858.

[19] The pwntools developers. pwntools: A CTF framework and exploit development library, 2026. URL https://github.com/Gallopsled/pwntools.

[20] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable (one step towards automatic exploit generation). In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1914–1927. ACM, 2018. doi: 10.1145/3243734.3243847.

[21] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium*, USENIX Security 18, pages 781–797. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei.

[22] I. Yun, D. Kapil, and T. Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium*, USENIX Security 20, pages 1111–1128. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/yun.