JNSA 定例会 ― clearwing（OSS 脆弱性発見エンジン）



Clearwing検証とその考察

同一標的の反復検証から整理した、検出力・所要時間・運用制約と、多層防御上の役割



プレゼンター
Kazuya Tsunoda
NTT Resonant Technology inc.

Remote TestKit

SAVE the DIGITAL WORLD
JNSA AGEST

目次

本日の流れ — 期待、現実、診断、そして処方。

期待　I. 防御上の位置 ／ II. clearwingとは ／ III. 検証の設計

現実　IV. 稼働の壁 ／ V. 検証結果 — ffmpeg 実在脆弱性の検出実験

診断　VI. 時間の壁 — なぜ縮まない ／ VII. 到達の壁 — なぜ届かない

処方　VIII. 順位は直せる ／ IX. 導入判断と撤退基準

I. 防御上の位置

既存防御の上に現れた新しい統制 — それが clearwing。

ネットワーク／エンドポイント層
FW・WAF・IDS/IPS・EDR

到達可能性と被害の拡大を抑える
既知の攻撃を広く・安く止める

開発プロセス層
コードレビュー・SAST・fuzzing

コードに残る欠陥を減らす
既知パターンを機械的に潰す

ソース層（本検証の対象）
clearwing

本検証

未知欠陥の候補探索を補助する
計算時間が大きい → 残余リスクの高い箇所へ限定投入

単独の防御線にしない。攻撃側は 1 個で勝つので、全件均等走査は防御側に不利な賭け方。
防御側の構造的優位の一つ＝ソースと型と履歴を context にできること。だからこそ、どこに計算を置くかを設計する。

clearwing の正体 — 2 本のパイプラインと共通基盤。

clearwing — 自律型 脆弱性スキャナ & ソースコードハンター

Glasswing 着想・Lazarus AI 製 ／ Rust 製 runtime genai-pyo3 で各社モデルに対応

① ネットワーク診断エージェント

ReAct ループ + 入出力 guardrail

live 標的をスキャン→悪用（人手承認）

② ソースコードハンター 本検証はこちら

段構えパイプライン（次ページ）

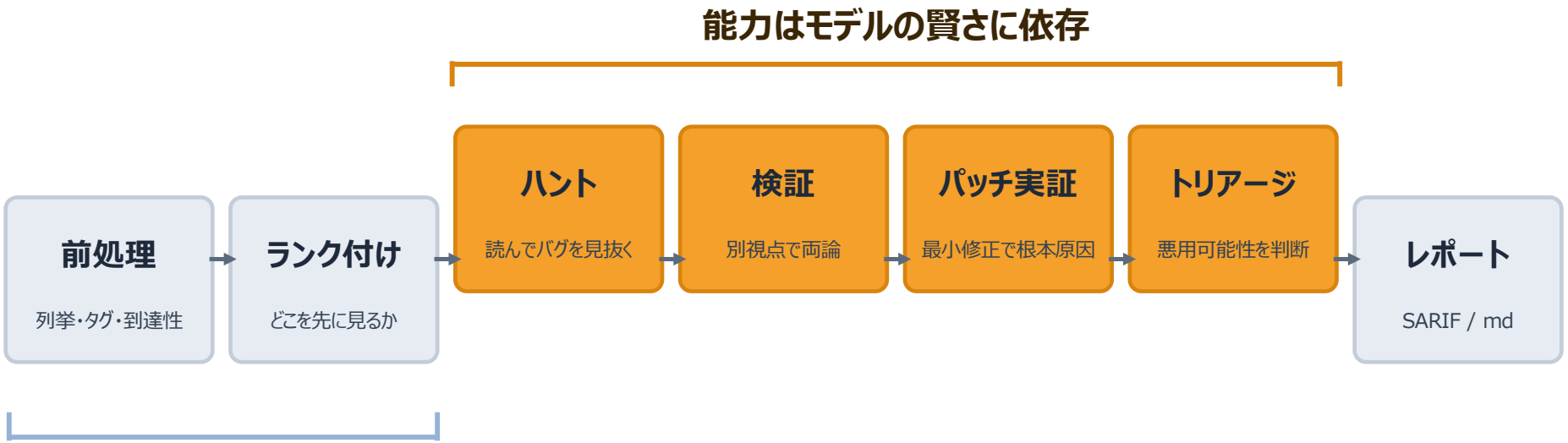
順位付け→per-file hunt→検証

共通基盤（両パイプラインが共有）

単一 Finding 型	sandbox (ASan/UBSan/MSan)	knowledge graph	memory	event bus	observability	safety (guardrail/audit)

II. clearwingとは｜足場と検出

足場は安価に改善できる。だが"検出"はモデルの頭脳しだい。



足場 ＝ ハーネスが担う（安価・機械的・改善できる）

II. clearwingとは｜検査の流れ

候補を絞り、探索し、独立に疑い、再現を試みる。

① ranker：調査対象のファイルを順位付けする

② hunter：仮説を立て、ツールと sandbox を使って探索する

③ verifier：hunter の履歴を渡さず、独立に反証を試みる

④ exploiter：PoC を組み立て、triggerable か確認する

II. clearwingとは｜なぜ賢いか

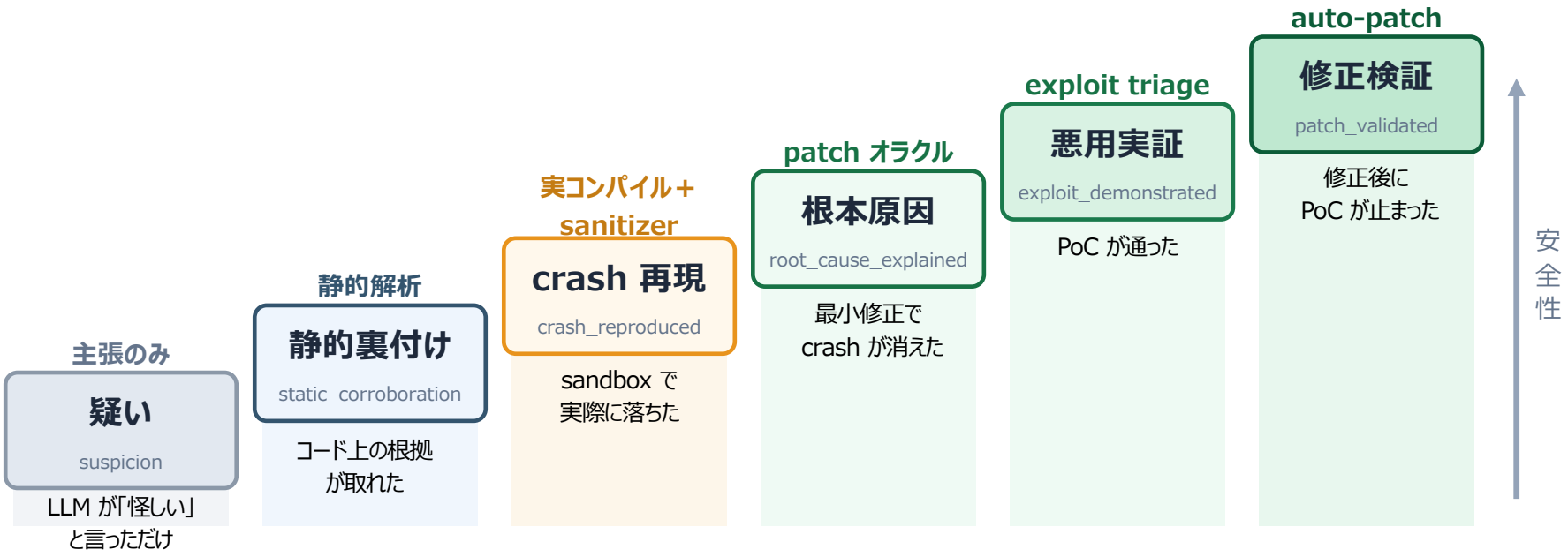
なぜ素のモデルに投げるより賢いのか。

	素のモデルに 1 ファイル		clearwing（足場あり）
見る範囲	1 ファイルの断片だけ	→	read_source_file / grep_source /find_callers で多段に読み歩く
与える文脈	プロンプトの記載が全て	→	seed クラッシュ・semgrep ヒント・entry point を外から注入
観点	汎用の一発質問	→	タグ・言語から専門家を自動選択（固有チェックリスト付き）
計算配分	一律	→	priority→tier で試行 1〜3 回、fast→standard→deep へ昇格
確定の仕方	単発の断定	→	別エージェントが反証、sandbox で再現・最小修正の裏取り

足場は『どこに計算を寄せるか』を決める — 見抜くのはモデルの頭脳。

II. clearwingとは｜なぜ賢いか

確度は『主張』ではなく『実証』で上がる。



clearwing は sandbox·patch·PoC で確度を上げられる設計。素のモデルは「疑い」で止まる。
ただし全 finding が実証段へ到達するとは限らず、static / verified=false で終わる場合もある。

その clearwing を測る — 対象は sourcehunt だけ。

sourcehunt / network / reveng / N-day・SBOM の 4 ドメイン

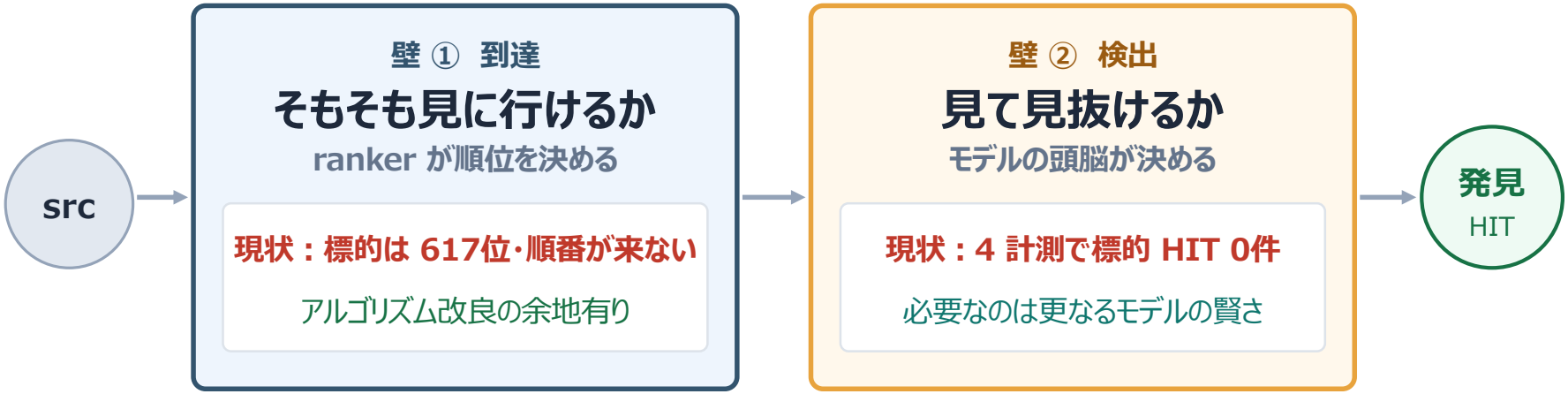
実行したのは、3 ファイルを対象にした sourcehunt のみ

network・reveng・N-day の検出性能は今回評価していない

製品の機能一覧と、今回確認できた能力を分けて説明する

III. 検証の設計 | 先に結論

壁は二つ、到達と検出。Clearwingは発展途上フェーズ。



両方を成立させる足場（ハーネス） = clearwing （読み歩き・文脈注入・専門家・計算集中・独立反証）

足場が無いと答えに届きにくい。ただし足場自体まだ改良余地が多い — これが現在地。

III. 検証の設計 | 持ち帰り

全ツリーを回す前に、時間を使う場所を設計する。

測る
- まず自社の既知バグで、モデル別の HIT 数と所要時間を測る

到達を上げる
- 絞る: 対象を git diff・重要 subsystem・variant hunt に限定
- 順位付けを直す: ranker改良で標的を上位に来るようにする

完走させる
- 長時間 run は予算・timeout・並列数・中断時の保存を先に決める

IV. 稼働の壁

まず回せるのか — メモリ容量で候補が絞られる。

ハード: DGX Spark ×2（GB10）— 1 node 121GB unified memory・合計 約242GB

	モデル・量子化	サイズ	Spark ×2 で	実測での挙動
○	DeepSeek-V4-Flash UD-Q8_K_XL	162 GB	○ 2 node で動作	各 node 約80GB に分散して常駐 — 現用構成
○	GLM-5.2 UD-IQ1_S（1bit）	217 GB	○ 2 node で動作	8 tok/s 止まり・標的 MISS（参考値・内部記録）— 載っても使えない
△	DeepSeek-V4-Flash UD-IQ3_XXS	104 GB	△ 1 node に載る	1 台構成は退けた — dual + Q8 が現行の正
×	Kimi K3 REAP55 + IQ1_M	342 GB	× 3 node 必要（算術）	約510W・55%枝刈＋1bit の二重劣化 — 品質は別評価
×	Kimi K3 Q2_K	865 GB	× 約 8 node 相当（算術）	実ロード試験なし — 算術による除外

載ること(収容)と使えること(検出力)は別 — GLM5.2は1bit まで削れば載るが 8 tok/s と遅すぎる

IV. 稼働の壁

同じモデルでも、量子化と構成で速度は 2 倍変わるが..

2 node 分散·Q8（162GB）
ほぼ無劣化 — 現用構成

prefill **346 tok/s**

decode **約 10 tok/s**

初期ロードに 10 分 19 秒（162GB）
— 起動時間も可用性コスト

単機·IQ2_XXS（約 2bit）
速い — が、精度を削った構成

prefill **865-1018**

decode **約 21 tok/s**

同一 box の llama.cpp 比較でも
decode 15.2→21 / prefill 477→865-1018

2×

decode 約10 → 21

同一モデルの decode 差
別ハード·別エンジン·
量子化の複合差

低bit量子化には品質低下リスクがある。本比較で測ったのは速度。
— tok/s を単独 KPI にせず速度と精度はセットで測る。

IV. 稼働の壁

設定ひとつで、所要時間は桁で変わる。

KV offload を切る

同一プロンプト 32,982 tokens で比較

KV offload 有効（既定）

77.1 秒

KV offload 無効（--no-kv-offload）

約 50 分

同一プロンプトで約 39 倍。
KV offload は llama.cpp
ローカル実行の必須設定

並列スロットを開ける

同時 4 リクエストの所要

-np 1

41.1 秒

-np 4 -cb

10.7 秒

スロット 1 は並列投入が直列化する
並列 8 も 20.7s で全件成功

代償は context 長

1 スロットで使える context

-np 1

256K

-np 4

64K

llama.cpp -np は総 context(-c)を
スロット数で分割（図は -c 262144）
長大 prompt の工程とは配分を設計する

他にも — 162GB の初期ロードに 10 分 19 秒 / model 一覧は 0.5 秒で 200、なのに生成は 272 秒後に 500 ERR

2 node 構成は片側の停止で endpoint 全体が止まる — 死活確認と切り分け手順を検証計画に最初から入れる

V. 検証結果 | 標的 | ffmpeg脆弱性の検出実験

検証する標的は作り物ではない。実際に修正された脆弱性である。

2026 年 3 月、FFmpeg 本体で修正された heap overflow

65536 スライスで、索引表の未割当マーカー 0xFFFF と衝突する

二つの防御を同時に回避し、96 バイト手前へ書き込みが起きる

我々は修正直前のコミットを取得し、同じ条件で検査した

V. 検証結果｜標的｜ffmpeg脆弱性の検出実験

実際のコード。三つのファイルに散っている。



単体ではどれも まっとうなコード。危険なのは 3 ファイルの組み合わせ。
検出には『複数ファイルを結ぶ推論』が要る＝モデルの頭脳（読み歩き）。

V. 検証結果｜実験の読み方 ｜ ffmpeg脆弱性の検出実験

時間と精度は、同じ実行方式の中でだけ比較する。

A｜現実条件：何も与えない/汎用ヒントのみ。検出力を測る

B｜lens 強制：正解に効く観点を与える。能力の上限を測る

line 2349 / 2351 を、番兵衝突という原因付きで finding として提出した場合だけ HITしたと定義

検証の結果 1/5 — opus-4-8 のみ標的に到達。



ranker 次第（標的は 617 位）

惜しいモデルもあったが frontier だけが正解に到達 1/5

frontier だけが正解に到達 1/5

現実条件 deep 4 計測で 1/5

観測された落ち方 — 同じ 1 件でも四通り

※ 各段は別々の計測・分母は不均質・N=1・優劣比較ではない

context 不足	推論不足	非収束・未提出	応答非適合
原因を結ぶ情報が入力に無い	値域と予約値の衝突を組めない	thinking終わらず処理非収束	LLMの応答が壊れ成果が0

V. 検証結果 | ② | ffmpeg脆弱性の検出実験

試したモデルと、その落ち方。

モデル	所要	findings	標的	考察
claude-opus-4-8	11分	8	○ **提出**	line 2349 でsentinel collision (== 0xFFFF) に到達。h264dec.h 型宣言も指摘。検証段は未通過
gpt-oss:120b	10分	4	×	最速だが対象がh264dec.c に集中。compile flag / assert / 未初期化。標的なし
qwen3.6-独自FT	128分	2	×	first_mb_addr shift wrap (1925行) + h264dec.c off-by-one。標的とは別筋
deepseek-v4-flash	86分	4	×	h264dec.c / h264_slice.c の memory_safety 4件。標的とは別筋
qwen3-235b	∞	—	×	thinkingが暴走して処理が未収束

opus-4.8 は標的を原因付きで提出した (line 2349, sentinel collision)

他の 4 モデルは標的行に到達せず。件数が多くても、標的とは別の欠陥

V. 検証結果｜運用の算術｜ffmpeg脆弱性の検出実験

3ファイル 10〜128分。絞らねば終わらない。

5 run の実測（旧 2 ファイル計測）を 4,980 ファイルへ線形外挿 — 単一スロット連続稼働換算

最速 gpt-oss:120b　7.5 分/file →	中央値　17.5 分/file →	最遅 DS4F　50 分/file →
622 時間 ≈ 26 日	**1,450 時間 ≈ 60 日**	**4,150 時間 ≈ 173 日**

① 順位を直す

自動化できる主要なレバー

617位 → 6位

- 母集団は変えず順番を早める
- LLM hunt より低コスト・前処理コストは発生

② 母集団を絞る

人手トリアージ

全ツリー → 差分

- 変更差分・サブシステムへ絞る
- 全件では617位でもh264関連に絞れば上位に

③ 観点を絞る

37 → 3〜8 観点

21h → 2〜5h

- 全網羅は 15〜21 時間
- 3〜8 観点に絞れば2〜5 時間の実用域

④ 時間で選ばない

時間 ≠ 到達

128分 × / 10分 ×

- deep は最速 10分も最遅 128分も不到達（0/4）
- 時間を伸ばしても届かない

時間を延ばすだけでは到達を保証できない。
対象範囲の限定と順位付けの改善で、計算時間を有効な場所へ配分する。

VI. 診断① 時間の壁｜コストの構造

なぜ時間がかかるのか — 速度でなく呼出回数の積。

ファイル数 × hunter 数 × 深さ x 再試行が呼出回数を増やす

各候補の後段に verifier・PoC・レポート処理が続く

長い context、収束しない thinking、timeout が run を引き延ばす

見積りは『秒 / token』でなく、工程別の呼出回数から作る

VI. 診断① 時間の壁｜並列化

並列は効く? — 効くが実行環境次第で context が分割される。

hunt は max_parallel(既定8)で並列化される
— 8 は同時実行数で、検査ファイル数ではない

【実測】同時4並列　-np 4 -cb
1件と比較して 41.1 秒 → 10.7 秒、 合計スループット 11 → 38 tok/s

【実測】並列 8 まで全リクエスト成功。abort・クラッシュ 0 件

3.8×
高速化

llama.cpp –np の既知の問題として context 長がスロット数(並列数)で分割されるため、
Clearwingの処理の途中で入出力トークンが溢れて処理不能となる問題が見つかっている。
vLLM(PagedAttention) ではこの問題は発生しない(context がスロットで分割されない)

VII. 診断② 到達の壁｜網羅率

なぜ届かない ― 『全件対象』は建前、完走しない。

『候補32・探索8』の選抜上限は無い ― 全 4,980 ファイルが hunt 対象

『8』は同時実行数(max_parallel)。検査ファイル数の上限ではない

『600』は LLM 再採点の枠。検査の足切りではない

上限なし

選抜枠は存在しない
全件が hunt 対象

VII. 診断② 到達の壁｜網羅率

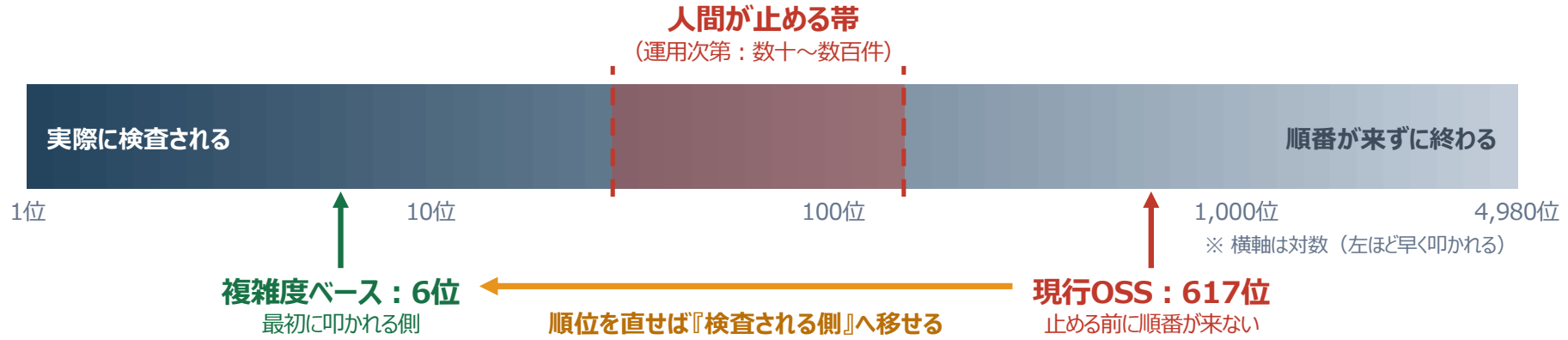
検査の順位の重要性を理解する。

全部検査するは建前、hunt は順位の高い順に叩く

⇒ 実際に検査されるのは、処理が完結した or 人が停止した際に処理できた分だけ。

コード上の自動停止は限定的：予算(ローカルLLMでは無効)、Tier概念はあるものの件数上限・全体 timeout は存在しない。実測として1ファイルで数時間を要する ⇒ 全件完走は非現実的。

ffmpeg の例: 全 4,980 ファイルが対象（OSS に選抜上限は無い）



VII. 診断② 到達の壁｜網羅率

tier A/B/C はあるが、『何件で止める』制御ではない。

- tier は priority から A(≥3.0)/B(≥2.0)/C(<2.0) を割り当てる
- 決めるのは順序(A→B→C)・試行回数(3/2/1回)・深さ(band)だけ
- つまり tier は『どの順で・何回・どれだけ深く』の配分機構
- **＋採点は各ファイルの先頭 16KB だけを読む(全体行数ではない)**

全部は見てない

VII. 診断② 到達の壁｜ランキング算法

現行rankerは危険な順に並べたものではない。

Priority = 0.5 × surface + 0.2 × influence + 0.3 × reachability

▼ 3 段だけ生きる
危険API文字列/タグ

▼ 全件 1 に潰れる
imports_by が無効化（>2000件）

▼ 床値 3 に潰れる
callgraph/taint が無効化

ffmpegの例：4,980 ファイルが、たった 8 段階にしか分かれない

残り 7 種類のスコアに散らばる 1,995 件（40%）	3.00 点の同点集団 2,985 件（60%）

▲ 標的は 3.00 点の同点集団内で 189 番目
同点内の並びは列挙順＝機械的に決まる。危険度の序列ではない

現行rankerのアルゴリズムは、粗い 3 段のふるい＋走査順で並べただけ。
C/C++ では SourceAnalyzer 由来の static hint が実質 0 — メモリ安全性検査なのに C だけ効きにくい

VII. 診断② 到達の壁｜運用指標

大きいほど時間で薄まる。だから順位が全て。

- 見る本数に固定枠は無い ― 全件対象だが、完走に非現実的な時間がかかる
- 大きいリポジトリほど、同じ時間で上位のどこまで到達できるかが落ちる
- 順位の解像度も粗く、surface 3段で大量に同点化し、同点内は列挙順

順位 = 到達率

止める前にどこまで順番が来るか？規模が大きいほど上位しか届かない。

VIII. 処方｜改善方向

順位を直せば、標的は『早く』検査対象になる。

現行ranker: h264_slice.c は 617 位 — 止める前に順番が来ない

改良版ranker: 分岐21位・ポインタ演算5位・全信号6位・構造24位

同じ全ファイル対象・同じ停止条件でも、標的が『先に』叩かれる

617→6

標的の順位(全信号等重み)
止める前に順番が来る位置へ

別途 30+ の未使用CVEでも複雑度集約は現行OSSを上回った

VIII. 処方｜改良案

12 の順位付け手法を、16 の実 CVE で採点した。

ranker 手法	上位8件	上位32件	上位200件	中央上位%	対現行 勝/分/敗
現行OSS（名前一致）	3	4	7	28%	—
複雑度集約（有望改良案）	5	7	8	9%	12/0/4
現行+構造 集約（探索的）	4	6	11	9%	14/0/2
全信号 集約	5	6	9	10%	12/0/4
セキュリティ集約	4	6	9	15%	12/0/4
分岐数	3	6	10	8%	12/0/4
行数	3	6	11	10%	13/0/3
ループ数	4	7	9	14%	9/0/7
添字演算	1	5	8	21%	8/1/7
構造ノード	4	6	10	10%	12/1/3
危険API	2	7	9	11%	8/0/8
攻撃面タグ	1	4	6	56%	6/0/10

上位K件の的中数（/16）と中央上位%。小さいKで多く当たり、中央%が小さいほど良い。

VIII. 処方｜改良案

16 の実 CVE で、脆弱ファイルの『検査される順』を並べた。

CVE ごとの脆弱ファイルの順位（全 12 手法）— 上段=順位 / 下段=上位%、小さいほど早く検査される

CVE / 対象	jq 2026-32316 C・全55件	libyang 2026-41401 C・全236件	OpenSC 2026-40528 C・全365件	wolfSSL 2026-6679 C・全1,313件	FreeRDP 2026-40033 C・全1,759件	OpenSSL 2026-34182 C・全2,580件	OpenSSL 2026-42768 C・全2,581件	OpenSSL 2026-45445 C・全2,581件	OpenSSL 2026-28386 C+asm・2,578件	Gitea 2026-27775 Go・全3,193件	jsrsasign 2026-4600 JS・全101件	junrar 2026-28208 Java・全118件	BouncyCastle 2026-5588 Java・全6,851件	TYPO3 2026-47345 PHP・全30件	Firecracker 2026-5747 Rust・全517件	gitoxide 2026-40034 Rust・全1,422件
現行OSS（名前一致）	32 58%	4 2%	88 24%	348 27%	334 19%	1,206 47%	922 36%	126 5%	961 37%	488 15%	8 8%	50 42%	3,396 50%	7 23%	322 62%	427 30%
複雑度集約（提案）	4 7%	8 3%	8 2%	421 32%	26 1%	1,035 40%	275 11%	856 33%	705 27%	250 8%	53 52%	13 11%	279 4%	2 7%	6 1%	420 30%
現行+構造 集約	5 9%	2 1%	2 1%	325 25%	44 3%	624 24%	107 4%	128 5%	543 21%	93 3%	30 30%	23 19%	875 13%	3 10%	53 10%	241 17%
全信号 集約	4 7%	3 1%	8 2%	616 47%	37 2%	752 29%	248 10%	489 19%	740 29%	146 5%	44 44%	17 14%	597 9%	4 13%	6 1%	442 31%
セキュリティ集約	4 7%	3 1%	12 3%	909 69%	65 4%	428 17%	474 18%	260 10%	586 23%	112 4%	49 49%	16 14%	1,119 16%	6 20%	3 1%	611 43%
分岐数	6 11%	3 1%	12 3%	645 49%	61 3%	394 15%	47 2%	232 9%	909 35%	67 2%	45 45%	15 13%	473 7%	2 7%	9 2%	444 31%
行数	5 9%	8 3%	12 3%	100 8%	45 3%	1,057 41%	197 8%	576 22%	274 11%	131 4%	30 30%	32 27%	627 9%	4 13%	53 10%	303 21%
ループ数	3 5%	3 1%	4 1%	611 47%	30 2%	1,220 47%	143 6%	2,153 83%	1,086 42%	1,210 38%	78 77%	21 18%	468 7%	4 13%	22 4%	1,122 79%
添字演算	9 16%	52 22%	25 7%	629 48%	97 6%	1,176 46%	1,488 58%	2,154 83%	382 15%	733 23%	75 74%	20 17%	577 8%	7 23%	19 4%	1,061 75%
構造ノード	2 4%	4 2%	6 2%	778 59%	45 3%	644 25%	89 3%	458 18%	845 33%	201 6%	49 49%	28 24%	749 11%	4 13%	19 4%	152 11%
危険API	2 4%	28 12%	24 7%	562 43%	116 7%	220 9%	1,215 47%	264 10%	569 22%	20 1%	89 88%	11 9%	3,996 58%	26 87%	1 0%	1,042 73%
攻撃面タグ	31 56%	8 3%	241 66%	746 57%	1,041 59%	350 14%	841 33%	1,475 57%	51 2%	2,088 65%	37 37%	22 19%	4,954 72%	26 87%	289 56%	1,014 71%

≤8位　≤32位　≤200位　≤600位　>600位

VIII. 処方｜改良案

現行が見ていないもの — 大きさ・複雑さ・履歴。

見ていない情報	現行 OSS の扱い	その結果（実測）	改良で何をしたか
大きさ	行数は先頭 16KB 分しか数えない	2,845 行を 457 行と採点	全長で行数を数え、順位付けに使う
複雑さ	分岐・添字演算を数えていない	複雑度シグナル 0 個	段2＝分岐数・添字演算・行数の順位平均
履歴	変更履歴を使っていない	標的 109 回変更／選ばれた側 20 回	今回は不採用（単独では 66 位止まり。差は実測したが順位付けには未組込）

改良効果(実測)

- **型宣言 h264dec.h を拾えた** 段3 companion＝include するヘッダを 3 件まで引き込むことに成功
- **617 位 → 6 位** 標的 h264_slice.c の優先順位の引き上げに成功 ※p32参照

どれも汎用で、答えを知らなくても使える情報。数えるだけで 617 位 → 6 位

『ファイル名に parse が入っているか』は、もう使わない

VIII. 処方｜改良案

賢そうな案ほど、測ると外れた。

FFmpegの実CVE由来267ファイルで検証。到達解析・被参照数・入口近接・密度・現行併用は、いずれも基準を下回った。

試した案	期待した理屈	実測	なぜ外れたか
× 入口から辿れないファイルを刈る	到達解析で絞れば精度が上がるはず	21→19（上位32件）	脆弱ファイルの 70% も一緒に落ちた
× 被参照数が多いファイルを加点	よく使われる＝影響が大きいはず	上位が log.c / mem.c で埋まる	「参照が多い＝危険」ではない
× 入口に近いファイルを加点	攻撃者入力に近いほど危険なはず	標的が 2 位 → 44 位 に後退	入口周辺(cmdutils.c 等)が上位を占める
× 複雑度を密度(分岐/行数)にする	大きいだけのファイルを排除できるはず	21→15（上位32件）	絶対数の方が良い。大きさ自体が情報
× 現行スコアを併用する	既存の知見も混ぜた方が良いはず	21→17（上位32件）	粗い同点スコアが順位を濁らせる

残ったのは「大きさ・分岐・添字を数えるだけ」— 単純な複雑度が、賢そうな理屈より強かった。

IX. 導入判断と撤退基準

導入の前に — 失う時間を知り、撤退基準を決めておく。

導入前に決めたい撤退基準（go / no-go）

再現率: pilot の go/no-go 閾値を事前に定める（例: 一度 HIT した条件で 50% 以上）

finding単価: GPU 時間・人手確認時間・完走率を記録する

完走性: timeout 率・再試行回数・verified 率に閾値を置く

拡大方法: pilot が満たすまで全ツリーへ広げない

クロージング

前提 **脅威側 — 前提が変わった**

ミュトス級の知能が、出揃いつつある。攻撃側も同等の知能を前提にできる — その前提に立つ。

実測 **計測した 4 モデルのうち 3 つはオープンウェイト — 高い知能はローカルでも入手・実行できる**

だから、防御側の強化は待ったなし。

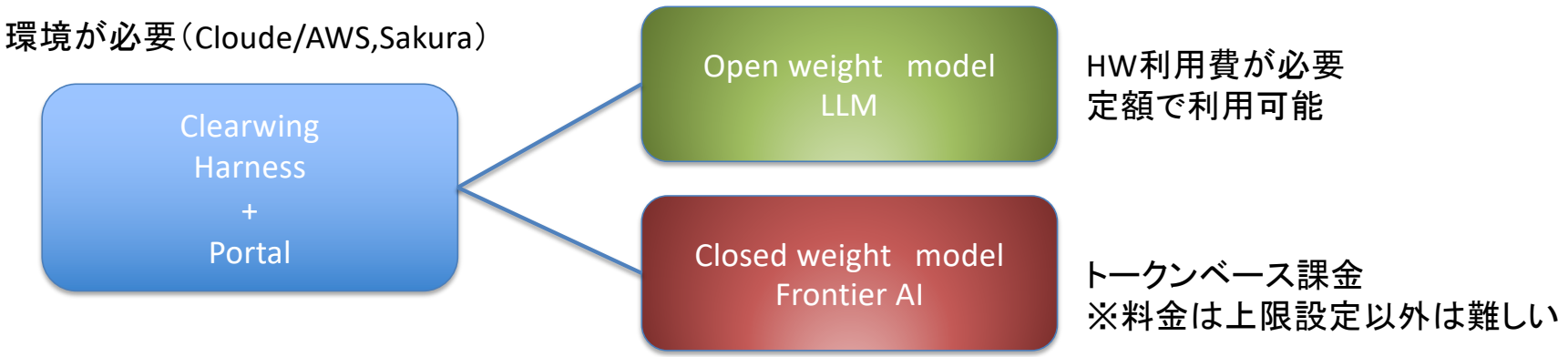
実測 **オープンウェイトも賢くなった — が、モデル肥大化で気軽に動かせる状態ではなくなった。**

ハード: DGX Spark ×2（GB10）— 1 node 121GB unified memory・合計 約242GB で考えてみる

GLM-5.2	Kimi K3	DeepSeek-V4-Flash	本編の実測	線形外挿
217GB・8 tok/s	865GB → 8 node	162GB / 2 node	10〜128 分	26〜173 日
1bit まで削って 2 node 動くが標的 MISS	Q2_K でも 8 node 相当 2 node では対象外	各 node 約80GB 常駐 初期ロード 10分19秒	3 ファイルの検査で (4 run の実測)	全ツリー 4,980 files 単一スロット・旧2ファイル基準

潤沢なスペックが無ければ、現実的な検査は成立しない — 本編で示したとおり。

ClearwingをJNSA会員向けに実践的環境として利用できるかの考察



■JNS会員向けに利用環境を検討する場合に、以下の形態が考えられる
・動作環境のDocker Imageの提供
・LLMの調達＋配置 or Frontier AIで課金する（IPAが資金提供するなど、利用料金の低減や補助も課題）
・利用料金整理とモデル化、オープンソースプロジェクト発展への寄与貢献などの役割整理も必要

■利用者の制限
・悪意ある利用者の排除が課題
・自身の所有するソースコード、システムだけをターゲットにするべきだが、どう管理するのか
・脆弱性情報管理の観点から、利用者の期間占有が望ましい

AGESTが提案するAIによるゼロデイ脆弱性ハンティング～自動修復ソリューション

2026/3/19日　米国のセキュリティ企業であるクラウドストライク社（CrowdStrike）が2026年度版グローバル脅威レポートを発表しました。
そのなかのいくつかのトピックスは非常に衝撃的であり、また予てよりの懸念が現実化したことを示しています。

ブレイクアウトタイムの最速記録

AIにより攻撃が加速し、サイバー犯罪 (eCrime) の平均ブレイクアウトタイムは29分に短縮されました。これは2024年と比べて65%の高速化となり、過去最速のブレイクアウトはわずか27秒で完了しました。ある侵入では初期アクセス取得から4分以内にデータ流出が開始されました。

ゼロデイとクラウドエクスプロイトの拡大

攻撃者は、初期アクセス、リモートコード実行、権限昇格のためにゼロデイを武器化しており、脆弱性の42%は公開前に悪用されました。クラウドを意識した侵入は全体で37%増加し、中でもインテリジェンス収集のためにクラウド環境を標的とする、国家主導型脅威アクターからの侵入は266%増加しました。

AIは新たな攻撃対象領域として、プロンプトは新たなマルウェアとして台頭

攻撃者は、90を超える組織で正規の生成AIツールに悪意のあるプロンプトを注入し、認証情報や暗号資産の窃取のためのコマンド生成に悪用しました。また、AI開発プラットフォームの脆弱性を悪用し、永続的なアクセスを確立してランサムウェアを展開したほか、信頼できるサービスになりすました悪意のあるAIサーバーを立ち上げ、機密データを傍受しました。

最短27秒でラテラルムーブメントが起こるなか、人間が知覚し検討する時間的余裕はなくなったと言える　⇒　攻撃AI vs 防御AI

AGESTが提案するAIによるゼロデイ脆弱性ハンティング〜自動修復ソリューション

ZERO SHIELD

AGEST ZERO SHIELDは、株式会社AGESTが商用リリースする
Clearwing を原点とする最新のAIセキュリティソリューションです

ZERO TEST

そしてパッチ急増で増加する回帰テスト工数削減の切り札となるのが
AIによる高速A/B回帰テストのAGEST ZERO TESTです

B. AGEST ZERO TEST：全自動A/B-AIテストプラットフォーム

1. テスト設計から課題管理までAIで一気通貫

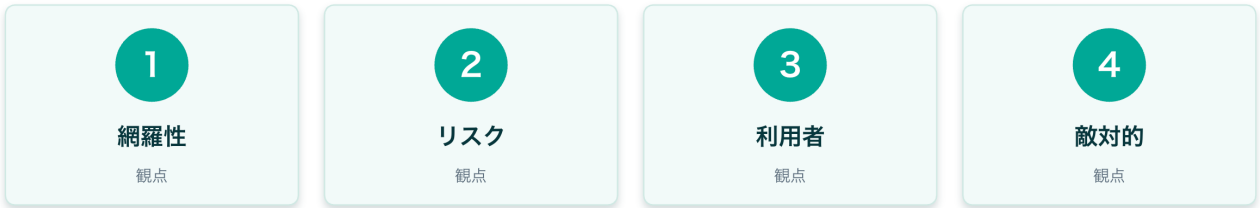


1つのツールの中で、設計→レビュー→実行→比較→課題化まで完結。ツール間の受け渡しが不要

約90%

工数削減

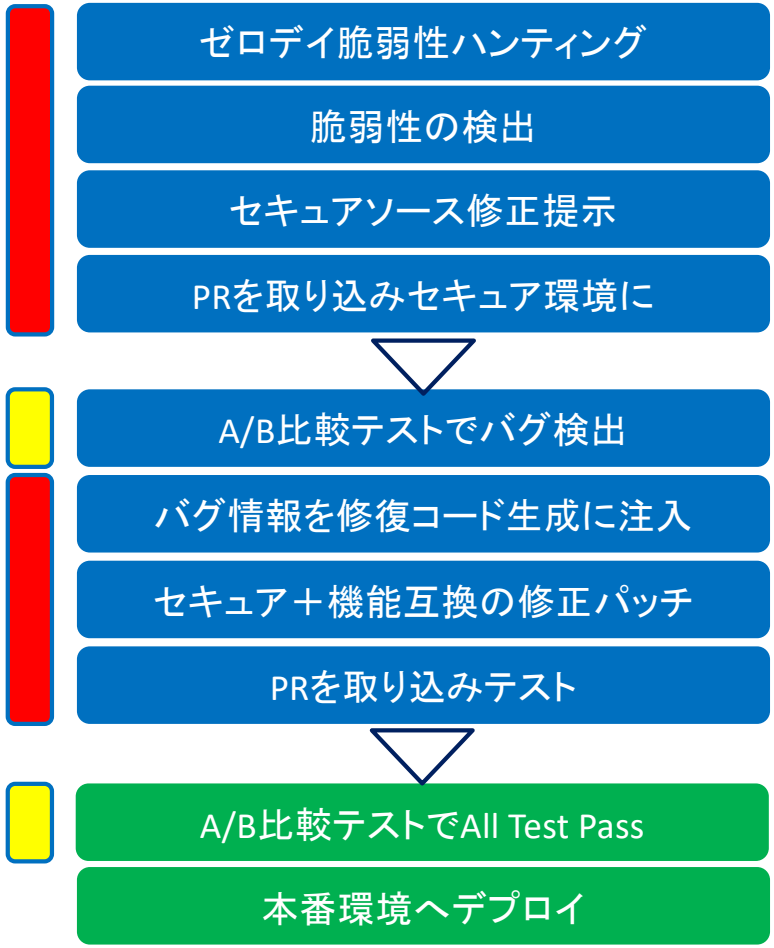
2. レビューは、AIによる“多層”で品質を担保



最大5人のAIレビューエージェントが、異なる観点で独立にレビュー → 合議で統合し単一AIの見落としを相互に補う

3. 回帰テスト専門なので仕様書がなくても即実行可能

A. ゼロデイ脆弱性検出～修復パッチ適用までの流れ



AGEST ZERO SHIELDの特徴

・オンプレミスシステムで完全動作し、外部にソースコード送信をしない
・トークン課金ではない（無制限ライセンス購入）

システムの問題を見つけるだけでなく、**修復とテストを一貫して実施する**ことで、常にセキュアな状態を維持することが可能になります

AGEST ZERO SHIELD

AGEST ZERO TEST

AIによる高速なA/B自動比較テストにより、パッチ適用前後のシステム影響を浮き彫りにし、システム機能を損なわない改修が実現できます

２つのプロダクトにより、システムを安全に保ちます