# The Mathematics of Project UNDERTOW

## Defensive Attack Surface Intelligence & Binary Analysis

Mark Konstantinov

April 2026

**Abstract**

Project UNDERTOW is a defensive security research tool engineered to narrow and map potentially valuable targets for manual reverse engineering through deterministic binary introspection and statistical analysis. Rather than relying on published vulnerability databases, the system ingests complete driver installer archives across software's version releases, performs deep static analysis of every Portable Executable (PE) binary, computes cross-version deltas capturing dozens of change dimensions, and mathematically ranks reverse-engineering targets for human researchers.

This document formally specifies the four mathematical subsystems that underpin UNDERTOW's reconnaissance pipeline:

1. A **Feature Vector Scoring Engine** that extracts a 27-dimensional named feature vector from each binary delta, applies percentile-capped population normalisation, computes a weighted linear combination base score, and amplifies it with a root-mean-square z-score anomaly measure.
2. A **True Patches Detection** mechanism that reuses the scoring architecture with a separate weight vector tuned to identify undisclosed code patches.
3. A **Generalised Extreme Studentised Deviate (ESD)** anomaly detector that compares each component to its own historical change pattern, using programmatic $t$-distribution approximations to avoid external statistical library dependencies.
4. An **Evolutionary Coupling Analysis** engine that computes Jaccard-based co-change matrices, detects coupling violations indicative of targeted patches, and infers transitive dependencies.

UNDERTOW, using these concepts, aims to transform binary analysis from a subjective manual process into a reproducible, data-driven pipeline, built on large amounts of data. The system's ceiling is explicitly defined: it ranks targets; actual vulnerability discovery occurs downstream in tools such as Ghidra. NVIDIA GeForce GPU drivers serve as the initial proving ground, with the architecture designed for multi-vendor expansion.

# Contents

**1 Introduction** **4**
1.1 System Context . . . 4
1.2 Aims of the Mathematics . . . 4
1.3 Applications . . . 5
1.4 Pipeline Context . . . 5

**2 Input: The Binary Delta** **6**
2.1 Delta Dimensions . . . 6
2.2 Why This Structure Matters . . . 7

**3 Deep PE Introspection Features** **8**
3.1 IMAGE_LOAD_CONFIG_DIRECTORY Parsing . . . 8
3.2 Rich Header Semantic Delta . . . 9
3.3 .rdata String Table Hash . . . 9
3.4 .pdata Function-Level Diff . . . 10

**4 The Feature Vector Scoring Engine** **10**
4.1 Design Rationale . . . 10
4.2 Notation . . . 11
4.3 Feature Vector Definition . . . 11
4.4 Population Normalisation . . . 12
4.4.1 Continuous Features . . . 12
4.4.2 Bounded and Binary Features . . . 13
4.5 Weighted Linear Combination (Base Score) . . . 13
4.6 Z-Score Anomaly Detection . . . 14
4.7 Final Research Priority Score . . . 15
4.8 Design Constraint: Purely Mathematical Scoring . . . 15

**5 Silent Fix Detection** **15**
5.1 Computation . . . 16
5.2 Threshold Classification . . . 16

**6 Generalised ESD Anomaly Detection** **16**
6.1 Rationale . . . 16
6.2 Formal Specification . . . 17
6.3 Applicability Threshold . . . 17
6.4 Programmatic Quantile Approximations . . . 18
6.4.1 Normal Quantile: Abramowitz & Stegun . . . 18
6.4.2 $t$-Distribution Correction: Cornish-Fisher Expansion . . . 18
6.5 Change Velocity . . . 18

**7 Evolutionary Coupling Analysis** **19**
7.1 Motivation . . . 19
7.2 Co-Change Matrix . . . 19
7.3 Coupling Violation Detection . . . 20
7.4 Transitive Coupling Inference . . . 20

**8 Threshold Parameters Summary** **21**

**9 Validation and Verification** **21**
9.1 Scoring Engine Validation . . . 21
9.2 ESD Test Validation . . . 22
9.3 Coupling Analysis Validation . . . 22
9.4 Ground Truth Confirmation (Future) . . . 22

**10 Implementation Notes** **23**
10.1 Technology Stack . . . 23
10.2 Performance Considerations . . . 23
10.3 Database Schema Highlights . . . 23

**11 Conclusion** **24**

**References** **25**

# 1 Introduction

## 1.1 System Context

Major technology vendors such as NVIDIA, Intel, AMD, and Broadcom ship hundreds of products containing thousands of components. Traditional vulnerability databases (the National Vulnerability Database, the CVE programme) document *known* vulnerabilities but structurally miss many categories of risk: abandoned legacy tools still bundled with current products, shared dependencies silently spanning product lines, and temporal maintenance patterns that reveal vendor negligence or urgency.

Project UNDERTOW addresses this gap through a **large-scale ingestion of software and performing binary analysis to establish patterns.** The system scrapes complete driver installers from vendor CDNs across different versions of the same software, extracts every PE binary, performs deep static fingerprinting, computes cross-version deltas, and ranks each changed binary by a custom priority scoring system. CVE and security bulletin data serve as *confirmatory evidence* rather than the primary intelligence driver.

UNDERTOW is a **reconnaissance instrument**. It ranks reverse-engineering targets—it does not discover vulnerabilities itself. Its ceiling is clearly defined: the output is a prioritised list of binaries with supporting statistical evidence explaining *why* a human researcher should spend manual effort on them. The actual vulnerability analysis occurs in tools such as Ghidra, IDA Pro, or WinDbg, etc.

The system has been developed and tested on consumer-grade hardware (16 GB RAM, 12 GB VRAM) and uses only freely available data sources. NVIDIA GeForce Game Ready Drivers serve as the initial proving ground; the architecture is vendor-agnostic by design.

## 1.2 Aims of the Mathematics

The mathematical subsystems in UNDERTOW were selected to answer three fundamental questions:

1. *How do we objectively rank the severity of undocumented binary changes across a large population without outlier distortion?*
   Answered by the **Feature Vector Scoring Engine** (Section 4).

2. *How do we detect when a single component breaks its own historical pattern of behaviour over time?*
   Answered by the **Generalised ESD Test** (Section 6).

3. *How do we infer hidden architectural dependencies and detect highly targeted, isolated code patches?*
   Answered by **Evolutionary Coupling Analysis** (Section 7).

## 1.3 Applications

The mathematics enable the following concrete applications in the security research workflow:

**Silent Patch Detection**
Identifying binaries that exhibit the mathematical fingerprint of a patch (exploit mitigation activation, security API changes, isolated modification of historically coupled components) despite having no published CVE.

**Recompilation Filtering**
Mathematically distinguishing routine compiler toolchain upgrades from genuine functional code changes, using Rich Header semantic analysis and `.rdata` string table stability as compiler-invariant anchors.

**Target Prioritisation**
Surfacing the most anomalous binary changes in a given ecosystem update by comparing each binary to both its peers (population anomaly) and its own history (temporal anomaly).

**Architectural Inference**
Discovering hidden dependencies between components that share no static import relationships but are revealed through co-change patterns over time.

## 1.4 Pipeline Context

To ground the mathematics in the system's data flow, the scoring engine operates at the following position in the pipeline:

$$\begin{aligned}
\text{CDN Download} &\longrightarrow \text{Extract installer via 7-Zip} \\
\text{Extract} &\longrightarrow \text{PE Fingerprint every binary} \\
\text{PE Fingerprint} &\longrightarrow \text{Store in } \texttt{component_registry} \\
\text{Cross-version pair} &\longrightarrow \text{Compute } \texttt{BinaryDelta} \\
\text{All deltas (one transition)} &\longrightarrow \text{Extract Feature Vectors} \\
\text{Feature Vectors} &\longrightarrow \text{Population Normalisation} \\
\text{Normalised matrix} &\longrightarrow \text{Weighted Scoring + Anomaly} \\
\text{Scored deltas} &\longrightarrow \text{Persist + Neo4j Sync} \\
\text{Full history} &\longrightarrow \text{ESD + Co-Change Analysis}
\end{aligned}$$

Every stage upstream of the scoring engine is deterministic: the same installer produces the same fingerprints, the same fingerprint pair produces the same delta, and the same delta produces the same feature vector. The only tuneable parameters are the weight vector and threshold constants, which are stored in the database and loaded at runtime.

# 2 Input: The Binary Delta

Before specifying the scoring approach, we define the input structure. The **BinaryDelta** is the computed difference between two consecutive versions of the same PE binary component. It captures every measurable change dimension extracted during static analysis.

## 2.1 Delta Dimensions

A single `BinaryDelta` record contains the following categories of change measurements:

**Hash and Size Changes**

Whether the SHA-256 hash changed (indicating any modification at all), the TLSH fuzzy hash distance of the `.text` section (a locality-sensitive measure of code similarity), absolute code size delta derived from `SizeOfCode` in the NT optional header, and function count delta derived from the `.pdata` exception directory. The entry point RVA (`AddressOfEntryPoint`) is also diffed: a changed entry point between versions without a corresponding structural overhaul is a strong anomaly signal.

**Import Surface Changes**

The sets of added and removed imported functions, partitioned into security-relevant APIs (e.g., `CryptProtectMemory`, `BCryptEncrypt`), dangerous-capability APIs (e.g., `CreateRemoteThread`, `WriteProcessMemory`), vendor-specific imports, and Windows system imports. Jaccard similarity coefficients quantify the magnitude of import set divergence across both the Windows and vendor subsets. All four import hash variants are compared: `windows_imphash`, `vendor_imphash`, `undertow_imphash` (UNDERTOW's canonical full-namespace hash), and `security_api_hash`.

**Delay-Load Import Surface Changes**

The sets of added and removed delay-loaded DLL names. Delay-loaded imports are resolved at runtime rather than load time; new delay-loaded DLLs can indicate architectural changes and expanded attack surface for DLL planting or hijacking.

**Export Surface Changes**

Added and removed exported function names, export ordinal changes (functions renamed at a fixed ordinal, ordinals shifted, or new functions added at a given ordinal position), indicating expansion, contraction, or interface restructuring of the binary's public surface.

**Section-Level Changes**

Per-section entropy deltas (the `.text` section entropy shift is particularly informative), per-section raw size deltas, the count of modified sections, and overlay size delta. The overlay transition boolean (a binary that *gained* appended data where none existed before, or *lost* it) is tracked separately from the size delta, as the transition itself is more semantically meaningful than the magnitude.

**Exploit Mitigation Transitions**
State transitions in the `IMAGE_LOAD_CONFIG_DIRECTORY`: Control Flow Guard (CFG) activation or deactivation, eXtended Flow Guard (XFG) activation or deactivation, stack cookie (buffer overflow canary) addition or removal, retpoline (Spectre v2 mitigation) addition or removal, changes in the CFG-protected function count, and growth in the raw load config structure size (which indicates that new security fields were added to the binary's load configuration by a newer toolchain).

**Compiler Forensics**
Rich Header semantic delta detecting whether all changes are attributable to a compiler toolchain upgrade (all `ProdID` entries identical, only `MinorCV` values changed). The `.rdata` string table hash provides a compiler-invariant anchor: if code changed but strings did not, the change is likely a recompilation artefact.

**Signature and Authenticity Changes**
Transitions in the Authenticode signature posture: whether the binary gained or lost a valid signature (`IsSigned` transition), whether the signing certificate common name changed between versions (indicating a certificate rotation or build pipeline change), and changes in the full certificate chain (leaf, intermediate, and root).

**Obfuscation Indicator Changes**
The *set* of obfuscation heuristic signals that were newly introduced or resolved between versions. A binary that gains obfuscation indicators (e.g., abnormal section names, high entropy across multiple sections, missing imports) between successive releases without any published changelog entry is a meaningful anomaly.

**Metadata Changes**
PDB GUID changes (recompilation indicator), security API hash changes, vendor and Windows import hash changes, and version resource changes (`FileVersion` and `ProductVersion` stored side-by-side for both the old and new binary to allow downstream correlation with vendor release notes).

## 2.2 Why This Structure Matters

Each dimension contributes differently to the scoring engine's assessment. A binary that gained CFG protection, added security APIs, changed its entry point, and changed in isolation from its historically coupled partners exhibits a very different profile than a binary that was merely recompiled with a newer toolchain. The feature vector (Section 4.3) extracts the numerically salient dimensions from this rich delta structure; the scoring engine (Section 4) then weighs, normalises, and ranks them.

# 3 Deep PE Introspection Features

Several of the 27 scoring features derive from PE structures that require explanation beyond standard import/export analysis. This section describes the four specialised introspection techniques and their rationale.

## 3.1 IMAGE_LOAD_CONFIG_DIRECTORY Parsing

The `IMAGE_LOAD_CONFIG_DIRECTORY` is pointed to by `DataDirectory[10]` in the PE optional header. It contains *definitive* (not heuristic) metadata about exploit mitigations compiled into the binary. The relevant fields and their bit positions in `GuardFlags` are:

| Bit | Constant | Meaning |
|---|---|---|
| 8 (`0x00000100`) | `CF_INSTRUMENTED` | CFG instrumentation present |
| 10 (`0x00000400`) | `CF_FUNCTION_TABLE_PRESENT` | CFG function table present |
| 19 (`0x00080000`) | `XFG_ENABLED` | XFG type-based validation |
| 20 (`0x00100000`) | `RETPOLINE_PRESENT` | Spectre v2 mitigation |

CFG is considered *active* when both bit 8 and bit 10 are set. The `GuardCFFunctionTable` then contains the RVA of every valid indirect call target; the Windows kernel validates indirect calls against this table at runtime, preventing ROP/JOP attacks. The `SecurityCookie` field holds the RVA of the stack canary value; a non-zero value indicates stack buffer overflow protection.

In addition to the `GuardFlags` bit fields, the raw `Size` field of the `IMAGE_LOAD_CONFIG_DIRECTORY` structure itself is tracked. The size encodes the toolchain era: each successive generation of Visual Studio and Windows SDK extended the structure with new security fields. A growth in `Size` between versions therefore indicates that the binary was rebuilt with a toolchain that emits a more complete security configuration block—an independent signal of deliberate hardening beyond the individual bit flags.

**Delta semantics.** The delta fields capture *state transitions* between consecutive driver versions:

- **CFGActivated**: component *gained* CFG—deliberate security hardening.
- **CFGDeactivated**: component *lost* CFG—potential security regression.
- **XFGActivated**: component gained XFG—advanced type-based hardening.
- **XFGDeactivated**: component *lost* XFG—potential security regression.
- **StackCookieAdded / StackCookieRemoved**: stack canary protection added or removed.
- **RetpolineAdded**: Spectre v2 mitigation enabled.
- **RetpolineRemoved**: Spectre v2 mitigation disabled—a deliberate regression signal.
- **GuardCFDelta**: change in CFG-protected function count (expansion indicates new protected code paths).

- **LoadConfigSizeGrew**: the raw structure size increased, indicating a newer toolchain era.

These transitions produce some of the highest-weighted features in the scoring engine (CFG and XFG activation each carry weight 3.0), because enabling or disabling exploit mitigations is among the strongest signals of a security-relevant change. Deactivation transitions (**CFGDeactivated**, **XFGDeactivated**, **RetpolineRemoved**) carry positive weights as security regression signals rather than negative weights—they are interesting to a researcher precisely because they are harmful, not because they should demote the binary.

## 3.2 Rich Header Semantic Delta

The PE Rich Header is an array of `PRODITEM` entries written by the Microsoft linker, each containing:

- `ProdID`: which Visual Studio tool produced the object file,
- `MinorCV`: which version of that tool,
- `Count`: how many object files it produced.

The delta computation builds maps from `ProdID` $\rightarrow$ `(MinorCV, Count)` for both the old and new version, then identifies:

- **Added ProdIDs**: new tools introduced in the build pipeline.
- **Removed ProdIDs**: tools dropped from the pipeline.
- **Version-bumped ProdIDs**: same tool, different version.

The boolean `IsCompilerUpgradeOnly` is set to `true` when all ProdIDs remain identical and only `MinorCV` values changed. This is a toolchain upgrade—the source was recompiled with a newer Visual Studio, but no functional code was modified. This distinction is critical: compiler upgrades produce real binary changes (different code generation, different hashes) that would otherwise rank highly in the scoring engine. The Rich Header delta provides the negative-weight feature (`rich_header_compiler_upgrade`, weight $= -3.0$) that actively demotes these false positives.

## 3.3 .rdata String Table Hash

The `.rdata` section contains string literals, vtable pointers, and constant data. String literals are **compiler-invariant**: if source code contains `printf("error: invalid buffer")`, that string appears in `.rdata` regardless of compiler version or optimisation level.

This invariance produces two derived correlation features:

`TextChangedWithoutRdata` Code (`.text`) changed meaningfully (TLSH distance $> 20$) but `.rdata` hash is identical. This suggests the change is a recompilation artefact—the actual program logic is unchanged. Assigned negative weight ($-1.5$).

`RdataChangedWithoutText` New strings appeared but no meaningful code change occurred. This suggests a configuration or resource update rather than a functional

patch. Assigned low positive weight (0.4).

When combined with the Rich Header delta, these two features form a high-confidence **recompilation noise filter**: if the Rich Header shows a compiler upgrade *and* `.rdata` is stable, the binary change is almost certainly maintenance noise and should be deprioritised.

### 3.4 .pdata Function-Level Diff

The `.pdata` section is an array of `RUNTIME_FUNCTION` structures (on x86-64), each containing `BeginAddress` and `EndAddress` RVAs of a function. By diffing the `BeginAddress` sets between versions, UNDERTOW identifies the specific RVAs of functions that were added or removed. When a researcher opens the binary in Ghidra, they can navigate directly to these addresses instead of manually searching through potentially hundreds of functions.

**Gated computation.** This analysis is computationally expensive (potentially thousands of RVAs per binary). It is only performed for binaries that pass a significance threshold: the binary must have `sha256_changed = true` *and* its feature vector score must place it above the population median. This avoids storing thousands of RVAs for uninteresting binaries.

The function count delta (`function_count_delta_abs`) enters the scoring engine as a continuous feature with weight 0.9.

## 4 The Feature Vector Scoring Engine

This section provides the complete mathematical specification of the scoring engine that replaced the naive boolean-checklist heuristics of UNDERTOW v1. The architecture enforces a strict separation: the Go source code defines the *algorithm* (how features combine); the PostgreSQL database stores the *coefficients* (how much each feature matters). This ensures that weight tuning never requires recompilation.

### 4.1 Design Rationale

Older heuristics relied on rigid boolean rules ("if CFG activated, add 10 points; if size increased, add 5 points"). This approach fails at scale for three reasons:

1. It treats all changes as absolute rather than relative to the population of changes in the same update.

2. A single extreme outlier (e.g., a binary whose size increased by 50 MB) compresses the normalised scores of the remaining population to near-zero under standard min-max normalisation.

3. It provides no mechanism to detect binaries that are anomalous *relative to their peers*—a binary with modest absolute scores that is nonetheless the most unusual change in the update.

The replacement architecture uses percentile-capped min-max normalisation to solve problem (2), population-relative z-score anomaly detection to solve problem (3), and a database-stored weight vector to replace the hardcoded constants of problem (1).

## 4.2 Notation

| | |
|---|---|
| $N$ | Number of changed binaries in a single version transition |
| $K = 27$ | Number of features in the feature vector |
| $\mathbf{F}$ | The $N \times K$ raw feature matrix; $F_{ik}$ is feature $k$ for binary $i$ |
| $\hat{\mathbf{F}}$ | The $N \times K$ normalised feature matrix |
| $\mathbf{W} \in \mathbb{R}^K$ | Weight vector loaded from `scoring_weights` table |
| $\mu_k, \sigma_k$ | Mean and standard deviation of column $k$ across all $N$ rows |
| $P_5^{(k)}, P_{95}^{(k)}$ | 5th and 95th percentiles of column $k$ |
| $\alpha_{\text{anomaly}}$ | Anomaly influence parameter (default: 0.3) |
| $C_{\text{anomaly}}$ | Anomaly cap (default: 3.0) |

## 4.3 Feature Vector Definition

Each `BinaryDelta` produces a 27-dimensional *named* feature vector. Features are classified into three types that determine their normalisation treatment:

- **Continuous**: raw numeric values with unbounded range; require population normalisation.
- **Bounded**: already constrained to $[0, 1]$; passed through unchanged.
- **Binary**: boolean indicators converted to $\{0, 1\}$; passed through unchanged.

| # | Feature Name | Type | Extraction | $W_k$ |
|---|---|---|---|---|
| 1 | `tlsh_distance` | cont. | $\max(d.\text{TLSHDist}, 0)$ | 1.0 |
| 2 | `code_size_delta_abs` | cont. | $\lvert d.\text{CodeSizeDelta}\rvert$ | 0.8 |
| 3 | `function_count_delta_abs` | cont. | $\lvert d.\text{FunctionCountDelta}\rvert$ | 0.9 |
| 4 | `security_api_added_count` | cont. | $\lvert\text{SecurityAPIAdded}\rvert$ | 2.5 |
| 5 | `security_api_removed_count` | cont. | $\lvert\text{SecurityAPIRemoved}\rvert$ | 2.0 |
| 6 | `dangerous_api_added_count` | cont. | $\lvert\text{DangerousAPIAdded}\rvert$ | 1.8 |
| 7 | `export_added_count` | cont. | $\lvert\text{AddedExports}\rvert$ | 1.0 |
| 8 | `export_removed_count` | cont. | $\lvert\text{RemovedExports}\rvert$ | 1.0 |
| 9 | `text_entropy_delta_abs` | cont. | $\lvert\Delta H_{\texttt{.text}}\rvert$ | 0.6 |
| 10 | `changed_section_count` | cont. | $\lvert\text{ChangedSections}\rvert$ | 0.5 |
| 11 | `overlay_size_delta_abs` | cont. | $\lvert d.\text{OverlaySizeDelta}\rvert$ | 0.3 |
| 12 | `guard_cf_delta_abs` | cont. | $\lvert d.\text{GuardCFDelta}\rvert$ | 1.2 |

| # | Feature Name | Type | Extraction | $W_k$ |
|---|---|---|---|---|
| 13 | `vendor_import_distance` | bounded | $1.0 - J_{\text{vendor}}$ | 1.5 |
| 14 | `windows_import_distance` | bounded | $1.0 - J_{\text{windows}}$ | 0.7 |
| 15 | `pdb_guid_changed` | binary | bool $\to$ float | 0.3 |
| 16 | `security_api_hash_changed` | binary | bool $\to$ float | 2.0 |
| 17 | `windows_imphash_changed` | binary | bool $\to$ float | 0.5 |
| 18 | `vendor_imphash_changed` | binary | bool $\to$ float | 1.0 |
| 19 | `cfg_activated` | binary | CFG off $\to$ on | 3.0 |
| 20 | `xfg_activated` | binary | XFG off $\to$ on | 3.0 |
| 21 | `stack_cookie_added` | binary | canary added | 2.5 |
| 22 | `retpoline_added` | binary | retpoline added | 2.0 |
| 23 | `text_changed_without_rdata` | binary | code changed, strings stable | $-1.5$ |
| 24 | `rich_header_compiler_upgrade` | binary | compiler-only change | $-3.0$ |
| 25 | `cfg_deactivated` | binary | CFG on $\to$ off | 1.5 |
| 26 | `stack_cookie_removed` | binary | canary removed | 1.5 |
| 27 | `rdata_changed_without_text` | binary | new strings, no code change | 0.4 |

Table 1: Complete feature vector registry with seed weights for the `research_priority` score type. Negative weights actively demote features associated with maintenance noise. The weight column ($W_k$) shows domain-knowledge initial estimates stored in the `scoring_weights` PostgreSQL table. The 27-dimensional vector represents the current scoring implementation; several additional delta dimensions described in Section 2 (entry point changes, signature transitions, delay-load surface changes, obfuscation indicator transitions, `xfg_deactivated`, `retpoline_removed`, `undertow_imphash_changed`, and `load_config_size_grew`) are candidates for future feature vector expansion and weight assignment during the next tuning cycle.

The bounded features (indices 13–14) are derived from Jaccard similarity: $J = |A \cap B|/|A \cup B|$, where $A$ and $B$ are the import sets of the old and new versions. The distance $1 - J$ measures *divergence*: a value of 0 means identical import sets; a value of 1 means completely disjoint.

## 4.4 Population Normalisation

Normalisation is applied *per-column* across the population of $N$ changed binaries in the **same version transition**. This is the key design choice: features are scaled relative to what changed in *this specific update*, not in absolute terms.

### 4.4.1 Continuous Features

For features with type `continuous`, percentile-capped min-max normalisation is applied:

$$\hat{F}_{ik} = \frac{\text{clamp}(F_{ik},\ P_5^{(k)},\ P_{95}^{(k)}) - P_5^{(k)}}{P_{95}^{(k)} - P_5^{(k)}} \tag{1}$$

where $\text{clamp}(x, a, b) = \max(a, \min(x, b))$.

If $P_{95}^{(k)} = P_5^{(k)}$ (zero interpercentile range), then $\hat{F}_{ik} = 0$ for all $i$—the feature carries no discriminative information in this population.

**Why percentile capping?** Standard min-max normalisation maps the observed minimum to 0 and maximum to 1. A single extreme outlier (e.g., one binary with TLSH distance 5000 when all others are below 100) compresses the entire rest of the population into the $[0, 0.02]$ range, destroying discriminative power. The 5th/95th percentile caps ensure that values beyond the 95th percentile are clamped to 1.0, preventing outlier domination while preserving the relative ordering of the non-extreme population.

The percentile bounds are themselves configurable parameters stored in the `scoring_weights` table:

| Parameter | Description | Default |
|---|---|---|
| `percentile_cap_low` | Normalisation floor percentile | 5.0 |
| `percentile_cap_high` | Normalisation ceiling percentile | 95.0 |

### 4.4.2 Bounded and Binary Features

For features with type `bounded` (already in $[0, 1]$) or `binary` ($\{0, 1\}$):

$$\hat{F}_{ik} = F_{ik} \tag{2}$$

No normalisation is applied. These features are inherently scaled and do not suffer from outlier distortion.

## 4.5 Weighted Linear Combination (Base Score)

The base score for binary $i$ is the weighted average of its normalised features:

$$S_{\text{raw}}^{(i)} = \frac{\sum_{k=1}^{K} \hat{F}_{ik} \cdot W_k}{\sum_{k=1}^{K} |W_k|} \tag{3}$$

The denominator normalises by the total *absolute* weight magnitude. This serves two purposes:

1. The raw ratio is bounded in $[-1, 1]$ regardless of how many features exist or how large the weights are.

2. Negative weights (e.g., `rich_header_compiler_upgrade` at $W = -3.0$) actively reduce the score when the corresponding feature fires, without breaking the mathematical bounds. Using $|W_k|$ rather than $W_k$ in the denominator ensures that

negative weights contribute to the normalisation factor and do not cause the denominator to shrink.

The raw score $S_{\text{raw}}^{(i)} \in [-1, 1]$ is then mapped to $[0, 1]$:

$$S_{\text{base}}^{(i)} = \text{clamp}\left(\frac{S_{\text{raw}}^{(i)} + 1}{2},\ 0,\ 1\right) \tag{4}$$

A base score of 0.5 indicates a binary with no distinguishing features (all normalised features at zero or cancelling out). Scores above 0.5 indicate positive security-relevance signals; scores below 0.5 indicate active demotion by negative-weight features.

**Informal interpretation.** Imagine 200 files changed in a driver update. The Weight Vector encodes domain knowledge: "security API additions are very important (2.5), compiler upgrades are anti-interesting ($-3.0$)." The Base Score applies these rules across the normalised feature matrix and produces a preliminary ranking. A binary that gained CFG and added security APIs will score highly; a binary that merely got recompiled with a newer Visual Studio will be actively demoted.

## 4.6 Z-Score Anomaly Detection

The base score captures *known* importance signals via the weight vector. The anomaly score captures *unknown* importance: a binary whose feature vector is statistically unusual relative to its peers, regardless of which specific features fired.

The z-score of feature $k$ for binary $i$ is:

$$Z_{ik} = \frac{\hat{F}_{ik} - \mu_k}{\max(\sigma_k, \epsilon)} \tag{5}$$

where $\epsilon = 0.001$ prevents division by zero for zero-variance columns.

The anomaly score is the **root-mean-square** (RMS) of the z-scores across all informative features:

$$S_{\text{anomaly}}^{(i)} = \sqrt{\frac{1}{K'} \sum_{k=1}^{K'} Z_{ik}^2} \tag{6}$$

where $K'$ is the number of features with $\sigma_k \geq \epsilon$ (features with zero variance are excluded from the anomaly calculation, as they provide no discriminative information).

The RMS formulation was chosen over the simple maximum z-score because it rewards binaries that are unusual across *multiple* dimensions simultaneously, rather than binaries that are extreme in a single dimension. A value of 1.0 means "on average, each feature is one standard deviation from the population mean." Values above 2.0 are strongly anomalous.

**Informal interpretation.** The Anomaly Score looks at a binary and says: "Not only did you trigger our weighted rules, but you acted completely differently than the other 199 files in this update across many dimensions simultaneously." A binary might

have a modest base score but an extremely high anomaly score if its combination of features—even individually unremarkable features—is collectively unlike anything else in the population.

### 4.7 Final Research Priority Score

The anomaly score amplifies the base score multiplicatively:

$$S_{\text{final}}^{(i)} = S_{\text{base}}^{(i)} \cdot \left(1 + \min\left(S_{\text{anomaly}}^{(i)},\ C_{\text{anomaly}}\right) \cdot \alpha_{\text{anomaly}}\right) \tag{7}$$

The parameters are:

| Parameter | Description | Default |
|---|---|---|
| $\alpha_{\text{anomaly}}$ | Anomaly influence factor | 0.3 |
| $C_{\text{anomaly}}$ | Maximum anomaly before capping | 3.0 |

The cap at $C_{\text{anomaly}}$ prevents extreme outliers from dominating the ranking (an anomaly score of 50 should not produce a 15× multiplier). With the default parameters, the maximum amplification factor is $1 + 3.0 \times 0.3 = 1.9$, so a binary's score can at most be nearly doubled by anomaly amplification.

The final score is clamped to $[0, 1]$:

$$S_{\text{final}}^{(i)} \leftarrow \text{clamp}\left(S_{\text{final}}^{(i)},\ 0,\ 1\right) \tag{8}$$

### 4.8 Design Constraint: Purely Mathematical Scoring

An immutable architectural rule: the scoring system must be purely mathematical—feature extraction → normalisation → weighted linear combination → anomaly adjustment. There are no `if/else` chains that add hardcoded constants based on boolean checks. The Go source defines the *algorithm*; the database stores the *coefficients*. This separation ensures that weight tuning—whether manual or via future automated feedback loops—never requires recompilation.

## 5 Silent Fix Detection

A **silent fix** is a binary change that exhibits security-patch characteristics but has no corresponding published CVE. Detecting silent fixes is one of the primary value propositions of UNDERTOW: it surfaces security-relevant changes that the vendor did not publicly disclose.

The silent fix score reuses the same mathematical machinery as the research priority score (Section 4) but with a different weight vector $\mathbf{W}_{\text{sf}}$ stored in the `scoring_weights` table with `score_type = 'silent_fix'`. This separate weight vector can assign higher

importance to features specifically associated with silent fixes (e.g., exploit mitigation activation, security API changes) and lower importance to features associated with general code evolution.

### 5.1 Computation

$$S_{\text{sf}}^{(i)} = S_{\text{base,sf}}^{(i)} \cdot \left(1 + \min\left(S_{\text{anomaly}}^{(i)},\ C_{\text{anomaly}}\right) \cdot \alpha_{\text{anomaly}}\right) \tag{9}$$

The anomaly score $S_{\text{anomaly}}^{(i)}$ is shared between both score types (it is a property of the binary's statistical position in the population, not of the weighting scheme).

### 5.2 Threshold Classification

The boolean `is_silent_fix` is set when:

$$\texttt{is\_silent\_fix} = \left(S_{\text{sf}}^{(i)} \geq \theta_{\text{sf}}\right) \wedge (\text{no published CVEs for this component in the target version}) \tag{10}$$

where $\theta_{\text{sf}}$ is the silent fix threshold, loaded from the database (default: 0.30). The conjunction with the CVE check is essential: a binary that scores highly *and* has a published CVE is a known patch, not a silent fix.

## 6 Generalised ESD Anomaly Detection

### 6.1 Rationale

While the Scoring Engine (Section 4) compares a binary to *other binaries in the same version transition*, we also need to compare a binary to *its own history across multiple version transitions*.

For each component, UNDERTOW builds a time-ordered series of change magnitudes: the TLSH distance (or 0 for versions where the component did not change) across all consecutive version pairs. The question is: "In this specific version transition, did this component change more than it typically does?"

The Generalised Extreme Studentised Deviate (ESD) test was selected because:

1. It detects one *or more* point anomalies in a univariate time series (unlike the simpler Grubbs test, which assumes at most one outlier).

2. It iteratively removes extremes to prevent **masking**: a massive anomaly artificially inflating the standard deviation and hiding smaller anomalies.

3. It provides a principled statistical threshold via the $t$-distribution, rather than arbitrary cutoffs.

## 6.2 Formal Specification

Given a time series $\{x_1, x_2, \ldots, x_n\}$ of change magnitudes and a maximum number of suspected anomalies $r$:

For $i = 1, 2, \ldots, r$:

1. Compute the mean $\bar{x}$ and standard deviation $s$ of the *remaining* data (after previous removals).

2. Find the most extreme studentised deviate:

$$R_i = \max_j \frac{|x_j - \bar{x}|}{s} \tag{11}$$

3. Compute the critical value $\lambda_i$ using the $t$-distribution:

$$\lambda_i = \frac{(n-i) \cdot t_{p,\,n-i-1}}{\sqrt{\left(n-i-1+t^2_{p,\,n-i-1}\right)(n-i+1)}} \tag{12}$$

   where $t_{p,\nu}$ is the $t$-distribution quantile with:

$$p = 1 - \frac{\alpha}{2(n-i+1)}, \qquad \nu = n-i-1 \tag{13}$$

   and $\alpha$ is the significance level (default: 0.05).

4. Remove the most extreme value from the dataset and repeat.

After all $r$ iterations, find the largest $i$ such that $R_i > \lambda_i$. All values removed in iterations $1, \ldots, i$ are declared anomalous.

**Intuition.** The ESD test iteratively pulls the most extreme outlier out of the dataset and asks: "Is this point statistically far enough from the rest to be considered a genuine anomaly, or is it merely the natural maximum of a bell-shaped distribution?" By removing points one by one and recomputing statistics, it prevents the masking effect where a single massive outlier inflates $s$ so much that smaller (but still anomalous) outliers appear normal.

## 6.3 Applicability Threshold

The ESD test requires $n \geq 10$ data points for statistically meaningful results. With the current 9 R580 GRD driver versions (producing 8 version transitions), this is borderline. The corpus expansion plan:

| **Priority** | **Versions** | **Cumulative $n$** |
|---|---|---|
| P1 | R575/R577 (577.11, 576.94, 576.40) | $\geq 11$ |
| P2 | R560/R565 (566.36, 560.94) | $\geq 13$ |
| P3 | R550 and older (555.85, 551.86, 546.33, 537.70) | $\geq 17$ |

At $n \geq 15$, the Cornish-Fisher approximation (Section 6.4) becomes highly accurate and the test is robust.

## 6.4 Programmatic Quantile Approximations

Because the implementation language (Go) lacks a native statistical library for $t$-distribution quantiles, UNDERTOW implements the computation natively using two classical approximations.

### 6.4.1 Normal Quantile: Abramowitz & Stegun

The rational approximation for the standard normal inverse CDF:

$$z_n = t - \frac{c_0 + c_1 t + c_2 t^2}{1 + d_1 t + d_2 t^2 + d_3 t^3} \tag{14}$$

where $t = \sqrt{-2\ln(1-p)}$ and the constants are:

$$\begin{aligned} c_0 &= 2.515517 \quad d_1 = 1.432788 \\ c_1 &= 0.802853 \quad d_2 = 0.189269 \\ c_2 &= 0.010328 \quad d_3 = 0.001308 \end{aligned}$$

This approximation has maximum absolute error $< 4.5 \times 10^{-4}$ across the full range, which is more than sufficient for the ESD critical value computation.

### 6.4.2 $t$-Distribution Correction: Cornish-Fisher Expansion

To correct for the heavier tails of the $t$-distribution relative to the normal, the Cornish-Fisher expansion is applied:

$$t_\nu \approx z_n + \frac{z_n^3 + z_n}{4\nu} + \frac{5z_n^5 + 16z_n^3 + 3z_n}{96\nu^2} \tag{15}$$

For $\nu \geq 30$, this is very accurate (error $< 10^{-3}$). For smaller degrees of freedom, it is conservative—producing slightly wider confidence intervals, which means the test is less likely to declare false anomalies.

**Why native implementation?** Avoiding external statistical libraries eliminates a dependency risk (Go's ecosystem has no canonical statistics package comparable to SciPy), keeps the binary self-contained, and ensures the computation is fully auditable. The polynomial approximations allow exact reproduction of the algorithm without reliance on lookup tables or floating-point library internals.

## 6.5 Change Velocity

Complementing the ESD test, UNDERTOW computes a **change velocity score** for each component:

$$v_{\text{component}} = \frac{\text{significant changes}}{\text{total versions seen} - 1} \tag{16}$$

where a "significant change" is a version transition where the component's SHA-256 hash changed. A velocity near 1.0 indicates a component that changes in every update (routine maintenance); a velocity near 0.0 indicates a stable component. Components with low velocity and an ESD-flagged anomalous transition are of particular interest: a normally stable component that suddenly changed dramatically.

# 7 Evolutionary Coupling Analysis

## 7.1 Motivation

Static dependencies (component A imports component B) are easy to extract from PE import tables but are often noisy: hundreds of components import `kernel32.dll`, which reveals nothing about architectural coupling. *Evolutionary* dependencies track components that change *together over time*, which reveals hidden architectural links invisible to static analysis.

The key insight for security research: if two components are highly coupled (they always change together) and one suddenly changes *without* the other, the change is likely a targeted, isolated fix rather than a broad feature update. Broad updates (API changes, struct redefinitions) require updating all coupled components; a change to one component alone suggests a surgeon went in, fixed a specific bug or vulnerability, and exited without touching the architecture.

## 7.2 Co-Change Matrix

Let $T_A$ be the set of version transitions where component $A$ was modified (SHA-256 changed). The co-change ratio between components $A$ and $B$ is the Jaccard coefficient applied to their transition sets:

$$r_{AB} = \frac{|T_A \cap T_B|}{|T_A \cup T_B|} \tag{17}$$

This is symmetric ($r_{AB} = r_{BA}$) and bounded in $[0, 1]$. A value of 1.0 means $A$ and $B$ changed in exactly the same set of version transitions; a value of 0.0 means they never changed together.

**Why Jaccard?** It was selected over alternatives (e.g., Pearson correlation of change magnitudes, or simple co-occurrence count) because:

1. It operates on discrete sets (version transitions), which is the natural representation—the question is "did they change together?" not "how much did they change?"

2. It normalises by the union, so a component that changes in every version (high individual activity) does not automatically appear coupled to everything.

3. It is efficiently computable in SQL via aggregate queries over the `binary_deltas` table.

The co-change matrix is stored in the `co_change_matrix` PostgreSQL table with columns: `canonical_id_a`, `canonical_id_b`, `co_change_count`, `total_possible`, `co_change_ratio`, `is_transitive`, and `transitive_via`.

## 7.3 Coupling Violation Detection

A **coupling violation** occurs in the current version transition if historically coupled components fail to change together. Formally, a violation is recorded for component $A$ if:

$$r_{AB} \geq \theta_{\text{coupling}} \quad \wedge \quad (\text{Changed}(A) \ \wedge \ \neg\, \text{Changed}(B)) \tag{18}$$

where $\theta_{\text{coupling}}$ is the coupling threshold (default: 0.75, loaded from `scoring_weights`).

Detection is symmetric: if $A$ changed without $B$, *or* $B$ changed without $A$, both are recorded. The violations are stored in the `coupling_violations` table with columns: `canonical_id`, `expected_partner`, `from_version`, `to_version`, `expected_ratio`, and `partner_changed`.

**Informal interpretation.** If Component A and Component B have been updated together 90% of the time over the last 3 years ($r_{AB} = 0.9$), they are highly coupled—likely sharing an internal API, struct definition, or build dependency. If a new driver release modifies Component A *without* Component B, something unusual happened. Broad feature updates and API changes would require updating both. A change to $A$ alone suggests a surgical intervention: a specific bug fix, a targeted security patch, or an isolated regression fix. This mathematical flag is one of UNDERTOW's most powerful beacons for security researchers.

## 7.4 Transitive Coupling Inference

Direct co-change data requires sufficient shared version transitions. For component pairs with limited direct history, UNDERTOW infers latent coupling through a transitive step.

If direct couplings $r_{AB} \geq \theta_{\min}$ and $r_{BC} \geq \theta_{\min}$ both exist, the latent coupling between $A$ and $C$ is:

$$r_{AC}^{(\text{transitive})} = r_{AB} \cdot r_{BC} \cdot \delta_{\text{decay}} \tag{19}$$

where $\delta_{\text{decay}}$ (default: 0.50) is a decay factor that prevents the transitive graph from becoming meaninglessly dense. Without the decay, a chain of marginally coupled components would produce high transitive coupling scores.

**Constraints on transitive inference:**

- Only direct (non-transitive) edges participate as source edges—no multi-hop chaining.
- $A \neq C$ (no self-loops).
- If a direct edge $A \leftrightarrow C$ already exists, the transitive ratio only overwrites if it is higher.
- $\theta_{\min}$ (default: 0.60) and $\delta_{\text{decay}}$ are loaded from `scoring_weights`, never hardcoded.

| Parameter | Description | Default |
|---|---|---|
| `coupling_min_ratio` | Min ratio for violation detection | 0.75 |
| `transitive_decay` | Decay factor for transitive inference | 0.50 |
| `transitive_min_direct` | Min direct ratio for transitive expansion | 0.60 |

**Informal interpretation.** Transitive inference says: "If $A$ always changes with $B$, and $B$ always changes with $C$, then $A$ and $C$ are architecturally related—even if they have never directly changed together in the observed history." The decay factor ensures that these inferred links are weaker than directly observed ones, reflecting the uncertainty introduced by the intermediate step.

# 8 Threshold Parameters Summary

All configurable parameters are stored in the `scoring_weights` PostgreSQL table with `is_threshold = true`. They are loaded at runtime and can be adjusted without recompilation.

| Parameter | Symbol | Default | Used In |
|---|---|---|---|
| `anomaly_influence` | $\alpha_{\text{anomaly}}$ | 0.3 | Eq. 7 |
| `anomaly_cap` | $C_{\text{anomaly}}$ | 3.0 | Eq. 7 |
| `silent_fix_threshold` | $\theta_{\text{sf}}$ | 0.30 | Eq. 10 |
| `coupling_min_ratio` | $\theta_{\text{coupling}}$ | 0.75 | Eq. 18 |
| `transitive_decay` | $\delta_{\text{decay}}$ | 0.50 | Eq. 19 |
| `transitive_min_direct` | $\theta_{\text{min}}$ | 0.60 | Eq. 19 |
| `percentile_cap_low` | $P_{\text{low}}$ | 5.0 | Eq. 1 |
| `percentile_cap_high` | $P_{\text{high}}$ | 95.0 | Eq. 1 |

# 9 Validation and Verification

## 9.1 Scoring Engine Validation

After a full dictionary rebuild across the version timeline, the following mathematical properties must hold:

1. **Non-degenerate score distribution**: $\text{stddev}(S_{\text{final}}) > 0$ across the population. If all scores are identical, the scoring engine has no discriminative power.

2. **Anomaly score variation**: $S_{\text{anomaly}}$ values must vary across the population (not all equal), confirming that z-score computation found feature-level variance.

3. **Weight loading**: Three `score_type` groups (`research_priority`, `silent_fix`, `threshold`) must load successfully from the database.

4. **Negative weight effect**: At least one binary with `rich_header_compiler_upgrade = 1` must score lower than a comparable binary without this flag, confirming negative weights function correctly.

## 9.2 ESD Test Validation

The ESD implementation is validated against five canonical test cases:

| Test Case | Series | Expected Result |
|---|---|---|
| Flat series | $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$ | 0 anomalies (zero variance) |
| Single spike | $[1, 1, 1, 1, 1, 1, 1, 1, 1, 100]$ | 1 anomaly (index 9) |
| Short series | $[1, 2, 3]$ | Skipped ($n < 10$) |
| Dual spike | $[1, 1, 1, 1, 1, 100, 1, 1, 1, 200]$ | 2 anomalies (both outliers) |
| Gradual trend | $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ | 0 anomalies (trend $\neq$ anomaly) |

## 9.3 Coupling Analysis Validation

1. Co-change matrix entries must exist for component pairs that both changed in at least one shared version transition.

2. Coupling violations must be detected when a component changes without its partner (for pairs with $r_{AB} \geq \theta_{\text{coupling}}$).

3. Transitive coupling entries must exist when direct couplings above $\theta_{\min}$ form a chain $A \to B \to C$.

## 9.4 Ground Truth Confirmation (Future)

As the CVE bulletin module ingests bulletins over time, it links CVEs to binary components. When a binary that the engine flagged as `security_patch_candidate` later receives a published CVE, that constitutes a true positive. This feedback enables tracking precision at rank $k$:

> Of the top-$k$ ranked targets, how many were later confirmed by published CVEs?

This is explicitly deferred to a future iteration. The current mechanism is manual weight tuning via the `scoring_weights` table. Future approaches include Bayesian weight updating, grid search over known CVE events, and genetic algorithm-based weight optimisation—all requiring a sufficient ground truth dataset (at least 20–30 confirmed CVE-to-binary mappings).

# 10 Implementation Notes

## 10.1 Technology Stack

The mathematical subsystems are implemented in Go 1.22+ with the following key dependencies:

| Component | Purpose |
|---|---|
| `saferwall/pe` | PE file parsing (pure Go, no CGO) |
| `glaslos/tlsh` | TLSH locality-sensitive hashing |
| `pgx/v5` | PostgreSQL driver with batch support |
| Neo4j Go driver | Knowledge graph synchronisation |
| `golang-migrate` | Database schema versioning |

## 10.2 Performance Considerations

**Population-level scoring**
: Weights are loaded once per version transition and applied across the entire population in a single pass, rather than loading per-binary.

**Batched persistence**
: Binary deltas are collected per version transition and inserted using `pgx.Batch` for amortised round-trip cost.

**Conditional TLSH**
: TLSH computation is skipped for binaries where SHA-256 has not changed (distance would be 0) and for binaries smaller than 256 bytes (TLSH minimum input size).

**Gated .pdata analysis**
: Function-level RVA diffing is only performed for binaries above the population median score, avoiding expensive analysis of uninteresting binaries.

## 10.3 Database Schema Highlights

The scoring weights table enforces the algorithm–coefficient separation:

```
CREATE TABLE scoring_weights (
    score_type   TEXT NOT NULL,
    feature_name TEXT NOT NULL,
    weight       FLOAT8 NOT NULL,
    description  TEXT,
    is_threshold BOOLEAN DEFAULT FALSE,
    updated_at   TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (score_type, feature_name)
);
```

Three `score_type` values partition the table: `research_priority` (27 feature weights), `silent_fix` (separate weight vector), and `threshold` (configurable parameters from Section 8). Seed values use `ON CONFLICT DO NOTHING` for idempotent migration reruns.

# 11 Conclusion

The mathematical models deployed in UNDERTOW transform the subjective art of binary triage into an objective, reproducible, data-driven pipeline. The four interconnected subsystems—Feature Vector Scoring, Silent Fix Detection, Generalised ESD Anomaly Detection, and Evolutionary Coupling Analysis—operate at complementary levels of analysis:

1. The **Feature Vector Scoring Engine** compares each binary to its *peers* in the same version transition, using population normalisation and z-score anomaly amplification to rank changes by security relevance.

2. The **Silent Fix Detection** mechanism applies a separate weight vector tuned for undisclosed security patches, flagging binaries that exhibit patch characteristics without a corresponding CVE.

3. The **Generalised ESD Test** compares each component to *its own history*, detecting version transitions where a component broke its own baseline change pattern.

4. The **Evolutionary Coupling Analysis** detects components that changed in isolation from their historically coupled partners, a hallmark of targeted surgical fixes.

Together, these models allow a security researcher to mathematically filter out software maintenance noise—compiler upgrades, routine recompilations, broad API updates—and isolate the true markers of defensive security patches. The system's output is not a vulnerability report but a ranked, evidence-backed guide that tells the researcher where to point their disassembler.

All parameters are stored in PostgreSQL and loaded at runtime. The algorithm is fixed in code; the coefficients are tuneable in the database. As the corpus grows and CVE confirmations accumulate, the weights can be refined through empirical feedback without modifying any source code—bridging the gap between domain-knowledge-driven initial estimates and data-driven optimisation.

The 27-dimensional feature vector represents the current scoring implementation. A second tuning cycle is planned to incorporate the additional delta dimensions described in Section 2 that are not yet represented in the scoring layer: entry point RVA changes, Authenticode signature transitions, delay-load import surface changes, obfuscation indicator transitions, the UNDERTOW-canonical import hash (`undertow_imphash`), and the full set of exploit mitigation deactivation signals (`xfg_deactivated`, `retpoline_removed`). Each of these dimensions is already present in the `BinaryDelta` structure and available

to the scoring engine; assigning appropriate weights requires a sufficient labelled corpus of confirmed CVE-to-binary mappings to avoid arbitrary coefficient assignment.

# References

[1] B. Rosner, "Percentage Points for a Generalized ESD Many-Outlier Procedure," *Technometrics*, vol. 25, no. 2, pp. 165–172, 1983.

[2] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards, 1964. (Rational approximation 26.2.17.)

[3] E. A. Cornish and R. A. Fisher, "Moments and Cumulants in the Specification of Distributions," *Revue de l'Institut International de Statistique*, vol. 5, no. 4, pp. 307–320, 1938.

[4] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[5] Microsoft Corporation, "PE Format," *Microsoft Learn*, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format. (IMAGE_LOAD_CONFIG_DIRECTORY specification.)

[6] OASIS, "Common Security Advisory Framework (CSAF) Version 2.0," OASIS Standard, 2022.

[7] NIST, "National Vulnerability Database API 2.0," https://nvd.nist.gov/developers.

[8] Trend Micro, "Trend Micro Locality Sensitive Hash (TLSH)," https://github.com/trendmicro/tlsh.

[9] Mandiant, "Tracking Malware with Import Hashing," FireEye Blog, 2014.

[10] IEEE Std 1016-2009, "IEEE Standard for Information Technology—Systems Design—Software Design Descriptions," IEEE, 2009.