TECHNICAL DOCUMENTATION

Harshil
DIGISURAKSHA PARHARI FOUNDATION

Download Link:

https://github.com/HarshilShiroya/Digisuraksha_340/tree/main/Internship_Work/Week_2/homography

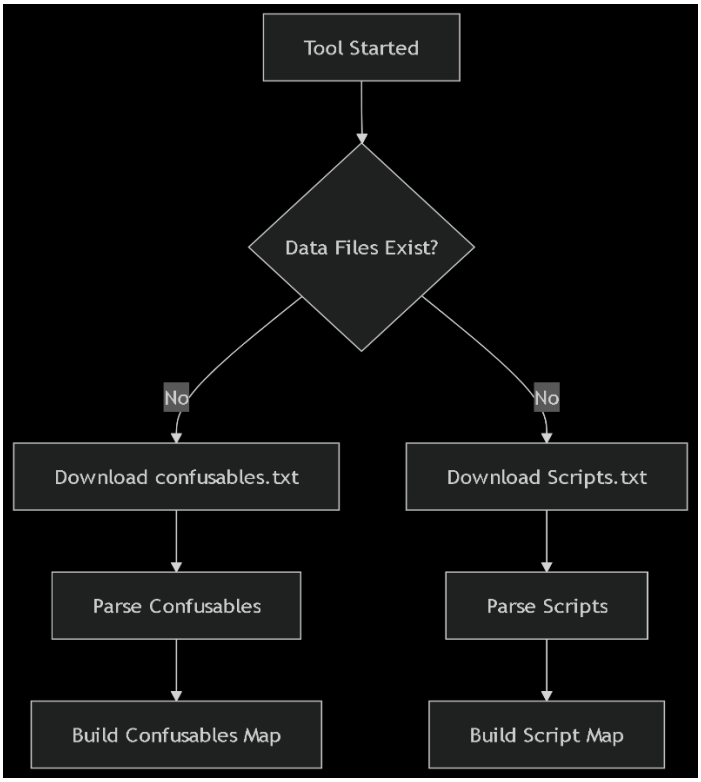
Overview

This document explains how the homograph detector tool works, detailing its internal processes and data requirements. The tool identifies potential Unicode homograph attacks by analyzing input strings through a multi-stage detection pipeline.

Input Processing Workflow

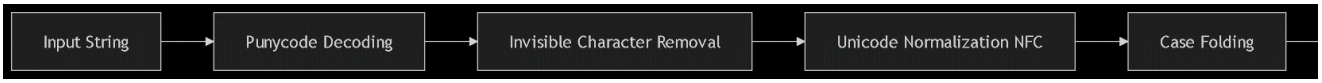
When you provide an input string to the tool, it undergoes the following processing stages:

1. Initialization (First Run Only)



When we will run this tool, it first check if data file is available or not if not then it will automatically download both files and parse it, so it will be ready when you give any input to it to check homography.

2. Input Processing Pipeline



1) Punycode Decoding (IDNA Processing)

- Input: Raw string (potential IDN domain)
- Process:
    - Detects ASCII-compatible encoded domains (RFC 5890)
    - Decodes xn-- prefixed labels using IDNA 2008 standards
    - Validates against UTS #46 normalization rules
    - Preserves both original and decoded forms for comparison
- Background:
    - Converts ACE (ASCII Compatible Encoding) to Unicode
    - Handles internationalized domain names (IDNs)
    - Example: xn--mgbh0fb.xn--kgbechtv → تونس.موقع

2) Invisible Character Removal

- Input: Unicode string (decoded if IDN)
- Process:
    - Scans for Unicode categories:
        - Cf (Format controls)
        - Cc (Control characters)
        - Mn/Nonspacing marks
    - Special case detection:
        - Zero-width joiners/non-joiners (U+200C-U+200D)
        - Bidirectional overrides (U+202A-U+202E)
        - Byte order marks (U+FEFF)
        - Records positions and types of removed characters
- Background:
    - Prevents visual spoofing via non-printing characters
    - Maintains positional integrity for diagnostics
    - Example: ppaypal → ppaypal (removes U+200C)

3) Unicode Normalization (NFC)

- Input: Sanitized Unicode string
- Process:
    - Applies Normalization Form C (Canonical Composition)
    - Decomposes then recomposes characters:
        - Canonical decomposition
        - Canonical reordering
        - Composition exclusion filtering
    - Handles compatibility equivalents in NFKC mode

- Background:
    - Ensures canonical equivalence (UTS #15)
    - Resolves multiple representations of same glyph
    - Example: e´ (U+0065 U+0301) → é (U+00E9)

4) Case Folding

- Input: Normalized Unicode string
- Process:
    - Applies full Unicode case folding (Section 3.13)
    - Special handling for:
        - Greek final sigma (context-sensitive)
        - Turkish dotted/dotless I
        - German sharp S (ß → ss)
    - Preserves script properties during transformation
- Background:
    - Implements case-insensitive matching (UTS #10)
    - Example: Ωmega → ωmega

5) Confusable Mapping

- Input: Case-folded string
- Process:
    - Consults UTR 39 confusables database
    - Applies single-script and mixed-script mappings
    - Handles multi-character confusables (e.g., rn → m)
    - Maintains mapping provenance metadata
- Background:
    - Uses skeleton algorithm (UTS #39)
    - Example: paypal → paypal (Cyrillic to Latin)

6) Skeleton Creation

- Input: Confusable-mapped string
- Process:
    - Generates canonical visual representation
    - Combines results from all previous transformations
    - Creates stable identifier for comparison
- Background:
    - Forms equivalence classes for visual similarity
    - Example: 𝒫ay𝓅𝒶ł → paypal

7) Script Analysis

- Input: Original string and skeleton
- Process:
    - Annotates each character with script property
    - Calculates:
        - Script distribution histogram
        - Script transition points
        - Mixed-script entropy
    - Validates against allowed script policy
- Background:
    - Uses Unicode Script.txt database
    - Implements UTS #39 mixed-script detection
    - Example: paypal → Latin+Cyrillic mix

8) Whitelist Comparison

- Input: Skeleton and original string
- Process:
    - Performs O(1) lookup in precomputed skeleton hash table
    - Compares:
        - Skeleton similarity (exact match)
        - Original string equivalence
        - Script consistency
    - Calculates deviation scores
- Background:
    - Implements minimum edit distance fallback
    - Example: paypal.com matches paypal.com skeleton

9) Result Compilation

- Input: All analysis artifacts
- Process:
    - Aggregates detection signals:
        - Confusable distance
        - Script mixing severity
        - Whitelist deviation
    - Generates:
        - Risk score (0.0-1.0)
        - Diagnostic trace
        - Visual diff highlighting
- Background:
- Applies weighted decision matrix
- Produces explainable security verdicts

- Example output:

```
{
  "risk": 0.92,
  "triggers": [
    {"type": "script_mixing", "scripts": ["Latin", "Cyrillic"]},
    {"type": "confusable", "pos": 0, "from": "p", "to": "p"}
  ]
}
```


Detailed Process Explanation

1. Punycode Decoding

- Purpose: Handle internationalized domain names (IDNs)
- Process:
  - Checks if input starts with xn--
  - Decodes to Unicode using IDNA 2008 standard
  - Example: xn--80ak6aa92e.com → appIe.com

2. Invisible Character Removal

- Targets:
  - Zero-width characters (U+200B, U+200C, U+200D)
  - Directional controls (U+202A-U+202E)
  - Format characters (U+FEFF, U+2060)
  - Non-spacing marks (diacritics)
- Process:
  - Scans each character
  - Removes invisible characters
  - Records removal positions

3. Unicode Normalization (NFC)

- Purpose: Handle equivalent character representations
- Process:
  - Combines characters and diacritics
  - Example: café = c + a + f + e´ → single composed character

4. Case Folding

- Purpose: Case-insensitive comparison
- Process:
  - Converts to lowercase using full Unicode case folding
  - Example: ExAmPle.COM → example.com

5. Confusable Mapping

- Data Source: confusables.txt (Unicode UTR 39)
- Process:
    - Replaces characters with their "skeleton" equivalents
    - Example mappings:
        - Cyrillic a (U+0430) → Latin a
        - Greek β (U+03B2) → Latin b
        - Mathematical g (U+1D600) → Latin g

6. Skeleton Creation

- Purpose: Create canonical representation for comparison
- Process:
    - Combines results from previous steps
    - Example: paypal.com → paypal.com

7. Script Analysis

- Data Source: Scripts.txt (Unicode Script Data)
- Process:
    - Identifies script for each character
    - Checks for:
        - Disallowed scripts (configurable)
        - Mixed scripts in single label
    - Example: paypal → Latin + Cyrillic = mixed script

8. Whitelist Comparison

- Purpose: Detect homographs of trusted entities
- Process:
    - Compares skeleton against precomputed trusted skeletons
    - Flags if skeleton matches but raw input differs
    - Example: paypal.com matches paypal.com skeleton

9. Result Compilation

- Output Includes:
    - Suspicious status (boolean)
    - Computed skeleton
    - Matched whitelist entry (if any)
    - Detailed reasons for flagging
    - Character-level analysis

Critical Data Files

The homograph detector relies on two essential Unicode data files to perform accurate confusable detection and script analysis. These files provide the foundational mappings and character properties required to identify deceptive strings. Below is a detailed breakdown of each file's role, structure, and processing methodology.

1. confusables.txt

This file defines a comprehensive mapping of visually similar characters (homoglyphs) across different scripts. It enables the detector to convert deceptive characters (e.g., Cyrillic а) into their Latin equivalents (a) for comparison.

- Source: Unicode Technical Report #39
- Location: https://www.unicode.org/Public/security/latest/confusables.txt
- Format:
  0061 ; 0430 ; MA # (a → а) LATIN SMALL LETTER A → CYRILLIC SMALL LETTER A
  0062 ; 0432 ; MA # (b → в) LATIN SMALL LETTER B → CYRILLIC SMALL LETTER VE
- Purpose: Provides mapping between visually similar characters

Processing Steps:

- Parsing: The detector loads the file into a hash table, mapping each deceptive character to its skeleton equivalent.
- Normalization: Ensures all mappings are in NFC form for consistency.
- Lookup Optimization: Uses a trie structure for efficient multi-character confusable resolution (e.g., ѕ → s).

2. Scripts.txt

This file categorizes every Unicode character by its script (e.g., Latin, Cyrillic, Greek), allowing the detector to identify mixed-script attacks (e.g., Latin o + Cyrillic с in ocr.com).

- Source: Unicode Character Database
- Location: https://www.unicode.org/Public/UCD/latest/ucd/Scripts.txt
- Format:
  0041..005A ; Latin # L& [26] LATIN CAPITAL LETTER A..LATIN CAPITAL LETTER Z
  0061..007A ; Latin # L& [26] LATIN SMALL LETTER A..LATIN SMALL LETTER Z
  0400..0484 ; Cyrillic # Mn [133] CYRILLIC CAPITAL LETTER IE WITH GRAVE..
- Purpose: Defines script associations for Unicode characters

Processing Steps:

- Binary Search: The detector builds an interval tree for fast script lookups.
- Script Inheritance: Characters like punctuation (Common) inherit the dominant script of surrounding text.
- Mixed-Script Detection: Computes script entropy per label—high entropy indicates suspicious mixing.

Maintenance and Updates

- Data Refresh: Run with --update-data annually or after Unicode releases
- Whitelist Management: Update JSON file as trusted entities change
- Version Compatibility: Test with new Python versions (3.8+ supported)

Working Demo: