The Enigma DSL: How A Wartime Cipher Became a Proto-Programming Language

Douglas C. Youvan

doug@youvan.com

www.youvan.ai

October 12, 2025

The WWII Enigma machine is usually told as a story about cryptography, intelligence, and the race to break a cipher. This paper argues a complementary thesis: Enigma's operating procedures constitute a domain-specific language (DSL) avant la lettre, and the analytical methods developed to attack it—Turing's logical frameworks, Welchman's diagonal board, Bombe/Colossus wiring tables—amount to early program semantics and hardware description. Enigma's "code" lived in key sheets and operating rituals: rotor order, ring settings, start positions, plugboard pairs, reflector choice, and a strict message procedure. That artifact looks strikingly like a modern configuration-as-code file whose interpreter is a physical cipher engine. We formalize this view with a compact grammar (EBNF), state-transition semantics, and permutation algebra; then trace the lineage forward to contemporary programming practice: assembly and microcode, regex and SQL, protocol specs and ciphersuites, HDLs and formal methods. The Enigma DSL was intentionally narrow—Turing-incomplete by design—yet it delivered high assurance through constrained expressivity and deterministic replay. By reading Enigma as a language, we illuminate deep continuities between cryptographic practice and today's software engineering: how we specify behavior, verify invariants, constrain degrees of freedom for safety, and compile intent into circuits. The machine that "spoke" in letters helped teach us how to speak to machines.

Keywords: Enigma, domain-specific language, Bombe, Colossus, Turing, rotor cipher, EBNF, operational semantics, permutation algebra, hardware description language, formal methods, configuration-as-code, safety by construction.

A collaboration with GPT-5. 75 pages.

Outline

1. Problem Statement and Thesis
   1.1 From cipher device to language view
   1.2 Why a DSL lens clarifies both history and practice
2. The Enigma Artifact as Language
   2.1 Alphabet, parameters, and state (rotor_order, rings, start, plugs, reflector)
   2.2 Procedure as program: configure -> step -> transform -> emit
   2.3 Key sheets as configuration-as-code
3. A Minimal Enigma Grammar (EBNF)
   3.1 Concrete syntax for operator directives
   3.2 Static checks: disjoint plugs, valid rotors, ranges
   3.3 Parsing to an AST and error models
4. Operational Semantics
   4.1 State transition: turnover and double-step rules
   4.2 Enciphering as composed permutations: plug^-1 ∘ rot^-1 ∘ R ∘ rot ∘ plug
   4.3 Determinism, invertibility, and replay logs
5. Algebraic Foundations
   5.1 Modeling rotors and plugs in S_26
   5.2 Reflector as an involution; consequences for self-inverse behavior
   5.3 Safety properties provable from group structure
6. Early Program Analysis in Practice
   6.1 Turing/Welchman notations as proto-pseudocode
   6.2 Bombe/Colossus wiring tables as HDLs
   6.3 Cribs and constraint propagation as search over state space
7. Lineage to Modern Languages and Systems
   7.1 Assembly and microcode: mnemonic control of hardware state
   7.2 Regex and SQL: constrained, declarative engines
   7.3 Protocol specs and ciphersuites: declarative crypto DSLs
   7.4 HDLs (Verilog/VHDL) and netlists: wiring as computation
8. Expressivity vs Assurance: A Design Trade-off
   8.1 Turing-incomplete by design: why small languages are safer
   8.2 Misuse vs break: operational failures and procedure hygiene
   8.3 Reproducibility, audit trails, and “witness” execution

1. Problem Statement and Thesis

Most accounts of Enigma emphasize cryptanalytic heroics or device mechanics. This paper reframes Enigma as a language artifact: the operational procedures, key sheets, and wiring conventions together form a tiny domain-specific language (DSL) interpreted by a physical cipher engine. The thesis is twofold. First, Enigma's "programs" are its configurations and rituals of use; these specify a state machine that deterministically transforms inputs (plaintext letters) into outputs (ciphertext letters). Second, the Allied response—Turing/Welchman notations, Bombe and Colossus wiring tables, crib-driven constraint search—constitutes early program analysis and hardware description practice. Reading Enigma through a DSL lens recovers design principles that still govern modern computing: minimal expressivity for assurance, deterministic replay for accountability, and formal interfaces between human procedure and machine behavior. The goal is not to anachronistically elevate Enigma into a general-purpose language, but to show how a constrained, safety-oriented language shaped both the attack surface and the birth of program semantics.

1.1 From cipher device to language view

A device view asks: how do rotors, notches, and reflectors permute letters?
A language view asks: what is the syntax and semantics by which operators direct that device?

- Syntax (surface form): key sheets encode rotor_order, ring_settings, start_positions, plugboard_pairs, and reflector choice. These are well-typed fields with small finite domains and validity constraints (e.g., disjoint plug pairs, legal rotor names).
- Static checks: before any run, operators validate that parameters lie in range. Violations are compile-time errors in today's terms.
- Semantics (meaning): a keypress applies a state transition (stepping with turnover rules) and a pure transformation (composition of permutations). Decoding(Encoding(m,S),S) = m is a semantic invariant under fixed S.
- Execution model: the machine is an interpreter of this language; the "program" is the configuration plus a fixed procedure (double-enciphered message key, header, body). Each message is a fresh, parameterized run.
- Observables and logs: headers, indicators, and message traffic are execution traces. Cribs are partial specifications that constrain legal runs.

Seen this way, Enigma is not a pile of parts; it is a small, austere language with a single instruction cycle (press) and a narrow but precisely defined state space. Its vulnerabilities are therefore language-level: ambiguity or sloppiness in procedure, low-entropy choices in message

keys, and repeated patterns that leak constraints—in modern terms, misconfiguration, weak randomness, and predictable I/O.

1.2 Why a DSL lens clarifies both history and practice

- Explains why “small beats big” in assurance. Enigma’s language is intentionally non–Turing-complete. Constrained expressivity yields predictable behavior, simple mental models, and easier auditing—exactly the rationale for modern safety DSLs (e.g., SQL, regex, P4, iptables, cipherspecs).
- Unifies operators, machines, and analysts under one model. Operators write programs (configs), Enigma interprets them, and codebreakers perform static and dynamic analysis (constraint solving, model checking, and differential testing via cribs). This triangulation matches today’s pipeline of config-as-code -> runtime -> verification.
- Locates the real failure modes. The core permutation algebra is sound; the break came from language misuse: repeated indicators, stereotyped phrases, procedural deviations, and human factors. This mirrors modern incidents where secure primitives fail due to configuration errors rather than cryptographic flaws.
- Illuminates the Bombe/Colossus leap. Bombe wiring tables are hardware descriptions; Colossus is a reconfigurable logic engine executing a higher-level spec (paper-tape patterns and switchboards). The path from DSL to HDL to analysis is the same stack we use now: spec -> compilation -> hardware execution -> witness logs.
- Provides portable design lessons. a) Prefer minimal DSLs at trust boundaries. b) Make invariants explicit and testable (decode(encode)=id). c) Treat procedures as code: version, lint, and review them. d) Capture deterministic traces for replayability and accountability.
- Reframes “crypto history” as “language history.” Turing’s notations, Welchman’s diagonal board, and crib algebra are early program semantics: naming states, composing transitions, and pruning search spaces. This lens connects wartime practice to today’s formal methods and secure-by-construction design.

In short, the DSL view clarifies why Enigma worked when faithfully used, why it failed under operational shortcuts, and how its ecosystem anticipated modern programming language design, verification, and hardware-software co-design.

2. The Enigma Artifact as Language

2.1 Alphabet, parameters, and state (rotor_order, rings, start, plugs, reflector)

Alphabet.
Finite set Sigma = {A..Z}. No digits, spaces, or punctuation in the core path; operators handle spacing externally.

Parameters (the program inputs).

- rotor_order: an ordered 3-tuple (or 4 for Naval M4) from a small catalog, e.g., (III, I, IV). Each rotor r has:
    - a fixed internal wiring permutation w_r: Sigma -> Sigma,
    - a turnover notch set N_r subset of Sigma indicating when the next rotor steps.
- rings (Ringstellung): a 3-tuple of ring offsets, e.g., (B, F, K). Shifts the labeling relative to the wiring; formally, a conjugation that reindexes w_r.
- start (Grundstellung): initial window positions, e.g., (Q, E, M). These are the live offsets that change as you type.
- plugs (Steckerbrett): up to 10–13 disjoint 2-cycles on Sigma, e.g., (A L)(P R)(T X). As a permutation, S_plug = product of disjoint transpositions; S_plug = S_plug^{-1}.
- reflector (Umkehrwalze): an involutive permutation U with no fixed points, U = U^{-1}, mapping Sigma to Sigma in 2-cycles (e.g., UKW-B).

State (the runtime).
S = (rotor_order, rings, pos, plugs, reflector), where pos is the current rotor window triple. Only pos evolves during execution; all other fields are constant for a message run. The rotor stack implements a composed permutation that depends on pos and rings.

Well-formedness (static checks).

- Rotor names exist; no duplicates if doctrine forbids them.
- Ring values and start positions are in A..Z.
- Plug pairs are disjoint and within A..Z.
- For naval variants, reflector compatibility is enforced.

These are the language's type rules: reject before execution.

2.2 Procedure as program: configure -> step -> transform -> emit

Enigma's program is a strict, single-instruction-cycle loop per keypress:

1. **configure**
   Load S_0 from the day's sheet: set rotor_order, rings, start -> pos_0, plugs, reflector.

2. **step**
   Advance pos with the double-step rule:

- Right rotor always steps each press.
- Middle rotor steps when its own notch is engaged (due to right rotor's position) *and* it causes the left rotor to step next press (the famous double step).
- Left rotor steps when the middle passes its notch.

This is the control flow of the language; it is the only branching.

3. **transform**
   Apply the composed permutation for the current state:

- Let A_pos be the position-induced shift (a rotation of indices), and A_ring the ring-induced relabeling.
- For rotors R = [R1(left), R2(mid), R3(right)] at positions p and rings r, define each effective rotor permutation as:
  w_eff(Ri) = shift(p_i)^{-1} o ring(r_i)^{-1} o w_Ri o ring(r_i) o shift(p_i).
- Forward path: S_plug -> w_eff(R3) -> w_eff(R2) -> w_eff(R1) -> U (reflector).
- Return path: w_eff(R1)^{-1} -> w_eff(R2)^{-1} -> w_eff(R3)^{-1} -> S_plug.
- Output letter is the image of the input under that composition.

Because S_plug = S_plug^{-1} and U = U^{-1}, the whole path is an involution for a fixed state S (decoding = encoding with same S).

4. **emit**
   Light the lamp for the output letter; append to ciphertext. Repeat.

Semantic invariants.

- For a fixed S: encipher_S(encipher_S(x)) = x.
- For a run: decode_S(ciphertext) = plaintext iff initial pos matches and the same procedure is followed.

- No letter can map to itself *for many configurations* when plugs and rotor wirings align in certain ways (historically noted constraint leaks); formally, self-maps are rare but not impossible in general rotor stacks without additional constraints.

Runtime trace.
The evolving pos and emitted letters form a deterministic trace given (S_0, plaintext). This is the language's execution log.

2.3 Key sheets as configuration-as-code

Key sheets are declarative programs: they specify *what* to enforce, not *how* to wire. They are minimal, audit-friendly, and replayable.

Typical structure (modernized, compact):

MACHINE

ROTOR: III-I-IV

RINGS: B-F-K

START: Q-E-M

PLUGS: A-L,P-R,T-X

REFL: B

PROC: double-encipher indicator; header->body; no word breaks

MESSAGE (operator adds)

INDICATOR: HZG # chosen at station; sent enciphered per PROC

TEXT: MEETATNINE SENDSUPPLIES

Why this is config-as-code:

- **Declarative fields** with finite domains and schema validity (lintable).
- **Versionable artifacts** (day, theater, service branch); diffs are meaningful.
- **Reproducible runs**: the tuple (config, indicator, ciphertext) is a complete witness; any simulator will arrive at the same output.

- **Role separation**: creators of the sheet (policy), operators (execution), and analysts (verification) share one source of truth.

Failure modes are configuration bugs:

- **Low entropy indicators** (e.g., repeated or predictable triples) reduce search space.
- **Procedure deviations** (violating PROC) leak structure (cribs).
- **Reuse of start or plugs** across contexts aids correlation attacks.
- **Ambiguous spec** (illegible handwriting, misread rotor order) is a parsing error leading to mismatched runs.

Modern parallels:

- TLS ciphersuites & parameters (cipher, mode, KDF, groups) as a small DSL.
- Firewall rule sets (iptables/P4) as constrained, ordered transforms on an alphabet (packet headers) with stateful stepping.
- Build files (Make/Bazel) where configuration orchestrates deterministic transforms and reproducible artifacts.

Best practices (then and now):

- Keep the grammar tiny and unambiguous.
- Validate preconditions (no overlapping plugs, legal rotor names).
- Log execution traces for replay (indicator, start, ciphertext).
- Rotate configs on a cadence; separate config authorship from runtime operation.
- Train to procedure; most breaches stem from procedural, not primitive, failures.

Net effect: Once we read the key sheet as a DSL program whose interpreter is an electro-mechanical VM, Enigma's strengths and weaknesses line up with today's lessons: small, strict languages improve assurance; sloppy ops turn strong primitives into weak systems.

3. A Minimal Enigma Grammar (EBNF)

We formalize a compact, auditable grammar for Enigma operator directives. The goal is: (i) a tiny, unambiguous syntax, (ii) statically checkable constraints, (iii) a clean AST suitable for interpreters, HDL emitters, and verifiers.

3.1 Concrete syntax for operator directives

Design principles

- Line-oriented, key: value pairs.
- Single machine block per file; optional message block(s).
- Strict tokens: A..Z letters only where letters are expected.
- Comments start with # and run to end of line.
- No implicit defaults; all required fields must appear.

EBNF (ISO-style, ASCII-safe)

```
File         ::= ws* MachineBlock (ws* MessageBlock)* ws*

MachineBlock  ::= "MACHINE" nl
                  RotorLine RingLine StartLine PlugLine ReflLine ProcLine?

MessageBlock  ::= "MESSAGE" nl
                  IndicatorLine TextLine+

RotorLine     ::= "ROTOR:" ws RotorList nl
RingLine      ::= "RINGS:" ws Triplet nl
StartLine     ::= "START:" ws Triplet nl
PlugLine      ::= "PLUGS:" ws (PlugList | "NONE") nl
ReflLine      ::= "REFL:"  ws Reflector nl
ProcLine      ::= "PROC:"  ws ProcSpec nl
```

```
IndicatorLine ::= "INDICATOR:" ws TripleLetters nl

TextLine      ::= "TEXT:" ws Letters nl


RotorList     ::= Rotor ( "-" Rotor )*

Rotor         ::= "I" | "II" | "III" | "IV" | "V" | "VI" | "VII" | "VIII" | "Beta" | "Gamma"


Triplet       ::= Letter ws? Letter ws? Letter

TripleLetters ::= Letter Letter Letter


PlugList      ::= PlugPair ( "," PlugPair )*

PlugPair      ::= Letter "-" Letter


Reflector     ::= "A" | "B" | "C" | "B-thin" | "C-thin"


ProcSpec      ::= ProcToken ( ws ProcToken )*

ProcToken     ::= "double-indicator" | "header-body" | "no-spaces" | "five-groups"


Letters       ::= Letter (Letter)*

Letter        ::= "A".."Z"


ws            ::= " " | "\t"

nl            ::= "\r"? "\n"
```

Example (valid)

```
MACHINE

ROTOR: III-I-IV
```

RINGS: B F K

START: Q E M

PLUGS: A-L,P-R,T-X

REFL: B

PROC: double-indicator five-groups

MESSAGE

INDICATOR: HZG

TEXT: MEETATNINE

TEXT: SENDSUPPLIES

Notes

- Rotor admits Beta/Gamma for naval M4 variants; validators will enforce consistency with Reflector later.
- ProcSpec is extensible; parsers should accept unknown tokens as warnings unless --strict is on.

3.2 Static checks: disjoint plugs, valid rotors, ranges

Static checks act like a type system. Reject early, fail loudly, and give actionable diagnostics.

C-1. Alphabet/range checks

- RINGS, START, INDICATOR: letters A..Z only.
- TEXT: letters A..Z only; if spaces are present, require no-spaces to be absent (or pre-strip at ingest with a warning).

C-2. Rotor catalog and arity

- ROTOR must list exactly 3 rotors for Army/Air Force (M3) or exactly 4 for Naval M4.
- Each named rotor must be in the catalog for the selected service/day; duplicates allowed only if doctrine permits (flag with warning otherwise).

- If 4 rotors are specified, the leftmost must be Beta or Gamma (or as per service rules), and Reflector must be a thin type (B-thin or C-thin).

C-3. Plugboard disjointness

- Each PlugPair uses distinct letters; across the whole PlugList, letters must be pairwise disjoint (no letter appears in more than one pair).
- Max pair count: enforce service doctrine (e.g., <=10 or <=13). If omitted, default to <=10 for safety with warning.

C-4. Procedural compatibility

- If double-indicator is set, require presence of INDICATOR in MESSAGE blocks; otherwise error.
- If five-groups is set, the executor should emit 5-letter groups; not a parse error but record in AST.

C-5. Reflector compatibility

- For M3: Reflector in {A, B, C}. For M4: Reflector in {B-thin, C-thin}. Mismatches are errors.

C-6. Doctrine-level lints (non-fatal unless strict)

- Low-entropy INDICATOR patterns (e.g., AAA, ABC): warn LINT-LOW-ENTROPY-IND.
- Reused START equals common defaults (e.g., AAA): warn.
- Suspicious TEXT trigrams (HEIL, WETTER) without pre-sanitization: warn about crib leakage (historical hygiene).

Error code taxonomy

- E-SYNTAX-* : lexical/grammar errors (unexpected token, missing field).
- E-TYPE-* : catalog/range violations (invalid rotor, reflector, arity).
- E-PLUG-* : plugboard disjointness/limit violations.
- E-PROC-* : contradictory or missing procedural requirements.
- E-M4-* : M4-specific constraints (thin reflector mismatch).
- W-LINT-* : hygiene warnings (entropy, patterns, style).
- W-EXT-* : unknown PROC token accepted in non-strict mode.

3.3 Parsing to an AST and error models

AST shape (JSON-like; comments for types)

```
{
  "type": "EnigmaSpec",
  "machine": {
    "rotors": ["III", "I", "IV"],         // list[str]
    "rings":  ["B", "F", "K"],            // list[str], len=3
    "start":  ["Q", "E", "M"],            // list[str], len=3
    "plugs":  [["A","L"],["P","R"],["T","X"]], // list[list[str]]
    "reflector": "B",                     // str
    "proc": ["double-indicator","five-groups"] // list[str]
  },
  "messages": [
    {
      "indicator": "HZG",                 // str, len=3
      "text": ["MEETATNINE","SENDSUPPLIES"]// list[str] lines
    }
  ],
  "meta": {
    "service": "auto",                    // inferred: M3 or M4
    "strict": true,                       // parser mode
    "source_hash": "sha256:..."           // reproducibility
  }
}
```

Elaboration phase (post-parse)

- Resolve rotor names to catalog entries:

- catalog["III"] -> { wiring: perm[26], notch: {'V'} }
- Compute arity: if 4 rotors present, set service="M4"; else service="M3".
- Normalize plugs to a permutation S_plug and verify involution property.
- Record constraints object for executors (e.g., turnover map per rotor).
- Attach lints array with non-fatal findings.

Parsing algorithm sketch

1. **Lexing**: produce tokens: KEY, COLON, ID, LETTER, HYPHEN, COMMA, NL; collapse runs of spaces/tabs into ws.
2. **Parsing**: LL(1) or a small Pratt parser suffices due to simple structure; enforce order inside MACHINE.
3. **Validation**: run static checks C-1..C-6, producing a typed MachineAST.
4. **Messages**: parse zero or more MESSAGE blocks; apply procedural checks (e.g., double-indicator requires INDICATOR).
5. **Elaboration**: resolve catalog, compute service, build permutations.

Error reporting model

- *Fail-fast for type errors*, accumulate all findings per section to reduce iteration cost.
- Each error includes:
    - code (e.g., E-PLUG-DUP-LETTER)
    - span (line:col..line:col)
    - message (human readable)
    - hint (corrective action)
    - example (when helpful)

Examples

Invalid: overlapping plug pairs

PLUGS: A-L,P-A

Report:

E-PLUG-DUP-LETTER at line 4: "A" appears in more than one plug pair.

Hint: Each letter may be present in at most one pair.

Invalid: M4 reflector mismatch

ROTOR: Beta-V-III-I

REFL: B

Report:

E-M4-REFLECTOR at line 5: Thin reflector required for 4-rotor stack (Beta/Gamma present).

Hint: Use REFL: B-thin or REFL: C-thin.

Warning: low-entropy indicator

INDICATOR: AAA

Report:

W-LINT-LOW-ENTROPY-IND at line 9: Indicator AAA is trivially guessable.

Hint: Choose uniformly at random from A..Z^3.

AST invariants (checked)

- rings.len == start.len == rotor_arity (3 or 4).
- plugs form a set of disjoint 2-cycles over A..Z.
- reflector in allowed set for inferred service.
- proc tokens deduplicated; unknown tokens preserved in proc_misc.

Executor interface

- Input: EnigmaSpec AST.
- Output: RunPlan with:
    - effective rotor order and turnover schedule,
    - plug permutation as 26x26 boolean or index map,
    - reflection permutation,
    - procedure flags (e.g., emit_five_groups: bool),
    - precomputed ring and position conjugation tables for fast encipher.

Testing strategy

- Golden files: valid/invalid samples with expected diagnostics.
- Property tests: for all random well-typed configs S and plaintext P, assert decode(encode(P,S),S) == P.
- Mutation tests: introduce single-character errors; expect specific error codes.

Compatibility and extensions

- The grammar admits future fields via additional KEY: lines; unknown keys produce W-EXT-KEY warnings but are preserved in meta.extensions.
- A CATALOG: stanza could be added to embed rotor/reflector definitions for research variants; validators would switch to the embedded catalog.

Outcome: This EBNF, static check suite, and AST/error model form a minimal, production-ready spec. It is small enough to implement twice (to cross-check), strong enough to prevent the historical failure modes, and clear enough to serve as the single source of truth for interpreters, HDL backends, and formal verifiers.

4. Operational Semantics

We give a small-step semantics for Enigma as a deterministic state machine over the alphabet A..Z. Positions are 0-based integers mod 26; letters map to {0..25}. The runtime state S contains fixed parameters and one evolving field, the rotor positions.

Let S = (R, ring, pos, plug, refl) where:

- R = [R1,R2,R3] (left..right; add R0 for M4 Naval if needed),
- ring = [k1,k2,k3] ring settings in 0..25,
- pos = [p1,p2,p3] window positions in 0..25 (evolves),
- plug is a permutation P over {0..25} with P = P^{-1},
- refl is a fixed involution U with U = U^{-1} and no fixed points.

We write a single keypress as a small-step transition:
(S, ch_in) -> (S', ch_out).

4.1 State transition: turnover and double-step rules

Notches. Each rotor Ri has a notch set N_i subset {0..25}. When the window of rotor i shows a notch value, rotor i+1 will step on the *next* press. Historical 3-rotor Army/Air Force: single notch per rotor. (Naval and some rotors may have two; rules are identical pointwise.)

Stepping order on each press (3-rotor case).
Let pos = [p1,p2,p3] (left..right) *before* the press.

1. Right rotor (R3) always steps: p3' = (p3 + 1) mod 26.
2. Middle rotor (R2) steps iff p3 was in N_3 *or* p2 was in N_2 (double-step trigger):
    - If p2 in N_2, then p2' = (p2 + 1) and (in step 3) left rotor will also step.
    - Else if p3 in N_3, then p2' = (p2 + 1).
    - Else p2' = p2.
3. Left rotor (R1) steps iff p2 was in N_2 (because the middle will carry over):
   p1' = (p1 + 1) if (p2 in N_2) else p1' = p1.

Collecting: pos' = [p1', p2', p3'].

Double-step phenomenon (worked example).
Suppose p2 in N_2 at time t (before press t):

- On press t:
    - Step 1: p3 increments.
    - Step 2: because p2 in N_2, p2 increments.
    - Step 3: because p2 in N_2 (checked on pre-step value), p1 increments.
      Thus p2 steps on press t.

Additionally, if after step t, p3' lands in N_3, then on press t+1 the middle steps again due to right-notch, causing p2 to step twice in two consecutive presses. This is the characteristic "double-step."

Naval M4 (four rotors).
Insert a thin, non-stepping leftmost rotor (Beta/Gamma) as R0 with pos0 fixed. Stepping logic applies to R1..R3 exactly as above; R0 never steps.

Ring setting effect on notches.
Notches are defined relative to the rotor's internal alphabet. If ring offset k_i shifts the labeling, the effective notch set at runtime is:

N_i^eff = { (n - k_i) mod 26 : n in N_i }.
All stepping checks use N_i^eff against the *pre-step* position p_i.

4.2 Enciphering as composed permutations: plug^-1 o rot^-1 o R o rot o plug

A single press performs:

1. compute pos' by stepping (Sec 4.1),
2. transform the input letter under a fixed permutation determined by (R, ring, pos').

Indexing.
Map input letter ch_in to x in {0..25}. Output index y maps back to ch_out.

Per-rotor effective wiring.
Each physical rotor Ri has a base permutation W_i over 0..25 (forward), and thus W_i^{-1} (reverse). Ring and position act by conjugation/shift.

For rotor i with ring offset k_i and position p_i' (the *post-step* position for this press), define:

- shift by s: Sh(s)(x) = (x + s) mod 26,
- ring conjugation: C_i = Sh(-k_i).

Effective forward mapping for Ri at this press:
F_i = Sh(-p_i') o C_i o W_i o C_i^{-1} o Sh(p_i').

Effective reverse mapping:
B_i = Sh(-p_i') o C_i o W_i^{-1} o C_i^{-1} o Sh(p_i').

Intuition: ring reindexes the wiring, position rotates the entry/exit contacts.

Whole-path permutation for one press.
Let P be the plugboard permutation, U the reflector, and rotors ordered left..right as [R1,R2,R3].

Forward path (rightward through stack):
x0 = P(x)
x1 = F_3(x0)
x2 = F_2(x1)
x3 = F_1(x2)

Reflect:
x4 = U(x3)

Reverse path (leftward back through stack):
x5 = B_1(x4)
x6 = B_2(x5)
x7 = B_3(x6)

Exit plug:
y = P(x7)

Thus ch_out = idx_to_letter(y).

Compact composition.
Let rot = F_1 o F_2 o F_3 (note: functional composition here is left-to-right by stage). The overall per-press permutation is:

Enc_S_press = P^{-1} o rot^{-1} o U o rot o P,

which matches the traditional “plug^-1 o rot^-1 o R o rot o plug” slogan (P = P^{-1} since P is an involution; we write P^{-1} for symmetry).

Self-inverse at fixed state.
For fixed (R, ring, pos') the mapping Enc_S_press is an involution:

- P = P^{-1}, U = U^{-1},
- (rot^{-1} o U o rot) is conjugate of U and hence also an involution,
- composition of involutions in this specific symmetric sandwich yields Enc_S_press^2 = id.

Caveat: across *multiple* presses the state changes (pos evolves), so the full-map over a message is not a single involution. Decryption succeeds by reproducing the *same* stepping sequence from the *same* start.

4.3 Determinism, invertibility, and replay logs

Determinism.
Given:

- machine parameters (R, ring, plug, refl),
- initial positions pos_0,
- plaintext sequence X = x_1..x_T in A..Z,
  there is a unique sequence of states pos_t and outputs Y = y_1..y_T produced by applying the stepping rule (Sec 4.1) and per-press permutation (Sec 4.2). No randomness

is used during a run (aside from human choice of message indicator before the run begins).

Invertibility (per message).
Let Exec(R,ring,plug,refl; pos_0; X) = Y with state trace pos_1..pos_T.

Then for decryption, running the *same* machine/config from the *same* pos_0 over Y yields X: Exec(R,ring,plug,refl; pos_0; Y) = X.

Sketch: Each press uses Enc_{pos_t}, an involution at that pos_t. Because both sender and receiver produce the same pos_t sequence from the same pos_0 by the same stepping rule, applying the same per-press involution in the same order recovers the input symbolwise.

Replayability and witness logs.
To make execution auditable and reproducible (modern “witness” idea), define a minimal append-only log per message:

header:

version: 1

rotor: [III, I, IV]

rings: [B, F, K] # as letters or 0..25

start: [Q, E, M]

plugs: [[A,L],[P,R],[T,X]]

reflector: B

proc_flags: ["double-indicator","five-groups"]

plaintext_len: T

hash_config: sha256(machine_block_canonical)

trace: # one row per press

- t: 1

pos_before: [Q,E,M] # letters or numbers (0..25)

step_flags: {L:false, M:false, R:true}

input: M

```
  output: X
  pos_after:  [Q,E,N]
- t: 2
  pos_before: [Q,E,N]
  step_flags: {L:false, M:maybe, R:true}
  input:  E
  output: Q
  pos_after:  [...]
...
footer:
 ciphertext: "XQKJM..."
 hash_full: sha256(header||trace||ciphertext)
```

Properties.

- *Deterministic replay:* Any compliant interpreter can recompute the same trace from header + plaintext (or verify ciphertext).
- *Local checkability:* At each row, you can verify that pos_after = step(pos_before) and output = Enc_{pos_after}(input).
- *Tamper evidence:* hash_full binds the whole run; hash_config binds the machine spec.
- *Privacy knobs:* In practice you may omit plaintext from the log and retain only (config, ciphertext) for receiver-side verification.

Complexity.
Per press cost is O(1) table lookups (a handful of array accesses). Over T letters: O(T). Stepping and conjugations are constant-time arithmetic mod 26.

Corner cases to test.

- Middle notch double-step: ensure p2 increments on consecutive presses across the notch passage; left increments exactly once at the crossover press.
- Multiple notches (if modeled): verify each notch independently triggers as per pre-step position.

- Ring offsets: verify that shifting the ring by +1 and position by -1 leaves effective contact alignment consistent (ring-position equivalence sanity check).
- Plug involution: ensure P(P(x)) = x and that overlapping plug pairs are statically rejected (Sec 3.2).

Takeaways.

- The semantics isolate all control flow into the stepping function and all data flow into a single conjugated-permutation sandwich.
- Determinism and per-press involution give clean proofs of correctness (decode(encode)=id) under matched runs.
- A simple witness log makes historical runs reproducible and modern implementations verifiable.

5. Algebraic Foundations

We model a single Enigma press at a fixed state as a permutation over the 26-letter alphabet. Let S_26 denote the symmetric group on {0,..,25}. All maps below are elements of S_26; composition is written as o, inverses as ^-1.

5.1 Modeling rotors and plugs in S_26

Rotors.
Each physical rotor Ri has a fixed wiring W_i in S_26 and (for the reverse path) W_i^-1. Ring offset k_i and window position p_i act by conjugation and shifts:

Sh(s)(x) = (x + s) mod 26

C_i = Sh(-k_i)

F_i(p_i,k_i) = Sh(-p_i) o C_i o W_i o C_i^-1 o Sh(p_i)

B_i(p_i,k_i) = Sh(-p_i) o C_i o W_i^-1 o C_i^-1 o Sh(p_i)

For a 3-rotor stack with rightmost index 3, define the forward rotor block

Rot = F_1 o F_2 o F_3

and the backward block

Rot^-1 = B_3 o B_2 o B_1

(These are mutual inverses by construction.)

Plugboard.
The plugboard is a product of disjoint transpositions (2-cycles). If the plug pairs are (a_1 b_1),...,(a_m b_m) with all letters disjoint, then

P = (a_1 b_1) (a_2 b_2) ... (a_m b_m)

Hence P = P^-1 and its cycle structure is entirely 2-cycles plus 1-cycles on unplugged letters.

Reflector.
U is an involution with no fixed points (a perfect matching on 26 letters):

U = (u_1 v_1) (u_2 v_2) ... (u_13 v_13), U = U^-1, U(x) != x for all x.

One-press map at fixed state.
With positions advanced for this press (Sec. 4), the effective enciphering permutation is:

E = P^-1 o Rot^-1 o U o Rot o P

This is the canonical “plug^-1 o rot^-1 o R o rot o plug” form.

5.2 Reflector as an involution; consequences for self-inverse behavior

Conjugation preserves involution.
For any g in S_26, define U' = g^-1 o U o g. Then:

U'^2 = g^-1 o U o g o g^-1 o U o g = g^-1 o U^2 o g = id.

Thus U' is also an involution. Since E is a conjugate of U by g = Rot o P, it follows that:

E^2 = id (at a fixed state)

This yields the per-press self-inverse property used by decryption (given the same state sequence).

No self-maps (“no letter enciphers to itself”).
Because U has no fixed points, conjugation preserves fixed-point counts: if x were fixed by E, then g(x) would be fixed by U. But U has none. Therefore E has no fixed points:

E(x) != x for all letters x.

This well-known Enigma property (a derangement each press) falls out immediately from group structure.

Parity.
Let sgn(·) be permutation parity. Transpositions are odd; a product of 13 disjoint transpositions is odd:

sgn(U) = (-1)^13 = -1.

Parity is invariant under conjugation, and sgn(P)=±1, sgn(Rot)=±1 cancel in pairs:

sgn(E) = sgn(P^-1) sgn(Rot^-1) sgn(U) sgn(Rot) sgn(P) = sgn(U) = -1.

So every per-press Enigma map is an odd derangement.

5.3 Safety properties provable from group structure

(S1) Decode(Encode) = id at matched states.
Given E_t for press t (positions stepped deterministically), encryption applies E_1,...,E_T; decryption applies the same sequence. Since each E_t is an involution and the sequences match, symbol-wise inversion holds:

E_t(E_t(x)) = x => E_T ... E_1 (E_1 ... E_T (x)) = x.

(S2) No plaintext letter can appear unchanged (per press).
Because E_t is a fixed-point-free involution, E_t(x) != x. Operationally, this forbids identity mappings and was historically exploitable (crib pruning).

(S3) Conjugacy class constraint.
All one-press maps E lie in the conjugacy class of U. Hence their cycle structure is always 13 disjoint 2-cycles (a perfect matching), merely relabeled by g = Rot o P. This sharply narrows the a priori search space: codebreakers can assume “odd derangement, all 2-cycles” without loss.

(S4) Plugboard invariants.
Conjugating by P before and after the rotor block preserves the derangement property and parity. Any valid plugboard (disjoint 2-cycles) yields an involutive, fixed-point-free E. Thus static plugboard checks (disjointness; at most m pairs) are sufficient to retain the algebraic guarantees.

(S5) Ring/position equivalence (sanity).
Because ring and position act by inner conjugations and shifts, the effective rotor maps satisfy:

F_i(p+1,k) = Sh(-1) o F_i(p,k+1) o Sh(1)

So changing ring vs. position in compensating ways yields conjugate effective maps. Implementations can use this to cross-check tables and catch off-by-one bugs.

(S6) Error localization and replay proofs.
Given a witness log of (pos_before, pos_after, input, output) per press, one can verify locally that:

pos_after = step(pos_before) (pure control)

output = E_pos_after(input) (pure data)

Since E_pos is a conjugate of U, checking E_pos^2 = id and "no fixed points" on each row is a constant-time certificate that the run obeys the Enigma semantics.

(S7) Provable limits on leakage (structure only).
The group-level constraints do not by themselves leak keys, but they do eliminate impossible hypotheses. For instance, any candidate mapping with a fixed point or wrong parity can be discarded immediately, shrinking search. This underlies crib-driven pruning and the logic of the Bombe.

(S8) Robustness to catalog choice.
Swapping to any valid rotor/reflector catalog merely changes W_i and U, but as long as U remains a fixed-point-free involution and W_i are bijections, all proofs above continue to hold. Thus the safety properties (involution, derangement, parity) are catalog-agnostic.

Bottom line: Modeling Enigma in S_26 makes its guarantees transparent: every press is an odd, fixed-point-free involution obtained by conjugating the reflector. That alone explains (i) why decryption is guaranteed when states match, (ii) why identity mappings never occur, and (iii) why attackers could exploit invariant structure to prune the key space—without touching the machine's mechanics.

6. Early Program Analysis in Practice

6.1 Turing/Welchman notations as proto-pseudocode

Abstraction of mechanics into logic.
Turing and Welchman replaced metal and wire with symbols and relations: letters became variables; rotor positions became parameters; Enigma's path became a composed function. Their worksheets used compact entailments like:

P[i] --E_t--> C[i] # plaintext → ciphertext at press t

E_t = P^{-1} ∘ Rot_t^{-1} ∘ U ∘ Rot_t ∘ P

Rot_t = F_1(t) ∘ F_2(t) ∘ F_3(t)

Chains and links.
Signals were represented as *chains* (linear letter relations) with *links* encoding equality constraints:

a ↔ b ↔ c … # a, b, c are letters at related places in the message

a = σ(b) # σ is a (partially known) permutation segment

Diagonal board as relational join.
Welchman's diagonal board added a second-order constraint: if plugboard maps x↔y, then every place x appears in any chain must pair with y. This is essentially a *global join* across all chains—relational algebra before SQL.

Operational pseudocode (modernized).

Given crib relations R = { (p_i, c_i) at offsets t_i }:

  Initialize Unknowns: rotor_order, ring, start, plugs

  For each relation r in R:

    Emit equation: c_i = E_{t_i}(p_i; Unknowns)

  Propagate equalities across all occurrences (diagonal constraint)

  Search Unknowns with pruning when any equation is unsatisfiable

Result: a human/machine-readable *specification* of Enigma runs, executed by people, Bombes, and eventually Colossus.

6.2 Bombe/Colossus wiring tables as HDLs

Bombe as a compiled circuit.
A Bombe menu (derived from crib alignments) compiled into a *wiring table* that instantiated many copies of Enigma's letter-transition constraints in parallel. This is a hardware description language in everything but name:

- **Modules:** rotor stacks at hypothesized offsets.
- **Nets:** letter contacts and plugboard equivalences.
- **Clocking:** stepping emulated mechanically/electrically across trials.
- **Assertions:** a contradiction lamp if any net demands x≠x.

HDL analogy (sketch).

```
module rotor_stack(input [4:0] x, output [4:0] y);
  // x,y ∈ 0..25 encoded in 5 bits
  // parameters: rotor, ring, pos
  // combinational table lookup = wirings + conjugations
endmodule

module bombe_menu(...);
  wire [4:0] n1, n2, n3, ...;   // letter nets
  rotor_stack A(.x(n1), .y(n2));
  rotor_stack B(.x(n3), .y(n4));
  // diagonal-board constraints as wire ties:
  assign n5 = n_alpha; assign n_beta = n_gamma;
  // contradiction detection:
  assert_unique(n_k);  // lights when violated
endmodule
```

Colossus as reconfigurable logic.
Colossus advanced this: switchboards + paper tape defined Boolean tests over streams (for Tunny/Lorenz), i.e., a *dataflow program*. Operators loaded *microprograms* by setting switches and patching cords—functionally equivalent to loading an FPGA bitstream.

Outcome: Bombe/Colossus workflows formalized the separation we still use: *spec → compile (menu/wiring) → execute (hardware) → observe assertions*.

6.3 Cribs and constraint propagation as search over state space

Crib = partial specification.
A crib provides aligned plaintext–ciphertext pairs at certain positions. Algebraically:

```
c_t = E_t(p_t; θ)          # θ are unknowns: rotor order, rings, start, plugs
```

Each equality restricts θ. Multiple equalities intersect to carve a tiny feasible subset out of an enormous key space.

Constraint graph.
Represent each letter occurrence as a node; edges encode known transforms (plug, rotor slice, reflector). A crib induces a *cycle* through these edges. Consistency requires the cycle to compose to identity. Any hypothesis θ that breaks identity is pruned.

Propagation algorithm (menu method).

Input: crib relations R, rotor-catalog, notch tables

For each rotor_order in catalog_permutations:

For each ring in 26^arity:

For each start in 26^arity (often searched by Bombe):

Initialize union-find over 26 letter nets

Apply diagonal constraints (plugboard equality)

For each relation r in R:

traverse path defined by r under (order, ring, start) to imply equality x = y

union(x, y)

if union creates contradiction (x ≠ x): prune branch

If no contradictions: emit candidate (order, ring, start, partial plug)

Heuristics that made it fast.

- **No fixed points per press:** discard any path implying E_t(x)=x.
- **Parity & cycle structure:** eliminate candidates with wrong permutation parity or cycle shapes.
- **High-value cribs:** stereotyped headers (e.g., weather) create longer, tighter cycles → stronger pruning.
- **Diagonal board:** global propagation couples distant constraints, exploding contradictions early.

Search as modern SAT/CSP.
Formally, we are solving a finite-domain CSP/SAT instance:

- Variables: rotor_order (small factorial), ring/start ($\mathbb{Z}/26$), plug pairs (matching on 26 nodes).
- Constraints: equalities induced by crib paths; disjointness for plugboard; turnover schedule.
- Engine: physical parallelism (Bombe drums) + human-guided branching → *complete* search on a drastically pruned space.

Guarantees & diagnostics.

- *Soundness:* any surviving θ satisfies all crib-implied equations (no contradictions).
- *Completeness (relative to crib):* if a key realizes the crib, it will be in the candidate set.
- *Explainability:* each elimination cites a violated equality on a specific path (a proof obligation), i.e., early "counterexample-guided pruning."

Synthesis.
Notations (proto-pseudocode) → menus/wirings (HDLs) → constraint search (CSP/SAT) form a clean toolchain. The same three layers—specification, hardware compilation, symbolic search—define modern verification and cryptanalysis pipelines. Enigma didn't just inspire computing; it rehearsed our entire programming workflow.

7. Lineage to Modern Languages and Systems

7.1 Assembly and microcode: mnemonic control of hardware state

Enigma's key sheet is mnemonic control of a finite state machine: rotor_order, ring, start, plugs, reflector define the machine state; each press is the sole instruction that both mutates state (step) and applies a combinational transform (encipher). This mirrors early assembly and microcode:

- **Assembly.** Mnemonics (e.g., LDA, ADD, JMP) are human-readable encodings of hardwired control paths. The Enigma DSL's identifiers (ROTOR, RINGS, START, PLUGS, REFL) are mnemonic handles for control fields that govern a single instruction's execution.

- **Microcode.** A microinstruction toggles control lines to steer data through functional units on a cycle. Likewise, one Enigma press routes a symbol through a fixed datapath whose routing is parameterized by config. The turnover logic is control flow; the permutation sandwich is dataflow.

Design lessons:

- A tiny instruction set with well-defined side effects is auditable. Enigma's single instruction (press) and explicit control fields anticipate RISC minimalism.
- Misconfiguration dominates failures: as with unsafe flags or errant microcode patches, small deviations from doctrine produce global, systemic weaknesses.

7.2 Regex and SQL: constrained, declarative engines

Regex and SQL exemplify declarative DSLs where the *engine* chooses the execution plan; users declare *what* rather than *how*. Enigma parallels this:

- **Regex.** Patterns compile to finite automata. The key sheet compiles to a fixed permutation network; the operator supplies text and the engine performs the run. Both languages prioritize predictability and bounded expressivity over Turing-completeness.
- **SQL.** A schema and query define constraints; the optimizer picks a plan. In Enigma, the schema is the catalog of rotors/reflectors; the "query" is the machine block; the plan is the effective per-press permutation given ring and start. Cribs at Bletchley act like constraints that eliminate impossible "plans" (keys) without enumerating all states.

Design lessons:

- Constrained syntax plus strong algebra (regular languages, relational algebra, permutation groups) enables correctness proofs and optimizer-like pruning.
- Deterministic semantics and small state reduce foot-guns; operators can reason locally about global effects, which is the core virtue of declarative DSLs.

7.3 Protocol specs and ciphersuites: declarative crypto DSLs

Modern secure systems are steered by configuration-as-code: TLS ciphersuites, AEAD choices, curves, KDFs, and modes. Enigma is the archetype:

- **Cipherspecs as programs.** TLS_AES_128_GCM_SHA256 is a programmatic directive that instantiates a pipeline (KDF -> key schedule -> AEAD). Enigma's ROTOR: III-I-IV, RINGS: B-F-K, PLUGS: ..., REFL: B instantiate the cipher's datapath.

- **Security posture via choice sets.** TLS 1.3 collapsed option sprawl to reduce misconfiguration risk; Enigma doctrine likewise constrained rotor reuse, plug count, and procedures to preserve assurance.
- **Operational hygiene.** Replayable transcripts (ClientHello etc.) serve as witness logs for postmortems; Enigma's headers/indicators and reproducible runs play the same role. Most incidents are not primitive breaks but config/ops failures (bad randomness, weak modes) — a historical constant from Enigma to today.

Design lessons:

- Prefer safe defaults and small choice surfaces; expose intent, not knobs.
- Treat procedures as code: lint, review, and version them; pair specs with conformance tests and executable references.

7.4 HDLs (Verilog/VHDL) and netlists: wiring as computation

Bombe menus and Colossus switchboards were proto-HDLs: human-authored wiring tables compiled into machines that implemented logical constraints. Today:

- **HDLs.** Verilog/VHDL describe structure and timing; tools generate netlists and bitstreams. Enigma's effective permutation per press is exactly a netlist composed of LUT-like tables (plugboard map, rotor tables, reflector), plus a tiny FSM for stepping.
- **FPGA thinking.** Changing a cable then is changing a routing constraint now. The "program" is the connectivity; the "execution" is signal propagation through that connectivity every cycle (keypress).
- **Verification.** Properties like involution and absence of fixed points become assertions. A UVM-style testbench for an Enigma core would replay witness logs and check local invariants per cycle.

Design lessons:

- Wiring-level specifications can be first-class programs when behavior is combinational plus small control; they are amenable to equivalence checking and property proofs.
- Separation of concerns — spec (menu/HDL), compile (layout/wiring), run (mechanism/FPGA), observe (lamps/assertions) — is the enduring co-design pattern from Bletchley to modern silicon.

Synthesis. Across these paradigms, the same virtues recur: a narrow, declarative surface; algebraic cores that admit proofs; deterministic traces; and cultural practices that treat configuration as code. Enigma's "language" sits at the root of this lineage.

8. Expressivity vs Assurance: A Design Trade-off

8.1 Turing-incomplete by design: why small languages are safer

Enigma's "language" is deliberately tiny: a finite alphabet, bounded parameter sets, and exactly one instruction (keypress) whose control flow is fully specified by the stepping rule. There is no unbounded recursion, no dynamic code loading, no user-defined control structures. In modern terms, it is **Turing-incomplete by construction**.

Why this helps:

- **Bounded state ⇒ bounded reasoning.** With only positions changing mod 26, every execution is a linear scan with constant-time steps. This makes both human and machine verification tractable.
- **Constrained surface area.** Few knobs (rotor_order, rings, start, plugs, reflector) mean fewer ways to be wrong. Compare to general languages where APIs, libraries, and control flow create combinatorial misconfiguration risks.
- **Algebraic core.** The semantics collapse to conjugations in S_26; properties like involution, odd parity, and fixed-point freedom become theorems, not test cases.
- **Audit-friendliness.** A small grammar and total semantics can be re-implemented independently to cross-check behavior. (In practice: two interpreters, one HDL core, differential tests.)
- **Principle of minimal expressivity at trust boundaries.** Where safety and correctness dominate (cryptography, access control, packet filters), **DSLs that can't express foot-guns** are preferable to powerful languages that merely *advise* you not to shoot yourself.

Design corollary: If a feature cannot be routinely verified and linted, consider removing it from the trusted language and pushing it to a higher layer with stricter interfaces.

8.2 Misuse vs break: operational failures and procedure hygiene

Historically, Enigma did not "break" because its core permutation algebra failed; it was **misused**. The same pattern recurs today: systems fall to weak keys, predictable nonces, bad rotations, and sloppy ops—not to textbook AES failures.

Typical Enigma-style failure modes (and modern analogues):

- **Low-entropy indicators → predictable nonces/IVs.** Reusing "AAA" indicators shrank the key space; reusing a GCM nonce collapses security.

- **Repeated phrasing (cribs) → structured, unredacted headers.** Weather boilerplate then; verbose, deterministic protocol banners now.
- **Procedure drift → config drift.** Deviating from "double-indicator" or grouping rules echoed today's habit of toggling "temporary" debug flags that never get removed.
- **Ambiguous artifacts → parse/validation gaps.** Illegible key sheets then; permissive YAML/JSON schemas now.

Hygiene, treated as code:

- **Make procedures executable.** Write the procedure as machine-checked policy (e.g., PROC: double-indicator five-groups) and reject runs that violate it.
- **Lint before run.** Enforce disjoint plugs, valid rotors, M4 reflector compatibility, indicator entropy, and banned plaintext trigrams. Fail closed.
- **Rotate and pin.** Rotate configs on cadence; pin catalogs/hashes so "same-named" rotors can't drift.
- **Two-person integrity.** Split duties: one prepares the machine block, another executes and signs the witness log.
- **Posture tests.** Run synthetic messages daily to prove decode(encode)=id on the live stack; this catches silent misalignment (off-by-one rings/start).

Principle: Treat *misuse* as your primary adversary. Engineer the language and tooling so wrong configurations are harder to author than right ones.

8.3 Reproducibility, audit trails, and "witness" execution

Assurance hinges on being able to **reproduce** a run and **prove** it obeyed the semantics. Enigma affords this because the execution is deterministic and the per-press map is locally checkable.

A minimal "witness" model:

- **Canonical spec.** The machine block is serialized deterministically (sorted keys, fixed whitespace) and hashed (hash_config).
- **Per-press trace.** For each keypress, record: t, pos_before, step_flags, input, output, pos_after. Hash the concatenated rows (hash_full).
- **Local verifiers.** A verifier, given the header and one row, checks:
    1. pos_after = step(pos_before) (control),
    2. output = Enc_{pos_after}(input) (data),

3. Enc_{pos_after}^2 = id and Enc_{pos_after}(x) ≠ x for all x (algebraic invariants).

- **End-to-end replay.** Anyone with (spec, plaintext) can re-derive the ciphertext and match hash_full; with (spec, ciphertext) they can decrypt if authorized, or at least validate the trace if plaintext is withheld.

Why this matters now:

- **Forensics without guesswork.** When incidents occur, the question isn't "what happened?" but "show the witness." Deterministic traces short-circuit he-said/she-said.
- **Independent re-implementation.** Two teams can implement the interpreter/HDL from the spec and achieve bit-identical runs; mismatches indicate either a defect or spec ambiguity, both of which are actionable.
- **Policy proofs at the edge.** Embedding invariants (e.g., "no fixed points," "strict five-group emission") as executable checks turns historical doctrine into enforceable policy.

A concise checklist (portable beyond Enigma):

1. **Language minimization:** keep the trusted surface Turing-incomplete and algebraic.
2. **Total semantics:** define small-step rules; forbid undefined behavior.
3. **Static gates:** reject malformed configs; lint hygiene issues with severity levels.
4. **Deterministic I/O:** normalize inputs; canonicalize specs; tie all artifacts to hashes.
5. **Witness by default:** emit per-step traces (or summarized commitments) and verify locally.
6. **Dual implementation:** maintain independent interpreter/HDL paths with differential tests.
7. **Ops as code:** version procedures, require reviews, and enforce two-person integrity.

Trade-off distilled: every increment of expressivity invites new classes of error and new proof obligations. Enigma shows that for safety-critical cryptographic control, **less language is more assurance**—provided the smaller language is paired with strong algebra, strict validation, and first-class witness execution.

9. Where the Analogy Breaks

9.1 No stored program or higher-order control flow

Enigma's "program" is a static configuration interpreted by fixed hardware; the only runtime evolution is rotor stepping. There is **no stored program** (no instruction memory), **no branching on data**, **no subroutines**, **no loops** beyond "press again." Consequently:

- **No higher-order behavior.** You cannot pass transformations as values, compose new operators at runtime, or abstract over procedures. All "composition" is baked into metal (wiring) and alignment (rings/positions).
- **No context sensitivity.** Output never changes policy based on input content; the device lacks guards like "if crib detected, then …". This sharply contrasts with CPUs/VMs where control depends on data.
- **No extensibility contracts.** Features can only be added by machining new rotors/reflectors or revising doctrine, not by linking a new code module.
- **Security upside and downside.** Upside: tiny attack surface; few emergent behaviors. Downside: no adaptive hardening (e.g., rate-limit on repeated patterns), no runtime checks beyond what operators enforce.

In short, Enigma is a **parameterized transducer**, not a stored-program computer. Treating it as a language is illuminating, but equating it with a programmable system overstates its dynamism.

9.2 Lack of abstraction boundaries and libraries

Modern languages survive complexity via **abstractions** (types, modules, interfaces) and **libraries** (curated, reusable components). Enigma has neither:

- **No types or modules.** All fields are raw symbols (A..Z). There is no type system to prevent, say, swapping "rings" with "start," or to prove that "plugs" are disjoint beyond out-of-band lint.
- **No encapsulation.** Every parameter is global and mutually visible; changing any one (plugboard, ring, start) perturbs the same permutation pathway. There are no "private fields" that confine errors.
- **No libraries or specs for variability.** You cannot import a "safe indicator generator" or a "crib-resistant emission policy." Doctrine lived on paper and in training, not in code with test suites.
- **No versioned interfaces.** A rotor named "III" is an agreement by convention; there is no semantic versioning to prevent "silent" catalog drift.

The result is a brittle system whose **complexity ceiling** is low. The machine's strength derives from algebraic simplicity, not from scalable software architecture; beyond a point, the only way to add capability is to complicate hardware or procedures (and thereby increase operator error).

9.3 Human factors: ritual, error, and coercion

Unlike modern systems that embed policy into code and verification, Enigma relied on **ritualized human practice**:

- **Ritual.** Repeated, memorized sequences (double-indicator, five-grouping, daily changes) stood in for enforceable invariants. Ritual builds muscle memory but is fragile under stress and turnover.
- **Error.** Most historical compromises were **operational**: low-entropy indicators, repeated phrases, misread settings, out-of-order steps. With no runtime guardrails, the device happily executed unsafe rituals.
- **Coercion and hierarchy.** Military command structures enforced speed and obedience, sometimes at odds with cryptographic hygiene (e.g., pressure to reuse settings for throughput). In modern terms: incentive misalignment and unsafe SLAs.
- **Audit scarcity.** There was no ubiquitous, tamper-evident logging. Post-hoc reconstruction depended on intercepted traffic and human recollection. Today we expect cryptographic transcripts, reproducible builds, and signed attestations.

These human realities limit the analogy to programming. A modern secure language **internalizes** safety—types, permissions, proofs, tests—so that the *default behavior is correct*. Enigma **externalized** safety into training and discipline. The core lesson travels (minimize expressivity, maximize determinism), but the **sociotechnical substrate** was fundamentally different: people were the type system, review process, and CI pipeline, and people are fallible.

10. A Modernized Enigma Toolchain (Design Sketch)

10.1 Reference grammar and interpreter

Goals. Single source of truth, tiny surface, total semantics. Two independent implementations (to cross-check): (A) a reference interpreter in a memory-safe language; (B) a second, minimal interpreter in another language.

Repo layout.

```
enigma/
  spec/
    enigma.ebnf            # authoritative grammar
    catalog.json           # rotor/reflector tables, notches, service rules
    schema.machine.json    # JSON Schema for canonicalized AST
  ref/
    parser/                # lexer+parser -> AST
    interp/                # small-step engine
    cli/                   # `enig run`, `enig check`, `enig log`
  hdl/
    verilog/               # generated RTL, testbench templates
  tests/
    golden/                # valid/invalid samples + expected diagnostics
    props/                 # property-based generators
  docs/
    spec.md                # human-readable spec
    invariants.md          # algebra + proofs (lightweight)
```

Grammar (authoritative excerpt, ASCII-safe).

- Same as Section 3 EBNF; MACHINE + optional MESSAGE blocks; strict tokens; comments # ....

Catalog (catalog.json).

```
{
  "rotors": {
    "I":    {"wiring": "EKMFLGDQVZNTOWYHXUSPAIBRCJ", "notches": ["Q"]},
    "II":   {"wiring": "AJDKSIRUXBLHWTMCQGZNPYFVOE", "notches": ["E"]},
    "III":  {"wiring": "BDFHJLCPRTXVZNYEIWGAKMUSQO", "notches": ["V"]},
    "IV":   {"wiring": "ESOVPZJAYQUIRHXLNFTGKDCMWB", "notches": ["J"]},
    "V":    {"wiring": "VZBRGITYUPSDNHLXAWMJQOFECK", "notches": ["Z"]},
    "Beta": {"wiring": "LEYJVCNIXWPBQMDRTAKZGFUHOS", "notches": []},
    "Gamma":{"wiring": "FSOKANUERHMBTIYCWLQPZXVGJD", "notches": []}
  },
  "reflectors": {
    "A":      "EJMZALYXVBWFCRQUONTSPIKHGD",
    "B":      "YRUHQSLDPXNGOKMIEBFZCWVJAT",
    "C":      "FVPJIAOYEDRZXWGCTKUQSBNMHL",
    "B-thin": "ENKQAUYWJICOPBLMDXZVFTHRGS",
    "C-thin": "RDOBJNTKVEHMLFCWZAXGYIPSUQ"
  },
  "rules": {
    "M3": {"arity": 3, "allowed_reflectors": ["A","B","C"]},
    "M4": {"arity": 4, "leftmost_thin": true, "allowed_reflectors": ["B-thin","C-thin"]}
  }
}
```

Interpreter (small-step core).

- Types:
    - Machine: {rotors[3|4], rings[3|4], start[3|4], plugs: transposition set, reflector}
    - State: Machine + pos[3|4]
- Functions:
    - step(pos, notches, rings) -> pos' (double-step rule, notches adjusted by rings)
    - enc_press(M, pos, letter) -> (pos', out)
    - enc_stream(M, pos0, text) -> (posT, cipher)
- Deterministic, side-effect-free core; CLI handles I/O, grouping, logging.

CLI.

```
enig check   machine.enig            # parse + static checks + lint
enig run     machine.enig < plaintext  # emit ciphertext + witness log
enig verify  log.json                # verify per-press invariants and hash
enig sim     --catalog catalog.json --json ast.json --text file.txt
```

Canonical AST (after parse+elaboration).

- JSON per Section 3.3; validated by schema; all letters normalized to 0..25 ints for engine.

10.2 Dual backends: software simulator and HDL emitter

Software simulator (ref/interp).

- Table-driven: precompute 26x26 matrices for each rotor at each (ring, pos) pair or compute on the fly with O(1) shifts.
- Throughput target: 100+ MB/s on commodity CPUs; vectorize lookups if desired (optional).

HDL emitter (hdl/verilog).

- Generate synthesizable Verilog from the AST/catalog:
    - plug.v: 26-entry ROM (or combinational case) for plugboard mapping (involution).

- rotor.v: parameterized module for forward/reverse maps; ring+pos handled by add/sub modulo 26.
- reflector.v: 26-entry ROM (involution, no fixed points).
- fsm.v: 3-state FSM for stepping (computes next pos with double-step logic).
- enigma_core.v: wires plug -> R3 -> R2 -> R1 -> reflector -> R1^-1 -> R2^-1 -> R3^-1 -> plug.

Module sketch (simplified).

```
module enigma_core(
  input        clk,
  input        valid_in,
  input  [4:0]  ch_in,         // 0..25
  input  [4:0]  pos_in [2:0],   // left..right
  output       valid_out,
  output [4:0]  ch_out,
  output [4:0]  pos_out [2:0]
);
  wire [4:0] s0 = plug(ch_in);
  wire [4:0] s1 = rotor_fwd(R3, pos_in[2], ring3, s0);
  wire [4:0] s2 = rotor_fwd(R2, pos_in[1], ring2, s1);
  wire [4:0] s3 = rotor_fwd(R1, pos_in[0], ring1, s2);
  wire [4:0] s4 = reflect(s3);
  wire [4:0] s5 = rotor_rev(R1, pos_in[0], ring1, s4);
  wire [4:0] s6 = rotor_rev(R2, pos_in[1], ring2, s5);
  wire [4:0] s7 = rotor_rev(R3, pos_in[2], ring3, s6);
  assign ch_out = plug(s7);

  // stepping FSM computes pos_out given pos_in and notches, ring offsets
```

```
  // valid_out asserted next cycle after valid_in
endmodule
```

Testbench template.

- Drives a known MACHINE config and plaintext stream; checks per-cycle:
    - pos_after == step(pos_before).
    - ch_out equals golden produced by software simulator.
    - Assertions: (i) self-inverse at fixed pos; (ii) no fixed-point maps in any cycle.

Build targets.

```
make sim-sw       # run software golden tests
make rtl-gen      # emit Verilog from catalog+AST variants
make sim-rtl      # iverilog/questa run against witness logs
make ci           # end-to-end: parse -> run -> log -> verify -> RTL check
```

10.3 Property tests and lightweight proofs (turnover invariants; decode(encode(m)) = m)

Property-based testing (QuickCheck/Hypothesis style).

- Generators:
    - Random well-typed machines: choose rotor_order (no duplicates unless allowed), uniform rings/start in 0..25, random plug matchings (<= m pairs), reflector consistent with arity.
    - Random plaintexts of varying length (including 0, 1, boundary lengths around notch positions).
- Properties:

1. decode(encode(P, M), M) == P
2. For each press: E_pos^2 == id (for all 26 letters) and E_pos(x) != x.
3. Stepping correctness: implement step_ref (pure spec) and assert engine FSM matches for all states.
4. Ring/position equivalence sanity:

5. encode(P, rings=(k1+1,...), start=(p1-1,...)) == encode(P, rings=(k1,...), start=(p1,...)) up to conjugacy

(Checked via per-press permutation comparison modulo shifts.)

6. M4 constraint: leftmost Beta/Gamma never steps; thin reflector enforced.

Golden vectors (deterministic).

- Include historically documented daily keys (public-domain) and known plaintexts; assert exact ciphertext match with spaces/grouping rules.
- Edge-case sequences crossing double-step boundaries (e.g., middle rotor notch on consecutive presses).

Lightweight proofs (machine-checkable notes).

- **Turnover invariant.** Define pre-step position p and effective notch set N_eff = { (n - k) mod 26 }. Prove by case-split:
    - If p2 in N2_eff, then p1' = p1 + 1, p2' = p2 + 1 (double-step trigger).
    - Else if p3 in N3_eff, then p2' = p2 + 1.
    - Else only p3 steps.
      Provide a small TLA+/Alloy model (optional) that explores all 26^3 states and checks the rule matches the FSM.
- **Decode(encode) = id.** For fixed press state, E = P^-1 o Rot^-1 o U o Rot o P is a conjugate of U, hence E^2 = id. Across a run, both sides generate identical pos_t, therefore symbol-wise involution composes to identity. Supply a two-line equational proof and an executable check:
- forall t, letter l: E_t(E_t(l)) = l
- **No fixed points.** Since U has no fixed points and E is its conjugate, E has none. Include an executable assertion in tests that for every pos: for all 26 letters, E_pos(x) != x. This doubles as a catalog sanity check (catches miswired reflectors).

Witness logs as test oracles.

- Every enig run emits log.json containing per-press traces and content hashes.
- enig verify log.json:
    - recomputes pos transitions,

    - reconstructs E_pos and checks involution + no-fixed-point,
    - replays the entire stream and matches hash_full.
- CI consumes a corpus of logs (short/long, M3/M4, boundary cases) to validate both software and RTL.

Negative tests (expected failures).

- Overlapping plug pairs -> E-PLUG-DUP-LETTER.
- M4 with non-thin reflector -> E-M4-REFLECTOR.
- Low-entropy indicators -> W-LINT-LOW-ENTROPY-IND (warning).
- Intentionally mis-specified notch to ensure property (no-fixed-point) fails and is caught.

Assurance posture summary.

- Small, total language; two independent engines; algebraic invariants executable as assertions; property-based fuzzing over the full state space; witness-logged runs; HDL equivalence against software golden. This yields a practical, verifiable chain from human-readable spec to bit-accurate execution—faithful to Enigma's spirit, but with modern reproducibility and safety.

11. Case Study: From DSL Spec to Verified Execution

11.1 Compiling a sample "key sheet" to permutations

Input (DSL)

MACHINE

ROTOR: III-I-IV

RINGS: B F K

START: Q E M

PLUGS: A-L,P-R,T-X

REFL: B

PROC: double-indicator five-groups

MESSAGE

INDICATOR: HZG

TEXT: MEETATNINE

TEXT: SENDSUPPLIES

Step A — Parse -> AST (canonicalized)

```
{
  "rotors": ["III","I","IV"],
  "rings":  [1,5,10],        // B,F,K -> 1,5,10 (A=0)
  "start":  [16,4,12],       // Q,E,M -> 16,4,12
  "plugs":  [["A","L"],["P","R"],["T","X"]],
  "reflector":"B",
  "proc": ["double-indicator","five-groups"],
  "service":"M3"
}
```

Step B — Static checks

- Arity 3, reflector B allowed (OK).
- Plug pairs disjoint; <= 10 pairs (OK).
- Rings/Start in 0..25 (OK).
- double-indicator present and INDICATOR provided (OK).

Step C — Elaborate catalog -> concrete tables

- Load rotor base wirings and notches:
    - III: W_III, notch V (21)
    - I: W_I, notch Q (16)
    - IV: W_IV, notch J (9)
- Reflector U = UKW-B.
- Build plug permutation P as product of transpositions: (A L)(P R)(T X).

Step D — Precompute effective rotor maps
For each rotor Ri with ring k_i and (per-press) position p_i, the forward map is:

F_i(p_i,k_i) = Sh(-p_i) o Sh(-k_i) o W_i o Sh(k_i) o Sh(p_i)

Backward map:

B_i(p_i,k_i) = Sh(-p_i) o Sh(-k_i) o W_i^{-1} o Sh(k_i) o Sh(p_i)

(Sh(s)(x) = (x + s) mod 26.)

At t=1 (first press), positions step first (Sec. 4):

- Compute pos_1 = step(pos_0=[16,4,12]) using notch-adjusted rule.
- With pos_1 fixed, instantiate F_1,F_2,F_3 and B_1,B_2,B_3 for that press.

Step E — One-press permutation
Define Rot = F_1 o F_2 o F_3. Then:

E_press = P^{-1} o Rot^{-1} o U o Rot o P

This is a 26-cycle table (really 13 disjoint 2-cycles); cache it per press.

Deliverables from compile

- A small struct with: notches (adjusted by rings), step() function, P, U, and closures to materialize F_i/B_i from (p_i,k_i).
- Optional per-press cache: E_pos for any (p1,p2,p3) observed during the run.

11.2 Executing and logging with deterministic replays

Execution loop

pos = start

cipher = []

for each line in TEXT (concatenated, stripped if PROC says no-spaces):

  for ch in line:

    pos' = step(pos)

    out = E_{pos'}(ch) // via P, F_i/B_i, U as above

    append(out, cipher)

    log row = {t, pos_before=pos, pos_after=pos', input=ch, output=out, step_flags}

```
      pos = pos'
group cipher into 5s if PROC includes five-groups
```

Witness log (excerpt)

```
{
  "header": {
    "version": 1,
    "rotors": ["III","I","IV"],
    "rings": [1,5,10],
    "start": [16,4,12],
    "plugs": [["A","L"],["P","R"],["T","X"]],
    "reflector":"B",
    "proc": ["double-indicator","five-groups"],
    "hash_config": "sha256:7f66...c3"
  },
  "trace": [
    {
      "t": 1,
      "pos_before": [16,4,12],
      "step_flags": {"L": false, "M": false, "R": true},
      "pos_after":  [16,4,13],
      "input": "M",
      "output":"X",
      "checks": {"involution_ok": true, "no_fixed_points": true}
    },
    {
      "t": 2,
```

```
    "pos_before": [16,4,13],
    "step_flags": {"L": false, "M": notch_R3, "R": true},
    "pos_after":  [16,5,14],
    "input": "E",
    "output":"Q",
    "checks": {"involution_ok": true, "no_fixed_points": true}
   }
   // ...
 ],
 "footer": {
   "ciphertext": "XQKJM ...",
   "hash_full": "sha256:b1a2...9d5"
 }
}
```


Deterministic replay

- Given header + plaintext, any conforming engine recomputes trace and ciphertext, matching hash_full.
- Given header + ciphertext, receiver decodes (same pos stepping) to recover plaintext; verifier can still check per-row invariants without plaintext.

Local verifications per row

1. pos_after == step(pos_before) — control-integrity.
2. E_{pos_after}^2 == id (spot-check by applying to all 26 letters or a fixed subset).
3. E_{pos_after}(x) != x for all x — derangement check.
4. output == E_{pos_after}(input) — data-integrity.

These checks make the log self-certifying.

11.3 Failure modes and diagnostics

A) Static failures (reject before run)

1. **Overlapping plug pairs**

PLUGS: A-L,P-A

Diag:

E-PLUG-DUP-LETTER line 4: letter 'A' appears in multiple plug pairs.

Hint: Each letter may occur in at most one pair.

2. **M4 reflector mismatch**

ROTOR: Beta-V-III-I

REFL: B

Diag:

E-M4-REFLECTOR line 5: Thin reflector required for 4-rotor stack.

Use REFL: B-thin or REFL: C-thin.

3. **Arity/range**

RINGS: B F 42

Diag:

E-TYPE-RANGE line 3: ring '42' not in A..Z.

4. **Procedure inconsistency**
   PROC: double-indicator but INDICATOR missing in MESSAGE.
   Diag:

E-PROC-REQUIRED: double-indicator requires INDICATOR in MESSAGE.

B) Runtime failures (bad doctrine; caught by lints or assertions)

1. **Low-entropy indicator**

INDICATOR: AAA

Diag:

W-LINT-LOW-ENTROPY-IND: indicator AAA is guessable; choose uniformly in A..Z^3.

2. **Crib leakage (stereotyped text)**
   TEXT contains WETTER or HEIL without pre-sanitization.
   Diag:

W-LINT-KNOWN-CRIB: stereotyped trigram 'WETTER' raises traffic analysis risk.

3. **Involution/fixed-point violation (indicates catalog bug)**
   If miswired reflector or rotor table loaded, per-row check fails:

E-SEM-INVOLUTION: E_pos^2 != id at t=137 (catalog/implementation mismatch).

or

E-SEM-FIXEDPOINT: E_pos maps 'G' -> 'G' at t=42; reflector must be fixed-point-free.

C) Divergence between implementations (software vs HDL)

- **Mismatch in stepping (double-step off-by-one)**
  RTL pos advances differ from software golden around middle-notch boundary.
  Diag:

E-STEP-DIVERGENCE t=101: sw=[p1=7,p2=16,p3=3], rtl=[7,15,3].

Hint: check pre-step notch evaluation vs ring offset.

- **ASCII vs index mapping error**
  If a letter-index conversion off-by-one occurs:

E-ENC-MAP-OFFBYONE: 'A' encodes differently (sw='X', rtl='W') at t=1.

D) Replay verification failures (tamper or operator error)

- **Hash mismatch**

E-LOG-HASH: hash_full does not match recomputed value; log may be tampered.

- **Trace doesn’t match recomputed stepping**

E-TRACE-STEP: pos_after in row 238 inconsistent with step(pos_before).

Possible manual edit or device desync during run.

Remediation playbook

1. **Fail closed** on E-*: do not emit ciphertext; produce diagnostics and suggested fix.
2. **Warn on W-***: emit ciphertext but annotate log; require sign-off for deployment traffic.

3. **Dual-path diff**: when RTL and SW disagree, dump minimal counterexample (header + short plaintext causing first divergence), then bisect.
4. **Catalog pinning**: require catalog.json hash in header; refuse run if unpinned or mismatched.

What "Verified Execution" means here

- *Semantic soundness*: per-press maps are provably conjugates of a fixed-point-free involution (U); checks enforce it.
- *Determinism*: same inputs produce the same trace and outputs.
- *Reproducibility*: independent engines (software, RTL) agree at the bit level.
- *Accountability*: the witness log plus hashes constitute a portable, tamper-evident proof of what was executed.

Outcome
This case study shows the full pipeline: a human-readable DSL spec compiles to concrete permutations; execution produces ciphertext and a witness; independent verifiers replay and attest correctness; and well-scoped diagnostics convert historical "doctrine errors" into actionable, machine-checked feedback.

12. Implications for Language and Security Design

12.1 Minimal DSLs for safety-critical pipelines

Premise. When correctness and auditability trump expressivity, prefer a Turing-incomplete DSL with total, algebraic semantics—exactly the Enigma pattern.

Design pattern (Minimal-DSL stack):

- **Surface:** tiny key:value grammar; no implicit defaults; explicit arity; comments only.
- **Static layer:** schema + lints → reject ill-typed artifacts (ranges, catalog membership, disjointness).
- **Semantic core:** closed algebra (e.g., permutations, regular sets, relational algebra) with proofs-as-invariants.
- **Execution:** deterministic interpreter with a single control path (or few), no dynamic code loading.
- **Attestation:** per-step witness logs + content hashes.

Where this fits today:

- **Crypto policy** (KMS/HSM templates): alg, key size, mode, rotation cadence, AAD policy.
- **Network guards** (P4/ACL/iptables): ordered transforms over a finite header alphabet.
- **Build/CI release gates:** signed steps with finite state transitions (allow/deny/promote).

Anti-patterns to avoid:

- Embedding general scripting (loops, HTTP calls) inside policy languages.
- Optional fields with magical defaults.
- Runtime mutability of spec (self-modifying "policy").

Success criteria:

- Two independent implementations agree bit-for-bit.
- Every construct has a small-step rule; no "undefined behavior."
- A novice can predict outcomes locally (compositional mental model).

12.2 Configuration provenance and cryptographic accountability

Premise. If "procedure is code," then **provenance is supply chain**. Track who authored what, with what catalog, and prove what actually ran.

Provenance model (portable):

- **Canonicalization:** deterministic serialization of the config (sorted keys, normalized whitespace, normalized encodings).
- **Identity & intent:** signer identity, purpose, and expiration baked into the header.
- **Pin catalogs/artifacts:** include cryptographic digests for rotor/reflector tables (modern: alg suites, keysets, container images).
- **Witness logs:** append-only per-step traces or succinct commitments (Merkle roots) binding inputs, state, outputs.
- **Dual attestation:** software and hardware (e.g., FPGA/TEE) each sign the same run; mismatches trigger quarantine.

Operational hooks:

- **Pre-flight:** check must pass (schema + lints) before promotion; refusal is fail-closed.
- **Runtime:** run emits witness; verify replays locally or on a separate verifier.
- **Post-facto:** logs are immutable, time-stamped, and anchored (e.g., transparency log).

Metrics to watch:

- % of runs with full witness present and verified.
- Mean time to root cause (witness-assisted).
- Drift rate: # of configs not pinned to catalog hash.

Accountability payoffs:

- **Forensics:** disputes collapse to hash comparisons and per-step checks.
- **Compliance:** auditors can replay representative runs without privileged access.
- **Safety culture:** operators learn to think in specs and proofs, not runbooks and lore.

12.3 Lessons for secure defaults and usable correctness

Premise. Most failures are **misuse**, not **primitive break**. Enforce safe behavior by default; make the right thing the easy thing.

Secure-by-construction defaults:

- **Minimal choice surfaces:** collapse option sprawl (e.g., fixed AEAD + fixed KDF) unless a justification is signed.
- **Entropy gates:** refuse low-entropy indicators/nonces; build in high-quality randomness providers.
- **Deterministic grouping/normalization:** remove operator variability (e.g., five-grouping, whitespace policy).
- **Safe arity and catalogs:** forbid dangerous combos (e.g., M4 without thin reflector → hard error).

Usability patterns (so humans succeed):

- **Explaining linters:** diagnostics with span, cause, fix, example (not just “invalid”).

- **Dry-run everything:** show the first N steps of execution (positions, outputs) before committing.
- **One-click witnesses:** logs emitted automatically; verification is a single command.
- **Guardrails not guidelines:** encode doctrine as validators; warnings for hygiene, errors for hazards.

Institutional practices:

- **Two-person integrity on config changes**; cryptographic sign-off required to run.
- **Golden vectors & property tests** in CI; every release must satisfy decode(encode)=id, no-fixed-points, and turnover invariants.
- **Independent re-implementation** mandate (software + HDL/TEE) with continuous differential testing.
- **Rotation & pinning SLAs:** routine rotation cadence enforced by the tool; rejects stale specs.

Red flags (design smells):

- A policy file that secretly enables code execution.
- Defaults that are insecure "for compatibility."
- Ambiguous text specs without an executable reference.
- Logs that can't be independently verified.

Bottom line. Enigma's austere "language" shows how **small, algebraic DSLs + strict validation + witness execution** deliver assurance out of proportion to complexity. Modern security design should follow suit: restrict expressivity at the trust boundary, make every run replayable, and shift correctness from human ritual to machine-checked invariants.

13. Conclusion: The Cipher That Taught Us to Program

13.1 Historical reframing

Enigma is typically narrated as hardware bravado and cryptanalytic genius. Reframed as a language artifact, it becomes an early lesson in how humans **specify** behavior for machines and how machines **interpret** constrained intent. Key sheets read as programs; rotors, plugs, and reflectors as a compact algebra; Bombe menus and Colossus switchboards as compiled circuits; cribs as declarative constraints. This view links wartime practice to the modern stack: grammar

→ semantics → compilation → execution → verification. The story is not that Enigma "was a computer," but that its ecosystem **forced** the habits that still underwrite trustworthy computation: minimal surfaces, deterministic semantics, reproducible runs, and analysis-by-constraints instead of brute force.

13.2 Practical patterns to carry forward

- **Prefer tiny DSLs at trust boundaries.** Keep policy languages Turing-incomplete with total semantics and algebraic cores; prove invariants instead of hoping tests hit them.
- **Treat procedure as code.** Version, lint, and pin catalogs; make misconfiguration harder than correct configuration.
- **Exploit structure for pruning.** Encode invariants (parity, involution, no fixed points) as executable assertions; discard impossibilities early, as Bletchley did.
- **Dual implementations for assurance.** Independent software + hardware (or TEE/FPGA) must agree bit-for-bit; disagreements produce actionable counterexamples.
- **Witness by default.** Emit per-step traces or cryptographic commitments; make replay and local checking a one-command act.
- **Human factors as first-class.** Replace ritual with guardrails: entropy gates, safe defaults, dry-runs, and explicit error models.

13.3 Future work: formalizing historical machines as living specs

Turning artifacts into executable, verifiable specifications would both honor history and sharpen contemporary practice:

1. **Open catalogs and reference grammars.** Publish rotor/reflector tables, notches, and doctrine as signed, versioned data with canonical EBNF and JSON schemas.
2. **Machine-checked semantics.** Small-step models in TLA+/Alloy/Coq for stepping rules and permutation properties; generate test oracles directly from proofs.
3. **Multi-backend toolchains.** From one spec, emit: a reference interpreter, synthesizable HDL, and a portable verifier that checks witness logs.
4. **Corpus of witness logs.** Curate public, reproducible runs (including double-step edge cases) as an assurance benchmark—usable for pedagogy, regression tests, and competitions.
5. **Comparative archaeology.** Apply the same DSL/semantics lens to Lorenz/Colossus, SIGABA, Typex, and Hagelin machines; map where guarantees diverge and why.

6. **Translational templates.** Package the pattern (minimal DSL + invariants + witnesses) for modern domains—KMS policies, network filters, release pipelines—so teams can adopt "Enigma-style assurance" without reinventing it.

Read this way, Enigma is not a relic but a **template**: a compact language, a rigorous semantics, and a culture of verifiable execution. The cipher that once hid meaning now illuminates how we should design systems that are small, checkable, and honest about what they do.

Appendices

A. Full EBNF and AST schema

A.1 Complete EBNF (ISO-style, ASCII-safe)

```
File         ::= ws* MachineBlock (ws* MessageBlock)* ws*

MachineBlock  ::= "MACHINE" nl
                  RotorLine RingLine StartLine PlugLine ReflLine ProcLine?

MessageBlock  ::= "MESSAGE" nl
                  IndicatorLine? TextLine+

# Lines
RotorLine     ::= "ROTOR:" ws RotorList nl
RingLine      ::= "RINGS:" ws Triplet nl
StartLine     ::= "START:" ws Triplet nl
PlugLine      ::= "PLUGS:" ws (PlugList | "NONE") nl
ReflLine      ::= "REFL:"  ws Reflector nl
ProcLine      ::= "PROC:"  ws ProcSpec nl

IndicatorLine ::= "INDICATOR:" ws TripleLetters nl
TextLine      ::= "TEXT:" ws Letters nl

# Tokens and composites
RotorList     ::= Rotor ( "-" Rotor )*
Rotor         ::= "I" | "II" | "III" | "IV" | "V" | "VI" | "VII" | "VIII" | "Beta" | "Gamma"
```

```
Triplet      ::= Letter ws? Letter ws? Letter
TripleLetters ::= Letter Letter Letter

PlugList     ::= PlugPair ( "," PlugPair )*
PlugPair     ::= Letter "-" Letter

Reflector    ::= "A" | "B" | "C" | "B-thin" | "C-thin"

ProcSpec     ::= ProcToken ( ws ProcToken )*
ProcToken    ::= "double-indicator" | "header-body" | "no-spaces" | "five-groups"

Letters      ::= Letter (Letter)*
Letter       ::= "A".."Z"

ws           ::= " " | "\t"
nl           ::= "\r"? "\n"

# Comments ignored anywhere: "#" {any char except NL} NL
```

Parse-time constraints

- Exactly one MACHINE block per file.
- ROTOR arity: 3 for M3; 4 for M4 (if Beta/Gamma present, it must be leftmost).
- RINGS and START length must equal rotor arity.
- PLUGS letters pairwise disjoint; ≤ doctrine max (default 10).
- REFL must be thin for M4.
- If ProcSpec contains double-indicator, every MESSAGE requires INDICATOR.

A.2 Canonical AST (post-elaboration)

```
{
  "type": "EnigmaSpec",
  "machine": {
    "service": "M3",             // "M3" | "M4" (inferred)
    "rotors": ["III","I","IV"],   // array<string>
    "rings":  [1,5,10],          // array<int 0..25>
    "start":  [16,4,12],         // array<int 0..25>
    "plugs":  [["A","L"],["P","R"],["T","X"]], // array<tuple<UpperAlpha,UpperAlpha>>
    "reflector": "B",            // string
    "proc": ["double-indicator","five-groups"], // array<string>
    "catalog_hash": "sha256:..."  // of catalog.json
  },
  "messages": [
    {
      "indicator": "HZG",        // optional string len=3
      "text": ["MEETATNINE","SENDSUPPLIES"] // array<string A..Z+>
    }
  ],
  "meta": {
    "strict": true,
    "source_hash": "sha256:...",  // canonicalized MACHINE block hash
    "lints": [                // optional non-fatal diagnostics
      {"code":"W-LINT-LOW-ENTROPY-IND","at": {"line": 12, "col":11}}
    ]
  }
```

}

JSON Schema (excerpt)

```
{
  "$schema":"https://json-schema.org/draft/2020-12/schema",
  "title":"EnigmaSpec",
  "type":"object",
  "required":["machine","messages","meta"],
  "properties":{
    "machine":{
      "type":"object",
      "required":["rotors","rings","start","plugs","reflector","proc","service","catalog_hash"],
      "properties":{
        "service":{"enum":["M3","M4"]},
        "rotors":{"type":"array","minItems":3,"maxItems":4,"items":{"type":"string"}},
        "rings":{"type":"array","items":{"type":"integer","minimum":0,"maximum":25}},
        "start":{"type":"array","items":{"type":"integer","minimum":0,"maximum":25}},
        "plugs":{"type":"array","items":{"type":"array","items":{"type":"string","pattern":"^[A-
Z]$"},"minItems":2,"maxItems":2}},
        "reflector":{"type":"string"},
        "proc":{"type":"array","items":{"type":"string"}},
        "catalog_hash":{"type":"string","pattern":"^sha256:[0-9a-f]{64}$"}
      }
    },
    "messages":{
      "type":"array",
      "items":{
```

```
      "type":"object",
      "required":["text"],
      "properties":{
        "indicator":{"type":"string","pattern":"^[A-Z]{3}$"},
        "text":{"type":"array","items":{"type":"string","pattern":"^[A-Z]+$"}}
      }
    }
  },
  "meta":{
    "type":"object",
    "required":["strict","source_hash"],
    "properties":{
      "strict":{"type":"boolean"},
      "source_hash":{"type":"string","pattern":"^sha256:[0-9a-f]{64}$"},
      "lints":{"type":"array","items":{"type":"object"}}
    }
  }
 }
}
```

B. Formal semantics (small-step rules; permutation proofs)

B.1 Domains and maps

- Alphabet Σ = {0..25}; A=0,…,Z=25.
- Plugboard P ∈ S_26, product of disjoint transpositions (involution).
- Reflector U ∈ S_26, perfect matching (involution, no fixed points).
- Rotor i base wiring W_i ∈ S_26; inverse W_i^{-1}.

- Ring offsets k_i ∈ Σ; positions p_i ∈ Σ.

Define shift Sh(s)(x) = (x + s) mod 26.

Effective rotor maps at press t (post-step positions p_i'):

F_i = Sh(-p_i') ∘ Sh(-k_i) ∘ W_i ∘ Sh(k_i) ∘ Sh(p_i')

B_i = Sh(-p_i') ∘ Sh(-k_i) ∘ W_i^{-1} ∘ Sh(k_i) ∘ Sh(p_i')

Rotor blocks:

Rot = F_1 ∘ F_2 ∘ F_3 // left..right

Rot^{-1} = B_3 ∘ B_2 ∘ B_1

One-press enciphering permutation:

E = P^{-1} ∘ Rot^{-1} ∘ U ∘ Rot ∘ P

B.2 Small-step transition

State S = (R, k, p, P, U) with fixed R (rotor ids), k (rings), P, U; evolving p.

Transition on input x ∈ Σ:

1. p' = step(p, k, notches(R)) // Sec. 4 turnover rule (pre-step notch check, ring-adjusted)
2. y = E_{p'}(x) // build E using p' in F_i/B_i
3. (S', x) → (S[p := p'], y)

B.3 Turnover correctness (case rule)

Let N_i be notch set for rotor i (as indices 0..25), effective N_i^eff = { (n - k_i) mod 26 | n ∈ N_i }. With p = [p1,p2,p3] before press:

- Always: p3' = p3 + 1 (mod 26)
- Middle: p2' = p2 + 1 iff (p3 ∈ N_3^eff) ∨ (p2 ∈ N_2^eff)
- Left: p1' = p1 + 1 iff (p2 ∈ N_2^eff)
- Otherwise unchanged.

Proof sketch: matches historical mechanism; ring offsets reindex notch observation; double-step arises when p2 ∈ N_2^eff.

B.4 Involution and derangement

Let g = Rot ∘ P. Then:

E = g^{-1} ∘ U ∘ g ⇒ E^2 = id   (conjugation preserves involution)

Fixed points: if E(x)=x, then U(g(x))=g(x); impossible since U has none. Hence each E is an **odd derangement** (parity preserved under conjugation; sgn(E)=sgn(U)=-1).

B.5 Whole-run correctness

Given plaintext X = x_1..x_T and initial p_0, define per-press E_t from p_t. Encryption yields y_t = E_t(x_t). Decryption runs same E_t in same order from same p_0; as each E_t is an involution, x_t = E_t(y_t). Therefore decode(encode(X,S_0),S_0) = X.

C. Reference interpreter pseudocode and property-based tests

C.1 Pseudocode (reference, side-effect-free core)

```
ALPHA = {ch: i for i, ch in enumerate("ABCDEFGHIJKLMNOPQRSTUVWXYZ")}

IALPH = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"


def sh(x, s): return (x + s) % 26


def build_plug(plugs):
    P = list(range(26))
    for a, b in plugs:
        ia, ib = ALPHA[a], ALPHA[b]
        P[ia], P[ib] = ib, ia
    return P  # involution array


def apply_perm(P, x): return P[x]


def rotor_effective(W, Winv, p, k, forward=True):
    # returns functions F(x) or B(x)
    def F(x):
```

```
        y = sh(x,  p)
        y = sh(y,  k)
        y = W[y] if forward else Winv[y]
        y = sh(y, -k)
        y = sh(y, -p)
        return y
    return F

def step(pos, rings, notches):
    p1, p2, p3 = pos; k1, k2, k3 = rings
    N1 = {(n - k1) % 26 for n in notches[0]}
    N2 = {(n - k2) % 26 for n in notches[1]}
    N3 = {(n - k3) % 26 for n in notches[2]}

    stepR = True
    stepM = (p3 in N3) or (p2 in N2)
    stepL = (p2 in N2)

    p3 = (p3 + 1) % 26 if stepR else p3
    p2 = (p2 + 1) % 26 if stepM else p2
    p1 = (p1 + 1) % 26 if stepL else p1
    return (p1, p2, p3), {"L": stepL, "M": stepM, "R": stepR}

def enc_press(cfg, state, x):
    # cfg: (rotors tables, rings, P, U, notches)
    (W1, W1i, W2, W2i, W3, W3i), rings, P, U, notches = cfg
```

```
    pos, _ = state
    pos2, flags = step(pos, rings, notches)

    F1 = rotor_effective(W1, W1i, pos2[0], rings[0], True)
    F2 = rotor_effective(W2, W2i, pos2[1], rings[1], True)
    F3 = rotor_effective(W3, W3i, pos2[2], rings[2], True)
    B1 = rotor_effective(W1, W1i, pos2[0], rings[0], False)
    B2 = rotor_effective(W2, W2i, pos2[1], rings[1], False)
    B3 = rotor_effective(W3, W3i, pos2[2], rings[2], False)

    s0 = apply_perm(P, x)
    s1 = F3(s0); s2 = F2(s1); s3 = F1(s2)
    s4 = apply_perm(U, s3)
    s5 = B1(s4); s6 = B2(s5); s7 = B3(s6)
    y  = apply_perm(P, s7)
    return (pos2, flags), y
```

Stream encode/decode

```
def run(cfg, start_pos, letters):
    state = (start_pos, None)
    out, trace = [], []
    for t, ch in enumerate(letters, 1):
        x = ALPHA[ch]
        state, y = enc_press(cfg, state, x)
        out.append(IALPH[y])
        trace.append({"t":t, "pos_after":state[0], "input":ch, "output":IALPH[y]})
    return "".join(out), trace
```

C.2 Property-based tests (sketch)

Generators

- Random well-typed configs: choose rotor order, uniform rings/start (0..25), random plug matchings (≤10 pairs), valid reflector.
- Random plaintexts length L ∈ {0..200}, with targeted cases around notch boundaries.

Properties

1. **Decode/encode identity**

for all cfg, start, Ptext:

C, tr = run(cfg, start, Ptext)

P2, _ = run(cfg, start, C)

assert P2 == Ptext

2. **Per-press involution & derangement**

for all cfg, pos:

build E_pos as table (apply enc_press on each letter with pos fixed)

assert for all x: E_pos[E_pos[x]] == x

assert for all x: E_pos[x] != x

3. **Step rule consistency**

for all rings, notches, pos:

pos2, flags = step(pos, rings, notches)

Verify double-step cases against a slow spec or enumerated truth table

4. **RTL equivalence (if RTL present)**

- Generate witness logs with SW; simulate RTL; ensure outputs and positions match per cycle.

D. HDL netlist template and testbench harness

D.1 RTL structure (Verilog, conceptual)

plug.v

```
module plug(input [4:0] x, output reg [4:0] y);
  always @* begin
    case (x)
      // 26 entries; pairs swapped; others map to themselves
      5'd0: y = 5'd11; // A->L
      5'd11: y = 5'd0; // L->A
      // ...
      default: y = x;
    endcase
  end
endmodule
```

reflector.v

```
module reflect(input [4:0] x, output reg [4:0] y);
  always @* begin
    case (x)
      // 26 entries from catalog (e.g., UKW-B)
      // each is a 2-cycle; no fixed points
      // ...
    endcase
  end
endmodule
```

rotor.v (parameterized)

```
module rotor #(
```

```
  parameter [129:0] W = /* 26*5 bits forward table */,
  parameter [129:0] Winv = /* inverse table */
)(
  input  [4:0] x,
  input  [4:0] pos,   // 0..25
  input  [4:0] ring,  // 0..25
  input        fwd,   // 1=fwd,0=rev
  output [4:0] y
);
  wire [4:0] a = x + pos;        // Sh(+p)
  wire [4:0] b = a + ring;       // Sh(+k)
  wire [4:0] c = fwd ? W[b] : Winv[b];
  wire [4:0] d = c - ring;       // Sh(-k)
  assign y = d - pos;            // Sh(-p)
endmodule
```

step_fsm.v (double-step logic)

```
module step_fsm(
  input        clk,
  input        rst,
  input        valid,         // strobe per char
  input  [4:0]  p1,p2,p3,      // before press
  input  [4:0]  n1,n2,n3,      // notch indices adjusted by ring
  output [4:0]  q1,q2,q3,      // after press
  output       stL,stM,stR     // flags
);
  wire hit2 = (p2 == n2);
```

```
  wire hit3 = (p3 == n3);


  assign stR = 1'b1;
  assign stM = hit3 | hit2;
  assign stL = hit2;


  assign q3 = p3 + 1;
  assign q2 = p2 + (stM ? 1 : 0);
  assign q1 = p1 + (stL ? 1 : 0);
endmodule
```

enigma_core.v

```
module enigma_core(
  input        clk,
  input        rst,
  input        valid_in,
  input  [4:0]  ch_in,
  input  [4:0]  pos_in [2:0],   // [0]=left, [1]=mid, [2]=right
  input  [4:0]  ring   [2:0],
  // rotor params supplied via module params or higher-level wrapper
  output       valid_out,
  output [4:0]  ch_out,
  output [4:0]  pos_out [2:0],
  output [2:0]  step_flags
);
  wire [4:0] p1 = pos_in[0], p2 = pos_in[1], p3 = pos_in[2];
  wire [4:0] n1, n2, n3; // notch indices precomputed externally or via ROM
```

```
  wire [4:0] q1,q2,q3; wire stL,stM,stR;
  step_fsm STEP(.clk(clk),.rst(rst),.valid(valid_in),
              .p1(p1),.p2(p2),.p3(p3),.n1(n1),.n2(n2),.n3(n3),
              .q1(q1),.q2(q2),.q3(q3),.stL(stL),.stM(stM),.stR(stR));

  wire [4:0] s0, s1, s2, s3, s4, s5, s6, s7;

  plug P0(.x(ch_in), .y(s0));
  rotor R3(.x(s0), .pos(q3), .ring(ring[2]), .fwd(1'b1), .y(s1));
  rotor R2(.x(s1), .pos(q2), .ring(ring[1]), .fwd(1'b1), .y(s2));
  rotor R1(.x(s2), .pos(q1), .ring(ring[0]), .fwd(1'b1), .y(s3));
  reflect U0(.x(s3), .y(s4));
  rotor R1r(.x(s4), .pos(q1), .ring(ring[0]), .fwd(1'b0), .y(s5));
  rotor R2r(.x(s5), .pos(q2), .ring(ring[1]), .fwd(1'b0), .y(s6));
  rotor R3r(.x(s6), .pos(q3), .ring(ring[2]), .fwd(1'b0), .y(s7));
  plug P1(.x(s7), .y(ch_out));

  assign pos_out[0]=q1; assign pos_out[1]=q2; assign pos_out[2]=q3;
  assign step_flags = {stL,stM,stR};
  assign valid_out  = valid_in; // 1-cycle comb path; register if timing requires
endmodule
```

D.2 Testbench harness (simulation-level)

Golden-driven testbench (pseudo-SystemVerilog)

```
module tb;
  logic clk=0, rst=1, valid_in=0;
```

```
  logic [4:0] ch_in, ch_out;
  logic [4:0] pos_in [2:0], pos_out [2:0];
  logic [4:0] ring   [2:0];
  logic [2:0] step_flags;
  enigma_core DUT(.*);

  // clock
  always #5 clk = ~clk;

  // load from JSON (preprocessed): catalog tables, initial pos/rings, input letters, expected outputs & positions
  initial begin
    rst = 1; repeat(2) @(posedge clk); rst = 0;

    pos_in[0]=left0; pos_in[1]=mid0; pos_in[2]=right0;
    ring[0]=k1; ring[1]=k2; ring[2]=k3;

    foreach (stim[i]) begin
      ch_in    = stim[i].in;  // 0..25
      valid_in = 1;
      @(posedge clk);
      valid_in = 0;
      // Allow 0-latency comb or 1-cycle registered depending on synthesis
      @(posedge clk);

      assert (ch_out    == stim[i].out)
```

```
      else $fatal("Enc mismatch t=%0d got=%0d exp=%0d", i, ch_out, stim[i].out);
    assert (pos_out[0]== stim[i].p1 && pos_out[1]== stim[i].p2 && pos_out[2]== stim[i].p3)
      else $fatal("Step mismatch t=%0d", i);
  end
  $display("PASS");
  $finish;
 end
endmodule
```

Assertions (optional SVA)

- **Involution at fixed pos**: bind a checker that, given frozen q1,q2,q3, wraps DUT twice and checks identity for all 26 inputs.
- **No fixed points**: sweep x ∈ 0..25 at fixed pos and assert enc(x) != x.

Artifacts

- vectors/ directory: JSON or CSV witness logs produced by software reference; conversion script outputs SV stim[] for the testbench.
- CI target runs: (i) parse → (ii) SW run → (iii) log → (iv) RTL sim vs. log.

D.3 Build & CI checklist

- **enig check**: schema + static lints pass.
- **enig run**: emits ciphertext + witness log; hashes computed.
- **enig verify**: per-row invariants and end-to-end replay succeed.
- **rtl-gen**: Verilog emitted from catalog and machine AST; tables checked against catalog hash.
- **rtl-sim**: testbench passes on golden logs (short, long, double-step edges).
- **Properties**: quick involution/derangement sweeps at random positions each run.
- **Differential**: software vs RTL bit-for-bit equality; first counterexample logged with minimal repro case.

Summary

These appendices provide the executable backbone: a complete grammar and typed AST; rigorous small-step semantics and algebraic guarantees; a reference interpreter with property tests; and synthesizable HDL with a verification harness. Together they operationalize the paper's thesis—**a small, auditable language with deterministic, witnessable execution**—in a form you can build, test, and trust.

References

Books & historical accounts

- Copeland, B. J. (Ed.). (2004). *The Essential Turing*. Oxford University Press.
- Gannon, P. (2006). *Colossus: Bletchley Park's Greatest Secret*. Atlantic Books.
- Hodges, A. (1983/2012). *Alan Turing: The Enigma* (revised ed.). Princeton University Press.
- Kahn, D. (1967/1996). *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet* (rev. ed.). Scribner.
- Sebag-Montefiore, H. (2000). *Enigma: The Battle for the Code*. Weidenfeld & Nicolson.
- Welchman, G. (1982). *The Hut Six Story: Breaking the Enigma Codes*. McGraw-Hill.

Primary/technical sources on Enigma, Bombe, and Colossus

- Rejewski, M. (1981). How Polish Mathematicians Deciphered the Enigma. *Annals of the History of Computing*, 3(3), 213–234.
- Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2), 230–265.
- Turing, A. M. (1941; reprinted in Copeland, 2004). *The Applications of Probability to Cryptography* (declassified report). In B. J. Copeland (Ed.), *The Essential Turing*.
- Sale, T. (2004). *The Colossus Computer 1943–1996*. M&T Books / Bletchley Park Trust.
- Copeland, B. J., & Proudfoot, D. (2004). The Turing–Welchman Bombe: An Historical and Technical Study. In B. J. Copeland (Ed.), *The Essential Turing* (technical appendix).

Programming languages, theory, and verification (for analogies in the paper)

- Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377–387.
- Claessen, K., & Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Proc. ICFP*, 268–279.
- Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis* (Revised ed.). MIT Press.

- Kleene, S. C. (1956). Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies* (pp. 3–42). Princeton University Press.
- Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Networking/crypto DSLs and hardware description

- McKeown, N., Anderson, T., Balakrishnan, H., et al. (2014). P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 87–95.
- Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. *RFC 8446*. IETF.
- IEEE Standards Association. (2005). *IEEE Std 1364-2005: Verilog Hardware Description Language*. (See also *IEEE Std 1800* for SystemVerilog.)

Supplementary/background

- Good, I. J., Michie, D., & Timms, G. (eds.) (1983). *General Report on Tunny* (declassified Bletchley Park report). HMSO / GCHQ release.
- Randell, B. (Ed.). (1980). *The Origins of Digital Computers: Selected Papers* (contains early material on Colossus and codebreaking context). Springer.

Note: Where declassified reports exist in multiple transcriptions (e.g., Turing's probability paper; the *General Report on Tunny*), the editions cited above are stable, widely referenced versions suitable for scholarly use.