

OpenAI: the Exponential Time Hypothesis is false

Source revision [679527a817e3](#) / [Lean source](#)

Statements: [exact_count](#), [not_eth](#), [not_seth](#), [satExponent_eq_zero](#), [length_encCNF](#)

For each fixed clause width, one deterministic native machine exactly counts satisfying assignments in subexponential time and polynomial space. Every fixed-width SAT exponent is zero, and ETH and SETH are false. Native formalization of OpenAI’s manuscript, revised 25 October 2026, by the contributors to #1226.

Scope. Explicit CNF semantics, model counts over the occurring variables, and binary formula/output encodings; the interpolation-scale exponent $n (\log \log (n + 16))^2 / \sqrt{\log(n + 16)}$ from §§3–6 and Proposition 6.4, where n is the number of variables, with polynomial time overhead and polynomial workspace in the encoded formula length. Includes all fixed widths and sizes; the count is exact on every formula of the given width. This does not establish the sharper Theorem 1.1 / Appendix D bound.

The four complexity theorems below are proved through SubexponentialSATBridge from the unconditional native counting endpoint. A fifth theorem, `length_encCNF`, gives the exact length of the input encoding.

The primary result is one deterministic machine for each fixed clause width that computes the number of satisfying assignments of every formula of that width. With n the number of distinct variables, its running time is exponential in a constant times $n (\log \log (n + 16))^2 / \sqrt{\log (n + 16)}$, multiplied by a polynomial in the encoded formula length. Its work space, including a separate charge for the encoded output, is polynomial in that same length. This is the interpolation-scale route of Sections 3–6 and Proposition 6.4 of [OpenAI2026ETH], not the sharper Theorem 1.1 / Appendix D bound.

The definitions here display the existing CNF semantics, encodings and fine-grained classes; they are not alternative library interfaces. The accompanying regression module checks their agreement with the canonical definitions. Machinery is existential. The bridge imports the public unconditional endpoint, whose native counter and consequences prove the displayed statements without any branch-certificate or exactness hypothesis.

What a reader inspects. Read the definitions below and the displayed Encoding foundation. The machine model is upstream `cslib`’s deterministic multi-tape Turing machine; no library module beyond Encoding is allowlisted, so no CNF implementation, complexity-class definition or counting construction enters the statements. Lean checks the bridge proofs; `#check_abstract` audits the dependency boundary of the definitions and theorem statements, not their English interpretation.

References and credit

- OpenAI, *The Exponential Time Hypothesis is False* (OpenAI2026ETH), revised 25 October 2026, Sections 3–6, Proposition 6.4, pp. 43–44, and Appendices A–B. Source revision: [openai/math@7ef5c0cc5ca4b0e072b7a050fca09ab24bc6fa3a](#).
- R. Impagliazzo and R. Paturi, *On the complexity of k -SAT* (ImpagliazzoPaturi2001): the exponential-time hypotheses and the infimum exponent.
- M. R. Garey and D. S. Johnson, *Computers and Intractability* (GareyJohnson1979): reasonable encodings.

The native port is the joint work recorded in #1226, including Samuel Schlesinger, Anton Kovsharov, Erik Demaine and the contributing Codex, Claude and Jetski workers. The native machine and encoding foundations

include Christian Reitwiessner's contributions. Construction provenance is recorded in Subexponential SATBridge.

Shared foundation definitions appear in the appendix: [ComplexityAbstracts.Encoding](#).

```
open ComplexityTests.Abstracts.Encoding
```

Formulas and model counting

A literal is a variable or its negation. Clauses are disjunctions of literals and formulas are conjunctions of clauses; repeated literals and clauses are permitted. A width bound counts literal occurrences, including repetitions.

A literal: the variable `var`, positive or negated. It is the pair `(var, positive)`.

```
abbrev Literal := ℕ × Bool
```

The variable of a literal.

```
abbrev Literal.var (l : Literal) : ℕ := l.1
```

Whether the literal is the variable itself (true) or its negation (false).

```
abbrev Literal.positive (l : Literal) : Bool := l.2
```

A clause is a disjunction of literals.

```
abbrev Clause := List Literal
```

A CNF formula is a conjunction of clauses.

```
abbrev Formula := List Clause
```

Evaluate a formula: every clause must contain a literal that is true under σ , that is, whose variable's value matches its polarity.

```
def eval (φ : Formula) (σ : ℕ → Bool) : Bool :=  
  φ.all fun clause => clause.any fun literal => σ literal.var == literal.positive
```

A formula is satisfiable when some total Boolean assignment makes it true.

```
def Satisfiable (φ : Formula) : Prop := ∃ σ : ℕ → Bool, eval φ σ = true
```

The variables occurring in φ .

```
def vars (φ : Formula) : Finset ℕ := (φ.flatten.map Literal.var).toFinset
```

The number of variables occurring in φ , independent of their numeric labels.

```
def numVars ( $\varphi$  : Formula) :  $\mathbb{N}$  := (vars  $\varphi$ ).card
```

The number of satisfying assignments of φ : assignments $\text{vars } \varphi \rightarrow \text{Bool}$ that make φ true when every other variable is set to false.

```
def modelCount ( $\varphi$  : Formula) :  $\mathbb{N}$  :=
  (Finset.univ.filter fun  $\sigma$  : vars  $\varphi \rightarrow \text{Bool} \Rightarrow$ 
    eval  $\varphi$  (fun v => if h : v  $\in$  vars  $\varphi$  then  $\sigma$  (v, h) else false) = true).card
```

Every clause of φ has at most k literals.

```
def HasWidth ( $k$  :  $\mathbb{N}$ ) ( $\varphi$  : Formula) : Prop :=  $\forall$  clause  $\in \varphi$ , clause.length  $\leq k$ 
```

Deterministic machine and resource conventions

The upstream model is a deterministic binary multi-tape Turing machine with finitely many states and a fixed finite number of work tapes. Each transition performs bounded head moves and symbol operations. The input tape is read-only, and output is write-only. Time counts transitions until halting; work space counts visited work-tape cells over every run prefix. No condition is imposed on bitstrings outside the chosen encoding's image.

Upstream ComputableInTimeAndSpace f encIn encOut t s asserts one such machine that, on every input x , halts with $\text{encOut } (f \ x)$ within $t \ x$ steps and $s \ x$ work cells. DecidableInTimeAndSpace is the case of the exact Boolean membership bit. The finite machine and all bounds precede the inputs. Time and space are independent budgets; a time bound does not supply polynomial space.

f is computable within time $O(T \ x)$ and space $O(S \ x)$, with the encoded output also of length $O(S \ x)$: output is charged even though its tape is not a work tape.

```
def ComputableIn0 { $\alpha$   $\beta$  : Type} (f :  $\alpha \rightarrow \beta$ ) (encIn :  $\alpha \hookrightarrow \text{List Bool}$ ) (encOut :  $\beta \hookrightarrow \text{List Bool}$ )
  (T S :  $\alpha \rightarrow \mathbb{N}$ ) : Prop :=
   $\exists$  c, Turing.MultiTapeTM.ComputableInTimeAndSpace f encIn encOut
    (fun x => c * (T x + 1)) (fun x => c * (S x + 1))  $\wedge$ 
     $\forall$  x, (encOut (f x)).length  $\leq c * (S x + 1)$ 
```

A decision problem is decidable in time $O(T \ x)$, with some independent space bound.

```
def DecidableInTime0 { $\alpha$  : Type} (A : Set  $\alpha$ ) (enc :  $\alpha \hookrightarrow \text{List Bool}$ ) (T :  $\alpha \rightarrow \mathbb{R}$ ) : Prop :=
   $\exists$  c :  $\mathbb{N}$ ,  $\exists$  t s :  $\alpha \rightarrow \mathbb{N}$ , ( $\forall$  x, (t x :  $\mathbb{R}$ )  $\leq c * T x + c$ )  $\wedge$ 
    Turing.MultiTapeTM.DecidableInTimeAndSpace A enc t s
```

Exponential time in a parameter, with a polynomial factor in the encoded input length.

```
def DecidableInExpTime { $\alpha$  : Type} (A : Set  $\alpha$ ) (enc :  $\alpha \hookrightarrow \text{List Bool}$ )
  ( $\mu$  :  $\alpha \rightarrow \mathbb{N}$ ) ( $\delta$  :  $\mathbb{R}$ ) : Prop :=
   $\exists$  e :  $\mathbb{N}$ , DecidableInTime0 A enc fun x => 2 ^ ( $\delta * \mu x$ ) * ((enc x).length + 1) ^ e
```

The exponential time hypotheses

The decision parameter is the number of distinct occurring variables, not the largest label. The bridge therefore uses the native dense-relabeling correspondence. Zero infimum exponent alone does not assert a polynomial-time algorithm.

k-SAT consists of satisfiable formulas whose clauses have at most k literals.

```
def KSAT (k : ℕ) : Set Formula :=
  {φ | (∀ clause ∈ φ, clause.length ≤ k) ∧ Satisfiable φ}
```

The infimum of all nonnegative exponential rates admitting a k-SAT decider.

```
noncomputable def satExponent (k : ℕ) : ℝ :=
  sInf {δ | 0 ≤ δ ∧ DecidableInExpTime (KSAT k) encCNF numVars δ}
```

ETH asserts that some positive exponential rate is unattainable for 3SAT.

```
def ETH : Prop :=
  ∃ δ : ℝ, 0 < δ ∧ ¬DecidableInExpTime (KSAT 3) encCNF numVars δ
```

SETH asserts that every improvement below rate one fails for some fixed clause width.

```
def SETH : Prop :=
  ∀ ε : ℝ, 0 < ε → ∃ k : ℕ, ¬DecidableInExpTime (KSAT k) encCNF numVars (1 - ε)
```

Input size

The polynomial factors below are polynomials in the length of `encCNF φ`, which is exactly determined by the formula.

The encoding is concise. Its length is exactly one bit per clause, plus $10 + 8 \cdot b$ bits for each literal occurrence, where $b = \text{Nat.size } v$ is the number of binary digits of its variable v . Thus the input length is linear in the number of literals and logarithmic in the variable labels, as for a reasonable encoding in the sense of Garey and Johnson (GareyJohnson1979).

```
theorem length_encCNF (φ : Formula) :
  (encCNF φ).length = φ.length + (φ.flatten.map fun l => 10 + 8 * Nat.size l.var).sum
```

Exact counting and the exponential time hypotheses

The primary theorem fixes the counter, constants and native machine for each k before every input. It includes $k = 0, 1$, empty formulas and empty clauses. On formulas with a clause longer than k the output is unspecified; the time and space bounds still hold there.

The scale uses natural logarithms and a shift by sixteen to cover all natural n; this changes only constants in the asymptotic bound. It is the scale of Proposition 6.4, not the smaller exponent of Theorem 1.1. The proof relabels the variables densely and runs the native declared-variable counter.

The interpolation-scale exponent, defined at every natural input.

```
noncomputable def interpolationScale (n : ℕ) : ℝ :=
  n * Real.log (Real.log ((n : ℝ) + 16)) ^ 2 / Real.sqrt (Real.log ((n : ℝ) + 16))
```

For each fixed width k , one machine counts the satisfying assignments of every width- k formula in time $2^{A \cdot n (\log \log n)^2 / \sqrt{\log n}} \cdot \text{poly}(L)$ and space $\text{poly}(L)$, where n is the number of variables and L the encoding length.

```
theorem exact_count (k : ℕ) :
  ∃ (count : Formula → ℕ) (A e b : ℕ),
    (∀ φ, HasWidth k φ → count φ = modelCount φ) ∧
    ComputableIn0 count encCNF encNat
    (fun φ => [(2 : ℝ) ^ (A * interpolationScale (numVars φ))] + * ((encCNF φ).length + 1) ^
e)
    (fun φ => ((encCNF φ).length + 1) ^ b)
```

The exponential time hypothesis is false.

```
theorem not_eth : ¬ETH
```

The strong exponential time hypothesis is false.

```
theorem not_seth : ¬SETH
```

Every fixed-width SAT exponent is zero.

```
theorem satExponent_eq_zero : ∀ k, satExponent k = 0
```

Appendix: representation details

The input is the encoded formula $\text{encCNF } \varphi$. Literal indices are binary; large or non-dense labels are not silently treated as small. A natural output uses canonical binary digits, with zero encoded by the empty word. The shared foundation spells out all framing and list conventions.

Encode each literal as a framed binary index followed by its polarity bit; frame the literals within clauses and the clauses within the formula. This is the native encCNF .

```
def encCNF : Formula ↔ List Bool :=
  encList (encList (encPair encNat encBool))
```

Appendix: shared foundations

These checked definitions specify the representations and machine bounds used above. They are displayed once here; their Lean sources remain part of the document's provenance.

Binary input conventions

[Lean source](#). Namespaces: ComplexityTests.Abstracts.Encoding.

Declarations: `frame`, `encodePair`, `encodeList`, `encodePair_injective`, `encodeList_injective`, `bits_injective`, `encPair`, `encList`, `encNat`, `encBool`, `encBits`, `encUnary`.

An encoding is an injective map to finite bitstrings; $\hookrightarrow$ records injectivity. These fixed conventions specify the inputs and outputs of the machines used in the abstracts. Framing makes component boundaries recoverable with linear overhead. Natural numbers use binary digits unless an abstract explicitly supplies a unary parameter. Changing binary to unary can change a complexity claim. The delimiter convention below fixes the precise machine input.

The conventions are those of Christian Reitwiessner’s native-machine encoding API. Their exact definitions are collected here so each theorem can be read before its serialization details.

Make a word self-delimiting: tag each bit by true and finish with false.

```
def frame (w : List Bool) : List Bool := w.flatMap (fun b => [true, b]) ++ [false]
```

Frame the first component, then append the second.

```
def encodePair {α β : Type} (a : α → List Bool) (b : β → List Bool) (p : α × β) :  
  List Bool := frame (a p.1) ++ b p.2
```

Concatenate the framed elements of a list.

```
def encodeList {α : Type} (enc : α → List Bool) (l : List α) : List Bool :=  
  l.flatMap (frame ∘ enc)
```

Injective encoding of a pair. Its length is $2 * |a \ x| + 1 + |b \ y|$.

```
def encPair {α β : Type} (a : α ↪ List Bool) (b : β ↪ List Bool) : α × β ↪ List Bool :=  
  ⟨encodePair a b, encodePair_injective a b⟩
```

Injective encoding of a list.

```
def encList {α : Type} (enc : α ↪ List Bool) : List α ↪ List Bool :=  
  ⟨encodeList enc, encodeList_injective enc⟩
```

Canonical binary digits, least significant first; zero has the empty encoding.

```
def encNat : ℕ ↪ List Bool := ⟨Nat.bits, bits_injective⟩
```

A Boolean is a single bit.

```
def encBool : Bool ↪ List Bool := ⟨fun b => [b], _⟩
```

A bitstring is its own encoding.

```
def encBits : List Bool  $\hookrightarrow$  List Bool := Function.Embedding.refl _
```

Unary size or security parameter: exactly n bits.

```
def encUnary :  $\mathbb{N}$   $\hookrightarrow$  List Bool :=  $\langle$ fun n => List.replicate n true, _ $\rangle$ 
```