GPU-Accelerated Synthesis of Mixed-Boolean Arithmetic: Beyond Caching

Gabriel Bathie
LaBRI
Université de Bordeaux
Bordeaux, France
gabriel.bathie@gmail.com

Nathanaël Fijalkow
CNRS, LaBRI
Université de Bordeaux
Bordeaux, France
nathanael.fijalkow@gmail.com

Abstract

Synthesizing Mixed-Boolean Arithmetic (MBA) expressions from input-output examples is central to program deobfuscation and also useful for compiler optimization, reverse engineering, and cryptanalysis. Existing MBA synthesizers are typically CPU-based and scale poorly on large specifications or complex targets. Recent GPU-accelerated synthesis methods achieve large speedups in qualitative settings, but they depend on caching observationally equivalent candidates; this strategy breaks down for MBA because candidate outputs are quantitative bitvectors and the behavioral space is enormous. We present SIMBA (**S**ynthesis of **M**ixed-**B**oolean **A**rithmetic), a GPU-accelerated MBA synthesizer built around cache-free bottom-up enumeration. SIMBA avoids language caches entirely and uses a GPU-oriented enumeration design that keeps work local and highly parallel. In experiments, SIMBA is substantially faster than prior MBA synthesis tools, handles larger specifications, and reaches expression sizes that existing methods fail to solve. These results show that cache-free GPU synthesis is a practical and scalable alternative to cache-based designs in quantitative domains.

1 Introduction

Program synthesis, the automated generation of programs from specifications, is a long-standing problem in computer science [Gulwani et al., 2017, David and Kroening, 2017]. Among the many formulations of program synthesis, *synthesis from input-output examples* (also known as inductive programming or programming by example) is both practically important and theoretically challenging: given a set of example pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, find a program e in a target language such that $e(\mathbf{x}_i) = y_i$ for all i.

In this paper, we study *Mixed-Boolean Arithmetic* (MBA) synthesis from examples. MBA expressions combine bitwise operations (AND $\wedge$, OR $\vee$, XOR $\oplus$, NOT $\sim$) with arithmetic operations (addition $+$, subtraction $-$, multiplication $\times$) over fixed-width bitvectors. This problem arises in several important domains. The primary application is *code deobfuscation*: MBA obfuscation [Zhou et al., 2007, Eyrolles, 2017] transforms simple expressions (e.g., $x + y$) into semantically equivalent but syntactically complex ones (e.g., $(x \oplus y) + 2 \cdot (x \wedge y)$), and recovering the original can be cast as MBA synthesis [Blazytko et al., 2017, David et al., 2020, Lee and Lee, 2023, Liu et al., 2021b, Reichenwallner and Meerwald-Stadler, 2023, Attias et al., 2025]. MBA synthesis also serves compiler superoptimizers [Bansal and Aiken, 2006, Sasnauskas et al., 2017], which search for simpler equivalent instruction sequences, and supports reverse engineering and cryptanalysis by extracting human-readable descriptions of complex binary computations.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Current MBA synthesis methods do not scale well. Stochastic systems such as Syntia [Blazytko et al., 2017] and its successors Xyntia [Menguy et al., 2021] and XSmir [Attias et al., 2025] can be effective in practice but do not provide completeness guarantees and struggle on hard instances. Algebraic simplifiers such as MBA-Blast [Liu et al., 2021b], SiMBA [Reichenwallner and Meerwald-Stadler, 2022], and GAMBA [Reichenwallner and Meerwald-Stadler, 2023] are strong on important subclasses but do not cover the full synthesis setting. SAT/SMT-based methods are principled but expensive at larger bit-widths, and neural approaches such as NeuReduce [Feng et al., 2020] trade soundness for speed.

A natural idea is to bring MBA synthesis to GPUs. Bottom-up enumeration is massively parallel, and prior work on regular-expression inference [Valizadeh and Berger, 2023] and the learning of Linear Temporal Logic (LTL) formulas from traces [Valizadeh et al., 2024] shows that GPU acceleration can deliver major speedups. But these methods rely on a language cache indexed by observational equivalence. That is effective in qualitative domains, where behavior on examples is Boolean. In MBA synthesis, behavior is quantitative: each candidate returns a w-bit value per example. With n examples, the behavioral space has size 2^{wn} (e.g., 2^{320} for $n = 10$, $w = 32$), so cache deduplication quickly becomes ineffective due to memory pressure and collisions.

We argue that GPU acceleration for MBA requires moving beyond cache-based designs. We introduce SIMBA, a cache-free GPU MBA synthesizer based on bottom-up enumeration by expression size. SIMBA is built around the following ideas:

1. **Cache-free on-the-fly evaluation.** Each candidate expression is decoded from a thread ID, evaluated against the specification, and immediately discarded. No intermediate results are stored, so the memory bottleneck of language caches disappears entirely.

2. **Bijective enumeration.** We define a bijection between integers and MBA expressions of a given size. Each GPU thread is assigned a unique integer, independently decodes it into an expression, and evaluates it with no inter-thread communication.

3. **Locally consistent encoding.** The bijection is designed so that consecutive integers decode to structurally similar expressions. Threads within the same GPU warp therefore follow nearly identical execution paths, minimizing branch divergence and keeping hardware utilization high.

Empirically, SIMBA synthesizes expressions substantially faster than prior MBA tools, handles larger specifications, and reaches larger target sizes. This shows that GPUs can speed up quantitative synthesis considerably, provided the algorithm is designed without a cache from the start. The source code of SIMBA is publicly available at `https://github.com/SynthesisLab/SIMBA`.

2 Problem Definition

2.1 Bitvectors and MBA Expressions

We work with w-bit bitvectors, i.e., elements of $\mathbb{B}^w = \{0,1\}^w$, which we also identify with integers in $\{0, \dots, 2^w - 1\}$ (unsigned) or $\{-2^{w-1}, \dots, 2^{w-1} - 1\}$ (signed) via the standard binary encoding. In practice, we use $w = 32$.

Definition 2.1 (MBA expressions)**.** Let $\mathcal{V} = \{x_1, \dots, x_k\}$ be a set of k variables ranging over $\mathbb{B}^w$. The set $\mathsf{MBA}(\mathcal{V})$ of *Mixed-Boolean Arithmetic expressions* over $\mathcal{V}$ is defined by the grammar:

$$e \ ::= \ x_i \in \mathcal{V} \mid -e \mid \sim e \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid e_1 \oplus e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2.$$

All operations are over $\mathbb{B}^w$, i.e., arithmetic is modular ($\bmod 2^w$) and bitwise operations act independently on each bit position. We write $[\![e]\!] : (\mathbb{B}^w)^k \to \mathbb{B}^w$ for the *semantics* of expression e, defined in the standard way.

Definition 2.2 (Size)**.** The *size* of an MBA expression e, written $\mathsf{size}(e)$, counts the number of operators and variables. In other words, it is the size of the tree representing the expression e.

2.2 The MBA Synthesis Problem

Definition 2.3 (Specification)**.** A *specification* is a set $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ of input-output pairs where each $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,k}) \in (\mathbb{B}^w)^k$ is a k-tuple of w-bit bitvector inputs and $y_i \in \mathbb{B}^w$ is the expected output. We require that the inputs are pairwise distinct: $\mathbf{x}_i \neq \mathbf{x}_j$ for $i \neq j$.

Definition 2.4 (MBA Synthesis Problem)**.** Given a specification $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ and a size budget $C \in \mathbb{N}$:

- **Soundness:** Find an expression $e \in \mathsf{MBA}(\mathcal{V})$ such that $[\![e]\!](\mathbf{x}_i) = y_i$ for all $i \in \{1, \ldots, n\}$.
- **Minimality:** Among all sound expressions, find one of minimum size: $\mathsf{size}(e) \leq \mathsf{size}(e')$ for all sound e'.
- **Bounded synthesis:** Find a sound expression with $\mathsf{size}(e) \leq C$, or report that none exists.

Observational equivalence. Two expressions e_1, e_2 are *observationally equivalent* with respect to specification S, written $e_1 \equiv_S e_2$, if $[\![e_1]\!](\mathbf{x}_i) = [\![e_2]\!](\mathbf{x}_i)$ for all i. The observational behavior of e given S is the vector $\mathsf{obs}_S(e) = ([\![e]\!](\mathbf{x}_1), \ldots, [\![e]\!](\mathbf{x}_n)) \in (\mathbb{B}^w)^n$.

Qualitative vs. quantitative. In prior GPU-based synthesis for LTL [Valizadeh et al., 2024], observational behavior is a Boolean vector in $\mathbb{B}^n$, since each formula either accepts or rejects each trace. With $n = 64$ traces, there are only 2^{64} possible behaviors, fitting in a single 64-bit machine word. In MBA synthesis, $\mathsf{obs}_S(e) \in (\mathbb{B}^w)^n$, giving 2^{wn} possible behaviors. With $n = 10$ examples and $w = 32$, this is 2^{320}, far too many to cache. This is why the language-cache approach of Valizadeh et al. [2024], Valizadeh and Berger [2023] does not carry over to MBA synthesis.

3 Algorithm

We now describe the SIMBA algorithm. The high-level structure is a bottom-up enumeration of MBA expressions by increasing size. Unlike prior GPU-based synthesizers, SIMBA does not use a language cache to store and deduplicate intermediate results; instead, it relies on an encoding of MBA expressions as integers.

3.1 Cache-Free Enumeration

SIMBA replaces the language cache with a *bijection* between integers and MBA expressions. Each GPU thread is assigned a unique integer identifier, decodes it into an expression on-the-fly, evaluates that expression against the specification, and then discards it. No intermediate results are stored between threads, and no inter-thread communication is needed beyond a final atomic write when a solution is found. The construction for this bijection is a classical technique in combinatorics, known as *ranking/unranking* [Knott, 1977, Grygiel and Lescanne, 2013].

The bijection does not shrink the search space; it removes the memory and synchronization costs of storing candidate programs. Cache-based GPU synthesizers (such as the algorithm of VFB [Valizadeh et al., 2024]) construct expressions by combining previously generated sub-expressions stored in a global memory array. Table 1 shows that this cache exhausts GPU memory before the search itself becomes too slow (VFB runs out of memory at size 11). In SIMBA, there is no such storage: each thread computes its candidate directly from its integer ID, using local registers and a small lookup table in constant memory.

We do not use random sampling either: it would evaluate some candidates twice and miss others, whereas exact minimal synthesis must cover every candidate.

Expression representation. Expressions are represented in *Reverse Polish Notation* (RPN). A formula of size s is a sequence of s tokens from the alphabet $\{x_0, \ldots, x_{k-1}\} \cup \{\sim, -, \wedge, \vee, \oplus, +, -, \times\}$, where binary operators pop two operands and push one result, and unary operators pop one and push one. Variables are the leaves (size 1), and operators increase size by exactly one (unary) or combine two sub-expressions (binary). The supported operator set consists of two unary operators ($\sim$ and $-$) and six binary operators ($\wedge$, $\vee$, $\oplus$, $+$, $-$, $\times$), for a total of 8 operator slots.

Counting table. The bijection is enabled by a precomputed table $\mathsf{T}[s][op]$ counting the number of valid RPN expressions of size s whose top-level operator is op (with $\mathsf{T}[s][8]$ denoting the total across

Equation 1 Recurrence for the table $\mathsf{T}[s][op]$.

$$\begin{aligned}
\mathsf{T}[1][8] &= k \quad \text{(one expr. per variable)}, \\
\mathsf{T}[s][op_{\text{unary}}] &= \mathsf{T}[s{-}1][8], \\
\mathsf{T}[s][op_{\text{comm}}] &= \sum_{j=1}^{\lfloor (s-1)/2 \rfloor} \mathsf{T}[j][8] \cdot \mathsf{T}[s{-}1{-}j][8], \\
\mathsf{T}[s][op_{\text{sub}}] &= \sum_{j=1}^{s-2} \mathsf{T}[j][8] \cdot \mathsf{T}[s{-}1{-}j][8], \\
\mathsf{T}[s][8] &= 2\,\mathsf{T}[s][op_{\text{unary}}] \\
&\quad + 5\,\mathsf{T}[s][op_{\text{comm}}] + \mathsf{T}[s][op_{\text{sub}}],
\end{aligned}$$

all operators). The table is computed on the CPU by the recurrence described in Equation 1, where commutative binary operators ($\wedge$, $\vee$, $\oplus$, $+$, $\times$) only consider splits where the left operand is no larger than the right one, $|L| \leq |R|$, which removes most symmetric duplicates (operands of equal size are still enumerated in both orders), while the non-commutative subtraction ($-$) counts all ordered pairs. Before enumeration begins, the table is transferred to the so-called *constant* memory of the GPU, a small segment of the device's memory that is read-only to threads but can be accessed with minimal latency.

Bijection and decoding. For each size s, the integers $\{0, \ldots, \mathsf{T}[s][8] - 1\}$ are in bijection with all valid RPN expressions of size s via the decoding function ϕ_s:

- **Base case** ($s = 1$): $\phi_1(n) = x_n$.
- **Find top operator**: scan the cumulative sums $\mathsf{T}[s][0], \mathsf{T}[s][0] + \mathsf{T}[s][1], \ldots$ until the prefix exceeds n; call this operator op, and let $n' = n -$ (sum before op).
- **Unary** ($op \in \{\sim, -\}$): $\phi_s(n) = op(\phi_{s-1}(n'))$.
- **Binary**: find the left-subtree size j such that n' falls in the block for split $(j, s{-}1{-}j)$, i.e., scan cumulative sums of $\mathsf{T}[j][8] \cdot \mathsf{T}[s{-}1{-}j][8]$ over valid j; let $n'' = n' -$ offset. Then $\phi_s(n) = \phi_j(\lfloor n''/\mathsf{T}[s{-}1{-}j][8] \rfloor)\ op\ \phi_{s-1-j}(n'' \bmod \mathsf{T}[s{-}1{-}j][8])$.

This decoding is computed entirely in thread-local registers; no global memory reads beyond the constant-memory table. A step-by-step example of the decoding with $k = 2$ variables is given in Section A.

Enumeration. The enumeration is shown in Algorithm 1. For each size s and each operator slot op, the host launches a GPU kernel with exactly $\mathsf{T}[s][op]$ threads. Thread tid decodes $\phi_s(\mathsf{tid} + \mathsf{offset}_{op})$ where $\mathsf{offset}_{op} = \sum_{op' < op} \mathsf{T}[s][op']$ is the cumulative count of preceding operator slots, evaluates the resulting expression against all n input-output pairs using an RPN stack interpreter, and checks if outputs match. If so, it atomically claims the solution slot and writes the formula. The host reads the solution buffer after each operator batch.

Algorithm 1 SIMBA: bijective cache-free enumeration of MBA expressions

Require: Specification $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, k variables, size bound C
Ensure: Expression e with $[\![e]\!](\mathbf{x}_i) = y_i$ for all i, or Not found

```
Compute counting table T on CPU; copy to GPU constant memory
for s = 1, 2, ..., C do
    offset ← 0
    for op ∈ {∼, ∧, ∨, ⊕, −, +, −, ×} do
        for all tid = 0, ..., T[s][op]−1 in parallel on GPU do
            e ← φ_s(tid + offset)    {decode integer to RPN expression}
            if [[e]](x_i) = y_i for all i then
                Atomically write e to solution buffer
            end if
        end for
        Check solution buffer on host; return e if found
        offset ← offset + T[s][op]
    end for
end for
return Not found
```

3.2 GPU-friendly Parallel Enumeration

GPU Architecture. Modern GPUs can execute operations in parallel on thousands of threads. However, they cannot execute *arbitrary* operations in parallel like CPUs; instead, they follow a paradigm called *Single Instruction, Multiple Threads* (SIMT) [Nickolls et al., 2008]. SIMT shares similarities with traditional SIMD (Single Instruction, Multiple Data) vector organizations by broadcasting a single instruction to multiple execution units (*threads*) simultaneously to maximize hardware throughput. The SIMT architecture can also handle some divergence, for example when threads of the same hardware group (called a *warp*) take different branches of an `if-else` statement. In this situation, threads that enter the `if` branch are executed first, while threads assigned to the `else` branch are temporarily masked out and remain idle. Once the first branch completes, the hardware inverts the execution mask to run the `else` branch. This process serializes the execution of the divergent paths, temporarily reducing the effective instruction throughput, until all threads in the warp complete the conditional block and execution reconverges.

Therefore, to maximize parallelism, one must ensure that threads of the same warp stay in sync as much as possible and do not take divergent branches. In our case, this means that threads of the same warp must evaluate *structurally similar* MBA expressions.

Locally Consistent Encoding of MBA Expressions. Each execution thread is given a unique integer identifier from $[0, n_{threads})$, with each warp being assigned identifiers from a contiguous slice of that range. In our algorithm, each thread is assigned a single MBA expression based on its identifier, and checks whether it satisfies the specification. We use a bijective mapping from integers to MBA expressions of a given size with the following properties:

- all threads are assigned different expressions (injectivity),
- all expressions are assigned to some thread (surjectivity),
- threads within the same warp process very similar formulas: most of the time, they differ only at the input variables.

To keep execution uniform within a warp, the design of SIMBA addresses three points:

- Decoding order: the low-order bits of a thread ID only change the deepest part of the expression tree (the rightmost leaf), while the high-order bits, which all threads of a warp share, fix the top-level operator and the left subtree.
- Register-only decoding: decoding runs entirely in thread-local registers and only reads a small table in constant memory, so it uses no global memory bandwidth.
- Uniform evaluation: the RPN stack interpreter is written so that structurally similar expressions follow the same instruction path with the same stack depth.

With this encoding, no work is duplicated and few threads are left idle by branch divergence. Each thread decodes its integer independently, in time linear in the formula size, without communicating with other threads.

Concrete mapping. The bijection ϕ_s is locally consistent by construction. Consider a kernel batch for a binary operator at size s with a fixed left-subtree size j. Within this batch, thread tid decodes to left sub-expression $\phi_j(\lfloor \mathsf{tid}/\mathsf{T}[s{-}1{-}j][8] \rfloor)$ and right sub-expression $\phi_{s-1-j}(\mathsf{tid} \bmod \mathsf{T}[s{-}1{-}j][8])$. Hence, consecutive thread IDs share the same left sub-expression and differ only in the right sub-expression. Within a warp of 32 threads, all threads therefore:

1. evaluate the *same* left sub-expression, executing the same decode path and producing the same intermediate values, ensuring no branch divergence;
2. evaluate consecutive right sub-expressions, whose decoding traces differ only in the final variable index, so divergence is again minimal.

For unary operators, consecutive threads decode expressions that differ only in the last token of the child sub-expression, which similarly minimizes divergence within a warp.

3.3 General Applicability

Our technique is not specific to MBA. It applies to any search space of labeled trees that can be counted by size, such as the programs generated by a context-free grammar. Symbolic regression [Schmidt and Lipson, 2009] is one example: candidate formulas are trees over arithmetic operators, their outputs are real numbers, and caching fails for the same reasons as for MBA.

4 Experiments

We evaluate SIMBA along three axes: (1) comparison with existing MBA synthesis tools on a new, challenging benchmark suite; (2) a quantitative analysis of why cache-based approaches fail to scale, and how SIMBA's cache-free design overcomes this bottleneck; and (3) an ablation study isolating the contribution of the locally consistent encoding.

4.1 Experimental Setup

Benchmarks. The existing benchmark suite proposed in Menguy et al. [2021] turned out too easy for SIMBA, as it solved 90% of the instances in less than a second. We therefore created a new benchmark suite of 2340 randomly generated MBA expressions following the exact same generation procedure with larger parameters. Three parameters determine the difficulty of an MBA synthesis problem: the expression size (s), the number of variables (k), and the number of examples in the specification (n).

To build the dataset, we generate 10 random expressions for every combination of $s = 5, 6, \ldots, 30$ and $k = 2, \ldots, 10$. The generated expressions might not be minimal: there may be smaller equivalent formulas, and they may depend on fewer than k variables. For example, if $k = 4$, the expression $e = x_2 + (x_0 \wedge x_0)$ has size 5 and simplifies to $e' = x_2 + x_0$, of size 3, and actually uses only two variables. To accurately measure an instance's difficulty, we redefine its size based on the smallest known formula (comparing our generated expression against the solver's eventual solutions). We then record its true variable count by taking the highest variable index present in that minimal formula, plus one. The above example would therefore be reported with $s = 3$ and $k = 3$ (the highest variable index is 2).

The distributions of size and number of variables across our benchmark suite are shown in Fig. 5 (Section B), covering a wide range of problem sizes and complexities.

Finally, we construct specifications by evaluating each expression on $n = 16$ examples, using uniformly random 32-bit values for each input. With this value of n, a formula that is not equivalent to the original one is very unlikely to produce the same outputs, so in practice any solution to the synthesis problem is equivalent to the original formula.

Baselines. We compare SIMBA against the following state-of-the-art synthesis solvers, which, like SIMBA, take only input-output examples as input:

- **VFB**: an implementation of the GPU algorithm of Valizadeh et al. [2024] for MBA instead of LTL formulas.
- **cvc5** [Barbosa et al., 2022]: a CPU SMT-based SyGuS solver (bitvector theory).
- **EUSolver** [Alur et al., 2017]: a CPU enumerative SyGuS solver based on divide-and-conquer.

We do not include CVC4, which is no longer maintained and has been superseded by CVC5. Other MBA tools solve a different problem. Deobfuscators such as MBA-Blast [Liu et al., 2021b], GAMBA [Reichenwallner and Meerwald-Stadler, 2023], NeuReduce [Feng et al., 2020] and ProMBA [Lee and Lee, 2023] simplify a given obfuscated expression, so they cannot run on input-output examples alone. Stochastic synthesizers such as Syntia [Blazytko et al., 2017], Xyntia [Menguy et al., 2021] and XSmir [Attias et al., 2025] do take examples as input, but they rely on heuristic local search and guarantee neither completeness nor minimality. To quantify this, we ran Xyntia on our benchmark suite: among the instances it solved, its output was at least twice as large as the minimal expression in 56.41% of cases, and at least five times as large in 17.95% of cases. Comparing running times with such tools would therefore not be meaningful.

Hardware. All CPU and H100 GPU experiments were conducted on nodes equipped with two Intel Xeon Platinum 8468 CPUs (48 cores @2.10 GHz each), 512 GB RAM, and eight NVIDIA H100 SXM5 GPUs with 80 GB of memory each. Experiments on A100 GPUs were conducted on nodes equipped with two AMD Milan EPYC 7543 CPUs (32 cores @2.80 GHz each), 512 GB RAM and eight NVIDIA A100 SXM4 GPUs with 80 GB of memory each. We report wall-clock times for solving each instance, excluding the overhead for parsing the input. Results on A100 GPUs and a stability analysis over 5 runs are reported in Section E; they show consistent behaviour across both hardware platforms and very little run-to-run variance.

4.2 Comparison with Existing Tools

We run SIMBA, VFB and CVC5 on our dataset, with a time limit of 60 seconds. An instance counts as solved if the solver returns, within the time limit, an expression that matches all examples and whose size equals that of the smallest known formula. Since SIMBA enumerates expressions by increasing size, its solutions are minimal by construction. We report the results in Figure 1. SIMBA solves nearly all instances up to size 11 and is the only algorithm capable of solving instances of sizes 12 through 16. In contrast, the performance of VFB and CVC5 degrades rapidly after size 6, and cannot solve instances of size greater than 11. No program solved instances of size more than 16. Similar conclusions can be drawn observing the influence on the number of variables, see Section C.

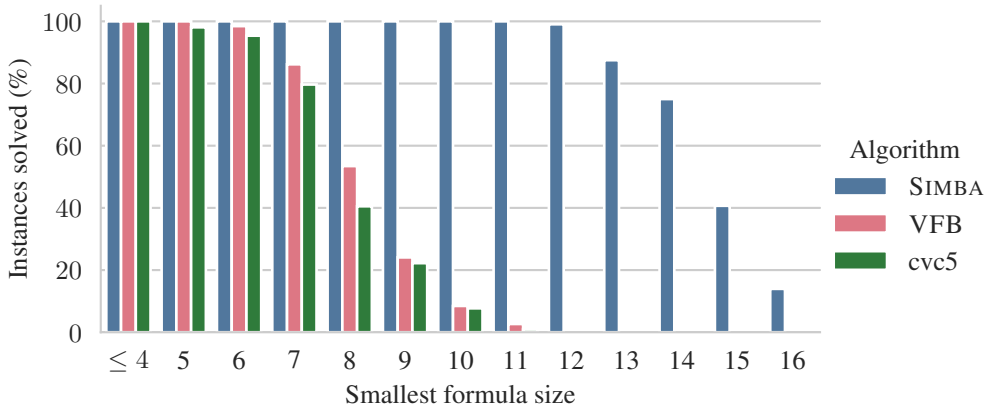

Figure 1: Percentage of instances solved by each algorithm across varying formula sizes.

In addition to the coverage gap, a head-to-head comparison of solving times reveals two further patterns. Against VFB: among the 615 instances solved by both, SIMBA is faster on 90% of them, with a median speedup of $1.6\times$. The other 634 instances solved by SIMBA are not solved by VFB. Against CVC5: on the 581 instances both solvers handle, SIMBA finishes in under one second on 96% of them (median: 0.67 s), indicating that they are rather simple instances. The remaining 668 instances solved only by SIMBA are beyond CVC5's reach within the time limit. Against EUSolver: it solves 514 instances (22.0%), against 1249 (53.4%) for SIMBA, and every instance it solves is also solved by SIMBA, always in under 2 seconds. On the instances solved by both, SIMBA is at least as fast on 75.66% of them, and at least 10 times faster on 56.53%.

SyGuS PBE-BV benchmarks. As an independent check, we also use the PBE-BV track of the SyGuS competition (2018 edition)[1]. Each of these instances comes with its own grammar, whereas SIMBA supports a single fixed MBA grammar, so we keep the 452 instances (out of 750) whose grammar is close to SIMBA's operator set. With a time limit of 60 seconds, SIMBA solves 96.4% of them, CVC5 90.7%, VFB 90.2% and EUSolver 88.3%. The gaps are smaller than on our suite, since these instances are small hand-written bitvector programs that existing tools already largely solve.

[1] `https://github.com/SyGuS-Org/benchmarks/tree/master/lib/PBE_BV_Track/from_2018`

4.3 Scalability

The scalability of SIMBA stems from its cache-free design. VFB maintains a language cache of all intermediate sub-expressions encountered during enumeration: each new expression of size s is built from cached sub-expressions of smaller sizes. This enables deduplication of semantically equivalent sub-expressions, but at the cost of memory that grows super-exponentially with formula size.

Memory scaling of cache-based approaches. Table 1 shows, for $k = 5$ variables, how the number of cached formulas and the resulting memory footprint grow with the enumeration size. VFB stores the observational behaviour of all expressions in the cache; it exhausts GPU memory already at size 11. Below that threshold, the cache reaches 2.1 GB at size 10. Memory therefore runs out well before VFB reaches the sizes that SIMBA solves.

Diminishing returns of deduplication. Table 1 also shows the fraction of all possible formulas actually stored in the cache, as well as the cumulative median time VFB spends to reach each size. At size 2 the cache is complete (100%), but the proportion falls rapidly as the search space grows exponentially. By size 11 the cache would need to store over 4 billion formulas, crashing the GPU long before that point. At size 10 only 7.4% of the 439 million candidate formulas are cached, yet those 32 million entries already occupy 2.1 GB. The search space grows faster than deduplication can prune it, so cache-based approaches are limited by memory.

Table 1: Cache size and efficiency of VFB for $k = 5$ variables. VFB runs out of GPU memory (OOM) once the number of stored formulas is too large, making it unable to search larger formula sizes.

Size	#MBA	#VFB cache	VFB mem	% cached	VFB cum. time (s)
2	15	15	<1 MB	100.0%	1.2
3	185	99	<1 MB	53.5%	2.4
4	875	166	<1 MB	19.0%	3.4
5	8,805	1,749	<1 MB	19.9%	4.3
6	60,715	6,874	<1 MB	11.3%	5.5
7	663,785	115,080	7.4 MB	17.3%	6.5
8	5,062,975	504,522	32.3 MB	10.0%	7.4
9	50,895,805	5,547,921	355.1 MB	10.9%	8.5
10	438,822,815	32,523,385	2.1 GB	7.4%	9.3
11	4,472,457,185	OOM	OOM	OOM	OOM

Cache-free scalability. SIMBA requires no cache at all: each thread independently decodes and evaluates its assigned expression on-the-fly, then discards it. The memory footprint is therefore constant with respect to formula size, bounded only by the GPU's register file and constant memory (used for a small pre-computed table of expression counts). This lets SIMBA enumerate expressions of sizes that would be inaccessible to cache-based approaches under any practical memory budget. The results in Fig. 1 confirm this: SIMBA is the only solver capable of solving instances of sizes 12 through 16.

Results for other variable counts ($k \in \{3, 4, 6, 7, 8\}$) follow the same pattern and are given in Section D.

4.4 Ablation Study

Effect of the Locally Consistent Encoding. Recall that SIMBA assigns each GPU thread a unique integer identifier, which it decodes on-the-fly into an MBA expression. This encoding is designed to be *locally consistent*: consecutive thread identifiers decode to structurally similar expressions. Since threads within the same GPU warp share a contiguous range of identifiers, they evaluate expressions that differ only in their input variables, following nearly identical execution paths. This minimizes branch divergence within the warp and keeps the GPU hardware fully utilized.

To measure the effect of this design choice, we introduce SIMBA-RTID, a variant of SIMBA in which thread identifiers are pseudo-randomly permuted before decoding. This shuffling preserves

the coverage of the search space: every expression is still evaluated exactly once, but destroys the locality structure since threads within the same warp now process structurally diverse expressions, inducing significant branch divergence.

To permute the threads, we use simple modular arithmetic: the thread with original ID i gets assigned the ID $\pi(i)$, where

$$\pi(i) = (i \cdot 2246822507) \mod n_{threads}.$$

The constant $C = 2246822507$ is taken from the MurmurHash3 hash function [Appleby, 2011]. Since the number of threads is always even, C is coprime with $n_{threads}$, and π is indeed a permutation over $\{0, \ldots, n_{threads} - 1\}$. The results of this experiment are reported in Fig. 2. We observe that SIMBA solves all instances that SIMBA-RTID solves, and is typically four times faster on them.

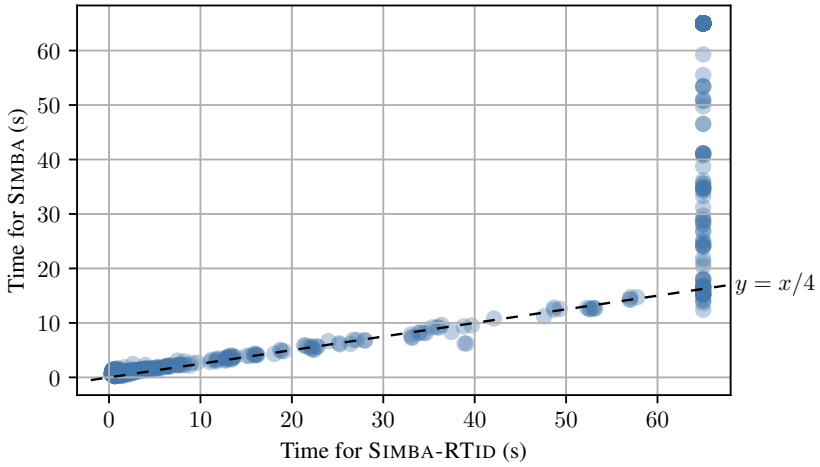

Figure 2: Comparison of the solving time of SIMBA-RTID and SIMBA on each instance. Each circle represents an instance; its x-coordinate is the solving time for SIMBA-RTID and its y-coordinate the time for SIMBA. The lines $x = 65$ and $y = 65$ correspond to timeouts of SIMBA-RTID and SIMBA, respectively. Most instances solved by both programs lie on the $y = x/4$ line, which means that SIMBA is 4 times faster than SIMBA-RTID.

5 Related Work

MBA obfuscation and deobfuscation. Mixed-Boolean Arithmetic was introduced as an obfuscation technique by Zhou et al. [2007], who observed that mixing bitwise and arithmetic operations creates expressions that are difficult to simplify. Eyrolles [2017] studied MBA obfuscation in depth, including reconstruction and simplification tools. MBA-Blast [Liu et al., 2021b] proposed a simplification method based on identifying the linear structure of MBA expressions, decomposing them into a combination of basis expressions. GAMBA [Reichenwallner and Meerwald-Stadler, 2023] extended this to general (not just linear) MBA expressions using an algebraic approach. Reichenwallner and Meerwald-Stadler [2022] gave efficient algorithms specifically for linear MBA deobfuscation. More recent work explores non-linear MBA obfuscation [Liu et al., 2021a] and architectural hardening against deobfuscation [Wei et al., 2026]. Formal verification of MBA obfuscation has been studied by Blazy and Hutin [2019].

Program synthesis for deobfuscation. Syntia [Blazytko et al., 2017] was the first to use stochastic program synthesis (Monte Carlo Tree Search) for black-box deobfuscation of binary code, including MBA-obfuscated expressions. Xyntia [Menguy et al., 2021] analyzed and improved local search-based deobfuscation, and proposed mitigations against it. QSynth [David et al., 2020] refined this approach with better sampling and synthesis strategies. Lee and Lee [2023] combined program synthesis with term rewriting for MBA deobfuscation. XSmir [Attias et al., 2025] augmented search-based synthesis with local inference rules to improve effectiveness. NeuReduce [Feng et al., 2020] took a neural approach, training a recurrent neural network to simplify MBA expressions, trading soundness for speed.

GPU-accelerated program synthesis. Valizadeh and Berger [2023] demonstrated the first GPU-accelerated program synthesizer for regular expression inference, using bitvector representations and a language cache to achieve orders-of-magnitude speedups. Valizadeh et al. [2024] extended this to LTL formula learning, introducing divide-and-conquer and relaxed uniqueness checks (RUCs) to scale further. Berger et al. [2025] presented a general framework for GPU-accelerated synthesis based on enumerating semantics rather than syntax. All these works operate in qualitative domains where observational equivalence classes can be compactly represented and cached. SIMBA is the first to address a quantitative domain, where such caches do not work and a cache-free algorithm is needed.

General program synthesis. Bottom-up enumerative synthesis [Alur et al., 2013, Udupa et al., 2013] is a foundational technique that systematically generates programs of increasing size. Observational equivalence reduction [Udupa et al., 2013] is the standard pruning technique: only one representative per equivalence class is kept. Syntax-guided synthesis (SyGuS) [Alur et al., 2013] provides a formal framework and annual competition. State-of-the-art SyGuS solvers like cvc5 [Barbosa et al., 2022] use sophisticated enumeration strategies but are CPU-based and do not exploit GPU parallelism.

6 Conclusion

We presented SIMBA, the first GPU-accelerated synthesizer for Mixed-Boolean Arithmetic expressions. Its cache-free enumeration removes the main limitation of prior GPU-based synthesizers, namely their reliance on language caches, which do not scale in quantitative domains. The same approach should apply to other quantitative domains where the output space is too large for caching.

Limitations. SIMBA has several limitations. First, the approach requires a GPU: the algorithm is designed around massively parallel thread execution and provides no benefit on CPU-only hardware. Second, despite the cache-free design, the number of candidate expressions still grows super-exponentially with formula size, so SIMBA remains practical only up to sizes around 16 in our experiments; very large target expressions remain out of reach within reasonable time budgets. Finally, SIMBA does not handle approximate or noisy oracles, and cannot yet synthesize expressions with numerical constants. Supporting constants would not require changing the enumeration: since candidates are evaluated on the fly rather than cached, a constant-solving step (for instance, solving linear equations over the examples) could be added to the evaluation kernel. This is harder for cache-based approaches, whose pruning relies on each candidate having a fixed behaviour on the examples.

Broader impacts. SIMBA is intended for defensive security applications such as malware deobfuscation, compiler superoptimization, and reverse engineering. Like any synthesis tool, it could in principle be used to generate or validate obfuscated code; however, obfuscation tools are already widely available, so we do not anticipate that SIMBA meaningfully lowers that barrier. We do not foresee significant negative societal impacts beyond those of existing program analysis tools.

Acknowledgments and Disclosure of Funding

The authors would like to thank Baptiste Mouillon for his comments on the initial algorithm design. This work was supported by the French PEPR Intelligence Artificielle SAIF project (ANR-23-PEIA-0006). The authors declare no competing interests.

References

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013. URL `https://ieeexplore.ieee.org/document/6679385/`.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336. Springer, 2017. doi: 10.1007/978-3-662-54577-5_18.

Austin Appleby. Murmurhash3 and smhasher. `https://github.com/aappleby/smhasher`, 2011. Accessed: 2026-04-24.

Vidal Attias, Nicolas Bellec, Grégoire Menguy, Sébastien Bardin, and Jean-Yves Marion. Augmenting search-based program synthesis with local inference rules to improve black-box deobfuscation. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 4529–4543. ACM, 2025. doi: 10.1145/3719027.3765134. URL `https://doi.org/10.1145/3719027.3765134`.

Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 394–403. ACM, 2006. doi: 10.1145/1168857.1168906. URL `https://doi.org/10.1145/1168857.1168906`.

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *Proc. of TACAS 2022, Part of ETAPS 2022*, volume 13243 of *LNCS*, pages 415–442. Springer, 2022. doi: 10.1007/978-3-030-99524-9_24. URL `https://doi.org/10.1007/978-3-030-99524-9_24`.

Martin Berger, Nathanaël Fijalkow, and Mojtaba Valizadeh. GPU accelerated program synthesis: Enumerate semantics, not syntax! *CoRR*, abs/2504.18943, 2025. doi: 10.48550/ARXIV.2504.18943. URL `https://doi.org/10.48550/arXiv.2504.18943`.

Sandrine Blazy and Rémi Hutin. Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 196–208. ACM, 2019. doi: 10.1145/3293880.3294103. URL `https://doi.org/10.1145/3293880.3294103`.

Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 643–659. USENIX Association, 2017. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko`.

Cristina David and Daniel Kroening. Program synthesis: Challenges and opportunities. *Philosophical Transactions of the Royal Society A*, 375(2104):20150403, 2017. doi: 10.1098/rsta.2015.0403.

Robin David, Luigi Coniglio, and Mariano Ceccato. QSynth – a program synthesis based approach for binary code deobfuscation. In *Workshop on Binary Analysis Research (BAR)*. NDSS, 2020.

Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université Paris-Saclay, 2017.

Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. NeuReduce: Reducing mixed boolean-arithmetic expressions by recurrent neural network. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, Findings of ACL, pages 635–644. Association for Computational Linguistics, 2020. doi: 10.18653/V1/2020.FINDINGS-EMNLP.56. URL `https://doi.org/10.18653/v1/2020.findings-emnlp.56`.

Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming*, 23(5):594–628, 2013.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010.

Gary D Knott. A numbering system for binary trees. *Communications of the ACM*, 20(2):113–115, 1977.

Jaehyung Lee and Woosuk Lee. Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2351–2365. ACM, 2023. doi: 10.1145/3576915.3623186. URL https://doi.org/10.1145/3576915.3623186.

Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. Software obfuscation with non-linear mixed Boolean-arithmetic expressions. In *Information and Communications Security, 23rd International Conference, ICICS 2021*, Lecture Notes in Computer Science, pages 276–292. Springer, 2021a. doi: 10.1007/978-3-030-86890-1_16.

Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast: Unveiling and simplifying mixed boolean-arithmetic obfuscation. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1701–1718. USENIX Association, 2021b. URL https://www.usenix.org/conference/usenixsecurity21/presentation/liu-binbin.

Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Search-based local black-box deobfuscation: Understand, improve and mitigate. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2513–2525. ACM, 2021. doi: 10.1145/3460120.3485250. URL https://doi.org/10.1145/3460120.3485250.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008. doi: 10.1145/1365490.1365500. URL https://doi.org/10.1145/1365490.1365500.

Benjamin Reichenwallner and Peter Meerwald-Stadler. Efficient deobfuscation of linear mixed boolean-arithmetic expressions. In Golden G. Richard III and Tim Blazytko, editors, *Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks, Los Angeles, CA, USA, 11 November 2022*, pages 19–28. ACM, 2022. doi: 10.1145/3560831.3564256. URL https://doi.org/10.1145/3560831.3564256.

Benjamin Reichenwallner and Peter Meerwald-Stadler. Simplification of general mixed boolean-arithmetic expressions: GAMBA. In *IEEE European Symposium on Security and Privacy, EuroS&P 2023 - Workshops, Delft, Netherlands, July 3-7, 2023*, pages 427–438. IEEE, 2023. doi: 10.1109/EUROSPW59978.2023.00053. URL https://doi.org/10.1109/EuroSPW59978.2023.00053.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009. doi: 10.1126/science.1165893. URL https://www.science.org/doi/abs/10.1126/science.1165893.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296. ACM, 2013. doi: 10.1145/2491956.2462174. URL https://doi.org/10.1145/2491956.2462174.

Mojtaba Valizadeh and Martin Berger. Search-based regular expression inference on a GPU. *Proc. ACM Program. Lang.*, 7(PLDI):1317–1339, 2023. doi: 10.1145/3591274. URL https://doi.org/10.1145/3591274.

Mojtaba Valizadeh, Nathanaël Fijalkow, and Martin Berger. LTL learning on GPUs. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*, Lecture Notes in Computer Science, pages 209–231. Springer, 2024. doi: 10.1007/978-3-031-65633-0_10. URL https://doi.org/10.1007/978-3-031-65633-0_10.

Haoming Wei, Tengyue Liu, Siyi Wang, and Chuanping Hu. Unifying mixed Boolean-arithmetic obfuscation by architectural and anti-generalization hardening. *Computers & Security*, 167:104866, 2026. doi: 10.1016/j.cose.2026.104866.

Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers*, Lecture Notes in Computer Science, pages 61–75. Springer, 2007. doi: 10.1007/978-3-540-77535-5_5. URL https://doi.org/10.1007/978-3-540-77535-5_5.

A Worked Example of the Bijection

We illustrate the decoding function ϕ_s of Section 3.1 on a small instance with $k = 2$ variables x_0, x_1. We first decode a unary expression, then a binary expression; the binary case reuses the unary one for its left operand. Throughout, operator slots are ordered as in Algorithm 1: $\sim, \wedge, \vee, \oplus, -, +, -, \times$. In RPN sequences, we write the unary minus token as `neg` to distinguish it from subtraction.

Counting table. Table 2 gives the counting table T of Equation 1 for $k = 2$ and sizes up to 4. At size 2, only unary operators can occur, each applied to one of the 2 variables. At size 3, the only split for binary operators is $(1, 1)$, so each binary operator has $\mathsf{T}[1][8] \cdot \mathsf{T}[1][8] = 4$ expressions. At size 4, commutative operators only use the split $(j, s-1-j) = (1, 2)$, giving $\mathsf{T}[1][8] \cdot \mathsf{T}[2][8] = 8$ expressions, while subtraction uses both splits $(1, 2)$ and $(2, 1)$, giving $8 + 8 = 16$ expressions.

Table 2: Counting table $\mathsf{T}[s][op]$ for $k = 2$ variables. The last column is the total $\mathsf{T}[s][8]$.

s	$\sim$	$\wedge$	$\vee$	$\oplus$	$-$	$+$	$-$	$\times$	total
1	0	0	0	0	0	0	0	0	2
2	2	0	0	0	2	0	0	0	4
3	4	4	4	4	4	4	4	4	32
4	32	8	8	8	32	8	16	8	120

Unary example: decoding $\phi_3(2)$. The decoding is illustrated in Fig. 3.

1. *Top operator.* At size 3, the operator blocks are $[0, 4)$ for $\sim$, $[4, 8)$ for $\wedge$, and so on. Since $2 < \mathsf{T}[3][\sim] = 4$, the top operator is $\sim$, with $n' = 2 - 0 = 2$.
2. *Child.* Since $\sim$ is unary, we decode $\phi_2(2)$. At size 2, the non-empty blocks are $[0, 2)$ for $\sim$ and $[2, 4)$ for $-$. Hence the operator is $-$, with $n' = 2 - 2 = 0$.
3. *Leaf.* Finally, $\phi_1(0) = x_0$.

Hence $\phi_2(2) = -x_0$ and $\phi_3(2) = \sim(-x_0)$, i.e., the RPN sequence x_0 `neg` $\sim$.

Binary example: decoding $\phi_4(109)$. The decoding is illustrated in Fig. 4.

1. *Top operator.* The cumulative sums of row $s = 4$ of Table 2 give the blocks $[0, 32)$ for $\sim$, $[32, 40)$ for $\wedge$, $[40, 48)$ for $\vee$, $[48, 56)$ for $\oplus$, $[56, 88)$ for $-$, $[88, 96)$ for $+$, $[96, 112)$ for $-$ and $[112, 120)$ for $\times$. Since $109 \in [96, 112)$, the top operator is $-$, with $n' = 109 - 96 = 13$.
2. *Split.* The two operands have total size $s - 1 = 3$. The split with a left operand of size $j = 1$ covers $\mathsf{T}[1][8] \cdot \mathsf{T}[2][8] = 8$ expressions, i.e., the range $[0, 8)$, and the split with $j = 2$ covers the next $\mathsf{T}[2][8] \cdot \mathsf{T}[1][8] = 8$ expressions, i.e., $[8, 16)$. Since $n' = 13 \in [8, 16)$, we get $j = 2$ and $n'' = 13 - 8 = 5$.

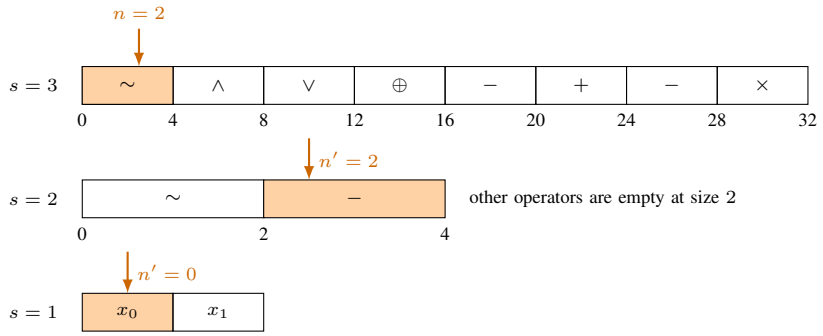

Figure 3: Decoding $\phi_3(2)$ with $k = 2$. At each level, the current index falls in the block of one operator (highlighted); the index relative to the start of that block is passed to the child. The result is $\sim(-x_0)$.

3. *Operands.* The right operand has size 1, with $\mathsf{T}[1][8] = 2$ expressions. The left operand is therefore $\phi_2(\lfloor 5/2 \rfloor) = \phi_2(2) = -x_0$, which we decoded in the unary example, and the right operand is $\phi_1(5 \bmod 2) = \phi_1(1) = x_1$.

Hence $\phi_4(109) = (-x_0) - x_1$, i.e., the RPN sequence x_0 `neg` x_1 $-$.

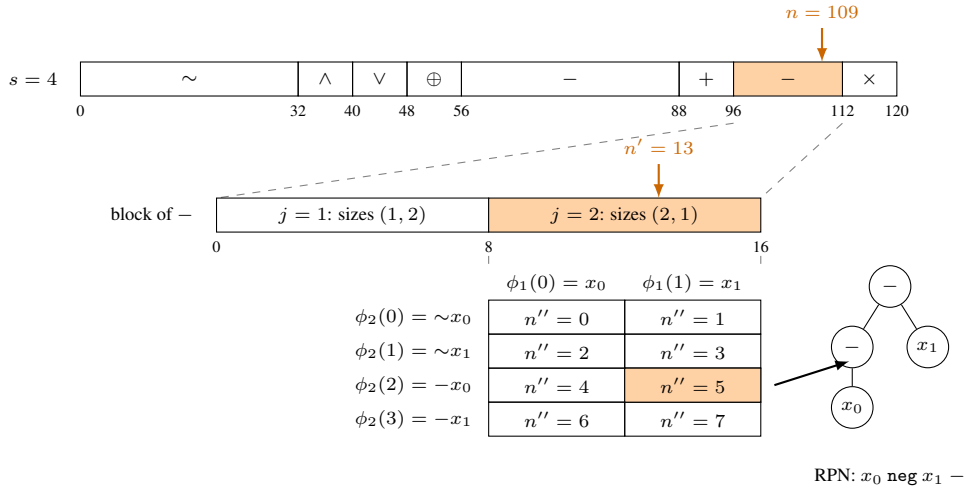

Figure 4: Decoding $\phi_4(109)$ with $k = 2$. Top: the index 109 falls in the block of $-$, at offset $n' = 13$. Middle: inside this block, $n' = 13$ falls in the split where the left operand has size $j = 2$, at offset $n'' = 5$. Bottom: the split block is a grid indexed by (left index, right index) $= (\lfloor n''/2 \rfloor, n'' \bmod 2) = (2, 1)$, which gives the expression $(-x_0) - x_1$.

Local consistency. The grid of Fig. 4 also illustrates why consecutive thread identifiers decode to structurally similar expressions. Table 3 lists the expressions decoded from identifiers 104 to 111, i.e., the whole block of the split $j = 2$ for subtraction. Consecutive identifiers share the same left operand, and differ only in the right operand, which is the last variable of the RPN sequence. In particular, all these sequences have the same shape, so the corresponding threads execute the same decoding and evaluation path.

Commutative operators. For a commutative operator, only splits with $j \leq s - 1 - j$ are counted, so the left operand is never larger than the right one. For instance, decoding $\phi_4(117)$ gives the top operator $\times$ (block $[112, 120)$, so $n' = 5$), the only split $j = 1$, and operands $\phi_1(\lfloor 5/4 \rfloor) = x_1$ and $\phi_2(5 \bmod 4) = \phi_2(1) = \sim x_1$, i.e., $x_1 \times (\sim x_1)$. The equivalent expression $(\sim x_1) \times x_1$, whose left operand is larger than its right one, is never enumerated.

Table 3: Expressions decoded from consecutive identifiers 104 to 111 at size 4 with $k = 2$. Pairs of consecutive identifiers share their left operand.

identifier n	$(\lfloor n''/2 \rfloor, n'' \bmod 2)$	expression	RPN
104	$(0, 0)$	$(\sim x_0) - x_0$	$x_0 \sim x_0 -$
105	$(0, 1)$	$(\sim x_0) - x_1$	$x_0 \sim x_1 -$
106	$(1, 0)$	$(\sim x_1) - x_0$	$x_1 \sim x_0 -$
107	$(1, 1)$	$(\sim x_1) - x_1$	$x_1 \sim x_1 -$
108	$(2, 0)$	$(-x_0) - x_0$	x_0 `neg` $x_0 -$
109	$(2, 1)$	$(-x_0) - x_1$	x_0 `neg` $x_1 -$
110	$(3, 0)$	$(-x_1) - x_0$	x_1 `neg` $x_0 -$
111	$(3, 1)$	$(-x_1) - x_1$	x_1 `neg` $x_1 -$

B Benchmark Suite

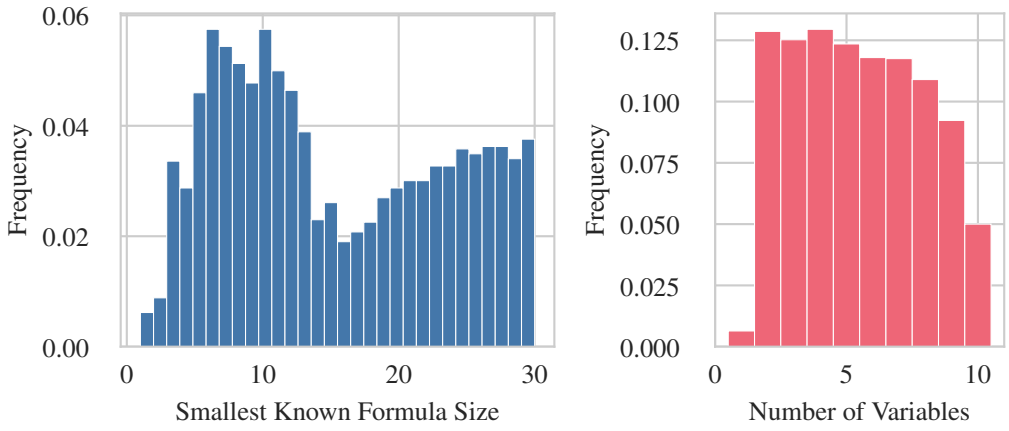

Figure 5: Distributions of formula size and number of variables across the benchmark suite.

C Comparison with Existing Tools

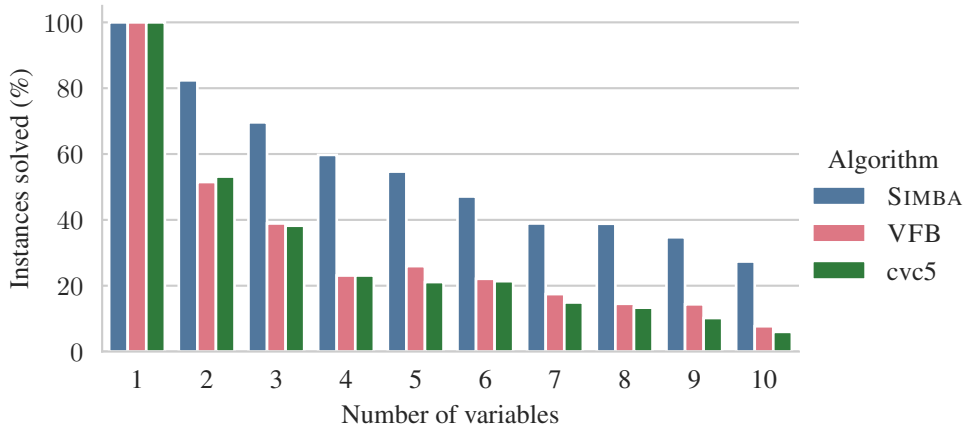

Figure 6: Percentage of instances solved by each algorithm across varying numbers of variables.

D Cache Size and Efficiency Tables

The following tables complement Table 1 in the main body by reporting cache size and efficiency for VFB across all variable counts $k \in \{3, 4, 6, 7, 8\}$.

Table 4: Cache size and efficiency of VFB for $k = 3$ variables.

Size	#MBA	#VFB cache	VFB mem	% cached	VFB cum. time (s)
2	9	9	<1 MB	100.0%	1.2
3	75	38	<1 MB	50.7%	2.4
4	333	71	<1 MB	21.3%	3.4
5	2,451	554	<1 MB	22.6%	4.3
6	14,877	2,112	<1 MB	14.2%	5.5
7	121,179	21,981	1.4 MB	18.1%	6.5
8	802,881	97,071	6.2 MB	12.1%	7.4
9	6,298,419	756,167	48.4 MB	12.0%	8.5
10	45,635,841	4,079,306	261.1 MB	8.9%	9.3
11	366,816,891	30,968,139	2.0 GB	8.4%	10.0
12	2,757,233,061	OOM	OOM	OOM	OOM

Table 5: Cache size and efficiency of VFB for $k = 4$ variables.

Size	#MBA	#VFB cache	VFB mem	% cached	VFB cum. time (s)
2	12	12	<1 MB	100.0%	1.2
3	124	65	<1 MB	52.4%	2.4
4	572	114	<1 MB	19.9%	3.4
5	4,988	1,051	<1 MB	21.1%	4.3
6	32,636	4,086	<1 MB	12.5%	5.5
7	311,932	55,253	3.5 MB	17.7%	6.5
8	2,243,196	244,245	15.6 MB	10.9%	7.4
9	20,140,668	2,298,267	147.1 MB	11.4%	8.5
10	161,176,188	13,036,731	834.4 MB	8.1%	9.3
11	1,475,205,756	31,589,728	2.0 GB	2.1%	10.0
12	12,299,156,092	OOM	OOM	OOM	OOM

Table 6: Cache size and efficiency of VFB for $k = 6$ variables.

Size	#MBA	#VFB cache	VFB mem	% cached	VFB cum. time (s)
2	18	18	<1 MB	100.0%	1.2
3	258	140	<1 MB	54.3%	2.4
4	1,242	227	<1 MB	18.3%	3.4
5	14,154	2,676	<1 MB	18.9%	4.3
6	101,466	10,586	<1 MB	10.4%	5.5
7	1,246,650	212,216	13.6 MB	17.0%	6.5
8	9,941,850	919,469	58.8 MB	9.2%	7.4
9	110,265,882	11,546,211	739.0 MB	10.5%	8.5
10	1,007,929,818	32,230,181	2.1 GB	3.2%	9.3
11	11,272,896,474	OOM	OOM	OOM	OOM

E Hardware Comparison

Figures 7 and 8 compare the number of instances solved within each time threshold by SIMBA and VFB on H100 and A100 GPUs respectively. The results are consistent across both hardware configurations: SIMBA substantially outperforms VFB on both GPUs, solving roughly twice as many

Table 7: Cache size and efficiency of VFB for $k = 7$ variables.

Size	#MBA	#VFB cache	VFB mem	% cached	VFB cum. time (s)
2	21	21	<1 MB	100.0%	1.2
3	343	188	<1 MB	54.8%	2.4
4	1,673	297	<1 MB	17.8%	3.4
5	21,287	3,860	<1 MB	18.1%	4.3
6	157,241	15,332	<1 MB	9.8%	5.5
7	2,142,679	359,149	23.0 MB	16.8%	6.5
8	17,695,293	1,536,260	98.3 MB	8.7%	7.4
9	214,233,831	21,655,933	1.4 GB	10.1%	8.5
10	2,053,008,573	OOM	OOM	OOM	OOM

Table 8: Cache size and efficiency of VFB for $k = 8$ variables.

Size	#MBA	#VFB cache	VFB mem	% cached	VFB cum. time (s)
2	24	24	<1 MB	100.0%	1.2
3	440	243	<1 MB	55.2%	2.4
4	2,168	376	<1 MB	17.3%	3.4
5	30,456	5,329	<1 MB	17.5%	4.3
6	230,392	21,222	1.4 MB	9.2%	5.5
7	3,446,264	570,096	36.5 MB	16.5%	6.5
8	29,274,616	2,407,325	154.1 MB	8.2%	7.4
9	383,703,544	33,244,298	2.1 GB	8.7%	8.5
10	3,823,512,056	OOM	OOM	OOM	OOM

instances within the 60-second limit. The absolute solving times are slightly faster on H100 GPUs than on A100 GPUs, as expected from the higher compute throughput of the H100, but the relative advantage of SIMBA over VFB is preserved across both platforms. The VFB curves stop early on both plots because VFB exhausts GPU memory (OOM): on the H100 this happens after about 2 seconds, while on the A100 the lower compute throughput delays reaching the same memory limit to about 6 seconds. Beyond that point, VFB solves no further instances.

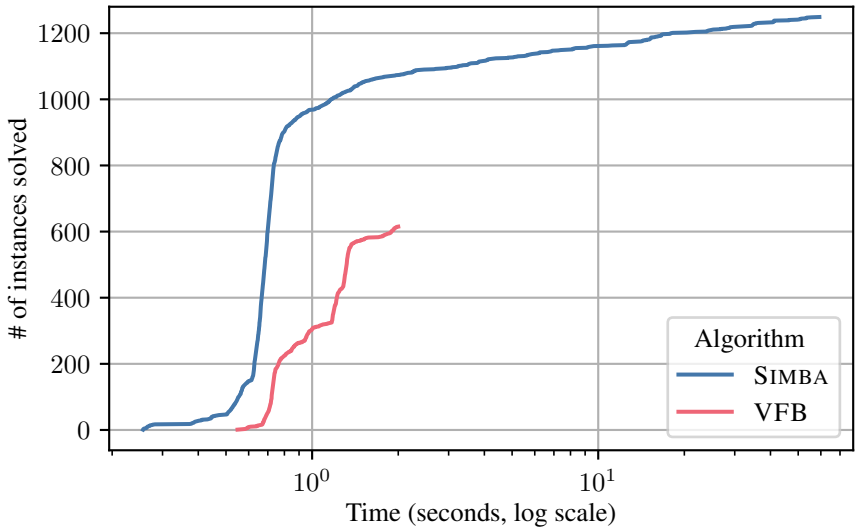

Figure 7: Number of instances solved under each time threshold (H100). The VFB curve ends when VFB runs out of GPU memory.

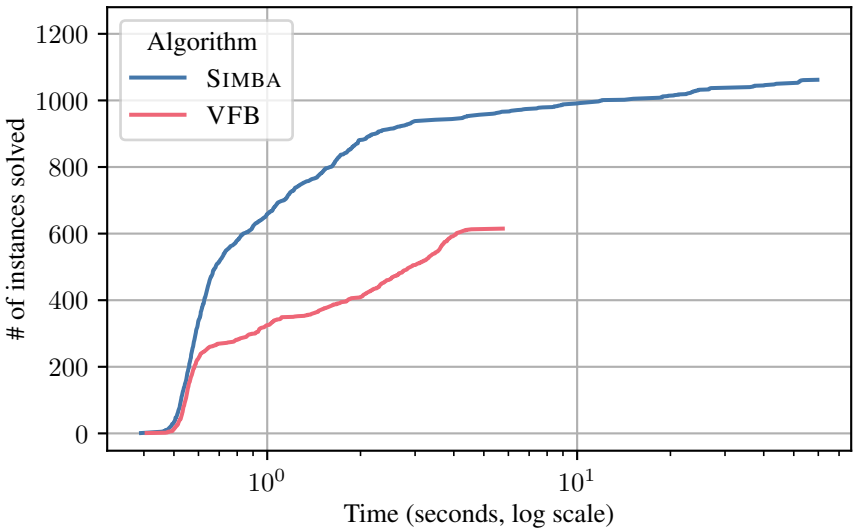

Figure 8: Number of instances solved under each time threshold (A100). The VFB curve ends when VFB runs out of GPU memory.

Stability of SIMBA. Figure 9 reports the number of instances solved by SIMBA within each time threshold, averaged over 5 independent runs on the H100 GPU. The shaded band, representing $\pm$ one standard deviation, is barely visible throughout the curve, indicating that SIMBA's performance is highly reproducible. This is expected given that SIMBA is a deterministic algorithm: the only source of variance is the scheduling jitter of the GPU runtime, which is negligible at the scale of our experiments.

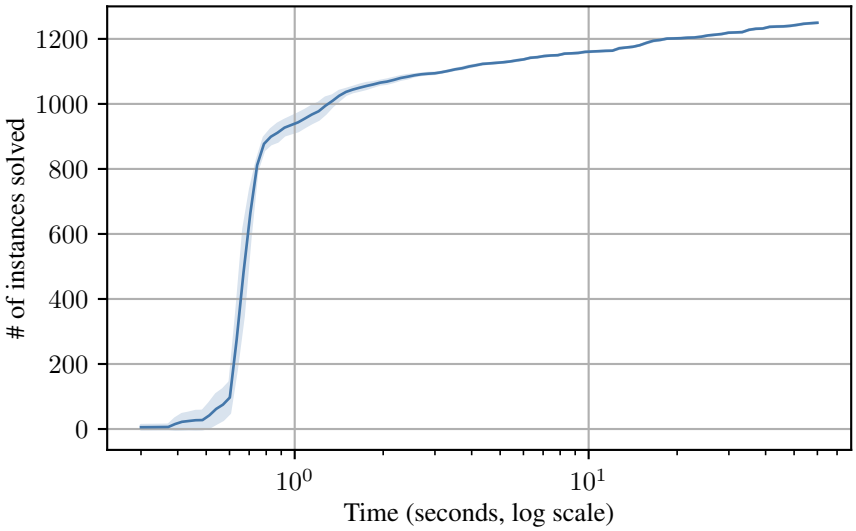

Figure 9: Number of instances solved by SIMBA under each time threshold on H100 GPU, averaged over 5 independent runs. The shaded area shows $\pm$ standard deviation.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state three concrete claims: cache-free GPU synthesis, a bijective enumeration encoding, and substantial speedups over prior MBA tools. All of them are directly supported by the algorithmic description in Section 3 and the experimental results in Section 4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in Section 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include formal theorems or proofs. The bijection between integers and MBA expressions is described constructively in Section 3 with sufficient detail to verify correctness, but is not presented as a formal theorem.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The algorithm is fully described in Section 3 (including pseudocode). Hardware configurations, baseline tools, benchmark generation procedure, time limit, and evaluation metrics are all specified in Section 4. Source code is publicly available at `https://github.com/SynthesisLab/SIMBA`.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The source code of SIMBA, together with the scripts used to generate the benchmark data and to run the baselines and the experiments, is publicly available at `https://github.com/SynthesisLab/SIMBA`, with instructions to reproduce the main experimental results.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 specifies the benchmark generation procedure (parameters s, k, n), the number of specifications ($n = 16$, 32-bit), the time limit (60 s), GPU/CPU hardware, and all baselines. SIMBA has no learnable parameters; the only configuration is the size bound, which is implicitly the timeout.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: SIMBA is a deterministic algorithm; run-to-run variance arises only from GPU scheduling jitter. Section E reports a stability analysis over 5 independent runs with ± 1 standard deviation bands, showing negligible variance. The main comparison metrics (percentage of instances solved) are therefore effectively deterministic.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 4 specifies the GPU models (NVIDIA H100 SXM5, A100 SXM4), GPU memory (80 GB each), CPU models, and RAM for both hardware configurations used. Wall-clock time is reported per instance with a 60-second time limit. Running all experiments requires about 20 hours of CPU usage (for dataset generation and running CVC5) and 5 hours of GPU (for running VFB and SIMBA). We estimate that the whole research project consumed 5 times this compute budget for preliminary experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research involves no human subjects, no data collection from individuals, and no datasets raising privacy or consent concerns. The work has been reviewed against the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Broader impacts are discussed in Section 6, covering positive applications (malware deobfuscation, compiler optimization, reverse engineering) and the limited dual-use risk (obfuscation tools are already widely available).

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper releases a synthesis tool and a benchmark suite of randomly generated mathematical expressions. Neither poses a high risk for misuse comparable to pre-trained generative models or scraped personal data.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All tools and datasets used are properly cited: cvc5 [Barbosa et al., 2022] (MIT license), EUSolver [Alur et al., 2017] (cited), the SyGuS-Comp PBE-BV benchmarks (linked), the Xyntia benchmark suite [Menguy et al., 2021] (cited), and the GPU hardware. Our new benchmark is generated from scratch following the same procedure as Menguy et al. [2021] with different parameters.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The new benchmark suite of 2340 MBA expressions is described in Section 4 (generation procedure, parameters, size distribution). The SIMBA source code and the scripts generating the benchmark suite are documented and publicly available at `https://github.com/SynthesisLab/SIMBA`.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human subjects; no IRB approval is required.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are not used as any component of the core methodology. Any use was limited to writing assistance and does not affect the scientific content.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.