

# Parse this! Summoning Context-Sensitive Inputs with GOBLIN

Anonymous Author(s)

## Abstract

Grammar-based fuzzers have shown immense promise in identifying bugs in software systems that have highly-structured and intricate input formats (e.g., XML). Many of the existing grammar-based fuzzers rely on context-free grammars (CFGs) to represent the target’s input structure. CFGs, however, are often insufficient to precisely capture many application input formats containing context-sensitive constraints. Application-specific fuzzers, albeit effective, lack generality to be adapted to new applications. In this paper, we present GOBLIN, a new input generation language and tool that helps bridge this gap. Given a context-free grammar annotated with semantic constraints, GOBLIN generates inputs that both conform to the grammar and satisfy the constraints. While a few prior techniques target this problem, our method is distinguished by: (i) support for constraint solving over arbitrary SMT theories (e.g., bitvectors, integers, strings); (ii) a minimal core input language with formal semantics that is smaller and less complex than prior work; and (iii) a shift from global constraints to local, production rule constraints, which enables easier integration with certain fuzzing workflows. GOBLIN’s input generation approach is inspired by DPLL-style SAT solvers and enjoys the following formal guarantees: *solution soundness*, *solution completeness*, and *refutation soundness*. In addition to comparing GOBLIN with prior work, we demonstrate its effectiveness by incorporating it into a grammar-based network protocol fuzzer.

## 1 Introduction

Real-world software systems, such as compilers, file parsers, and network protocol stacks, often have highly complex input specifications. These systems are notoriously difficult to test robustly, and oversights in testing can lead to unintended behavior and security vulnerabilities [23, 25–28]. Fuzzing is a common testing approach, but current approaches face significant limitations. Greybox and whitebox fuzzers leverage information from binaries and source code to help craft test cases. However, these resources may not be available, particularly when testing proprietary software.

In blackbox settings, one then needs to capture the input specification without access to internal information from the system under test. Here, a common approach is **grammar-based fuzzing** [18], where context-free grammars (CFGs) are used to capture input structure. However, while CFGs are often sufficient to describe syntactic requirements on input, they lack the expressiveness to specify **semantic constraints**, which require context sensitivity. For instance, a network message might contain a variable-length payload field  $f_1$  and a length field  $f_2$ , where the contents of  $f_2$  must equal the length of  $f_1$  (called a **length constraint**). For many common file formats (e.g. CSV, XML, and more), performing context-free grammar-based fuzzing without taking these semantic constraints into account will lead to virtually none of the generated inputs passing early parsing stages of the system under test [30].

In order to capture more expressive constraints, prior work relies on fuzzers tailor-made for their target applications (e.g., CSmith

[33] and Frankencert [8]). While these fuzzers are often very effective, they are not easily generalizable. To that end, we present GOBLIN (**G**rammar-**O**riented **B**ranching with **L**ogical **I**nference and **n**-**T**hreading), a new input generation language and tool that helps bridge this gap. Given a context-free grammar annotated with semantic constraints, GOBLIN generates inputs that both conform to the grammar and satisfy the constraints. While a few prior techniques target this problem, our method is distinguished by: (i) support for constraints over **arbitrary Satisfiability Modulo Theory (SMT)** [7] **theories** (e.g., bitvectors, integers, strings); (ii) a minimal core input language that is smaller and less complex than prior work; and (iii) a shift from global constraints to local, production rule constraints, which enables easier integration with certain fuzzing workflows. To our knowledge, GOBLIN is the first tool to support efficient generation of semantically valid inputs for real-world protocols involving rich constraints over diverse SMT theories in a fully blackbox setting.

**Prior Work.** The most relevant prior work is ISLa [30], another system for context-sensitive input generation. Much like GOBLIN, ISLa takes as input both a context-free grammar and a set of context-sensitive constraints and outputs a string in the language of the grammar that also respects the constraints. However, GOBLIN distinguishes itself from ISLa in several ways. Most notably, GOBLIN supports constraints over a wider array of data types, has a more minimal input language, considers *local* constraints rather than global constraints, and performs chronological backtracking with an incremental SMT backend (compared to ISLa’s nonchronological backtracking with a non-incremental SMT backend).

Fandango [2] also targets semantics-aware input generation, but it uses search-based heuristics in place of constraint solving for higher efficiency. While Fandango is effective for many examples, its effectiveness is highly dependent on the input constraints and built-in set of fitness functions, and it cannot establish formal guarantees.

Our approach is reminiscent of constraint logic programming (CLP), which extends logic programming to also handle constraint satisfaction. In fact, Dewey et al. [14] presented an approach for CLP-based language fuzzing and applied it to JavaScript programs. In principle, GOBLIN can be framed in terms of CLP—in the full version of the paper [3], we define an alternative semantics of GOBLIN in terms of a CLP problem. However, we cannot rely on CLP solvers in practice, as they are currently incapable of handling arbitrary SMT theories (e.g., bitvectors, strings) or their combinations.

**Approach.** We identify three core insights of our approach that distinguish GOBLIN from previous approaches while maintaining formal guarantees of *solution soundness* (every GOBLIN output is a member of the language of the input grammar with constraints), *refutation soundness* (if GOBLIN reports UNSAT, then the input grammar’s language is empty), and *solution completeness* (if the input grammar’s language is nonempty, then GOBLIN will find a member).

First, our approach conceptually **views context-free grammars as generators of algebraic datatype (ADT) terms** rather than

117 **strings**. The ADT view allows each leaf to be assigned a type—  
 118 e.g., `Int`, `BitVec(n)`, `String`—and enables seamless integration with  
 119 SMT solvers capable of reasoning over these theories. This allows  
 120 us to specify and natively perform constraint solving in various  
 121 domains that are inaccessible to ISLa, e.g., bitvector constraints  
 122 over network packets. In addition, this approach brings the benefits  
 123 of static typing to the language, while prior work [2, 30] requires  
 124 *unsafe casting* for non-string constraints.

125 Second, our approach represents a **minimal core language**  
 126 **without sacrificing expressiveness**. Minimizing the set of *core*  
 127 *language features* clarifies the formal foundations, simplifies the  
 128 implementation, and avoids unnecessary special cases. Finally, it  
 129 provides a solid foundation as an intermediate language for a future,  
 130 more usable surface-level language. In particular, we envision a  
 131 future surface-level language which follows in the footsteps of well  
 132 established and appreciated concepts from theoretical computer  
 133 science such as attribute grammars and CLP systems.

134 Third, our approach conceptually views semantic constraints at  
 135 the **production-rule level** rather than globally. This shift does not  
 136 affect expressiveness and allows GOBLIN to integrate naturally with  
 137 fuzzing workflows involving *grammar-level* mutations, in which  
 138 grammar rules themselves are mutated rather than concrete inputs.  
 139 This offers two distinct benefits: (i) mutated grammars function  
 140 as a form of *root cause analysis*; and (ii) grammar-level mutations  
 141 provide an extra mechanism for ensuring diversity of fuzzed inputs.

142 We build confidence in the correctness of GOBLIN’s main al-  
 143 gorithm by framing it as a calculus and proving three theorems:  
 144 solution soundness, refutation soundness, and solution complete-  
 145 ness (for this, see the full version of the paper at [3]).

146 **Evaluation and findings.** We perform an experimental analysis  
 147 with four case studies. Three of them compare directly to prior  
 148 work (XML, CSV, and Scriptsize C input generation), and they  
 149 demonstrate GOBLIN’s ability to handle complex constraints and  
 150 offer competitive performance. The fourth case study (WiFi SAE  
 151 input generation) demonstrates GOBLIN’s utility and ability to  
 152 handle certain constraints and fuzzing workflows that cannot be  
 153 handled by previous approaches.

154 **Contributions.** This paper makes the following contributions: (i)  
 155 we present a general approach for generating inputs that satisfy  
 156 both context-free syntactic constraints and SMT-expressible seman-  
 157 tic constraints; (ii) we implement and introduce GOBLIN, an input  
 158 generator that supports a minimal yet expressive DSL for defining  
 159 CFGs with production rule constraints over arbitrary SMT theo-  
 160 ries; and (iii) we evaluate GOBLIN on four real-world case studies,  
 161 showing that it scales to complex, constraint-rich input formats.

## 163 2 Motivation

164 We explain how GOBLIN integrates well with real-world fuzzers,  
 165 and we discuss the relationship between GOBLIN and prior work  
 166 more concretely.

### 168 2.1 Integration with Real-World Fuzzers

170 Grammar-based fuzzers use the implementation-under-test’s input  
 171 language (represented as a grammar) as a guide for generating  
 172 meaningful test inputs, especially when the test-target has an intri-  
 173 cate input format. Without loss of generality, there are two potential

175 classes of grammar-based fuzzers (*i.e.*, Class I and Class II). Class  
 176 I fuzzers (e.g., [15, 16, 34]) generate a concrete input  $x$  from the  
 177 input language  $\mathcal{L}$  such that  $x \in \mathcal{L}$ . They then apply mutation  
 178 operations on  $x$  to generate new test inputs  $x'$ s. Class II fuzzers  
 179 (e.g., [12, 21, 32]) directly mutate the original language  $\mathcal{L}$  (*i.e.*, the  
 180 grammar production rules) to obtain a new language  $\mathcal{L}'$ . They then  
 181 aim to obtain concrete test inputs  $y$  such that  $y \in \mathcal{L}'$ . The claimed  
 182 advantage of Class II fuzzers is that the mutation operations can  
 183 themselves be input structure aware. In addition, when a concrete  
 184 input triggers a bug in the system under test, the mutated gram-  
 185 mar that was used to generate the input can help with root cause  
 186 analysis.

187 Given a language  $\mathcal{L}$ , generating a concrete input  $x$  such that  
 188  $x \in \mathcal{L}$  is essentially the problem that GOBLIN solves. The language  
 189  $\mathcal{L}$  supported by GOBLIN goes beyond context-free grammars and  
 190 also includes constraints over the grammar elements. GOBLIN’s  
 191 Domain-Specific Language (DSL) for capturing the input language  
 192  $\mathcal{L}$  of the target allows attaching constraints to each production  
 193 rule of  $\mathcal{L}$ ’s grammar, which we refer to as local, production-level  
 194 constraints.

195 Supporting Class II fuzzers in input generators (e.g., GOBLIN,  
 196 ISLa, and Fandango) raises a natural question: *If a fuzzer performs*  
 197 *grammar production-level mutations, then what should happen to*  
 198 *the semantic constraints?* In ISLa [30], the input is a *pair*  $\langle \text{grammar},$   
 199  $\text{constraints} \rangle$ , where the constraint language allows quantification,  
 200 structural predicates over the generated term, and pattern match-  
 201 ing over the generated term. In some cases, it may be possible to  
 202 mutate the constraints in an analogous way to the production rule  
 203 mutations; however, this is not clear in general. On the other hand,  
 204 if we attach constraints locally to production rules rather than the  
 205 entire input grammar, constraint mutation becomes much more  
 206 natural. Example production rules with local constraints are shown  
 207 below in Listing 1—the curly braces after each production rule each  
 208 include a set of constraints that must be satisfied every time the  
 209 production rule is used in a derivation. More concretely, the first  
 210 production rule and constraint denote that in any derivation, for  
 211 every instance of the production rule, the `<GROUP_ID>` nonterminal  
 212 must be equal to a bitvector of width 16 with value 13.

213 **Listing 1: Grammar-level mutations**

```

214 1 <COMMIT> ::= <SEQ> <GROUP_ID> <SCALAR>
215 2   { <GROUP_ID> = int_to_bv(16, 13);
216 3     <SEQ> > int_to_bv(16, 4) };
217 4 <RG_CONT> ::= <RG_LENGTH> <RG_TY> <RG_LIST>
218 5   { <RG_LENGTH> <- length(<RG_LIST>); };
219 6   ~
220 7 <COMMIT> ::= <SEQ> <RG_LIST> <SCALAR>
221 8   { <SEQ> > int_to_bv(16, 4); };
222 9 <RG_CONT> ::= <RG_LENGTH> <GROUP_ID>
223 10  { <GROUP_ID> = int_to_bv(16, 13); };
224
225
```

226 Listing 1 also illustrates a *crossover mutation*, where the rules be-  
 227 fore the arrow denote a fragment of the grammar before mutation,  
 228 and the rules after the arrow denote the same grammar fragment  
 229 after mutation. In this mutation, two production rules are selected,  
 230 and one nonterminal from each production rule is chosen to swap  
 231 places (here, `<GROUP_ID>` and `<RG_LIST>`). It is easy to maintain

233 the semantic constraints over `<GROUP_ID>` and `<SEQ>` (they follow  
 234 the corresponding nonterminals); the lack of clarity is localized to  
 235 the constraint over `<RG_LIST>`. In our setting, grammar-level mutations  
 236 bring more benefits: grammar-level mutations can help boost  
 237 the diversity of fuzzed inputs, and further, due to the underlying  
 238 constraint solving, GOBLIN performs better when performing many  
 239 shorter generation rounds compared to one long generation round.  
 240

## 2.2 Working Example

We now introduce a working example which we will reference throughout the paper, aiming to give informal intuition for how GOBLIN works. In Listing 2, we see a grammar and constraints describing the format of a simple network packet.

247 **Listing 2: Working example in Goblin DSL**

```
248 1 <Packet> ::= <Type> <Len> <Payload>
249 2 { <Len> <- int_to_bv(8, 16 + length(<Payload>));
250 3   <Type> = int_to_bv(8, 1) =>
251 4     <Payload>.<Byte> < 32; };
252 5   <Payload> :: <Byte> <Payload>
253 6   | <Byte> { (<Byte> bvand 0b11110000)
254 7     = 0b10100000; };
255 8   <Type> :: BitVec(8)
256 9   { <Type> = int_to_bv(8, 0) lor
257 10    <Type> = int_to_bv(8, 1); };
258 11 <Len> :: BitVec(8);
259 12 <Byte> :: BitVec(8);
```

261 Line 1 describes a production rule for `<Packet>`, a single option producing three nonterminals `<Type>`, `<Len>`, and `<Payload>`. Lines 2, 262 3, and 4 (surrounded by curly braces) denote semantic constraints on 263 the production rule—that is, a **length constraint** which expresses 264 that `<Len>` should have a value equal to sixteen plus the length of 265 the `<Payload>`, and another constraint that restricts the length of 266 the `<Payload>` when `<Type>` takes value 1. The semicolon separating 267 the constraints should be interpreted conjunctively. Lines 8, 268 11, and 12 denote **type annotations**, which ascribe types to leaf 269 symbols in the grammar. As in lines 9 and 10, type annotations can 270 also carry semantic constraints (which we call **refinement types**).  
 271

## 2.3 Comparison with Prior Work

**Goblin and ISLa.** As mentioned in Section 1, the most relevant prior work is ISLa [30], another system for context-sensitive input generation. GOBLIN distinguishes itself from ISLa in four main ways.

272 First, the semantics of ISLa restrict it to only natively supporting  
 273 string and integer constraints. This means that the example  
 274 presented in Listing 2 could not be directly expressed in ISLa. Tech-  
 275 nically, one could encode the bitwise and constraint (`bvand`) in  
 276 terms of string constraints at the SMT-LIB level, but this would  
 277 be incredibly tedious and less performant. Further, the support for  
 278 integer constraints is limited in the sense that it relies on unsafe  
 279 string to integer casts. In fact, in ISLa’s formal semantics [30], the  
 280 semantics of quantification over integers is defined as quantifica-  
 281 tion over strings that can be casted to integers. ISLa’s reliance on  
 282 strings comes from the fact that ISLa conceptually views grammars  
 283 as generating *strings* rather than *ADT terms*, and so each nonter-  
 284 minal symbol has a string type. In addition to reliance on unsafe  
 285

291 casting, this introduces an obstacle for SMT constraints over string  
 292 variables representing string nonterminal symbols—the SMT solver  
 293 may return a model where the string variable does not respect the  
 294 input CFG. To deal with this, ISLa approximates the input CFG as  
 295 a regular expression, a process they call “grammar to regex” [30].  
 296 With the ADT view, GOBLIN sidesteps this issue entirely. While  
 297 ISLa’s string and integer constraints are sufficient for some do-  
 298 mains (e.g., compiler fuzzing), it is inadequate for applications with  
 299 non-string, non-integer constraints (e.g., network protocol stacks).

300 Second, the GOBLIN language is minimal compared to ISLa’s.  
 301 Many features that are part of ISLa’s core language are absent from  
 302 GOBLIN, but can still be encoded. For example, ISLa considers **struc-**  
**tural constraints** which constrain the structure of the generated  
 303 term. More concretely, ISLa can encode **definition before use** con-  
 304 straints for compiler input generation through a predicate before  
 305 and existential quantification, both native to the input language.  
 306 However, GOBLIN can express these constraints, despite not having  
 307 built-in structural predicates or existential quantification. Because  
 308 GOBLIN is an *ADT term generator* with *local constraints*, we can  
 309 encode such constraints akin to how one would do so in a CLP  
 310 system or attribute grammar. Minimality is not necessarily a ben-  
 311 efit or end goal in itself—however, we argue that a well-designed  
 312 *intermediate language* serves to clarify the formal foundations,  
 313 simplify the implementation, and avoid unnecessary special cases.  
 314 In the future, we envision designing and supporting a surface-level  
 315 input language with more usability features and syntactic sugar.  
 316

317 Third, GOBLIN supports local production rule constraints rather  
 318 than global constraints, as discussed in Section 2.1.

319 Fourth, GOBLIN’s style of backtracking is chronological, while  
 320 ISLa’s is nonchronological. With chronological backtracking, GOB-  
 321 LIN incrementally expands and backtracks a candidate solution,  
 322 which enables an incremental SMT backend. This backend enables  
 323 GOBLIN to both detect unsatisfiable states quickly and update partial  
 324 solutions as new constraints are encountered.

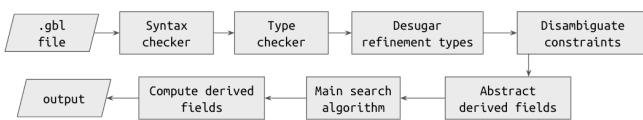
**GOBLIN and Fandango.** Fandango [2] also targets semantics-  
 325 aware input generation, but it uses search-based heuristics in place  
 326 of constraint solving for higher efficiency. Fandango is incredibly  
 327 efficient for many examples, and its constraint language (general  
 328 Python code) has unparalleled expressivity. However, Fandango’s ef-  
 329 fectiveness is highly dependent on the input constraints and cannot  
 330 establish formal guarantees. Conceptually, the approach produces  
 331 instances using a genetic algorithm which attempts to steer towards  
 332 semantically correct instances through fitness functions that favor  
 333 those instances that are the “closest” to satisfying the semantic  
 334 constraints. However, synthesizing suitable fitness functions from  
 335 scratch based on the input grammar and constraints is currently  
 336 out of reach, so Fandango [2] chooses from a set of built-in fitness  
 337 functions. If none of the built-in fitness functions works for an  
 338 input, Fandango struggles to produce even a single valid instance.  
 339 Several simple examples demonstrate this phenomenon—for exam-  
 340 ple, (i) an input grammar with two nonterminals `<nt1>` and `<nt2>`  
 341 with a single semantic constraint `<nt1> == <nt1>` (one equality  
 342 constraint), (ii) an input grammar with two nonterminals with a  
 343 single semantic constraint `str(<nt1>) == str(<nt1>) + "foo"`  
 344 (one string constraint), and (iii) an input grammar with an integer  
 345 nonterminal and a single semantic constraint `int(<nt>) in {1,`  
 346 `2, 3}` (one set membership constraint). In addition, when we tried  
 347

349 to use Fandango to synthesize a valid input for the WiFi SAE packet  
 350 grammar and constraints that we explore as part of our experimen-  
 351 tal evaluation of GOBLIN (see Section 8), Fandango was not able to  
 352 produce a single instance. The Fandango examples demonstrating  
 353 our encodings for these problems is available at [3].

354 In short, choosing or synthesizing a suitable fitness function for  
 355 any arbitrary set of input constraints is an extremely difficult prob-  
 356 lem. It could potentially be remedied through user-supplied fitness  
 357 functions, but this detracts from the generality of the approach. If  
 358 the constraints are non-monotonic, a suitable fitness function may  
 359 not exist in the first place.

### 3 Design Overview

361 Next, we give an informal overview of GOBLIN’s input generation  
 362 process. The main pipeline is depicted in Figure 1, and formal  
 363 description follows in the full version of the paper [3].



372 **Figure 1: Main pipeline.**

373 **Context-free language emptiness.** As a preprocessing step, with  
 374 the Syntax checker module, we implement a standard algorithm  
 375 to determine whether or not the *context-free* version of the in-  
 376 put grammar (i.e., without any extra semantic constraints) has an  
 377 empty language [19]. At the context-free level, language emptiness  
 378 stems from non-well-founded recursion (i.e., recursion without a  
 379 base case), leading to infinite derivations. Intuitively, the algorithm  
 380 proceeds by iteratively expanding a set of nonterminals that are  
 381 known to be able to produce terminal strings until reaching a fix-  
 382 point, and then checking if the set contains the start symbol. In  
 383 fact, we perform a more conservative check by mandating that *all*  
 384 reachable nonterminals must be a member of this set, as violation  
 385 likely signals a modeling error.

### 386 Listing 3: Syntax checks

387 1 <SAE\_PACKET> ::= <COMMIT> | <CONFIRM>;  
 388 2 <COMMIT> ::= <FIELD1> <RG\_ID\_LIST>;  
 389 3 <RG\_ID\_LIST> ::= <RG\_ID> <RG\_ID\_LIST>;  
 390 4 <CONFIRM> ::= <FIELD2> <FIELD3> <FIELD4>;  
 391 5 ...

392 An example input rejected by our syntax checker (inspired by real  
 393 experience with our WiFi case study) is in Listing 11. While the  
 394 language of the grammar is non-empty through a derivation involv-  
 395 ing <CONFIRM>, the <COMMIT> nonterminal cannot derive a finite  
 396 string due to a missing base case in the definition of <RG\_ID\_LIST>. Hence,  
 397 the input is conservatively rejected, as no finite derivation  
 398 through <COMMIT> is possible.

399 **Other syntax checks.** Other syntax checks include the detection  
 400 of dangling identifiers and invalid dot notation references.

401 **Type checking.** Because the language requires type annotations  
 402 for leaf-level nonterminals, we can perform standard bidirectional

403 type checking techniques with two mutually recursive functions  
 404 `check_type_expr` and `infer_type_expr`.

### 405 Listing 4: Type checking

406 1 <PACKET> ::= <F1> <F2> <ALGO>  
 407 2 { <F1> = <F2> bvplus <ALGO>. <ALGO\_ID>; };  
 408 3 <ALGO> ::= <ALGO\_ID> <ALGO\_SETTINGS> ;  
 409 4 <F1> :: BitVec(16); <F2> :: BitVec(16);  
 410 5 <FIELD3> :: BitVec(16); <ALGO\_ID> :: BitVec(16);  
 411 6 <ALGO\_SETTINGS> :: BitVec(8);

412 For example, in Listing 4, to perform type checking of the constraint  
 413 on line 2, we call `infer_type_expr` to infer the type of the whole  
 414 expression, and we call `check_type_expr` to see if it matches the  
 415 expected type, which is always `Bool`. Type inference is necessary  
 416 in our setting since the bitvector types are *dependent types* which  
 417 depend on the length of the bitvector. Notice that we cannot tell if  
 418 the subexpression with the `bvplus` operation is well-typed without  
 419 inferring the types of both arguments and checking if their lengths  
 420 are equal. With this type checking, we can specify constraints  
 421 over network packets without having to perform unsafe casts from  
 422 strings to bitvector types, where the input string may or may not  
 423 be convertible to a bitvector, and bitvector operations may not be  
 424 well-defined for the given operands depending on their lengths.

425 **Refinement type constraints.** GOBLIN’s input language supports  
 426 semantic constraints on type annotations, which we call *refinement*  
 427 types. We support refinement types by desugaring type annotation  
 428 constraints to production rule constraints by “Inlining” the type an-  
 429 notation constraint for some nonterminal `<nt>` at every production  
 430 rule option containing `<nt>` (on the right-hand side).

### 431 Listing 5: Refinement types

432 1 <S> ::= <NAT> <NAT>;  
 433 2 <NAT> :: Int { <NAT> >= 0; };  
 434 3 ~  
 435 4 <S> ::= <NAT> <NAT> { <NAT> >= 0; };  
 436 5 <NAT> :: Int;

437 We demonstrate the inlining with a simple example in Listing 5.

438 **Disambiguating dot notation.** Notice that in Listing 5, after in-  
 439 lining, the constraint `<NAT> >= 0` applies to a production rule  
 440 with two instances of the `<NAT>` nonterminal. Under the hood, the  
 441 constraint is disambiguated to `<NAT>[0] >= 0` and `<NAT>[1] >= 0`,  
 442 where we take the semantics of *implicit universal quantification*.  
 443 This is discussed in more detail in Section 4.

444 **Derived fields.** Some constraints are not amenable to analysis  
 445 by SMT solvers. For example, checksums, constraints over cryp-  
 446 tographic fields, and constraints involving non-linear arithmetic  
 447 are highly expensive and/or difficult to translate to SMT-LIB [6].  
 448 We handle these constraints separately from the rest of the search  
 449 problem through a language feature called *derived fields*, in which  
 450 syntax `<field> <- <expression>` denotes that the value for non-  
 451 terminal `<field>` can be computed with expression `<expression>`.  
 452 For example, in Listing 2, the `<Len>` nonterminal is a derived field  
 453 since its value can be calculated as defined on line 2. To make de-  
 454 rived fields computable outside the solver, we need to detect and  
 455 reject two scenarios: (i) we disallow mutual (cyclic) dependencies  
 456 between derived fields, and (ii) we disallow including derived fields

in (non-derived) semantic constraints. To detect cyclic dependencies, we borrow an efficient technique of circularity detection in attribute grammars [10, 35]. We build a directed graph for each production rule. There is a node for each nonterminal on the right-hand side of the production rule and an edge from  $\langle nt1 \rangle$  to  $\langle nt2 \rangle$  when  $\langle nt1 \rangle$  is a derived field defined in terms of  $\langle nt2 \rangle$ . If this graph contains a cycle, then the input is rejected for a cyclic dependency. This algorithm is a fast over-approximation that is computable in polynomial time, but works for “every practical example” [10]. To detect when derived fields are mentioned in (non-derived) semantic constraints, we simply scan every semantic constraint for dot notation expressions mentioning nonterminals that are associated with derived fields.

#### Listing 6: Derived fields

```
480 1 <S> ::= <LEN> <TYPE> <PAYLOAD>
481 2 { <LEN> <- length(<PAYLOAD>);
482 3   <LEN> < 64; };
483 4 <LEN> :: BitVec(8); <TYPE> :: BitVec(8);
484 5 <PAYLOAD> :: List(Bool);
```

The example in Listing 6 is rejected since derived field  $\langle LEN \rangle$  is mentioned in the semantic constraint  $\langle LEN \rangle < 64$ . In this case, the user can remedy the situation by replacing the derived field operator  $\langle - \rangle$  with an equality constraint. (Also, in Listing 6, if  $\langle TYPE \rangle$  were a derived field defined by an expression containing  $\langle LEN \rangle$ , then there would be a cyclic dependency between  $\langle TYPE \rangle$  and  $\langle LEN \rangle$ .) After these checks, these derived fields are removed from the grammar and replaced with opaque, “stub” nonterminals that record the position of the derived field with symbolic leaf symbols rather than the input production rules.

#### Listing 7: Derived fields

```
496 497 1 <S> ::= sym_leaf <TYPE> <PAYLOAD>;
498 2 <TYPE> :: BitVec(8); <PAYLOAD> :: List(Bool);
```

Assuming  $\langle LEN \rangle$  remains a derived field, Listing 6 gets translated to Listing 7, where the  $\langle LEN \rangle$  nonterminal is replaced by a special symbolic leaf value. The `sym_leaf` value will be present in the term generated by the main search algorithm, say,  $(S \ (\text{LEN } \text{sym\_leaf}) \ (\text{TYPE } 0b00000000) \ (\text{PAYLOAD } []))$ . Here, we can compute the value of `sym_leaf` to give output term  $(S \ (\text{LEN } 0b00000000) \ (\text{TYPE } 0b00000000) \ (\text{PAYLOAD } []))$ .

**Polymorphic length function.** Derived fields enable language features that don’t rely on SMT support. An example is GOBLIN’s *polymorphic length function*, which computes the length of an input nonterminal of any type. Supporting such an operator at the SMT-LIB level is difficult—one strategy is to synthesize an ADT for the input nonterminal and then synthesize a length function that takes a term of that ADT as input. However, we can support this easily in GOBLIN by simply including evaluation rules in GOBLIN’s internal expression evaluator (used for computing derived fields).

#### Listing 8: Polymorphic length

```
516 517 1 <S> ::= <LEN> <PAYLOAD>
518 2 { <LEN> <- length(<PAYLOAD>); };
519 3 <PAYLOAD> :: <F1> <F2> <L>;
520 4 <F1> :: BitVec(16); <F2> :: BitVec(16);
521 5 <L> :: List(Bool);
```

Listing 8 demonstrates the application of the polymorphic length function to a nonterminal containing two bitvectors and a list of Booleans. The length function works by adding all the lengths of the concrete leaf values of the input term (string length for strings, bit width for bitvectors, and so on). Here, it would compute a value equal to the length of  $\langle L \rangle$  plus 32.

**Main solving algorithm.** The main search algorithm will be described in detail in Section 5. At a high level, the solving algorithm iteratively builds a *candidate solution* by walking through the grammar and choosing production rules to expand. Whenever applicable semantic constraints are encountered, they are eagerly asserted to an SMT solver to detect unsatisfiable states as quickly as possible. When unsatisfiable states are detected, the search backtracks to the most recent decision (i.e., most recent choice of production rule to expand) if possible. Otherwise, the search will either report that the input problem is unsatisfiable, or continue the search, depending on greater context.

**Serialization.** GOBLIN produces ADT terms rather than strings. More specifically, GOBLIN produces Lisp-style terms, where each term is either (i) a concrete leaf-level value (say, `1000` or `"foo"`), (ii) a symbolic leaf-level value for derived fields (say, `sym_leaf_27`), or (iii) a constructor string paired with a list of subterms (say,  $(S \ (\text{LEN } 0b00000000) \ (\text{TYPE } 0b00000000) \ (\text{PAYLOAD } []))$ ) from the **Derived fields** paragraph. To produce strings, the user defines a serialization function mapping these Lisp-style terms to strings outside of GOBLIN.

## 4 GOBLIN’s Language and Semantics

In this section, we give a more precise characterization of GOBLIN’s syntax and semantics.

### 4.1 Syntax

We define the language syntax using a standard extension of Backus-Naur form [4] where the `*` operator denotes zero or more instances, the `+` operator denotes one or more instances, and the square brackets denote optional elements.

```

581
582     <S> ::= <element>+
583
584     <element> ::= <ty_annot> | <prod_rule>
585
586     <ty_annot> ::= <nt> :: <type> [ { <constraint>+ } ]
587
588     <prod_rule> ::= <nt> ::= <nt>+ [ { <constraint>+ } ]
589
590         [ | <nt>+ [ { <constraint>+ } ] ]+;
591
592     <constraint> ::= <derived_field>; | <expr>;
593
594     <derived_field> ::= <nt> <-> <expr>
595
596         <expr> ::= <expr> <binop> <expr> | <unop><expr> |
597             <f>(<expr>, ..., <expr>) | <p>(<expr>, ..., <expr>)
598
599             <nt_expr> | constant
600
601             <nt> ::= <identifier>
602
603             <type> ::= Bool | Int | String | Set(<type>) ...
604
605                 <f> ::= length | bv_cast | ...
606
607                 <p> ::= leq | geq | ...
608
609             <binop> ::= land | lor | ...
610
611             <unop> ::= lneg | minus | ...
612
613
614
615 In the BNF, we did not enumerate all the types, predicate symbols, function symbols, and constants. In principle, we support every type, predicate symbol, and function symbol that has a direct translation to SMT-LIB [6] or cvc5's [5] support for non-standard theories [11]. (In practice, we have not yet implemented support for all of the types, function symbols, and predicate symbols, but will gladly extend the input language where demand arises.)


Dot notation. Before formally defining the language semantics, we first informally discuss a tricky aspect of the language: ambiguous dot notation references. Consider Listing 9.


```

#### Listing 9: Ambiguous dot notation reference

```

616
617 1 <A> -> <B> <B> <C> { <B>. <D> > <C>; };
618 2 <B> -> <D> <D>;
619 3 <D> :: Int;
```

Above, the **nonterminal expression**  $\langle B \rangle . \langle D \rangle$  intuitively could refer to either the first or second child  $\langle D \rangle$  of either the first or second occurrence of  $\langle B \rangle$ . GOBLIN treats all ambiguous references of this form as **implicitly universally quantified** over the structure of generated terms—that is, one can view the above constraint as internally desugaring to  $\langle B \rangle[0]. \langle D \rangle[0] > \langle C \rangle[0]; \langle B \rangle[0]. \langle D \rangle[1] > \langle C \rangle[0]; \langle B \rangle[1]. \langle D \rangle[0] > \langle C \rangle[0]; \langle B \rangle[1]. \langle D \rangle[1] > \langle C \rangle[0]$ , where the bracket notation  $[i]$  of a nonterminal symbol uniquely indicates which occurrence of the nonterminal symbol is being referenced. Furthermore, also consider Listing 10, where  $\langle B \rangle$  gets a separate production rule also referencing  $\langle D \rangle$ .

#### Listing 10: Ambiguous dot notation reference II

```

634 1 <A> -> <B> <B> <C> { <B>. <D> > <C>; };
635 2 <B> -> <D> <D> | <D>;
636 3 <D> :: Int;
```

At term generation time, if  $\langle B \rangle$ 's second production rule is chosen, then (e.g.) constraint  $\langle B \rangle[0]. \langle D \rangle[1] > \langle C \rangle[0]$  is considered trivially satisfied, since  $\langle B \rangle[0]$  does not have a child  $\langle D \rangle[1]$  (instead, it has a single child,  $\langle D \rangle[2]$ ).

## 4.2 Semantics

To capture the language semantics, we first formally define an **augmented context-free grammar** (ACFG)  $G$ . Intuitively,  $G$  is a context-free grammar (CFG) where each production rule optionally carries additional (context-sensitive) constraints, and where we replace nonterminal symbols with type annotations, which specify that the given nonterminal symbol ranges over the given type.

More concretely,  $G$  is a 4-tuple  $(N, R, \Gamma, S)$ , where  $N$  is a finite set of nonterminal symbols,  $R : N \times N^* \times C^*$  is a relation capturing the production rules where each production rule additionally carries zero or more semantic constraints,  $\Gamma$  is a set of  $N \times \Lambda$  pairs denoting type annotations, and  $S \in N$  is the distinguished start symbol. Additionally, we require that each nonterminal symbol either (mutually exclusively) is mentioned on the left-hand side of one or more production rules, or has exactly one type annotation.

Rather than working with raw strings, we define the semantics of an ACFG  $G$  as the set of valid abstract syntax trees producible by the grammar (denoted  $\mathcal{L}_{\text{AST}}(G)$ ). We define an **abstract syntax tree** (AST)  $tr$  for a given ACFG  $G$  as a 3-tuple  $(V, \vec{E}, \ell)$ , where  $(V, \vec{E})$  denotes a directed graph without (even undirected) cycles, and  $\ell : V \rightarrow N \cup (\bigcup \Lambda)$  denotes the label of a given vertex with its associated nonterminal symbol, or value ranging over its given type in  $G$ . We abuse notation to allow  $\ell$  to also return a nonterminal symbol associated integer index for disambiguation (as discussed in Section 4.1) where convenient. Also,  $\text{get\_children} : \text{AST} \times V \rightarrow V^*$  takes the expected semantics, and we abuse notation by using  $\in$  to mean both set and list membership. Additionally,  $tr$  has two syntactic well-formedness requirements:

- (i)  $\ell(\text{root}(G)) = S$  (the AST is rooted at the start symbol);
- (ii) for every  $v \in V$ , either
  - (a) for every  $v_1, \dots, v_n \in \text{get\_children}(tr, v)$ ,  $(\ell(v), (\ell(v_1), \dots, \ell(v_n)), \_) \in R$ , or
  - (b) there is only a single  $v_1 \in \text{get\_children}(tr, v)$  and  $(\ell(v), \tau) \in \Gamma$ , or
  - (c)  $v$  is a leaf vertex with  $\ell(v) \in \tau$ , where  $\tau$  is the type of  $\ell(\text{parent}(v))$  ( $\ell(\text{parent}(v))$  must be associated with a type annotation).

Informally, each node in the tree is either a non-leaf node with children representing an application of some production rule in  $G$ , a non-leaf node with a single child representing some type annotation in  $G$ , or a well-typed leaf.

Next, we define the meanings of terms in a production rule constraint for a given AST  $tr$  by defining interpretation function  $\mathcal{I}_{tr}(t)$ , which outputs the denotation of term  $t$  in AST  $tr$ . It is analogous to first-order logic, except (i) the variable assignment is determined by the structure and labels of  $tr$  rather than by a mapping of variable names to values, (ii) terms evaluate to a distinguished symbol  $T$  if they reference nonterminal expressions that are not present in the given tree, and (iii) we assume all terms are well-typed (e.g., primitive values take the expected types, each function symbol has a given set of input types and an output type, and each function

symbol is passed the correct number of arguments with the correct types). For primitive values (e.g., integers), the denotation is defined by the identity function. Otherwise,

$$\begin{aligned}
 I_{tr}(\langle nt \rangle[i]) &= \top \text{ if } \\
 &\quad nt, \_ \in R \text{ and } \\
 &\quad \{v \in \text{get\_children}(tr, \text{get\_root}(tr)) \mid \ell(v) = \langle nt \rangle[i]\} = \emptyset \\
 I_{tr}(\langle nt \rangle[i]) &= \ell(v) \text{ for the unique } \\
 &\quad v \in \text{get\_children}(tr, \text{get\_root}(tr)) \text{ if } nt, \_ \in \Gamma \\
 I_{tr}(\langle nt \rangle[i]) &= I_{tr'}(\langle nt \rangle[i]) \text{ for } tr' \text{ rooted at the unique } \\
 &\quad v \in \text{get\_children}(tr, \text{get\_root}(tr)) \text{ such that } \ell(v) = \langle nt \rangle[i] \\
 &\quad \text{if } nt, \_ \in R \\
 I_{tr}(\langle nt \rangle[i].\langle nt\_expr \rangle) &= \top \text{ if } \\
 &\quad \{v \in \text{get\_children}(tr, \text{get\_root}(tr)) \mid \ell(v) = \langle nt \rangle[i]\} = \emptyset \\
 I_{tr}(\langle nt \rangle[i].\langle nt\_expr \rangle) &= I_{tr'}(\langle nt\_expr \rangle), \text{ for } tr' \text{ rooted at the unique } \\
 &\quad v \in \text{get\_children}(tr, \text{get\_root}(tr)) \text{ such that } \ell(v) = \langle nt \rangle[i] \\
 I_{tr}(f(t_1, \dots, t_n)) &= \top \text{ if some } I_{tr}(t_i) = \top; \text{ otherwise,} \\
 I_{tr}(f(t_1, \dots, t_n)) &= I_{tr}(f)(I_{tr}(t_1), \dots, I_{tr}(t_n))
 \end{aligned}$$

We define a satisfaction relation  $\models_G$  that captures whether or not a given constraint in  $G$  is satisfied by a given abstract syntax tree. It proceeds as in quantifier-free first-order logic, except it also uses  $\top$  analogously to the term semantics.

$$\begin{aligned}
 I_{tr} \models_G \varphi &\text{ if some subterm } t_i = \top; \text{ otherwise,} \\
 I_{tr} \models_G p(t_1, \dots, t_n) &\text{ if } (I_{tr}(t_1), \dots, I_{tr}(t_n)) \in I_{tr}(p) \\
 I_{tr} \models_G \neg\varphi &\text{ if } I_{tr} \not\models_G \varphi \\
 I_{tr} \models_G \varphi_1 \wedge \varphi_2 &\text{ if } I_{tr} \models_G \varphi_1 \text{ and } I_{tr} \models_G \varphi_2
 \end{aligned}$$

While we presented the semantics for predicate symbols and function symbols generically, the GOBLIN’s actual predicate and function symbols have fixed interpretations and take the expected semantics.

Below,  $\text{CFG}(\cdot)$  is a function that takes an ACFG as input and removes its semantic constraints, returning a standard CFG. Also,  $\text{get\_subtrees}(\cdot, \cdot)$  is a function that takes as input an AST  $t$  and a nonterminal symbol  $\langle nt \rangle$ , and returns all subtrees in  $t$  rooted at vertices  $v$  with  $\ell(v) = \langle nt \rangle$ . Finally,  $\text{resolve} : C \rightarrow \mathcal{P}(C)$  performs the desugaring of ambiguous dot notation references discussed previously, and we lift  $\models_G$  to sets of constraints as expected. Put together, the semantics of an input,  $\llbracket G \rrbracket$ , is defined as the set of ASTs in the language of  $G$  that also satisfy all the semantic constraints.

$$\begin{aligned}
 \llbracket G \rrbracket &= \{t \mid t \in \mathcal{L}_{\text{AST}}(\text{CFG}(G)) \wedge \forall (\text{lhs}, \_, \text{constraints}) \in R. \\
 &\quad \forall s \in \text{get\_subtrees}(t, \text{lhs}). \forall \varphi \in \text{constraints}. \\
 &\quad s \models_G \text{resolve}(\varphi)\}
 \end{aligned}$$

## 5 GOBLIN’s Constraint Solving Approach

Here, we describe GOBLIN’s core algorithm for constraint solving, continuing to reference our working example from Listing 2.

**Basic search strategy.** Intuitively, our search strategy mimics a DPLL-style [13] search in which a candidate solution, which we call a **derivation tree** (formally defined in the full version of the paper [3]), is incrementally constructed through node expansions

and backtracking until either a solution is found or the input grammar  $G$  is deemed unsatisfiable (i.e.,  $\llbracket G \rrbracket = \emptyset$ ). For conciseness, we represent derivation trees as strings denoting the expansion so far, where open leaves in the derivation tree are represented by nonterminal symbols. We can consider an open derivation of the working example in Listing 2 as  $\langle \text{Type} \rangle \langle \text{Len} \rangle \text{ sym\_leaf } \langle \text{Payload} \rangle$  to denote a situation where  $\langle \text{Type} \rangle$  and  $\langle \text{Len} \rangle$  are still unexpanded, and  $\langle \text{Payload} \rangle$  has been expanded a single time with the first option, producing byte  $\text{sym\_leaf}$  whose value has not yet been determined. If the constraint set associated with the derivation tree is ever deemed unsatisfiable, we backtrack the last decision (in this case, to  $\langle \text{Type} \rangle \langle \text{Len} \rangle \langle \text{Payload} \rangle$ ). On the other hand, if the derivation tree has no more open nonterminals and the constraint set is satisfiable, we instantiate all the instances of  $\text{sym\_leaf}$  with concrete values and output the solution. Intuitively, the problem is difficult because (i) the search space is over derivation trees of potentially unbounded size, and (ii) the search space is inherently both *syntactic* (over various derivations of the context-free aspect of the input grammar) and *semantic* (handling the annotated production rule constraints). We must craft the search algorithm to overcome a series of obstacles related to these key differences.

**Derivation tree normalization.** When constructing a candidate solution, some “decisions” are forced—namely, production rule expansions for nonterminals with only one production rule option, and type annotation expansions. To trim down the search space, we always normalize the candidate derivation tree by performing these forced expansions before making any decisions. Then, when backtracking, we always backtrack to a normalized derivation tree. In the working example, the search would start with derivation tree  $\langle \text{Packet} \rangle$ . However, since  $\langle \text{Packet} \rangle$  only has one expansion option, it takes a normalization step to  $\langle \text{Type} \rangle \langle \text{Len} \rangle \langle \text{Payload} \rangle$ . But here,  $\langle \text{Type} \rangle$  and  $\langle \text{Len} \rangle$  also only have one expansion option, so we fully normalize to  $\text{sym\_leaf1 } \text{sym\_leaf2 } \langle \text{Payload} \rangle$ . We cannot normalize any further, since  $\langle \text{Payload} \rangle$  has more than one expansion option. Then, upon backtracking, we will never backtrack to  $\langle \text{Packet} \rangle$ , since it is not in normal form.

**Incremental solving.** Two universally desirable qualities of search algorithms are (i) the ability to quickly identify when a candidate solution is infeasible (“fail fast”), and (ii) the ability to efficiently mend candidate solutions which are promising but imperfect. We achieve both of these qualities through an application of incremental SMT solving techniques. Intuitively, relevant semantic constraints are asserted as eagerly as possible to the solver instance. Whenever decisions are made (in this case, the choice of a production rule option to explore), an assertion level is pushed, and whenever a decision is backtracked, the assertion stack is popped, undoing the assertions associated with the corresponding decision. We achieve quality (i) because we can detect and backtrack as soon as the set of relevant constraints becomes unsatisfiable, rather than fully expanding a candidate derivation before realizing that it is an infeasible candidate. Similarly, we achieve quality (ii). Consider a situation where two decisions are made. Suppose each decision has an associated set of constraints, and the conjunction of all the constraints is satisfiable. After pushing the first set of constraints and querying for satisfiability, we do not have to immediately instantiate a model into the candidate term. Instead, we allow the partial solution to remain stored into the solver state, and we only retrieve a

813 solution when all constraints are pushed. If we retrieve a model and  
 814 instantiate it into the candidate term immediately, we risk unnecessarily  
 815 stumbling into an unsatisfiable state. In comparison, prior  
 816 work [30] instantiates models from SMT queries immediately into  
 817 the candidate term. Incremental solving is possible in our approach  
 818 because we iteratively build and backtrack a candidate solution,  
 819 rather than performing non-chronological backtracking where the  
 820 algorithm jumps between completely unrelated derivation trees  
 821 [30]. For the working example, once we normalize the derivation  
 822 tree to `sym_leaf1 sym_leaf2 <Payload>`, we eagerly assert all  
 823 *applicable* semantic constraints we encountered during the expan-  
 824 sion step. In this case, we assert `<Type> = int_to_bv(8, 1) =>`  
 825 `<Payload>. <Byte> < 32` and `<Type> = int_to_bv(8, 0) lor`  
 826 `<Type> = int_to_bv(8, 1)`. But, we do not assert the constraint  
 827 associated with `<Len>`, since it is a derived field. Also, we do not  
 828 assert the constraint on line 6, since it has not been encountered  
 829 yet. We check that the constraint set is satisfiable, but we refrain  
 830 from instantiating specific values into the derivation tree.  
 831 **IDS.** For highly recursive grammars, even without context sen-  
 832 sitivity, complete and terminating algorithms for the generation  
 833 of members of the language is nontrivial. Unstrategic approaches  
 834 may diverge or generate terms that are too large to handle once  
 835 context-sensitive constraints are introduced. We employ a clas-  
 836 sic search strategy, iterative deepening search (IDS), which ex-  
 837 haustively explores the search space to a given depth limit before  
 838 restarting and increasing the depth limit. Our formal approach  
 839 is parametric with respect to the specific choice of production  
 840 rule expansion to explore next. As much as possible, we separate  
 841 the *core theory* of the approach from *specific strategies and im-*  
 842 *plementation decisions*. In the working example, consider a depth  
 843 limit of 1 with current derivation tree `sym_leaf_1 sym_leaf_2`  
 844 `<Payload>` (with depth 0). Say we expand `<Payload>` with the first  
 845 option to `sym_leaf_1 sym_leaf_2 sym_leaf_3 <Payload>`, and  
 846 then again to `sym_leaf_1 sym_leaf_2 sym_leaf_3 sym_leaf_4`  
 847 `<Payload>`. However, this pushes the current depth of 2 beyond the  
 848 depth limit of 1. So, we backtrack the last decision to `sym_leaf_1`  
 849 `sym_leaf_2 sym_leaf_3 <Payload>`. But here, expanding to the  
 850 second option (and normalizing) to `sym_leaf_1 sym_leaf_2`  
 851 `sym_leaf_3 sym_leaf_4` still pushes beyond the depth limit, so  
 852 we backtrack to `sym_leaf_1 sym_leaf_2 <Payload>` and instead  
 853 expand (and normalize) to `sym_leaf_1 sym_leaf_2 sym_leaf_3`.  
 854 Since we took the second expansion option at `<Payload>`, we assert  
 855 `(<Byte> bband 0b11110000) = 0b10100000`. Since the constraint  
 856 set is satisfiable and there is nothing left to expand, we instantiate  
 857 the model from the SMT solver into the derivation tree to give  
 858 `0b00000000 sym_leaf2 0b00000000`. Finally, computing the de-  
 859 rived (length) field yields `0b00000000 0b00011000 0b00000000`.  
 860 If we had not found any solution at depth limit 1, we would have  
 861 restarted and increased the depth limit to 2.  
 862 **Multiple solutions.** While we present our approach in terms of  
 863 a search problem that seeks a single solution, it can be extended  
 864 to produce multiple solutions. After finding a solution, we (i) pop  
 865 to the zeroth assertion level, (ii) assert a blocking clause which  
 866 prevents the same solution from being found multiple times, and  
 867 (iii) push an assertion frame to start our next search from the first  
 868 assertion level. With this strategy, we ensure that blocking clauses  
 869 are never popped from the assertion stack.

**Unsatisfiable inputs.** GOBLIN will detect and report if it concludes  
 871 that an input grammar is unsatisfiable (i.e., its language is empty).  
 872 GOBLIN derives UNSAT when the current constraint set is unsatisfi-  
 873 able, there is no decision to backtrack, and no candidate solutions  
 874 have been discarded due to hitting the depth limit.

**Goblin and SMT semantics.** We use an incremental SMT solver  
 876 as a backend reasoning engine. However, Goblin’s semantics (de-  
 877 scribed earlier in Section 3) are defined in terms of a satisfaction  
 878 relation  $\models_G$  between abstract syntax trees and Goblin formulas,  
 879 while SMT semantics are defined in terms of a satisfaction relation  
 880  $\models_T$  between first-order models and first-order logic formulas, mod-  
 881 uro theories of interest. We bridge this gap in semantics by defining  
 882 two functions, `universalize_c` and `universalize_m`, which translates  
 883 Goblin constraints and models to SMT constraints and models.  
 884 Intuitively, the universalization functions replace dot notation ex-  
 885 pressions with variables, where the variable name is produced by  
 886 serializing the path from the root node to the corresponding leaf  
 887 node in the candidate derivation tree. In the working example, the  
 888 universalized version of Goblin constraint `<Type> = int_to_bv(8,`  
 889 `1) => <Payload>. <Byte> < 32` is `packet__type = int_to_bv(8,`  
 890 `1) => packet__payload__byte < 32`.

## 6 Formal Approach

In this section, we present a formal description of the approach de-  
 895 scribed in Section 3 by formulating it as a calculus. We present some  
 896 preliminaries, give an intuitive description of the calculus, and then  
 897 prove that the calculus offers certain guarantees, namely solution  
 898 soundness, refutation soundness, and solution completeness.

**Derivation trees.** We define a **derivation tree** as an abstract syn-  
 900 tax tree that is potentially unfinished. More precisely, the labeling  
 901 function  $\ell$  is updated to have type  $V \rightarrow N \cup (\bigcup \Lambda) \cup \{\text{None}\}$ ; case  
 902 (c) of syntactic restriction (ii) is updated to allow  $\ell(v) = \text{None}$ ;  
 903 and we introduce a new case (d) of syntactic restriction (ii) which  
 904 allows  $v$  to be a leaf vertex with  $\ell(v) \in N$ . We denote the language  
 905 of derivation trees of a grammar  $G$  as  $\mathcal{L}_{DT}(G)$ , which is analogous  
 906 to  $\mathcal{L}_{AST}(G)$  but for derivation trees rather than ASTs.

**Configurations.** The calculus operates on **configurations**, which  
 908 are either 7-tuples  $(DT, O, C, A, DS, \text{visited}, L)$  or the distinguished  
 909 symbol  $\perp$ , denoting that there is no solution. Each rule of the cal-  
 910 culus updates the current configuration to a new configuration, called  
 911 a **conclusion**, given that the specified preconditions hold. In each  
 912 rule, unprimed variables denote the current configuration, and the  
 913 primed variables represent the updated configuration. If the primed  
 914 version of some configuration variable is left undefined in the con-  
 915 clusion of a rule, it is assumed that the configuration variable is left  
 916 unchanged. A rule **applies** to a configuration  $C$  if all the rule’s pre-  
 917 conditions hold for  $C$ . A configuration is considered **saturated** if it  
 918 is not  $\perp$  and no rule applies. The **initial configuration** for a given  
 919 ACFG  $G = (N, R, \Gamma, S)$  is the seven-tuple  $(DT, O, \{\}, \{\}, [], \{\}, L)$ ,  
 920 where  $DT$  is the derivation containing only a single node labeled  
 921 with  $S$ , and  $O$  is at the zeroth assertion level with no assertions  
 922 pushed, and  $L$  is some natural number (initial depth limit).

**Proof trees.** A **proof tree** is a directed tree where every node maps  
 924 to a configuration, and all the children of each node are obtained  
 925 by the application of some applicable rule to the node. A proof  
 926 tree  $T'$  **derives** from a proof tree  $T$  if  $T'$  is obtainable by applying

929 a rule to one of  $T$ 's leaves. A **derivation** is a finite or countably  
 930 infinite sequence of proof trees such that each proof tree in the  
 931 sequence derives from the previous proof tree and every proof tree  
 932 has an initial configuration at the root. Then, a **refutation** is a  
 933 finite derivation where all leaves of the final derivation tree are  $\perp$ ,  
 934 and a **solution** is a finite derivation where all leaves of the final  
 935 derivation are saturated.

936 Note that since each rule in the calculus only produces a single  
 937 resulting configuration, every proof tree is a path with a single leaf.  
 938 **Configuration description.** The configuration variables are now  
 939 described more intuitively.

- 940 (1)  $DT$  is a derivation tree that represents the (potentially un-  
     finished and backtrackable) construction of an output term.
- 941 (2)  $O$  represents an incremental SMT oracle, supporting oper-  
     ations push, pop, assert, and check\_sat
- 942 (3)  $C$  represents a set of relevant constraints (with respect to  
      $DT$ ).
- 943 (4)  $A$  represents the set of constraints that have been asserted  
     since the last pop() of the incremental SMT oracle.
- 944 (5)  $DS$  represents a decision stack of derivation trees, support-  
     ing operations push, pop, and is\_empty.
- 945 (6)  $\text{visited} : \mathcal{DT} \times V \rightarrow \mathcal{P}(R)$  represents a map of ( derivation  
     tree, vertex ) pairs to the set of production rule expansions  
     that have been explored.
- 946 (7)  $L$  is a natural number representing the current depth limit.

947 A configuration is **satisfiable** if  $DT$ , or some derivation tree in  
 948  $DS$ , can be extended to an AST in  $\mathcal{L}_{\text{AST}}(G)$ , and otherwise, it is  
 949 **unsatisfiable**. Every AST in  $\mathcal{L}_{\text{AST}}(G)$  that is an extension of  $DT$   
 950 or some derivation tree in  $DS$ , only using expansions not captured  
 951 by visited, is called a **model** of the configuration.

952 Analogously, an ACFG  $G$  is called **satisfiable** if its language is  
 953 nonempty, and otherwise it is **unsatisfiable**.

954 **Informal rule explanations.** DECIDE is a rule for expanding an  
 955 open derivation tree at some vertex  $v$  associated with a nonterminal  
 956 with more than one production rule option. We choose an arbitrary  
 957 production rule option and perform the expansion. NORMALIZEPR  
 958 is a rule for normalizing an open derivation tree. It is similar to  
 959 DECIDEPR, but for “forced” expansions where a given nonterminal  
 960 has exactly one production rule option, so no choice needs to be  
 961 made. NORMALIZETA is analogous to NORMALIZEPR, but for type an-  
 962 notations. Note that we never make decisions for type annotations,  
 963 since there is only one expansion option (one child with a value of  
 964 the corresponding type). ASSERT is a rule for asserting an applicable  
 965 constraint at the current assertion level to the incremental SMT  
 966 oracle. BACKTRACKDEPTH is a rule for reverting the last decision  
 967 and backtracking when the depth limit has been reached at some  
 968 open leaf. BACKTRACKUNSAT is a rule for reverting the last decision  
 969 and backtracking when the set of asserted constraints becomes  
 970 unsatisfiable. RESTARTDEPTH and RESTARTUNSAT are analogous to  
 971 BACKTRACKDEPTH and BACKTRACKUNSAT, except they apply when  
 972 there is no prior decision to backtrack. Instead, there is a full restart,  
 973 and the depth limit is increased. SOLVE is a rule for instantiating a  
 974 finished derivation tree with concrete values, assuming the set of  
 975 asserted constraints is satisfiable. FAIL is a rule for giving up when  
 976 the current set of asserted constraints is unsatisfiable, and there  
 977 is no prior decision to backtrack. Moreover, the precondition on  
 978

979  $bd?$  enforces that we have not backtracked due to the depth limit  
 980 (as opposed to backtracking due to an unsatisfiable constraint set)  
 981 since the last restart.

982 **Other preliminaries.** The set of **active assertions** for  $O$  is defined  
 983 as the set of assertions at the current, or lower, assertion levels of  $O$ .  
 984 A constraint  $\varphi$  is considered **applicable** in a derivation tree  $dt$  if for  
 985 all terms  $t$  in  $\varphi$ ,  $I_{dt}(t) \neq \top$  (that is, every dot notation expression  
 986 references children that are actually present in the derivation tree).  
 987 We use  $M$  to denote a model returned by check\_sat; open\_leaves ::  
 988  $\mathcal{DT} \rightarrow \mathcal{P}(V)$  returns the set of open leaves (that is, leaves with  
 989  $\ell(v) \in N$ ) in the input derivation tree; applies ::  $\mathcal{DT} \times C \rightarrow \text{Bool}$   
 990 returns whether or not the input constraint applies to the input  
 991 derivation tree; expand ::  $\mathcal{DT} \times V \times N^* \rightarrow \mathcal{DT}$  expands the  
 992 input derivation tree at the given vertex with the given child node  
 993 labels; new\_dt ::  $N \rightarrow \mathcal{DT}$  takes a nonterminal  $n$  as input and  
 994 produces a new derivation tree with a single vertex of label  $n$ ;  
 995 is\_normalized ::  $\mathcal{DT} \rightarrow \text{Bool}$  is shorthand for whether or not the  
 996 input derivation is normalized (i.e., whether or not NORMALIZEPR  
 997 or NORMALIZETA is applicable); universalize\_c ::  $\mathcal{DT} \times V \times C \rightarrow C$   
 998 rewrites the input constraint, phrasing the nonterminal expressions  
 999 in terms of absolute paths from the root of the current derivation  
 1000 tree to vertex  $v$ ; and universalize\_m ::  $\mathcal{DT} \rightarrow \mathfrak{M}$  translates a  
 1001 derivation tree to a first-order logic model by creating a variable  
 1002 and value for each path in the input derivation tree from root to  
 1003 leaf.

1004 **Lemmas.** Before presenting the main results, we first present some  
 1005 useful helper lemmas.

1006 First, we tie Goblin’s semantics, defined in Section 3, with SMT  
 1007 semantics to soundly enable reasoning with SMT solvers, relating  
 1008 Goblin’s satisfaction relation  $\models_{\mathcal{G}}$  with SMT satisfaction, denoted  
 1009  $\models_{\mathcal{T}}$  (first-order logic satisfaction with respect to any relevant theo-  
 1010 ries). This gap is bridged by the universalize\_m and universalize\_c  
 1011 functions, which translate derivation trees to SMT models and  
 1012 Goblin constraints to SMT constraints, respectively. Intuitively, the  
 1013 functions perform a *flattening* where each path in the derivation  
 1014 tree from root to leaf gets a corresponding SMT variable. Note  
 1015 that the function and predicate symbols are directly translated—  
 1016 the functions and predicates in GOBLIN’s constraint language have  
 1017 direct SMT-LIB [6] analogues.

$$\begin{aligned}
 \text{universalize\_c}(dt, v, \varphi_1 \wedge \varphi_2) &= \\
 \text{universalize\_c}(dt, v, \varphi_1) \wedge \text{universalize\_c}(dt, v, \varphi_2) &= \\
 \text{universalize\_c}(dt, v, \neg\varphi) &= \\
 \neg\text{universalize\_c}(dt, v, \varphi) &= \\
 \text{universalize\_c}(dt, v, p(t_1, \dots, t_n)) &= \\
 p(\text{universalize\_c}(dt, v, t_1), \dots, \text{universalize\_c}(dt, v, t_n)) &= \\
 \text{universalize\_c}(dt, v, f(t_1, \dots, t_n)) &= \\
 f(\text{universalize\_c}(dt, v, t_1), \dots, \text{universalize\_c}(dt, v, t_n)) &= \\
 \text{universalize\_c}(dt, v, \langle nt_1 \rangle \langle nt_2 \rangle \dots \langle nt_n \rangle) &= \\
 \langle \ell(\text{root}(dt)) \rangle \dots \langle \ell(v) \rangle \langle nt_1 \rangle \langle nt_2 \rangle \dots \langle nt_n \rangle &= \\
 \text{universalize\_m}(dt) &= \{(p, v) \mid \\
 l \in \text{leaves}(dt) \wedge p = \langle \ell(\text{root}(dt)) \rangle \dots \langle \ell(\text{parent}(l)) \rangle \wedge v = \ell(l)\} &= \\
 \end{aligned}$$

1045	$\text{is\_normalized}(DT) \quad \text{depth}(DT) \leq L \quad v \in \text{open\_leaves}(DT) \quad \ell(v), NTs, D \notin \text{visited}(DT, v)$	1103
1046	$\{\varphi \in C \mid \text{applies}(DT, \varphi) \wedge \neg(\varphi \in A)\} = \emptyset \quad D^* = \text{universalize\_c}(DT, v, D)$	1104
1047	$\text{DECIDE } DT' \leftarrow \text{expand}(DT, v, NTs) \quad C' \leftarrow C \cup D^* \quad \text{visited}' \leftarrow \text{add}(\text{visited}, (DT, v), (\ell(v), NTs, D))$	1105
1048	$DS' \leftarrow DT :: DS \quad O' \leftarrow \text{push}(O) \quad \text{depth}' \leftarrow \text{depth}[DT' \mapsto \text{depth}(DT) + 1]$	1106
1049		1107
1050		1108
1051	$v \in \text{open\_leaves}(DT) \quad \text{depth}(DT) \leq L \quad  \{\ell(v), NTs, D \in R\}  = 1 \quad D^* = \text{universalize\_c}(DT, v, D)$	1109
1052	$\text{NORMALIZEPR } DT' \leftarrow \text{expand}(DT, v, NTs) \quad C' \leftarrow C \cup D^*$	1110
1053		1111
1054	$v \in \text{open\_leaves}(DT) \quad \text{depth}(DT) \leq L \quad \ell(v), \tau \in \Gamma$	1112
1055	$\text{NORMALIZETA } DT' \leftarrow \text{expand}(DT, v, [])$	1113
1056		1114
1057	$\text{BACKTRACKUNSAT } \begin{array}{l} \text{check\_sat}(O) = \text{UNSAT} \quad DS = dt :: DS_2 \quad V_b = \text{vertices}(DT) - \text{vertices}(dt) \\ DT' \leftarrow dt \quad DS' \leftarrow DS_2 \quad O' \leftarrow \text{pop}(O, 1) \quad A' \leftarrow \emptyset \\ C' \leftarrow C - \{\text{universalize\_c}(DT, v', \varphi) \mid \ell(v'), \varphi \in R \wedge v' \in V_b\} \end{array}$	1115
1058		1116
1059		1117
1060		1118
1061	$\text{BACKTRACKDEPTH } \begin{array}{l} \text{depth}(DT) > L \quad DS = dt :: DS_2 \quad V_b = \text{vertices}(DT) - \text{vertices}(dt) \\ DT' \leftarrow dt \quad DS' \leftarrow DS_2 \quad O' \leftarrow \text{pop}(O, 1) \quad bd?' \leftarrow \text{true} \quad A' \leftarrow \emptyset \\ C' \leftarrow C - \{\text{universalize\_c}(DT, v', \varphi) \mid \ell(v'), \varphi \in R \wedge v' \in V_b\} \end{array}$	1119
1062		1120
1063		1121
1064		1122
1065		1123
1066	$\text{SOLVE } \begin{array}{l} \text{open\_leaves}(DT) = \emptyset \quad \{\varphi \in C \mid \text{applies}(DT, \varphi) \wedge \neg(\varphi \in A)\} = \emptyset \quad \text{get\_model}(O) = \mathcal{M} \quad \text{depth}(DT) \leq L \\ \mathcal{D}\mathcal{T}' \leftarrow \{\text{instantiate}(DT, \mathcal{M})\} \end{array}$	1124
1067		1125
1068		1126
1069	$\text{RESTARTUNSAT } \begin{array}{l} \text{check\_sat}(O) = \text{UNSAT} \quad DS = [] \quad bd? \\ L' \leftarrow L + 1 \quad DT' \leftarrow \text{new\_dt}(S) \quad O' \leftarrow \text{push}(\text{reset}(O)) \quad C', A' \leftarrow \emptyset \quad bd?' \leftarrow \text{false} \quad \text{visited}' \leftarrow \{\} \end{array}$	1127
1070		1128
1071		1129
1072	$\text{RESTARTDEPTH } \begin{array}{l} v \in \text{open\_leaves}(DT) \quad \text{depth}(DT) > L \quad DS = [] \\ L' \leftarrow L + 1 \quad DT' \leftarrow \text{new\_dt}(S) \quad O' \leftarrow \text{push}(\text{reset}(O)) \quad C', A' \leftarrow \emptyset \quad bd?' \leftarrow \text{false} \quad \text{visited}' \leftarrow \{\} \end{array}$	1130
1073		1131
1074		1132
1075		1133
1076	$\text{FAIL } \begin{array}{l} \text{check\_sat}(O) = \text{UNSAT} \quad DS = [] \quad \neg bd? \\ \perp \end{array}$	1134
1077		1135
1078		1136
1079		1137
1080	To tie the semantics of Goblin to the semantics of SMT, then intuitively, we must demonstrate that a Goblin constraint $\varphi$ associated with some production rule in the derivation tree is satisfied by the tree exactly when some corresponding SMT model satisfies the corresponding SMT version of the constraint (where the corresponding SMT model and constraint are defined using <code>universalize_m</code> and <code>universalize_c</code> ).	1138
1081		1139
1082		1140
1083		1141
1084		1142
1085		1143
1086		1144
1087		1145
1088		1146
1089		1147
1090		1148
1091	LEMMA 6.1. Consider an arbitrary Goblin derivation tree $tr$ , which has an arbitrary applicable semantic constraint $\varphi$ at some arbitrary vertex $v$ . We denote the subtree of $tr$ rooted at $v$ by $tr'$ . Then,	1149
1092	$tr' \models_{\mathcal{G}} \varphi$ iff $\text{universalize\_m}(tr) \models_{\mathcal{T}} \text{universalize\_c}(tr, v, \varphi)$ .	1150
1093	PROOF. Assume $\varphi$ includes some arbitrary dot notation term $t$ . We denote the Goblin interpretation of $t$ in $tr'$ by $I_{tr'}(t)$ (as in Section 3), and analogously, we denote the SMT interpretation of the universalized version of $t$ by $I_{\text{universalize\_m}(tr)}(\text{universalize\_c}(tr, v, t))$ . We argue that	1151
1094	$I_{tr'}(t) = I_{\text{universalize\_m}(tr)}(\text{universalize\_c}(tr, v, t))$ .	1152
1095	The left-hand side is defined in the Goblin semantics as the value in $tr'$ at the leaf node $l$ found along the path (from root note $v$ ) denoted by $t$ . The right-hand side is defined as the value of the	1153
1096		1154
1097		1155
1098		1156
1099		1157
1100		1158
1101		1159
1102		1160

variable whose name was constructed by serializing the path from the root of  $dt$  to  $v$ , and from  $v$  to the corresponding leaf node denoted by  $t$ . Notice that this leaf node is precisely the same as  $l$ , hence, the values are equal.

We argued that the interpretations assigned to dot notation terms is equivalent between Goblin semantics and SMT semantics (using our universalization functions). Besides dot notation, for *applicable constraints*, the satisfaction relations  $\models_{\mathcal{G}}$  is defined equivalently to  $\models_{\mathcal{T}}$ . Hence, the claim holds.  $\square$

Using this lemma, we can identify derivation tree constraints taken from  $R$  with universalized SMT constraints produced by the `universalize_c` assumption (see `DECIDE`, `NORMALIZEPR`, and `NORMALIZETA`, which translate the derivation tree constraints to SMT constraints with `universalize_c` before adding them to the constraint set). Further, we can identify SMT models with derivation tree leaf values (see `SOLVE`, which instantiates the SMT model from  $O$  into  $DT$ ). Note that with the universalization functions, we avoid name clashes between variables in semantic constraints associated with nonterminals that are explored multiple times in the same derivation.

1161 Next, we present two lemmas to couple the state of  $O$  to  $DT$   
 1162 in every configuration (intuitively, the assertions active in  $O$  are  
 1163 exactly those that are relevant to  $DT$ ).  
 1164

1165 **LEMMA 6.2.** *For every configuration of every derivation, every*  
 1166 *active assertion in  $O$  is applicable in  $DT$ .*

1167 **PROOF.** The relevant rules are those that introduce new asser-  
 1168 tions or shrink  $DT$ , namely, ASSERT, BACKTRACKUNSAT, BACKTRACK-  
 1169 DEPTH, RESTARTUNSAT, and RESTARTDEPTH. We proceed by struc-  
 1170 tural induction on the rules, mentioning only these relevant cases.  
 1171 (1) ASSERT only adds assertions that are applicable in  $DT$  from the  
 1172 third precondition. (2) BACKTRACKUNSAT and BACKTRACKDEPTH  
 1173 both update  $DT$  to be the old head of  $DS$  and pop  $O$ . Now notice  
 1174 that the only rule which pushes to the decision stack or pushes an  
 1175 assertion level is DECIDE. By the inductive hypothesis, when  $DT$   
 1176 was pushed to the decision stack by DECIDE,  $O$ 's active assertions  
 1177 were all applicable to  $DT$ . Notice that every push to  $DS$  has a cor-  
 1178 responding push to  $O$  (see DECIDE), and analogously for pops (see  
 1179 BACKTRACKDEPTH and BACKTRACKUNSAT). Hence, in the backtrack  
 1180 rules, when we pop to retrieve  $DT$  from  $DS$  while simultaneously  
 1181 popping  $O$ , we return exactly to the same state of  $O$  from when  $DT$   
 1182 was pushed. Therefore, all the asserted constraints are applicable.  
 1183 (3) RESTARTUNSAT and RESTARTDEPTH both trivially satisfy the  
 1184 claim, as they reset the assertion stack (popping all assertions).  $\square$   
 1185

1186 **LEMMA 6.3.** *For every configuration of every derivation, every*  
 1187 *constraint from the input grammar which is applicable in  $DT$  is in  $C$ .*

1188 **PROOF.** New expansions of  $DT$  involving new production rule  
 1189 constraints only occur in DECIDE and NORMALIZEPR. In both cases,  
 1190 all constraints associated with the chosen production rule (including  
 1191 the applicable ones) are added to  $C$ .  $\square$   
 1192

1193 **LEMMA 6.4.** *For input grammar  $G$ , for every configuration of every*  
 1194 *derivation,  $DT \in \mathcal{L}_{DT}(G)$ .*

1195 **PROOF.** By structural induction on the rules. The relevant cases  
 1196 are DECIDE, NORMALIZEPR, and NORMALIZETA. In DECIDE and NOR-  
 1197 MALIZEPR, if  $DT \in \mathcal{L}_{DT}(G)$ , then  $DT' \in \mathcal{L}_{DT}(G)$  since  $DT'$  is  
 1198 constructed by expanding  $DT$  according to one of the production  
 1199 rules in  $R$ . In NORMALIZETA, if  $DT \in \mathcal{L}_{DT}(G)$ , then  $DT' \in \mathcal{L}_{DT}(G)$   
 1200 since  $DT$  is expanded by filling in a symbolic leaf value (i.e., an  
 1201 uninstantiated leaf with  $\ell(v) = \text{None}$ ) for some vertex associated  
 1202 with a type annotation in  $\Gamma$ . The other cases do not expand  $DT$ , so  
 1203 we trivially get that  $DT' \in \mathcal{L}_{DT}(G)$ .  $\square$   
 1204

1205 Finally, we present lemmas that are directly relevant for refuta-  
 1206 tion and solution soundness (respectively).

1207 **LEMMA 6.5.** *Every rule except for SOLVE and BACKTRACKDEPTH*  
 1208 *preserves models.*

1209 **PROOF.** By cases. (1) DECIDE updates  $DT$  by making a new de-  
 1210 cision and expanding an open leaf according to a production rule  
 1211 option in  $R$ . The models associated with  $DT$  can be partitioned into  
 1212 those that use this expansion for the open leaf, and those that use  
 1213 a different expansion. The prior set is still captured in  $DT'$ , while  
 1214 the latter set are captured by the new head of  $DS'$ , namely,  $DT$ . (2)  
 1215 NORMALIZEPR and NORMALIZETA both expand  $DT$  without altering  
 1216 the decision stack. Since the expansions are forced (a precondition  
 1217

1218 is that there are no alternative expansions), they cannot lose mod-  
 1219 els. (3) ASSERT does not directly alter  $DT$  or  $DS$ , so it only relevant  
 1220 through Lemmas 6.3 and 6.2, which are used in the SOLVE case. (4)  
 1221 BACKTRACKUNSAT discards  $DT$ , transfers the head of  $DS$  to  $DT'$ ,  
 1222 and maintains the rest of  $DS$ . We know from Lemma 6.2 that all the  
 1223 current constraints are relevant for  $DT$ , and yet the constraint set  
 1224 is unsatisfiable. Hence,  $DT$  cannot be extended to any models, so  
 1225 it is safe to discard. Since the rest of the derivation trees in  $DS$  are  
 1226 shifted/maintained, no models are lost. (5) RESTARTUNSAT removes  
 1227 everything but the root from  $DT$ , and it does not change  $DS$ , so it  
 1228 cannot possibly remove models. (6) RESTARTDEPTH proceeds by  
 1229 the same argument as RESTARTUNSAT (7) For FAIL, We know from  
 1230 Lemma 6.2 that all the current constraints are relevant for  $DT$ , and  
 1231 yet the constraint set is unsatisfiable. Hence,  $DT$  cannot be extended  
 1232 to any models. But further,  $DS$  is empty, so there cannot be any  
 1233 models in the unprimed configuration. Since there are no models  
 1234 in the unprimed configuration, losing models is impossible.  $\square$   
 1235

1236 Below, we use *productive* to denote that a nonterminal is capable  
 1237 of deriving some finite string.

1238 **LEMMA 6.6.** *For an input ACFG  $G$  such that every nonterminal*  
 1239 *is productive, if the calculus does not terminate at depth limit  $n$ , it*  
 1240 *enumerates all trees of depth  $\leq n$  which could possibly extend to a*  
 1241 *model before moving on to depth  $n + 1$ .*

1242 **PROOF.** The calculus implements an iterative deepening search,  
 1243 which exhaustively explores the search space of derivation trees  
 1244  $\leq L$  using DECIDE to store decisions to be backtracked later, BACK-  
 1245 TRACKDEPTH to enforce the depth limit, and RESTARTDEPTH to  
 1246 increase the depth limit once the current limit has been thoroughly  
 1247 explored.

1248 First, we argue by cases that IDS always *makes progress* in the  
 1249 sense that it is impossible to have an infinite chain of rule applica-  
 1250 tions that does not include DECIDE. ASSERT relies on applications  
 1251 of DECIDE to collect formulas in  $C$  to assert. BACKTRACKDEPTH  
 1252 and BACKTRACKUNSAT rely on decisions to backtrack. SOLVE al-  
 1253 ways produces a saturated conclusion (argued in Theorem 6.7).  
 1254 RESTARTUNSAT relies on BACKTRACKDEPTH for the precondition in-  
 1255 volving  $bd?$ , which in turn relies again on DECIDE. RESTARTDEPTH,  
 1256 NORMALIZEPR, and/or NORMALIZETA could, in principle, chain in-  
 1257 definitely without allowing an application of DECIDE. However,  
 1258 this is impossible with our assumption that each nonterminal is  
 1259 productive.

1260 Notice that calculus does not enumerate every possible derivation  
 1261 tree due to the presence of BACKTRACKUNSAT and RESTAR-  
 1262 TUNSAT, which discard derivation trees without exploring every  
 1263 possible expansion. However, whenever these rules are invoked,  
 1264 the constraint set is in an unsatisfiable state. Since the constraint  
 1265 set is applicable to  $DT$  by Lemma 6.2, no extension of  $DT$  could  
 1266 possibly lead to a model. Thus, it is safe to discard  $DT$  and refrain  
 1267 from exploring its possible extensions.

1268 Since IDS always makes progress and all other derivation trees  
 1269 are enumerated by IDS, the claim holds.  $\square$   
 1270

1271 **Theorems.** We now present the main theorems.

1272 **THEOREM 6.7.** *(Refutation soundness) Every refutation has an*  
 1273 *initial configuration associated with an unsatisfiable grammar.*

**PROOF.** First, we argue that no refutation can include an application of SOLVE. Assume for a contradiction that SOLVE is applied somewhere in a refutation. Then, the conclusion of this SOLVE rule must not be a saturated configuration, because the proof tree must have  $\perp$  at every leaf. However, this is impossible, as after applying SOLVE, none of the other rules can be applicable: DECIDE, NORMALIZEPr, and NORMALIZETA require some open leaf, but one cannot exist (see the preconditions to SOLVE). ASSERT requires the existence of some  $\varphi \in C$  such that  $\varphi$  is applicable but not in  $A$ , but this is also impossible due to a precondition of SOLVE. BACKTRACKUNSAT requires that the current asserted constraint set by unsatisfiable, but this cannot be the case since  $O$  just returned a model from check\_sat. BACKTRACKDEPTH requires the depth of  $DT$  to be above  $L$ , but this contradicts a precondition of SOLVE. RESTARTUNSAT and RESTARTDEPTH proceed with the same arguments as BACKTRACKUNSAT and BACKTRACKDEPTH. FAIL proceeds with the same argument as BACKTRACKUNSAT.

Second, we argue that every refutation containing BACKTRACKDEPTH can be rewritten a different refutation that does not contain BACKTRACKDEPTH. (We do this because BACKTRACKDEPTH does not necessarily preserve models.) Notice that due to the precondition on  $bd?$ , no proof tree can include an application of BACKTRACKDEPTH, followed by an application of FAIL, without an application of RESTARTDEPTH or RESTARTUNSAT in between. Therefore, consider an arbitrary proof tree containing BACKTRACKDEPTH. Then, consider the last application of RESTARTDEPTH or RESTARTUNSAT, and take the subtree rooted at the conclusion of this application as the new refutation. (This conclusion is a valid initial configuration, just with a higher depth limit.)

Since SOLVE and BACKTRACKDEPTH are never present, we can argue from structural induction on proof trees using Lemma 6.5 that any proof tree with a leaf containing  $\perp$  must also have an unsatisfiable root.  $\square$

**THEOREM 6.8. (Solution soundness)** *Every solution has an initial configuration associated with a satisfiable grammar.*

**PROOF.** We argue that every saturated leaf of a solution corresponds to a model of the input grammar. The arguments proceeds in three steps. We argue that (1) the conclusion of an application to SOLVE must be saturated, (2) the conclusion of an application of SOLVE corresponds to a model, and (3) the conclusion of any rule except for SOLVE cannot be saturated.

We already argued step (1) in Theorem 6.7.

For step (2), we can directly take  $DT'$  as the model. We know  $DT'$  is syntactically valid from Lemma 6.4, and we know it is semantically valid with respect to all the applicable constraints from Lemmas 6.3, 6.2, and 6.1, along with the precondition that all applicable constraints in  $C$  must be asserted since the last pop (second precondition). Finally, the inapplicable constraints are trivially satisfied, according to the semantics in Section 3.

We argue step (3) by contradiction. Assume the conclusion of another rule is saturated. First, the depth of  $DT$  must be  $\leq L$ , or else BACKTRACKDEPTH or RESTARTDEPTH would be applicable. Second, the derivation tree cannot have any open leaves, or else DECIDE, NORMALIZEPr, or NORMALIZETA would be applicable (assuming  $DT$  has depth  $\leq L$ ). Also, the derivation tree cannot have any applicable

but unasserted constraints, or else ASSERT would be applicable. Finally, the current constraint set cannot be unsatisfiable, or else BACKTRACKUNSAT or RESTARTUNSAT would be applicable. However, since all of the above hold, it must be the case that SOLVE is applicable, a contradiction with the assumption that the conclusion was saturated.  $\square$

In solution completeness, note that the precondition of nonterminal productivity is enforced in GOBLIN by a syntactic check before the main solve algorithm.

**THEOREM 6.9. (Solution completeness)** *For every satisfiable grammar  $G$  such that every nonterminal is productive, there exists a solution with an initial configuration built from  $G$ .*

**PROOF.** The result directly follows from Lemma 6.6.  $\square$

Notably, the calculus is refutationally incomplete in the sense that there might not exist a refutation for unsatisfiable initial configurations. This is unsurprising, as the unsatisfiability of the initial configuration may be an inductive property (e.g., a grammar denoting a list of integers whose individual elements are negative but whose total sum must be positive). Refutations are not discussed in prior work [30], so the presence of refutations and refutation soundness in our calculus is an improvement over prior work, despite the lack of refutation completeness.

## 7 CLP Semantics

We present an alternative language semantics for GOBLIN as a *translational semantics* to CLP problems. While the semantics described in Section 3 works naturally with our proofs, the CLP semantics is also useful to get a sense for how GOBLIN relates to CLP. More concretely, we define a translation of an input ACFG to a CLP problem, which we model as a set of constrained horn clauses (CHCs). Each CHC is of the form  $H \leftarrow B_1, \dots, B_n, \phi$ , where  $H$  and  $B_1$  through  $B_n$  are terms, and  $\phi$  is a constraint over the free variables from the prior terms.

Then,

$$\llbracket G \rrbracket = \bigcup_{nt, N, C \in R} \llbracket nt, N, C \rrbracket_{\text{pr}} \cup \bigcup_{nt, \tau \in \Gamma} \llbracket nt, \tau \rrbracket_{\text{ta}}$$

where

```

1393
1394      $\llbracket nt, \tau \rrbracket_{ta} = nt(NT) \leftarrow NT \in \tau$ 
1395
1396      $\llbracket nt, N, C \rrbracket_{pr} = \llbracket nt, N, C \rrbracket_{lhs} \leftarrow \llbracket nt, N, C \rrbracket_{rhs}$ 
1397
1398          $\cup \bigcup_{e \in nt\_exprs(C)} \llbracket e \rrbracket_{fe}$ 
1399
1400      $\llbracket nt, N, C \rrbracket_{lhs} = nt(nt_i(N_1, \dots, N_m))$ , where each  $N_j \in N$ 
1401         for the  $i$ th production rule option for  $nt$ 
1402
1403      $\llbracket nt, N, C \rrbracket_{rhs} = \bigwedge_{m \in N} m(M_i) \wedge \bigwedge_{e \in nt\_exprs(C)} \llbracket e \rrbracket_{nt\_expr} \wedge \llbracket C \rrbracket_{b\_expr}$ 
1404         for the  $i$ th occurrence of  $m$  in  $N$ 
1405
1406      $\llbracket nt_1.nt_2 \dots nt_n \rrbracket_{fe} = \bigwedge \llbracket nt_2 \dots nt_n \rrbracket_{fe}(nt_2, nt_n)$ 
1407          $\wedge \text{append}(nt_{n-1}, \dots, nt_{n-i})$ 
1408
1409      $\llbracket nt_1.nt_2 \rrbracket_{fe} = nt_1\_nt_2(nt_{1,i}(q_1, \dots, q_n), [NT_{2,1}, \dots, NT_{2,m}])$ 
1410         for the  $i$ th production rule option for  $nt_1$ , where
1411             each  $q_j$  is  $NT_{2,k}$  if the  $j$ th nonterminal in the  $i$ th
1412             production rule for  $nt_1$  is  $nt_2$ , and  $_$  otherwise,
1413             with  $k$  ranging from 1 to  $m$ 
1414
1415      $\llbracket nt_1 \dots nt_n \rrbracket_{nt\_expr} = nt_1 \dots nt_n(NT_1, NT_1 \dots NT_n)$ 
1416
1417      $\llbracket C \rrbracket_{b\_expr} = \bigwedge_{c_i \in \text{generalize}(C)} c_i$ 
1418

```

Above, `generalize` performs the disambiguation of dot notation constraints analogous to `resolve` as discussed in Section 4, and `nt_exprs(.)` retrieves a set of all dot notation expressions present in the input constraints. Also, we use square brackets to denote list literals and `append` to denote a multi-arity list append function. The pattern “ $_$ ” denotes a universal match for an unused value. In the encoding, we use both lowercase and uppercase names, which tie to corresponding nonterminals in the input grammar. By convention, we use the lowercase names to denote constructor symbols and the uppercase names to denote variables (e.g. `nt` and `NT`, both referring to the same nonterminal in the input grammar).

Intuitively, we partition the CHCs into those we need for type annotations and those we need for production rules. The type annotation CHCs simply state that the given nonterminal must be a member of its given type; the production rule CHCs are more complicated. First, the production rule CHC definitions are split into the left-hand side (LHS) and right-hand side (RHS). They use CLP terms to capture the structure of the grammar and CLP constraints to capture the semantic constraints from  $C$ . In addition to these LHS and RHS definitions, each production rule also generates CHCs which we call *field extractors*, defined by  $\llbracket \cdot \rrbracket_{fe}$ , which enable passing variables between clauses to support the dot notation constraints. The field extractor CHCs are invoked by calls defined by  $\llbracket \cdot \rrbracket_{nt\_expr}$ .

## 8 Implementation and Evaluation

**GOBLIN Implementation.** We implemented GOBLIN in ~9.9K lines of OCaml 5.1.1 code. For our underlying constraint solver, we use and fully support cvc5 [5] version 1.3.0. The main pipeline is depicted in Figure 1.

**Code availability.** GOBLIN’s source code and experimental evaluation details are available at [3].

**Empirical Evaluation.** We evaluate GOBLIN on four case studies. The first three case studies, XML, CSV, and Scriptsize C input generation, serve as a direct comparison to prior work [30] and demonstrate GOBLIN’s capability of handling complex constraints. In these case studies, we evaluate the correctness, efficiency, diversity, and length of generated inputs. The fourth case study involves generation of valid WiFi SAE commit and confirm frames, which cannot be handled directly by prior work due to the presence of rich bitvector constraints. Rather than taking a fixed grammar and constraints (as performed by prior work and in the prior two case studies), we demonstrate how GOBLIN’s local production rule constraints can be easily integrated into a fuzzing workflow involving *grammar-level production rule mutation operations*.

### 8.1 XML, CSV, and Scriptsize C

We present our main results for the XML, CSV, and Scriptsize C case studies in Table 1. We collected the results with 1-hour runs of GOBLIN and ISLa locally on a Macbook Pro with an M2 chip and 16GB RAM. Our main research questions are:

- RQ1. How *efficient* is GOBLIN compared to ISLa?
- RQ2. How *diverse* are GOBLIN’s outputs, compared to ISLa?
- RQ3. What is the *size* of GOBLIN’s outputs, compared to ISLa?

Both case studies involve *structural constraints*. In XML, there is a **definition before use** constraint that mandates that XML namespaces must be used only in scopes in which they are defined, a **no redefinition** constraint that mandates that XML namespaces are not redefined, and a **matching tags** constraint that matching closing and opening tags have the same identifiers. Scriptsize C has analogous definition before use and no redefinition constraints. In CSV, there is a **column count** constraint that mandates that all rows in the CSV file contain the same number of columns. In ISLa, structural constraints are supported through built-in **structural predicates**. However, in GOBLIN’s DSL, structural predicates are not built-in. *Nonetheless, these constraints can still be encoded in GOBLIN.* (The encoding will be discussed in Section 8.2.)

**Correctness.** Based on our formal soundness guarantees, we can confidently claim that all generated inputs are correct with respect to the input specification. Prior work [2, 30] use *precision* as a metric for evaluation, where precision denotes the percentage of inputs successfully parsed by the system under test. We argue this metric is not relevant, since it is a metric of the system under test rather than of the input generator. However, we note that for CSV and XML, we formulated constraints equivalent to those in prior work.

**Efficiency.** To address RQ1, we compare the efficiency of GOBLIN to ISLa in terms of the number of inputs generated per second. GOBLIN outperforms ISLa by a wide margin for XML inputs and by a narrow margin for Scriptsize C inputs. We hypothesize that incremental solving boosts performance by allowing quicker backtracking and enabling the mending of partial solutions. Notably, GOBLIN struggles on CSV inputs. The semantic constraint, which mandates that all rows have the same number of columns, entirely restricts the *derivation strategy* at the context-free level, rather than the values of leaf terms. A current weakness of GOBLIN is that it does not use any special heuristic for choosing production rule expansions (only coin flips), which we believe explains the poor performance compared to ISLa.

**Diversity.** To address RQ2, we compare efficiency based on *accumulated k-path coverage* [17], which measures how many paths of length  $k$  from the grammar are actually present in the generated inputs, where a *path* refers to a sequence of nonterminal or terminal symbols. A higher  $k$ -path coverage is more diverse and thus better. As in [30], we present results with  $k = 3$ . GOBLIN performs very well, reaching near full coverage on every case study. However, we note that because GOBLIN generates ADT terms rather than strings, we do not use any terminal symbols in our encoding (terminal symbols are generated from leaf nonterminals, optionally with semantic constraints), which differs from ISLa’s encoding which includes terminal symbols. So, although GOBLIN performs well, we cannot claim that the comparison with ISLa is direct.

**Length.** To address RQ2, we compare the mean length of inputs generated by GOBLIN and ISLa. GOBLIN’s inputs, on average, are more concise than ISLa’s. We report this metric to be informative, but we argue that there is no clear benefit or drawback to having shorter or longer inputs. Further, one can always impose additional semantic constraints or grammar-level mutations to either increase or decrease the length of generated inputs.

## 8.2 Structural Constraint Encoding

It may be unclear how one would express structural constraints in GOBLIN. Intuitively, we use local production rule constraints, along with dot notation and support for rich data types (e.g., sets), to specify the desired properties akin to how one would do so with an attribute grammar or a CLP system. The encodings are available along with GOBLIN’s source code at [3]. As a simpler example, consider a grammar encoding two lists with the structural constraint that every element in the second list must also be present in the first:

**Listing 11: Structural constraint**

```

1541 <S> ::= <L1> <L2>
1542   { <L2>.::_def<sub>s</sub>_down = <L1>.::_def<sub>s</sub>_up; } ;
1543 <L1> ::= <_def<sub>s</sub>_up> <str> <L1>
1544 { <_def<sub>s</sub>_up> = union(singleton(<str>),
1545   <L1>.::_def<sub>s</sub>_up); }
1546   | <_def<sub>s</sub>_up> <str>
1547     { <_def<sub>s</sub>_up> = singleton(<str>); }
1548 <L2> ::= <_def<sub>s</sub>_down> <str> <L2>
1549   { member(<str>, <_def<sub>s</sub>_down>);
1550     <L2>.::_def<sub>s</sub>_down = <_def<sub>s</sub>_down>; }
1551   | <_def<sub>s</sub>_down> <str>
1552     { member(<str>, <_def<sub>s</sub>_down>); }
1553 <str> :: String;
1554 <_def<sub>s</sub>_down> :: Set(String);
1555 <_def<sub>s</sub>_up> :: Set(String);

```

We specify the constraint by introducing extra nonterminal symbols into the grammar, preceded by underscores, to serve as means of passing context in the style of attribute grammars. We remind the reader that GOBLIN is designed as a minimal intermediate language, and a surface-level language with proper support for attributes and other features is left for future work.

Tool	Subject	Constraints	Efficiency (inputs/minute)	k-Path coverage	Mean length
GOBLIN	C	def-use; redef;	108.53	95.4	18.75
ISLa	C	def-use; redef;	101.70	50.72	24.01
GOBLIN	XML	def-use; redef; tags	100.00	84.44	11.71
ISLa	XML	def-use; redef; tags	48.00	75.37	38.03
GOBLIN	CSV	col count	19.5	100.00	88.45
ISLa	CSV	col count	91.83	100.00	487.08

**Table 1: Results for each subject, tool, and associated metrics.**

## 8.3 WiFi: WPA3-SAE Handshake Frames

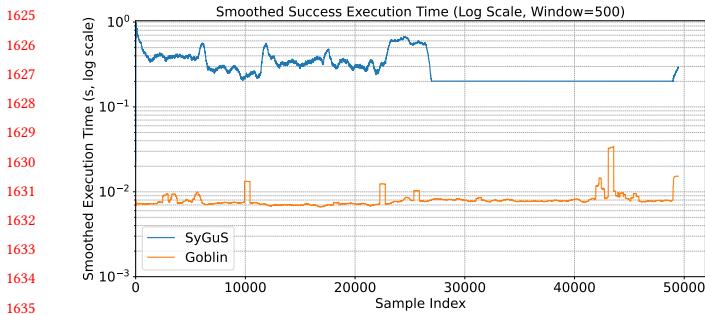
We now present the results of our WiFi authentication case study. We focus on the WPA3-SAE handshake as this is the state-of-the-art WiFi authentication protocol [20] with complex packet formats consisting of (1) rich bitvector constraints, (2) inter-field dependencies, and (3) non-linear fields (cryptographic elements). We instantiate GOBLIN with an existing grammar-based black-box fuzzer SAECRED [12] that generates context-sensitive SAE commit and confirm frames. SAECRED reduces input generation to a SyGuS problem and uses a SyGuS solver [1] to generate byte sequences satisfying the grammar constraints. To evaluate GOBLIN, we replace SAECRED’s input generator with GOBLIN (referred to as GOBLIN-SAECRED) and compare it with the original SAECRED fuzzer.

**Evaluation setup.** We run GOBLIN-SAECRED and SAECRED for 24 hours each on an Intel Core Ultra 9 185H with 64 GB RAM. The black-box target was hostapd v2.10, a popular open-source WiFi access point software. We use the same grammar and constraints for both fuzzers, which we obtained from the SAECRED authors.

**Evaluation criteria.** We propose three evaluation criteria: (1) *performance* – we measure the time taken for Goblin and SyGuS to generate byte sequences from the mutated grammar, (2) *expressiveness* – we measure the number of grammar-level mutations Goblin and SyGuS can handle, and (3) *effectiveness* – we measure the number of bugs uncovered in hostapd by the two fuzzers.

**Performance.** Both variations of SAECRED were able to generate 97,182 mutated grammars for byte serialization. Out of those, 49,460 passed the GOBLIN language-emptiness checks. Figure 2 shows the (log-scaled) smoothed time taken by GOBLIN-SAECRED and SyGuS-SAECRED to generate byte sequences from the 49,460 mutated grammar instances. GOBLIN-SAECRED outperforms SyGuS-SAECRED by a wide margin. Table 2 shows the summary statistics of the time taken to generate the byte sequences. GOBLIN-SAECRED is 36x faster on average than SyGuS-SAECRED.

**Expressiveness.** Out of the 97,182 mutated grammar instances, some instances were unsolvable due to the presence of unsatisfiable constraints; however, the expressive capabilities of both approaches were vastly different. Table 2 shows that GOBLIN-SAECRED was able to handle 49,460 grammar instances whereas SyGuS-SAECRED was only able to handle 23,904 grammar instances, leading to 73,278 failures as opposed to GOBLIN-SAECRED’s 47,722. This shows that



**Figure 2: Performance comparison of GOBLIN-SAECRED and SyGuS-SAECRED on WiFi SAE commit and confirm frames.** The x-axis shows the grammar instance, and the y-axis shows the log-scale time taken to produce byte serializations.

Metric / Method	Goblin	SyGuS	Unit
Successes	49,460	23,904	count
Failures	47,722	73,278	count
<b>Total Grammars: 97,182</b>			
Mean Time	0.0082	0.2968	seconds
Median Time	0.0074	0.2014	seconds
Std. Dev. Time	0.0588	0.3606	seconds

**Table 2: Comparison of GOBLIN-SAECRED and SyGuS-SAECRED in terms of success/failure count and execution time. Goblin on average performs 36x faster than SyGuS.**

GOBLIN is able to handle a wider variety of grammar-level mutations than SyGuS-based input generator of SAECRED.

**Effectiveness.** The 24 hour fuzzing campaigns resulted in GOBLIN-SAECRED uncovering all 4 known bugs from the original SAECRED paper [12], including denial-of-service and downgrade attacks.

## 9 Related Work

The most relevant related work was discussed in Sections 1 and 2: context-free grammar-based fuzzing [18], ISLa and Fandango [2, 30], system-specific input generators [8, 33], and CLP approaches [14]. **SyGuS.** Another related formalism is syntax-guided synthesis (SyGuS) [1], which presents a synthesis problem in terms of a function to synthesize that must be in the language of some user-defined grammar while also satisfying user-defined semantic constraints. In principle, one could use a SyGuS engine (e.g., cvc5SY [29]) as the backend reasoner for a tool like GOBLIN (as in SAECRED [12]). But, there are notable obstacles. For recursive grammars, in order to specify that a constraint applies to *all* instances of a production rule expansion during a derivation, the corresponding SyGuS constraint must be expressed as a recursive function, which is nontrivial to synthesize and often suffers from performance problems in a SyGuS setting. Further, by default, SyGuS solvers only support semantic constraints from the top level (i.e., predicates of type  $\tau \rightarrow \text{Bool}$ , where  $\tau$  is the type of the function to synthesize). Hence, supporting local production rule semantic constraints in general is nontrivial.

**Attribute grammars.** Attribute grammars [22] are a formalism built on top of context-free grammars, where the grammar can additionally assign *attributes* to each production rule. Attributes are additional variables that can be used to communicate context-sensitive information throughout a parse tree, and can be defined in terms of descendants (called *synthesized attributes*) or ancestors (called *inherited attributes*). One can use attributes to express and check semantic properties; however, prior work (e.g., [31]) on attribute grammars has focused on *parsing* rather than *generation*. **Property-based testing.** Property-based testing (PBT), pioneered by QuickCheck [9], is a testing strategy in which the programmer annotates expected invariants in the source code, and a testing tool generates inputs to check whether or not the invariants are ever falsified. These approaches differ from the one employed by GOBLIN in two main ways: (i) GOBLIN operates with system-level input rather than internal inputs, and (ii) GOBLIN allows for the annotation of rich semantic constraints and employs heuristics that are more in depth than *generate-and-test* approaches typical in PBT. **Context-sensitive parsing.** In general, most work on grammars with context-sensitive constraints has focused on parsing rather than input generation. For example, Parsley [24] is a tool that uses an input language similar in spirit to GOBLIN’s, but it is a tool for synthesizing verified parsers for languages with context-sensitive constraints (rather than for input generation).

## 10 Discussion

**Limitations.** Compared to prior works, ours is most similar in spirit to ISLa [30], and we experience a similar set of tradeoffs. For non-derived fields, the constraint language space is restricted to being easily translatable to SMT-LIB [6]. Moreover, constraint solving can be expensive for some inputs.

## 11 Conclusion

We presented GOBLIN, a system for generating context-sensitive inputs. Compared to prior work, GOBLIN is novel in its capability to handle constraints from arbitrary SMT theories, its generality despite using a minimal core language, and its amenability to integration with grammar-based fuzzing workflows due to its semantics of local production rule constraints.

## References

- [1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo MK Martin, Mukund Raghoothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emilia Torlak, and Abhishek Upadhyay. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.
- [2] JOSÉ ANTONIO ZAMUDIO AMAYA, MARIUS SMYTZEK, and ANDREAS ZELLER. 2025. FANDANGO: Evolving Language-Based Testing. (2025).
- [3] Anonymous. 2025. Parse this! Summoning Context-Sensitive Inputs with Goblin. [https://anonymous.4open.science/r/goblin\\_anonymous-44E1](https://anonymous.4open.science/r/goblin_anonymous-44E1). Anonymous submission.
- [4] John W Backus. 1959. The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *ICIP Proceedings*. 125–132.
- [5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.
- [6] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

- 1741 [7] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook*  
1742 *of model checking*. Springer, 305–343.  
1743 [8] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly  
1744 Shmatikov. 2014. Using frankencerts for automated adversarial testing of certifi-  
1745 cate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security*  
1746 *and Privacy*. IEEE, 114–129.  
1747 [9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for  
1748 random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN*  
1749 *international conference on Functional programming*, 268–279.  
1750 [10] Bruno Courcelle and Paul Franchi-Zannettacci. 1982. Attribute grammars and  
1751 recursive program schemes I. *Theoretical Computer Science* 17, 2 (1982), 163–191.  
1752 [11] cvc5 Developers. 2025. cvc5 Nonstandard Theories. <https://cvc5.github.io/docs/cvc5-1.0.2/theories/theories.html#non-standard-or-extended-theories>.  
1753 [12] Muhammad Daniyal Pirwani Dar, Rob Lorch, Aliakbar Sadeghi, Vincenzo Sorciglio,  
1754 Hélène Gollier, Cesare Tinelli, Mathy Vanhoef, and Omar Chowdhury. 2025.  
1755 *Sacred: A State-Aware, Over-the-Air Protocol Testing Approach for Discovering*  
1756 *Parsing Bugs in SAE Handshake Implementations of COTS Wi-Fi Access Points*.  
1757 In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3691–3709.  
1758 [13] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Pro-  
1759 gram for Theorem-Proving. *Commun. ACM* 5, 7 (1962), 394–397. doi:10.1145/  
1760 368273.368557  
1761 [14] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using  
1762 constraint logic programming. In *Proceedings of the 29th ACM/IEEE international*  
1763 *conference on Automated software engineering*, 725–730.  
1764 [15] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and  
1765 Ernest Kurniawan. 2022. {BrakTooth}: Causing havoc on bluetooth link manager  
1766 via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*,  
1767 1025–1042.  
1768 [16] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. 2020. Grey-  
1769 hound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and*  
1770 *Secure Computing* 19, 2 (2020), 817–834.  
1771 [17] Nikolas Havrikov and Andreas Zeller. 2019. Systematically covering input  
1772 structure. In *2019 34th IEEE/ACM international conference on automated software*  
1773 *engineering (ASE)*. IEEE, 189–199.  
1774 [18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code  
1775 fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, 445–458.  
1776 [19] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to  
1777 automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001),  
1778 60–65.  
1779 [20] IEEE Std 802.11. 2020. Wireless LAN Medium Access Control (MAC) and Physical  
1780 Layer (PHY) Spec.  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798 [21] Intiaz Karim, Fabrizio Cicala, Syed Raful Hussain, Omar Chowdhury, and Elisa  
1799 Bertino. 2020. ATFuzzer: dynamic analysis framework of AT interface for Android  
1800 smartphones. *Digital Threats: Research and Practice* 1, 4 (2020), 1–29.  
1801 [22] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical*  
1802 *systems theory* 2, 2 (1968), 127–145.  
1803 [23] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel  
1804 Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engi-  
1805 neering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11  
1806 (2019), 2312–2331.  
1807 [24] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anan-  
1808 tharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. 2020. Research Report:  
1809 The Parsley Data Format Definition Language. In *2020 IEEE Security and Privacy*  
1810 Workshops (SPW). 300–307.  
1811 [25] NIST. 2023. CVE-2021-1903. <https://nvd.nist.gov/vuln/detail/cve-2023-29548>.  
1812 [26] NIST. 2023. CVE-2021-30247. <https://nvd.nist.gov/vuln/detail/cve-2021-30247>.  
1813 [27] NIST. 2023. CVE-2023-29548. <https://nvd.nist.gov/vuln/detail/cve-2023-29548>.  
1814 [28] NIST. 2024. CVE-2024-52924. <https://nvd.nist.gov/vuln/detail/CVE-2024-52924>.  
1815 [29] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare  
1816 Tinelli. 2019. cvc4sy: Smart and fast term enumeration for syntax-guided synthe-  
1817 sis. In *International Conference on Computer Aided Verification*. Springer, 74–83.  
1818 [30] Dominic Steinböfel and Andreas Zeller. 2022. Input invariants. In *Proceedings*  
1819 *of the 30th ACM joint european software engineering conference and symposium on*  
1820 *the foundations of software engineering*. 583–594.  
1821 [31] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An  
1822 extensible attribute grammar system. *Science of Computer Programming* 75, 1–2  
1823 (2010), 39–54.  
1824 [32] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superior: Grammar-  
1825 aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Soft-*  
1826 *ware Engineering (ICSE)*. IEEE, 724–735.  
1827 [33] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and under-  
1828 standing bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference*  
1829 *on Programming language design and implementation*. 283–294.  
1830 [34] Michal Zalewski. 2014. American Fuzzy Lop (AFL). In *USENIX WOOT*. <https://lcamtuf.coredump.cx/afl/>.  
1831 [35] Jialun Zhang, Greg Morrisett, and Gang Tan. 2023. Interval parsing grammars  
1832 for file format parsing. *Proceedings of the ACM on Programming Languages* 7,  
1833 PLDI (2023), 1073–1095.  
1834 Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856