# *Firebolt*: Finding Bugs in Programmable Data Plane Generators

Jiamin Cao*, Yu Zhou†, Chen Sun†, Lin He*, Zhaowei Xi*, Ying Liu*
*Tsinghua University †Alibaba Group

## Abstract

Programmable data planes (DP) enable flexible customization of packet processing logic with domain-specific languages such as P4. To relieve developers from lengthy codes and tedious hardware details, many researches propose DP program generators that take high-level intents as input and automatically convert intents into DP programs. Generators must be correct, otherwise they may produce buggy programs or DP logic that is inconsistent with intents. Nevertheless, existing verification tools are designed to verify individual DP programs, not generators. They either cannot achieve high bug coverage or cannot debug generators with high scalability.

This paper presents *Firebolt*, a blackbox testing tool designed to dig out faults in DP program generators, including security vulnerabilities, intent violations, and generator crash. *Firebolt* achieves high bug coverage by using syntax-guided intent generation to construct a comprehensive, syntactically correct, and semantically valid intent set. To avoid intent explosion, *Firebolt* designs an intent space pruning approach that eliminates redundant intents while preserving representative ones. For high scalability, *Firebolt* automatically formalizes DP programs and intents for verification. We apply *Firebolt* to three popular open-source DP generators. Evaluation results demonstrate that *Firebolt* can detect 2× bugs with 0.1% to 0.01% human efforts compared to existing tools.

## 1 Introduction

Programmable network devices [1, 2] together with domain-specific programming languages (*e.g.* P4 [3]) have enabled many in-data-plane (DP) network functions, such as monitoring [4–6], security [7–9], routing [10–12] and so on. Meanwhile, booming DP functions heavily burden programmers with lengthy codes (100s to 1000s lines of code, LoC [13]) and manual consideration of tedious hardware constraints [14]. To this end, a growing body of research proposes DP generators [14–27], which provide high-level declarative primitives to easily express *intents*, and a compiler to convert intents into platform-specific DP programs and table entries. DP generators can reduce LoC by 80% [14] with resource optimizations that would otherwise require manual efforts. Above benefits encourage researchers and industries to design various DP generators for network monitoring [16, 17] and even *mission-critical* functions like routing [14] or security [21, 22].

Considering the prevalence of DP generators, guaranteeing their *correctness* becomes a must-be-solved problem. However, our study (§2) reveals that three types of mistakes including program security vulnerabilities such as out-of-bound register access, intent-program inconsistency, and generator crash, may happen to advanced DP generators [16, 17, 21], which can result in serious mistakes such as missing attacks and undesired packet processing procedure.

Unfortunately, little attention has been devoted to guaranteeing the correctness of DP generators. Existing tools focus on finding security vulnerabilities in DP programs [28–34], or verifying the consistency between high-level intents and DP programs [28–31]. However, these tools are not designed for debugging DP generators and thus fall short in two aspects: (1) *Coverage.* Existing verification tools aim to verify individual intent-program pairs instead of finding all bugs in advanced DP generators. As intents are numerous, even infinite, verification tools can hardly cover all generator faults. (2) *Scalability.* To check intent-program consistency, verification tools require massive human-written specifications of intents (100s to 1000s of LoC) for one program, which is error-prone and time-consuming.

This paper presents *Firebolt*, a blackbox testing tool designed to dig out DP generator faults including security vulnerabilities, intent violations, and crash with high coverage and scalability. The key idea of *Firebolt* is thoroughly constructing intents as test cases to achieve high coverage, and automatically producing specifications of intents with little human intervention for verification to achieve high scalability.

However, realizing such a tool is challenging in three aspects: *intent generation* that should contain every reasonable intent, *intent explosion* that results in unacceptably long testing time due to numerous intents, and *intent diversity* that hampers automatic specification derivation for verification. In response, *Firebolt* proposes the following innovative designs.

- *Intent generation. Firebolt* should generate a comprehensive intent space containing every reasonable intent for high coverage. However, random composition of intent grammar symbols can produce infinite intents, which is impractical for testing. We first generate syntactically correct intents based on intent grammar in Backus-Naur form (§4.1). We then identify semantic dependencies between grammar symbols and filter semantically valid intents (§4.2).

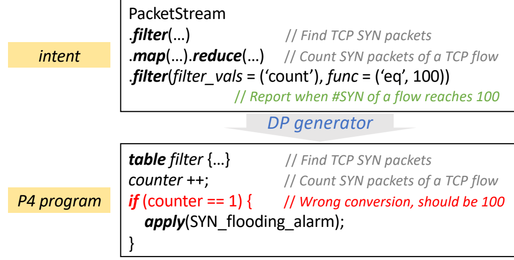



Figure 1: **False SYN flooding alarms due to intent violation.**

- *Intent explosion.* Due to wide parameter range and cyclic symbol reference, there may still exist massive or even infinite redundant intents that are syntactically correct and semantically valid, which compromises testing efficiency. To handle intent explosion, we design *intent space pruning* to eliminate redundant intents while keeping representative ones (§4.3). Remaining intents are input into generators to find crash bugs or to generate DP programs for verification.
- *Intent diversity.* The high diversity of intents and corresponding DP programs for one generator and across generators makes it challenging to devise a uniform approach for verification. To achieve high scalability, *Firebolt* first formalizes all DP programs into unified Z3 formulas [35] (§5.1). Next, instead of manually translating (1000s of) intents into specifications, we write specifications for intent grammar symbols, and automatically compose symbol specifications into the intent specifications (§5.2). Finally, we uniformly verify intent specifications and Z3 formulas to detect intent violations and security vulnerabilities (§5.3).

We apply *Firebolt* to three popular open-source DP generators, *i.e.*, Marple [16], Sonata [17], and Poise [21]. In all test cases of the three generators, *Firebolt* discovered 19 bugs including 3 security vulnerabilities, 13 intent violations, and 1 crash bug, while existing verification tools merely cover 10. Moreover, *Firebolt* requires 0.1% to 0.01% human-written LoC compared to existing tools under equal bug coverage.

## 2 Motivation

If a DP generator fails to faithfully translate programmer intents (intent violations), or produces logically flawed programs (security vulnerabilities), or even crashes under reasonable input intents, it adversely affects production efficiency and introduces instability into online DP functions. Below we present two example bugs that have been detected by *Firebolt* to reveal the consequences of faulty generators.

**#1: False SYN flooding alarm due to intent violation**. As presented in Figure 1, a SYN flooding monitoring function counts per-flow TCP SYN packet number. If a counter exceeds a threshold, a SYN flooding alarm is produced. However, due to a bug in Sonata [17], an advanced monitoring function generator, the threshold is wrongly configured as 1 instead of the originally intended 100, which results in massive false alarms that completely violates the monitoring goal.

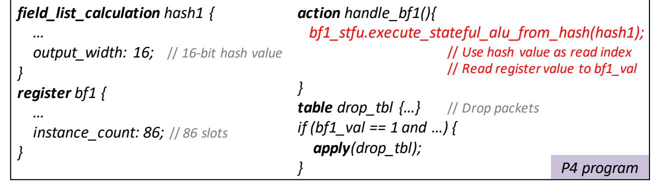



Figure 2: **Connection legality misjudgement due to security vulnerabilities**.

**#2: Connection legality misjudgement due to security vulnerabilities**. Poise [21] takes intents as input and generates context-aware security policies in programmable devices. It generates an in-DP bloom filter to track illegal connections. However, the bloom filter has 86 slots but is indexed by a 16-bit variable (0 to 65535), as shown in Figure 2. If the index exceeds 86, a random value outside the bloom filter will be returned. Such a vulnerability could incur wrong judgement of connection legality and lead to potential security leakage.

To eliminate above mistakes, programmers have to review generated DP programs and check intent-program consistency, which costs extra human efforts. Some recent tools are designed to find security vulnerabilities in DP programs [28–34] or verify the intent-program consistency [28–31]. However, using these tools to debug DP generators requires massive human efforts to verify each intent-program pair, and cannot fully cover all intents. Therefore, on modifying the intent or expressing a new intent, these tools must be repetitively executed to ensure program correctness. Unfortunately, doing so brings the scalability problem. To verify intent-program consistency, 100s to 1000s of LoC must be written manually to convert intents into specifications [16, 31]. Such LoC is comparable to the DP program, which is error-prone, time-consuming, and not scalable.

Instead of verifying individual programs, we propose *Firebolt* to debug generators. *Firebolt* thoroughly generates intents as test cases to detect generator faults, and automatically derives specifications from intents to improve scalability.

## 3 Overview

The key idea of *Firebolt* is thoroughly constructing intents as test cases to achieve high coverage, and automatically producing verification specifications to achieve high scalability. *Firebolt* workflow includes two major steps, *i.e.*, intent generation and program verification, as shown in Figure 3.

**§4 - Intent generation**. To thoroughly explore the intent space and cover all generator faults with little redundancy, *Firebolt* takes the intent grammar and semantic constraints of a DP generator as input and generates all possible and correct intents. Meanwhile, *Firebolt* losslessly prunes the generated intent space to eliminate redundancy and produces final test cases, which are then input into DP generators to generate DP programs or to find crash bugs.

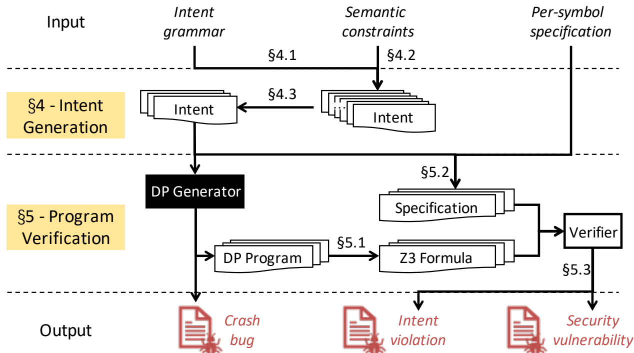


Figure 3: **Workflow of *Firebolt*.**

**§5 - Program verification**. To automatically verify the correctness of generated DP programs, the key is to derive the specification of intents for verification. *Firebolt* provides a general and high-level specification to express every intent grammar symbol, and automatically derives the specification of each intent based on per-symbol specifications. Meanwhile, *Firebolt* formalizes the output DP programs using Z3 formulas. Finally, *Firebolt* checks whether the Z3 formulas (1) are consistent with intent specifications and (2) have security vulnerabilities.

# 4 Intent Generation

The design goal of intent generation is producing a group of intents that (1) thoroughly covers all correct intents so that all generator faults can be discovered and (2) possesses little redundancy so that generator testing can be efficient. To achieve the above goals, we start by comprehensive intent generation with both syntactical correctness (§4.1) and semantic validity (§4.2) in mind. Next we analyze the source of redundant intents and propose the intent space pruning approach (§4.3).

## 4.1 Syntax-Guided Intent Generation

A DP generator must expose the intent grammar for expressing intents. For example, as shown in Figure 1, Sonata [17] provides primitives like ***filter***, ***map***, and ***reduce*** and parameters like *eq* and *count*. The intent grammar describes lawful function calls, parameter ranges, and syntax, which serves as the foundation to generate possible intents from scratch.

We refer to syntax-guided synthesis [36, 37], a common approach in program synthesis that finds the desired program by searching the program space described by a grammar. We leverage its idea and redefine the intent generation problem as: given the grammar $G$ of a DP generator, we need to explore the intent space and generate all syntactically correct intents.

**Intent grammar formalization**. Syntax-guided intent generation takes grammar as input. However, different generators can provide grammar in various formats [14, 16, 17]. We need to formalize grammars of generators into a unified expression. We observe that Backus-Naur form (BNF) [38] is the most common context-free grammar (CFG) [39] for describing the syntactic structure of programming languages. Most DP generators [14, 16, 21–23] provide a BNF syntax specification. Thus, *Firebolt* uses BNF for grammar formalization.

**Algorithm 1** Syntax-Guided Intent Generation

```
1:  function SYGUG(G)
2:      Q = Queue(G.n_r)                          ▷ Initialize
3:      while Q.size() > 0 do
4:          n = Q.front()
5:          Q.pop()
6:          if n.has_nt() then
7:              A = n.first_nt
8:              for A → β ∈ G.R[A] do
9:                  n = αAγ --(A→β)--> n_1 = αβγ   ▷ Grow graph
10:                 Q.push(n_1)
11:             end for
12:         else
13:             OUTPUT(n)              ▷ Output generated intent
14:         end if
15:     end while
16: end function
```

Note that *Firebolt* can also work with non-BNF grammars by adopting the expansion rules of these grammars during syntax-guided intent generation.

**Preliminaries of BNF**. Dark rectangle in Figure 4 shows partial BNF expression of Marple [16] intent grammar. A grammar $G$ expressed in the BNF format is a quadruple $\langle N, S, \Sigma, R\rangle$, where (1) $N$ is a finite set of non-terminal symbols that can be expanded to one or more terminal and non-terminal symbols, (2) $S$ is the start symbol in $N$, (3) $\Sigma$ is a finite set of terminal symbols that can appear in an intent, (4) $R$ is a finite subset of $N \times (N \cup \Sigma)^*$, where each member $(A, \beta) \in R$ is called an expansion rule and is written as $A \to \beta$. A sequence of non-terminal and terminal symbols in $(N \cup \Sigma)^*$ is called a *sentential form*, which represents an intermediate intent.

**Syntax-guided intent generation**. With a grammar $G$ in BNF format, a possible intent must start with $S$ and then be replaced with expansion rules in $R$ until there are no non-terminal symbols that need to be expanded, *i.e.*, it ends with a composition of terminal symbols in $\Sigma$. Therefore, generating all possible syntactically correct intents can be visioned as growing a single-rooted ($S$) graph with one or more rules in $R$, and collecting all leaf nodes ended with any one symbol in $\Sigma$.

We formalize the above process as follows. We define the intent generation graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}\rangle$ as a directed labeled graph derived from grammar $G = \langle N, S, \Sigma, R\rangle$, with the nodes $\mathcal{N} \subseteq (N \cup \Sigma)^*$ and the edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times R$. Each node in $\mathcal{N}$ has a sentential form which can be derived from the start symbol $S$. Each edge in $\mathcal{E}$ represents a non-terminal symbol expansion according to an expansion rule. At node $n_1$ whose sentential form is $\alpha A \gamma$, where $A$ is a non-terminal symbol, we can apply the expansion rule $A \to \beta \in R$ and derive a child node $n_2$ with a new sentential form $\alpha\beta\gamma$. $\mathcal{G}$ has a root node $n_r$ with sentential form $S$, and many (maybe infinite) leaf nodes.

Algorithm 1 depicts the procedure of growing graph $\mathcal{G}$ using *depth first search*. We maintain a *queue* $Q$ to store nodes in $\mathcal{G}$. First, $Q$ is initialized with the root node $n_r$ (line 2). Then, in each iteration (line 3-15), we pop the first node $n$ in $Q$. If $n$ has no non-terminal symbols in its sentential form, we output $n$ as an intent (line 13). Otherwise, we find the first non-terminal symbol $A$ in $n$, and apply all possible expansion rules of $A$ to generate child nodes (line 9), which are then appended to $Q$ for further expansion. We repeat this process until $Q$ is empty and all syntactically correct intents are generated.

## 4.2 Semantic Constraint Injection

Syntax-guided intent generation can cover syntactically correct intents. Nonetheless, some syntactically correct intents do not make sense, or, say, are semantically invalid [40]. Below we first introduce two types of semantically invalid intents. Next we identify semantic constraints between grammar symbols, and present the semantic constraint expression and injection mechanisms to filter semantically valid intents.

**Semantically invalid intents**. Besides conforming to the syntax, intents must also comply with semantic constraints. Below we identify two types of semantically invalid intents.

- *Uncompilable intents.* Syntactically correct intents cannot guarantee successful compilation. Typical examples include a variable that is not declared before it is referenced, a variable reference whose dimension is inconsistent with the declaration, or a variable that is repeatedly defined. Numerous such intents violate the semantic constraints of the intent grammar, and therefore should be ruled out.
- *Incomplete intents.* Some intents are semantically incomplete. For example, the *map* primitive in Marple [16] distributes incoming data according to certain match fields, and assigns a computing expression to process a temporary variable, which is meaningless if it is not referenced later. Therefore, an intent with a *map* primitive in the end of a query is considered incomplete and should be ruled out.

**Semantic constraint identification**. The existence of semantically invalid intents indicates that the above mentioned syntax-guided intent generation graph contains unreasonable leaf nodes (incomplete intents) or even invalid expansion rules (branches) for intermediate nodes (uncompliable intents). Thus, we should identify semantic constraints of intent grammar, and leverage the constraints to supervise the intent generation process. Overall, we classify the constraints into *exclusion constraints* and *dependency constraints*.

- *Exclusion constraints:* indicate that if an expansion rule $r_1$ on a node $n_1$ exists, the expansion rule $r_2$ on a node $n_2$ is not valid. We formally express them as:
$$if\ \exists\ r_1\ on\ n_1,\ then\ \nexists\ r_2\ on\ n_2$$
A typical example is that one variable name (such as the name of a packet stream) cannot be defined repeatedly.
- *Dependency constraints:* indicate that only if an expansion rule $r_1$ on a node $n_1$ exists, the expansion rule $r_2$ on a node $n_2$ is valid. We formally express them as:
$$if\ \exists\ r_1\ on\ n_1,\ then\ \exists\ r_2\ on\ n_2$$
A typical example is that a ***Map*** primitive must operate on a stream that has been previously defined.

```
Marple.bnf
⟨prog⟩        ::=  ⟨aggFun⟩*⟨streamStmt⟩+
⟨streamStmt⟩  ::=  ⟨streamName⟩ = ⟨streamQuery⟩
⟨streamName⟩  ::=  R⟨number⟩
⟨streamQuery⟩ ::=  ⟨map⟩ | ...
⟨map⟩         ::=  map (⟨streamName⟩, [⟨mapCol⟩], [⟨mapExpr⟩])

# Example 1: StreamName cannot be defined repeatedly
if ∃ ⟨r1⟩ on ⟨n1⟩, ∄ ⟨r2⟩ on ⟨n2⟩, ⟨n2⟩ ↔ ⟨n1⟩
⟨n1⟩ :   *, ⟨prog⟩⟨streamStmt⟩⟨streamName⟩, *
⟨r1⟩ :   ⟨streamName⟩ →*
⟨n2⟩ :   *, ⟨prog⟩⟨streamStmt⟩⟨streamName⟩, *
⟨r2⟩ :   ⟨r1⟩

# Example 2: Map query operates on a stream that has been defined
if ∃ ⟨r1⟩ on ⟨n1⟩ , ∃ ⟨r2⟩ on ⟨n2⟩ , ⟨n2⟩ → ⟨n1⟩
⟨n1⟩ :   *, ⟨prog⟩⟨streamStmt⟩⟨streamQuery⟩⟨map⟩⟨streamName⟩, *
⟨r1⟩ :   ⟨streamName⟩ ↛ T
⟨n2⟩ :   *, ⟨prog⟩⟨streamStmt⟩⟨streamName⟩, *
⟨r2⟩ :   ⟨r1⟩
⟨n1⟩. ⟨streamStmt⟩ ≠ ⟨n2⟩. ⟨streamStmt⟩
```

Figure 4: **Semantic constraint examples in Marple [16].**

With the above classification in mind, we thoroughly analyze the intent grammar and derive semantic constraints. Missing constraints can result in some semantically invalid intents left as test cases and slightly compromise the testing efficiency, which is considered acceptable. However, wrongly-written constraints will incur wrong deletion of semantically valid intents and impair generator fault coverage. Therefore, we must guarantee the correctness of the constraints. We have investigated several advanced generators [16, 17, 21] and observe that each of them corresponds to <20 semantic constraints, which is acceptable for manual inspection.

**Semantic constraint expression and injection**. Identified semantic constraints should be uniformly encoded for injection. To clearly express a constraint, we need to clearly specify the constraint type ($\exists$ or $\nexists$) and elements ($r_1$, $n_1$, $r_2$, and $n_2$). Next we showcase two semantic constraints in Marple [16].

Example 1 shown in Figure 4 presents an exclusion constraint. A Marple program $\langle prog \rangle$ includes multiple streams $\langle streamStmt \rangle$, each with name $\langle streamName \rangle$. The names of streams should not be defined repeatedly. $n_1$ and $n_2$ describe the node where stream names are defined. Naturally, there exists an expansion trace from the start symbol $\langle prog \rangle$ to the current symbol $\langle streamName \rangle$. We use $\leftrightarrow$ to indicate that the two nodes can appear in any order on the path. We use $*$ to match any expansion traces in $n$ and any expansion rules in $r$. By making $r_2 = r_1$, the constraint prevents $\langle streamName \rangle$ from taking duplicate values of $r_1$ in any other node $n_2$.

Example 2 in Figure 4 presents a dependency constraint. If a $\langle map \rangle$ query in Marple operates on a stream with a name other than $T$ (the name of the original input stream in Marple),

this stream should be previously defined. We use $n_2 \to n_1$ to constrain the ordering between two nodes, *i.e.*, $n_2$ should be an ancestor node of $n_1$. By making $r_2 = r_1$, the constraint guarantees that a stream is defined before referenced.

We present all identified semantic constraints of Marple [16], Sonata [17], and Poise [21] in Appendix A. During intent generation, an expansion rule $r$ on node $n$ is rejected if it violates exclusion constraints, and a leaf node is rejected if not all dependency constraints are satisfied on its path.

## 4.3 Intent Space Pruning

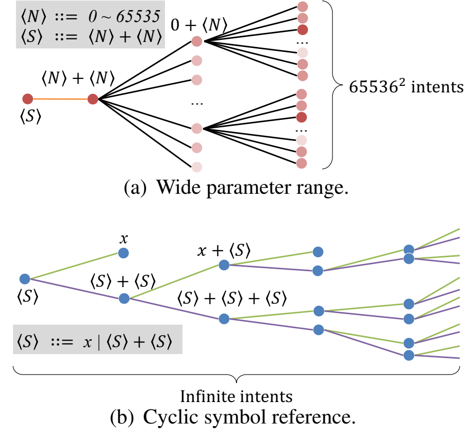
Despite we guarantee the syntactical correctness and semantic validity of intents, the massive expansion rules and their combinations may still build an extremely large or even infinite intent space, due to two reasons, as illustrated in Figure 5.

- *Wide parameter range.* A non-terminal symbol may have many possible expansion rules, which corresponds to a large node degree. For example, a 16-bit integer has 65536 possible values. Worse still, if a sentential form has multiple such non-terminal symbols, the exponential combination can lead to an explosion of the intent space.
- *Cyclic symbol reference.* A non-terminal symbol may return to itself after expansion, *i.e.*, the cyclic symbol reference, which corresponds to an infinite depth of the intent derivation graph. For example, an arithmetic expression has an expansion rule of $\langle S \rangle ::= \langle S \rangle + \langle S \rangle$. The circular expansion leads to an infinite number of possible intents.

For all generated intents of a DP generator, we observe that most intents would not cause any bugs, while many could cause the same bug. To strike a balance between coverage and efficiency, we propose the following two mechanisms to prune the intent space without loosing intent representativeness.

**Method #1: Intra-symbol representativeness**. To handle wide parameter range, we propose to *keep representative expansion rules*, which include three categories.

- *Boundary rules.* Boundary values in numbers, including minimum and maximum values (*e.g.*, 0 and 65535 for a 16-bit parameter), usually represent some extreme cases or conditions, and should be included in the test cases.
- *Random rules.* In addition to boundary values, we should take random values from values other than boundary values (*e.g.* one value from 1 to 65534 for a 16-bit parameter).
- *Previously selected rules.* When the same non-terminal symbol is expanded multiple times in a sentential form, the choices of expansion rules are actually correlated. In this case, the previously selected random rules should also be included in the latter non-terminal symbol expansions. For example, Marple [16] uses the query name to identify a query. Suppose a former expansion rule defines the name of a query $Q$, and a latter expansion rule references the name of a query (maybe query $Q$, maybe not), value $Q$ should be included in the latter rule to keep representativeness.




(a) Wide parameter range.

(b) Cyclic symbol reference.

Figure 5: **Two types of intent space explosion.**

**Method #2: Inter-symbol combination representativeness**. For cyclic symbol reference, we can think of an expansion circle as a non-terminal symbol returning to itself with zero to many intermediate non-terminal symbols. To handle cyclic symbol reference induced intent explosion, we should break infinite symbol recurrence without losing representativeness.

To this end, we refer to the combinatorial testing (CT) theory [41] for software testing in the software engineering field. Provided that a software is composed of multiple features, the CT theory indicates that a minimal set of test cases for the software should include *individual* features and combinations of *two* distinct features, which are enough to effectively test the software and find most bugs. If a test case containing three or more features causes a software bug, the root cause may still lie in the interaction of two features among them.

Inspired by the CT theory, to effectively test the DP generator with high efficiency, we can first extract all distinct features, *i.e.*, combination of non-terminal symbols, according to the intent grammar. Then, we prune the intent space and only keep sentential forms that are either (1) individual features, or (2) possible combinations of $n$ distinct features, where $n = 2$. Our evaluation results in §6.2 reveal that using a higher combination factor (e.g. $n = 3$) cannot find more bugs in the DP generator, which proves the effectiveness of applying the CT theory for DP generator testing.

Finally, we introduce how we extract distinct features, *i.e.*, combinations of non-terminal symbols. Recall that the CT theory limits the recurrence of the same feature to two times. Therefore, each feature should only include distinct symbols. Suppose an intent grammar has $k$ non-terminal symbols. By picking a random number (1 to $k$) of distinct symbols and organizing them in all possible sequences, we can generate $N = \sum_{i=1}^{k} \mathrm{A}_k^i$ features where A stands for the permutation symbol, *i.e.*, $\mathrm{A}_n^m = n!/(n-m)!$. With the combination factor $n = 2$, there exist at most $(N + \mathrm{A}_N^2)$ sentential forms composed of non-terminal symbols. Suppose there are $s$ terminal symbols and $p$ parameter value options, we produce at most $(N + \mathrm{A}_N^2) \times s \times p$ test cases, which are *finite* and feasible for testing. Our evaluation results in §6.2 show that using the

above pruning methods, *Firebolt* will generate <10K intents for testing three advanced DP generators [16, 17, 21].

So far, we have thoroughly explored the intent space to generate syntactically correct and semantically valid intents with little redundancy. We feed these reasonable intents into the DP generator to find crash bugs or to generate DP programs for verification, which we will introduce in the next section.

## 5 Program Verification

In this section, we introduce how *Firebolt* verifies the correctness of the generated DP programs from two aspects, *i.e.*, whether there are potential security vulnerabilities, and whether the DP programs are consistent with corresponding intents. We use Z3 [35], a Satisfiability Modulo Theories (SMT) solver, which can take (1) Z3 *formulas* and (2) Z3 *assertions* as input, and formally verify whether the formulas satisfy the assertions. In the rest of this section, we first introduce how we automatically formalize the generated DP programs as Z3 formulas (§5.1). Next, to avoid manually converting 1000s of intents into Z3 assertions, we provide a general and flexible specification to express every symbol of the intent grammar, and automatically compose symbol specifications into intent specifications, which will then be converted into Z3 assertions (§5.2). Finally, we check intent-program consistency and detect security vulnerabilities (§5.3).

### 5.1 DP Program Formalization

We use the popular $P4_{16}$ language as an example to illustrate how to formalize DP programs into Z3 formulas. $P4_{14}$ programs can be first converted into $P4_{16}$ programs using open-source P4 compiler suite [13] and then formalized into Z3 formulas by *Firebolt*. Gauntlet [42] has proposed approaches to convert partial P4 programs into Z3 formulas, but does not cover the formalization of P4 *table entries* and *externs*, which are essential to faithfully convert P4 programs. Our formalization solution is built atop Gauntlet. Below we first introduce the idea and capability of Gauntlet, and then introduce how we formalize externs and table entries.

**Formalizing each programmable block**. A $P4_{16}$ program is composed of several programmable blocks (*e.g.*, packet parser, ingress control flow, egress control flow, packet deparser, etc.). We provide an example of a P4 *match-action table* residing in the control flow block in the head of Figure 6. For each block, Gauntlet performs the following conversion.

- *Input parameter → free Z3 variable*. Two special types of Z3 variables, *i.e.*, Z3_INVALID and Z3_UNDEFINED, are defined to represent invalid and undefined parameter values.
- *Function → Z3 operation* that refers to input Z3 variables. For example, a table lookup function is converted into a reference to the resulting action index. Operations like parameter initialization or invalidation can refer to Z3_INVALID and Z3_UNDEFINED special variables.

```
Part of P4_16 Program: Match-Action Table
action a0 (z) {y = z;}
table t0 {
  key = k0 : exact;
  actions = {
    no_op;
    a0;
  }
  default_action = no_op;
}
```

```
Table entries of t0:
1 => a0(1)
2 => a0(2)
```

```
Input free Z3 variables:
(_ BitVec 32) k0               // Match key of table t0
(_ BitVec 32) t0_index         // Action index of table t0
(_ BitVec 32) a0_z             // Parameter of action a0
Output Z3 expressions (with table entries):
(_ BitVec 32) y = (ite (= k0 1) 1 (ite (= k0 2) 2
                    Z3_UNDEFINED))
Output Z3 expressions (without table entries):
(_ BitVec 32) y = (ite (= t0_index 1) a0_z Z3_UNDEFINED)
```

Figure 6: **Examples of formalizing match-action tables.**

- *Output parameter → output Z3 expression* that is the result of executing Z3 operations on the input Z3 variables.

**Formalizing table entries**. Gauntlet assumes that the contents of the table are unknown, and does not include the configuration of table entries in the output Z3 expressions. However, generating correct table entries is also critical for the DP generator, as table entries also reflect intents. For instance, a ***filter***(ip.src=192.168.1.1) intent segment in Sonata indicates that subsequent operations will operate on special flows, which corresponds to a table entry in generated DP programs.

To formalize table entries, we use a nested if-then-else statement to imitate a match-action table call, as shown in Figure 6. When table entries are provided, for each parameter modified by the table (parameter $y$ in this example), we use an if-then branch to express the modification in Z3 expressions as follows: if the key of incoming data matches a specific table entry ($k0 = 1$ or $k0 = 2$), the corresponding action is executed ($y = 1$ or $y = 2$). If the key does not match any entry, the default action is executed ($y$ is not assigned an initial value, and is therefore undefined). When no table entry is specified, we assume that all actions in the table are executable. We use a free Z3 variable ($t0_index$) to indicate the index of the action to be executed, and a separate free Z3 variable for each action parameter ($a0_z$ for parameter $z$). A table call can then be represented by a nested if-then-else statement with each branch representing the execution of one action.

**Formalizing externs**. P4 programs often operate on extern objects such as stateful memory and hash calculations. Gauntlet interprets externs as a function call that returns an arbitrary value. However, an accurate translation of externs is critical, since externs can maintain program internal states and may be modified and referenced. For example, the SYN flooding alarm program shown in Figure 1 maintains a counter that will later be compared to a threshold, which should be embedded in the Z3 formula. Below we introduce our approaches to handle stateful memory and hash calculation, respectively.

```
Part of P4_16 Program: Register Reading and Writing
register<bit<32>>(32w1024) reg;
reg.read(r_index, r_value);        // r_value = reg[r_index]
reg.write(w_index, w_value);       // reg[w_index] = w_value
Input free Z3 variables:
(_ BitVec 32) reg_read_value
(_ BitVec 32) r_index, w_index, w_value
Output Z3 expressions:
(_ BitVec 32) reg_instance_count = 32w1024
(_ BitVec 10) reg_write_index = w_index
(_ BitVec 32) reg_write_value = w_value
(_ BitVec 10) reg_read_index = r_index
(_ BitVec 32) r_value = reg_read_value
```

(a) Register reads and writes

```
Part of P4_16 Program: Hash Calculation
// Update hash_table_index with hash value
hash(hash_table_index, HashAlg.crc32, 32w0, inKey, 32w1024);
Input free Z3 variables:
(_ BitVec 32) crc32_hash_value     // Hash value
(_ BitVec 96) inKey                // Hash key
Output Z3 expressions:
(_ BitVec 32) hash_table_index = crc32_hash_value
(_ BitVec 32) crc32_hash_width = 32w0
(_ BitVec 128) crc32_hash_field = inKey
(_ BitVec 32) crc32_hash_size = 32w1024
```

(b) Hash calculation

Figure 7: **Examples of converting externs into Z3 formulas.**

- *Stateful memory.* We take register, an indexed array of stateful cells, as an example to illustrate our approach, which also applies to other types of stateful memory such as counters. A naive method to formalize a register is to define a free Z3 variable to represent the initial value and generate an output Z3 expression to represent the new value for each cell in the register. In this way, register reading can be converted into referencing the Z3 variable, and register writing can be converted into updating the Z3 expression. Then formalizing a register array with $n$ instances requires $2*n$ Z3 variables and expressions. Furthermore, we notice that for most commercial switches, a register array can be read/written only once in the switch pipeline. Thus, we only need to maintain the index and values for at most two register cells, as shown in Figure 7(a). Besides, we use another Z3 variable to store the size of the register, which can be used to detect out-of-bound register access in §5.3.

- *Hash calculation.* Hash is used to map large data to fixed-size values. We define a free Z3 variable to represent the computed hash value. Meanwhile, we store the parameters that impact the hash value, *e.g.*, the hash key and hash size, in the output expressions, as shown in Figure 7(b). Then, we can flexibly adjust the effect of the hash mapping by imposing the mapping relationship between the hash value and hash parameters when checking the Z3 expressions. For example, a conflict-free hash implies the one-to-one mapping between the hash key and hash value. In this way, we can avoid the complex hardware-specific hash calculation while maintaining the properties of hash operations.

## 5.2 Intent Formalization

The intent generation process can produce thousands of (§6) intents for one DP generator. Manually converting intents that are composed of different symbols of the same generator, or intents belonging to different generators, is time-consuming and not scalable. We observe that intents are generated by expanding non-terminal symbols. Therefore, instead of converting each intent, our key idea is first writing the specifications of each symbol in the grammar, and then automatically composing symbol specifications into intent specifications, which will finally be converted into Z3 assertions.

**Symbol specification**. To uniformly express highly-diversified intent grammar symbols across generators, we need to design an expression format that should be *general* enough to specify various symbols, and *flexible* enough for composition. The reason why we do not directly use Z3 assertions as the specification is that Z3 assertions are logical expressions that are low-level and counter-intuitive.

We propose to uniformly express each symbol as a high-level *function* written in python-like expressions. The function specification satisfies above requirements. It is general enough to specify the format and semantics of input, logic, and return values of individual symbols, and flexible enough for composition by sequentially performing function calls and correlating input and output of different functions.

Specifically, we regulate that one function consists of two segments, a declaration function **DECL_FUNC** that defines internal states of symbols, and an execution function **EXEC_FUNC** that describes the processing logic of input parameters, internal states, and output values. Stateless symbols such as a flow ***filter*** maintain no internal states and therefore can be expressed with only **EXEC_FUNC**. A typical execution function often takes network packets as input, and starts by PARSE-ing input packets into a series of header fields, *e.g.*, Ethernet → IPv4 → TCP. For each packet, we also generate standard metadata fields regulated by the P4 grammar, such as the output port. Next, symbol logic can operate on packet headers, metadata fields, and symbol internal states by packet modification, counting, forwarding, or other actions, and finally provide return values or return directly.

Figure 8 takes the $\langle groupby \rangle$ symbol in Marple [16] as an example to illustrate how to construct a stateful symbol specification. An entire $\langle groupby \rangle$ symbol is formatted as $\langle groupby \rangle ::= groupby(\langle stream \rangle, \langle columns \rangle, \langle aggFunc \rangle)$, which groups *streams* according to specific *columns* with the aggregation function $aggFunc$. In the declaration function DECL_FUNC, a key-value storage is declared for each possible option $\langle var \rangle$ in the child symbol $\langle aggFunc \rangle$. The EXEC_FUNC updates the internal states with no packet parsing, modification, or forwarding. First, we initialize a variable $tuple$ with the tuple contained in the input stream whose name is $\langle streamName \rangle$. Then we get the aggregation key from the aggregation field in $\langle columns \rangle$. Using the aggregation key,

```
# ⟨groupby⟩    ::=    groupby (⟨streamName⟩, ⟨columns⟩, ⟨aggFunc⟩)
# ⟨aggFunc⟩    ::=    def ⟨aggFun⟩ (⟨vars⟩, ⟨columns⟩): ⟨codeBlock⟩
DECL_FUNC() =
  states = []
  for var in <aggFunc>.<vars>.exec():
    KEY_VALUE_STORAGE REG_NAME_var
    states.append(REG_NAME_var)
EXEC_FUNC(stream_list) =
  tuple = <streamName>.exec(stream_list)
  key = tuple[<columns>.exec()]
  old_state = [reg[key] for reg in states]
  new_state = <aggFunc>.exec(old_state, tuple)
  states.update(key, new_state)
  tuple.append(new_state)
  return tuple
```

Figure 8: **An example for constructing stateful specifications of non-terminal symbols: $\langle groupby \rangle$ in Marple [16].**

we read the old states and execute the aggregation function $\langle aggFunc \rangle$. The output values of the aggregation function are used to update states of $\langle groupby \rangle$. Also, the output states are included in *tuple* for future usage in subsequent symbols.

**Symbol specification composition**. For each generated intent, *Firebolt* constructs its specification based on its generation path, *i.e.*, the non-terminal symbol expansion process and the final terminal symbols. Starting from the semantics of the start symbol, *Firebolt* recursively appends the semantics of child symbols in the expansion rules by sequentially connecting their FUNCs to automatically construct the final specification.

**Specification conversion into Z3 assertions**. A Z3 assertion is a series of algebra expressions connected with logical operators, *e.g.* $(x > 10)\&\&(y < 5)$. We use a special type of Z3 assertions, *i.e.*, $implies(f, g)$, to verify DP programs. *implies* is an implication operator, which assumes a condition expression $f$ on the input Z3 variables and asserts that the output satisfies the implication expression $g$. After expressing intents as a series of functions, we can identify how each parameter (*e.g.*, header fields, forwarding port, and internal states) is modified by the functions. Therefore, we develop a tool that can automatically convert the functions into the *implies* Z3 assertions that will be used to verify the DP programs.

## 5.3 Program Correctness Verification

With the Z3 formulas representing semantics of DP programs and intents formalized into Z3 assertions, we first check whether each P4 program is consistent with the corresponding intent. Then, we summarize security vulnerabilities from the literature, and introduce how to detect them in each program.

**Intent-program consistency**. To verify consistency, we take the Z3 formulas as input and use the Z3 constraint solver [35] to verify the Z3 assertions, *i.e.*, a set of $implies(f, g)$. Specifically, we need to check the following three types of consistency, each with a different set of $f$ and $g$.

- *Packet parsing consistency.* It indicates that parsed headers and header fields are ordered consistently with the original intent, and each header field is parsed correctly. During DP program formalization, when formalizing the parser block, *Firebolt* treats the entire input packet header as a free Z3 variable, and outputs individual Z3 expressions to represent different header fields. We verify the parsing consistency by asserting that for each header field in the parsing part of the intent specification, (1) there is a corresponding output Z3 expression, (2) a failed parsing implies an invalid value. For example, the first 16 bits of the Ethernet header should be 0x800. The parsing of Ethernet fails if an Ethernet header does not start with 0x800, and (3) a successful parsing implies a correct parsed value. For example, given the condition that the first 16 bits of the header are 0x800, we should be able to obtain the Ethernet.dstAddr field by extracting specific bits from the input header variable.
- *Packet deparsing consistency.* It refers to the correct order and values of headers and header fields in the output packet. The checking of the deparser block is similar to the parser block. We omit details here for brevity.
- *Packet processing consistency.* Packet processing includes packet modification, forwarding and state updates, and corresponds to the behavior of several programmable blocks of a P4 program, *e.g.*, both the ingress control flow and the egress control flow. To construct a complete Z3 formula, we first concatenate the Z3 formulas of individual related programmable blocks into a complete block. For each output Z3 expression of a block, if it is an input Z3 free variable of latter blocks, we replace the corresponding input with the current output, and recompute the output expressions of latter blocks. We iterate this process until the output of a block is no longer referenced by any blocks, which becomes the final output of the DP program and completes the Z3 formula for packet processing. Then, for each modified packet field and metadata in the Z3 formula, we extract related operators and construct an individual Z3 expression, which can be checked against corresponding intent specifications.

**Security vulnerabilities**. Security vulnerabilities are intrinsic flaws of DP programs without corresponding intents. For example, out-of-bound register access may cause unexpected behaviors and even online risks, and is never intended. We observe that security vulnerabilities can be converted into special Z3 assertions and verified against DP programs as introduced above. Therefore, we summarize security vulnerabilities highlighted by previous literature [28–32, 34], and introduce how to express them as Z3 assertions.

- *Invalid header access* may occur when the validity of a header is not checked before referencing it. To detect this bug, for each output Z3 expression, we assert that (1) each referenced header (Z3_h) belongs to a branch in an if-then-else statement, and (2) the if-condition in this branch includes a validity check (*i.e.*, Z3_h != Z3_INVALID).
- *Implicit packet drops* occur when the egress_port is not specified. To detect this bug, in the output Z3 expression

Table 1: **(1) Bugs detected by *Firebolt*, and (2) efficiency of *Firebolt* when debugging the three DP generators.**

| DP Generator Under Test | # Generated Intent | # Detected Bugs / # Intents Causing Bugs | | | Human-written LoC | | | Test-Case Size (Min / Max) | | | Running Time (Total / Average) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Crash Bug | Security Vulnerability | Intent Violation | Intent Grammar | Semantic Constraints | Per-Symbol Specification | Intent LoC | P4 Program LoC | # Table Entries | Intent Generation | Program Verification |
| **Marple** | 7341 | 1 / 12 | 1 / 7329 | 2 / 23 | 93 | 70 | 323 | 1 / 32 | 211 / 481 | 0 / 0 | 168s / 23ms | 1204s / 164ms |
| **Sonata** | 7912 | 0 / 0 | 2 / 7912 | 5 / 243 | 34 | 10 | 178 | 1 / 19 | 253 / 375 | 7 / 43 | 27s / 3ms | 926s / 162ms |
| **Poise** | 2362 | 0 / 0 | 2 / 2362 | 6 / 362 | 25 | 25 | 132 | 1 / 12 | 704 / 893 | 1 / 12 | 23s / 10ms | 355s / 150ms |

of egress_port, we assert that no branch results in the undefined special variable, *i.e.*, Z3_UNDEFINED.

- *Out-of-bound register access* occurs when the read/write index exceeds the register array size. Since we use separate output Z3 expressions to record the read/write index and the array size in §5.1, we can detect this bug by comparing the array size and the index range. For direct access, where *index* is assigned an exact value, we assert that $index < size$. For non-direct access, *i.e.*, the *index* expression includes symbolic variables, such as hash values, we assert that the range of the symbolic variables is within the allowed range.
- *Decapsulation errors* happen when invalid headers are deparsed in the deparser. To detect this bug, we assert that for each output header expression, no branch results in the invalid special variable (*i.e.*, Z3_INVALID).
- *Forbidden writes* happen when a P4 program tries to write certain metadata values which are read-only, but the P4 compiler allows the program to write them. To detect this bug, we assert that for all read-only metadata fields, the values remain unchanged, *i.e.*, the output values are the original undefined values (*i.e.*, Z3_UNDEFINED).

# 6 Evaluation

We implement *Firebolt* with ∼1200 lines of Python code for intent generation, and ∼800 lines of C++ code for program verification. Our verifier is built atop Z3 [35], and can verify both $P4_{16}$ and $P4_{14}$ programs with the aid of P4 compiler suite [13]. All experiments were conducted in a Ubuntu 16.04 virtual machine with 4GB RAM and two 2.3GHz CPU cores.

We use *Firebolt* to test three popular open-source DP generators, including two for network telemetry, *i.e.*, Marple [16, 43] and Sonata [17, 44], and one for security policy enforcement, *i.e.*, Poise [21, 45]. Marple and Sonata both use sequential composition of data flow operators to construct telemetry queries, while Marple is more complicated by supporting self-defined variable names in each query and dependencies between queries. Poise is relatively simpler and enforces security policies by filtering customized packet header fields. Finally, we implement two advanced DP program verification tools (*i.e.*, Aquila [31] and p4v [30]) for comparison.

Our evaluation intends to answer the following questions.

- *Bug coverage.* We first discuss all discovered generator bugs by *Firebolt*. (§6.1) Next, we prove the bug coverage of *Firebolt* by (1) showing the intent representativeness of *Firebolt*, and (2) comparing the number of bugs discovered by *Firebolt* and existing verification tools over open-source intents and programs of the generators. (§6.2)
- *Efficiency.* We introduce (1) the human efforts required by *Firebolt* to debug the generators, (2) the size of intents, P4 programs, and table entries that are generated and verified by *Firebolt*, and (3) the running time of intent generation and program verification. (§6.3)
- *Scalability.* We first compare the human efforts, *i.e.*, lines of hand-written codes, required by *Firebolt* and existing verification tools to debug the three generators. Then we evaluate the time required by *Firebolt* when verifying larger programs and more table entries. (§6.4)

## 6.1 Bug Analysis

As shown in Table 1, we find that all three generators have bugs, and discover 5 security vulnerabilities, 13 intent violations, and 1 crash bug in total. Below we introduce the detected bugs. To the best of knowledge, this is the first effort that comprehensively analyze and reason DP generator faults.

**Security vulnerability**. *Firebolt* finds security vulnerabilities in all generated programs of all three DP generators.

- *Invalid header access* is a common bug. All generated programs refer to some headers without checking validity.
- *Out-of-bound register* access is found in Poise, which may use a hash value that exceeds the size of the the register as the read/write index for the register.
- *Implicit drops* happen in Sonata. Generated programs never explicitly specify the egress port of input packets.

**Intent violation**. Due to the high intent diversity, intent violations are the most insidious bugs that cannot be easily detected by the developers of DP generators. Next we introduce intent violations in the three DP generators, respectively.

For Sonata, *Firebolt* finds 5 types of intent violation bugs in 243 generated programs out of 7912 programs in total.

- *Bug #1: Incorrect query combination.* When a *filter* query with the *eq* (=) function follows a *reduce* query, Sonata converts it into comparing the result of *reduce* and the default value of 1, regardless of the true value in the filter query. This bug may lead to false attack alarm (§2).
- *Bug #2, #3, #4: Missing/Incomplete table entries.* Sonata designs a *filter(k,v,f)* symbol to filter packets whose key ($k$) fields satisfy function ($f$) of value ($v$) fields. To implement a *filter* query in DP programs, both match-action tables and table entries should be generated. However, there are three

Table 2: **The number of detected bugs and generated intents with different intent space pruning methods.**

| Pruning Method | | # Detected Bugs | | | # Generated Intents | | |
|---|---|---|---|---|---|---|---|
| | | Marple | Sonata | Poise | Marple | Sonata | Poise |
| Intra-Symbol | None | ❶ ⓵ | ❶❷ ⓶ | ❶ | > 50K | > 50K | > 50K |
| | *r=1* | ①❶⓵⓶ | ❶❷⓵⓶⓷⓸⓹ | ❶❷⓵⓶⓷⓸⓹⓺ | 7341 | 7912 | 2362 |
| | *r=2* | ①❶⓵⓶ | ❶❷⓵⓶⓷⓸⓹ | ❶❷⓵⓶⓷⓸⓹⓺ | 14812 | 12384 | 3804 |
| Inter-Symbol | None | ❶ | ❶❷ ⓶⓷ | ❶❷⓵⓶⓷⓸⓹⓺ | > 50K | > 50K | 2362 |
| | *n=1* | ①❶⓵⓶ | ❶❷ ⓶⓷⓸⓹ | ❶❷⓵⓶⓷⓸⓹⓺ | 2346 | 3523 | 2362 |
| | *n=2* | ①❶⓵⓶ | ❶❷⓵⓶⓷⓸⓹ | ❶❷⓵⓶⓷⓸⓹⓺ | 7341 | 7912 | 2362 |
| | *n=3* | ①❶⓵⓶ | ❶❷⓵⓶⓷⓸⓹ | ❶❷⓵⓶⓷⓸⓹⓺ | > 50K | > 50K | 2362 |

○ Crash Bug ● Security Vulnerability ● Intent Violation

cases where table entries can be missing or incomplete. First, when a *filter* query operates on a variable, *e.g.*, a counter, table entries are forgotten. Second, when a *filter* query has a *mask* function, Sonata translates it into an LPM table, but forgets to include the prefix length in table entries. Third, when a *filter* query has a *geq* ($\geq$) function and does not follow a *reduce* query, no table entries are generated.

- *Bug #5: Incorrect mask translation.* Sonata uses bit-wise AND for *mask* operations in a *map* query. It translates the mask $m$ into 0xFF...F (F occurs $m/4$ times). When the mask length is not a multiple of 4, the translation is incorrect.

For Poise, *Firebolt* finds 6 types of intent violation bugs in 362 generated programs out of 2362 programs in total.

- *Bug #1: Incorrect list comparer.* Poise provides list comparer (*in* and *notin*) to check whether a value is in a list. However, *notin* is wrongly equated with *in* when translated.
- *Bug #2, #3, #4: Incorrect comparison operator.* Poise translates the comparison operators ($>$ and $<$) into a match-action table with *range* match and a table entry representing the comparison range. However, this range incorrectly includes the boundary value that should be excluded. That is, $>$ and $<$ are translated into $\geq$ and $\leq$. Besides, Poise sets a default range (0$\sim$10000) for comparisons without considering the real range of variables.
- *#5: Missing table entries.* Poise provides the *monitor* expression *count(p)* that counts the number of packets satisfying a predicate *p*. However, table entries are not generated. Thus, no packets would satisfy the predicate and be counted.
- *#6: Missing action parameters.* Poise uses registers to maintain states. However, for some registers, the read/write actions do not specify the index to read or write.

For Marple, *Firebolt* finds 2 types of intent violations in 23 generated programs out of 7329 generated programs in total.

- *Bug #1: Incorrect infinity translation.* Marple uses *infinity* to represent a variable that exceeds its pre-defined upper limit. However, it assigns a fixed value $2^{31}-1$ to *infinity* without considering the actual upper bound.
- *Bug #2: Incorrect key storage*. Marple uses 32-bit registers to store keys in the *groupby* query. When storing a value with a width greater than 32 bits, *e.g.*, the ingress timestamp, the stored value would be the truncation of the value.

Table 3: **Detected bugs by existing verification tools.**

| DP Generator Under Test | # Intents | # Detected Bugs | | |
|---|---|---|---|---|
| | | Crash Bug | Security Vulnerability | Intent Violation |
| Marple | 14 | 0 | 1 (❶) | 0 |
| Sonata | 13 | 0 | 2 (❶❷) | 1 (⓶) |
| Poise | 7 | 0 | 2 (❶❷) | 4 (⓶⓷⓸⓺) |

**Crash bug**. *Firebolt* finds one crash bug in Marple, while Sonata and Poise do not report any crash bugs. Marple converts the division expression $(a/2^b)$ into a right-shift operation $(a >> b)$, but sets the maximum shift width to a fixed value of 8. According to the code, the generator would crash when the exponent $b$ satisfies a legal value of $8 < b < log_2 a$.

## 6.2 Bug Coverage

To evaluate the bug coverage of *Firebolt*, first, we compare the bugs detected using different intent space pruning methods to demonstrate that the intent generation approach of *Firebolt* is able to thoroughly cover the intent space to find bugs. Then, we compare the bugs detected between *Firebolt* and existing verification tools to demonstrate that the automatic testing of *Firebolt* can detect more bugs, compared to verifying handwritten test cases using existing tools.

**Intent representativeness**. We examine whether our two intent space pruning methods (§4.3) compromise intent representativeness by checking the bug coverage of generated intents. For each type of pruning method, we configure the extent of the other method as the default value (the number of random rules $r = 1$ for intra-symbol and the combination factor $n = 2$ for inter-symbol), vary the pruning extent of the current method, and check the resulting bug coverage. As there exist infinite possible intents, obtaining all intents is impractical. We randomly generate 50K intents ($> 10\times$ intents generated by *Firebolt*) as the baseline.

We present the results in Table 2. The *None* row represents the baseline, where only 6 bugs are discovered (19 by *Firebolt*). This is because a limited number of intents (50K) represent a very small fraction of the entire intent space. Intra-symbol pruning can greatly reduce the intent space, but increasing the number of random rules ($r$) from 1 to 2 does not increase the bug coverage. For inter-symbol pruning, we can see that a small combination factor ($n = 1$) can find many bugs, but misses one interactive bug, *i.e.*, Sonata's *reduce-then-filter* bug, which can be found when $n = 2$. Further increasing the combination factor ($n = 3$) cannot find more bugs, but greatly increases generated intents from O(1K) to above 50$K$.

Therefore, compared to random intent generation, *Firebolt* intent pruning can maintain representativeness with much fewer test cases. Moreover, the recommended pruning extent is $r = 1$ for intra-symbol pruning and $n = 2$ for inter-symbol pruning, which is adequate for comprehensive bug detection.

**Comparison with existing tools**. We compare the bug coverage of *Firebolt* with that of existing verification tools [30,31]. For existing verification tools, we collect all open-source

manually-written intents [43–45] as input to find as many bugs as possible. For each collected intent, we manually write the corresponding specification for each verification tool and perform program verification. The results are shown in Table 3. With a limited number of O(10) hand-written intents, the existing verification tools can only discover 10 bugs out of 19 by *Firebolt*, and cannot find other bugs undetected by *Firebolt*. This highlights the high bug coverage of *Firebolt* over existing tools. By delving into these open-source intents, we find that the developers of DP generator did make an effort to write different examples, but the hand-written test cases struggle to efficiently find all bugs in DP generators.

To find more bugs with existing tools, we can use the (1000s of) intents generated by *Firebolt* as input for existing tools. However, doing so requires manually writing specifications for 1000s of intents, which is time-consuming and error-prone. We will analyze the scalability issues in §6.3.

## 6.3 Efficiency

Next, we evaluate the debugging efficiency of *Firebolt* by counting the lines of input human-written codes (Figure 3), the lines of intents that *Firebolt* generates, the size of generated test cases (including intents, P4 programs, and table entries), and the running time for intent generation and program verification. We summarize the results in Table 1.

**Human-written LoC**. In general, *Firebolt* requires a limited number of O(10) LoC for intent grammar, O(10) LoC for semantic constraints, and O(100) LoC for per-symbol specifications. Although the per-symbol specifications occupy the majority of human-written LoC, writing specifications is also required for existing verification tools, and *Firebolt* is still the most efficient. We further discuss *Firebolt* scalability in §6.4.

**Test case size**. As *Firebolt* utilizes pruning mechanisms to generate representative intents, the resulting intents are relatively small, *i.e.*, from one LoC to 10s of LoC. For the same reason, only a few table entries are generated. Marple even has no output table entries, since it uses flexible expressions in the P4 program to implements the intents. Finally, corresponding P4 programs are often with 100s of LoC. This is because generated programs contain many necessary components for all intents such as the definition of headers and parsers. Thus, even the smallest P4 program contains 100s of LoC.

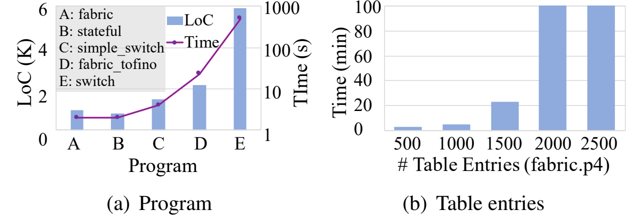
**Running time**. *Firebolt* generates O(1K) intents for each DP generator. Intent generation and program verification in all scenarios can be done within 25 minutes. DP generators with more semantic constraints (*e.g.*, Marple) take more time to generate a correct intent (23ms vs 3ms for Sonata). This is because relatively more semantically invalid intermediate sentential forms will be detected and rejected during generation.

## 6.4 Scalability

**Manual effort**. We compare the manual efforts (*i.e.*, lines of specifications) required by *Firebolt* and existing verification tools, Aquila [31] and p4v [30], to debug a DP generator with equal bug coverage. This means that both *Firebolt* and verification tools take thousands of intents generated by *Firebolt* as input. As shown in Table 4, *Firebolt* requires O(100) of per-symbol LoC to automatically generate the specifications of all intents. In comparison, verification tools require O(100) to O(1K) LoC to convert one intent. Converting all intents means O(100K) to O(1M) LoC. Under equal bug coverage, *Firebolt* consumes merely 0.1% to 0.01% manual efforts compared to existing tools. Moreover, human-written specifications can be faulty, which further reflects the scalability of *Firebolt*.

Table 4: **Comparing lines of human-written specifications.**

| DP Generator Under Test | Verifying One Program | Verifying All Generated Programs | Finding All Bugs (1 Bug / 1 Program) |
|---|---|---|---|
| **p4v** | O(1K) | O(1M) | O(10K) |
| **Aquila** | O(100) | O(100K) | O(1K) |
| **Firebolt** | O(100) | | |




(a) Program (b) Table entries

Figure 9: **Time needed to verify larger P4 programs with more table entries.**

**Scaling to larger test case**. We evaluate the scalability of *Firebolt* when verifying larger P4 programs with more table entries. We use several open-source or vendor-supplied P4 programs instead of the small programs generated by *Firebolt*.

First, we measure the time required by *Firebolt* to verify P4 programs of different sizes. For each program, we manually write <3 entries for each table, and also write the corresponding specifications for verification. As shown in Figure 9(a), *Firebolt* requires more time for verification as the complexity of the P4 program increases. Nevertheless, even the most complex switch.p4 can be verified in 8 minutes.

Next, we compare the verification time when installing different numbers of table entries to the same P4 program `fabric.p4`. As shown in Figure 9(b), the number of table entries has a larger impact on the verification time than the size of P4 program. When the number of entries does not exceed 1500, *Firebolt* can complete the verification in <30 minutes. When the number of entries exceeds 2000, *Firebolt* takes >100 minutes for verification. The increase is not linear because the table entries are converted into if-then-else branches, resulting in an exponential increase in the size of the generated Z3 formulas. This non-linear scalability of verification time with table entries has also been recognized in other verification works [31] and solved using encoding optimizations. Currently, *Firebolt* by design generates small test cases without losing bug coverage using intent space pruning. For upcoming generators, we may encounter larger intents, programs, and table entries that become more time-consuming

to verify. In that case, *Firebolt* can refer to the optimization techniques in existing verification tools [30, 31] to accelerate the verification of individual intent-program pairs.

## 7 Discussion

**Human effort required by *Firebolt***. *Firebolt* requires three inputs, including grammar, semantic constraints, and per-symbol specification to debug a DP generator. First, the grammar should already be provided by the designers of DP generators [14, 16, 21–23] so that the DP generator can be used correctly by others. Second, accurate semantic constraints also help to better use the DP generator. We summarize the semantic constraints of three advanced DP generators in Appendix A. They all have <20 semantic constraints that can be classified into four types, *i.e.*, banned variable redefinition, necessary variable definition, illegal variable reference, and special ones. The former three types account for the majority of the constraints and are closely related to variable definitions and references. Semantic constraints can shrink the intent space, and missing some semantic constraints will not affect the bug coverage but merely produce more intents. Third, *Firebolt* requires manually writing per-symbol specification. However, compared with existing tools, *Firebolt* saves significant human efforts by automatically composing per-symbol specifications into intent specifications.

**Cross-platform generality of *Firebolt***. The formalization phase of *Firebolt* considers extern behaviors because they are critical for the correctness of DP generators. However, the semantics of extern behaviors are target-specific. Currently, *Firebolt* supports two common extern implementations including stateful memory and hash calculation. Since externs can be taken as arithmetic operations on some variables (*e.g.*, temporal variable, metadata field, and packet header), they can always be converted into logical Z3 formulas. As a future work, we would like to extend *Firebolt* to support user-defined extern semantics to improve cross-platform generality.

## 8 Related Work

**Data plane generator**. To simplify DP programming, a growing body of research proposes data plane generators which convert high-level intents into platform-specific DP programs. DP generators provide primitives to specify developer intents in different domains, *e.g.*, query primitives for monitoring tasks [15–20], measurement and control primitives to specify security policies [21, 22] and routing policies [23], and some other intent languages for their own purposes [14, 24–27]. DP generators greatly relieve the burden of DP programming, but their own correctness is not guaranteed. In this paper, we design a blackbox-based testing system to debug them.

**Data plane program verifier**. Several efforts have been proposed to verify DP programs. P4-assert [28] and Vera [29] translate P4 programs into other language models (SEFL and C) and rely on existing symbolic execution framework (SymNet [46] and Klee [47]) to analyze the behavior of the resulting programs. p4v [30] and Aquila [31] use Dijkstra's classic verification approach by formalizing the P4 program in Guarded Command Language (GCL) and using the Z3 theorem prover [35] to check whether the specifications hold. Some other tools, such as bf4 [32] and P6 [34], utilize various techniques such as static verification, code changes and runtime checking to ensure that the deployed P4 program is bug-free. However, using these tools to debug DP generators cannot cover all generator bugs and requires massive manual efforts to verify each possible intent-program pair. Different from all of them, *Firebolt* can automatically generate representative intents as test cases for high coverage and automatically produce intent specifications for high scalability.

**Testing in networking**. Testing is a popular technique to find bugs in network systems by generating and running many test cases. Metha [48] tests network verification tools by generating network configurations as test cases and comparing the tool's output with that of the actual router. p4pktgen [49] tests P4 programs by generating test cases using symbolic execution. Gauntlet [42] and P4Fuzz [50] both test the P4 compiler by generating random P4 programs. If the intermediate representation (IR) of the compiler is accessible, Gauntlet compares the transformed programs after different compiler passes, otherwise it generates packets to test the behavior of the P4 program to debug the compiler. P4Fuzz compares the output of different compilers to find potential bugs. *Firebolt* also uses generation-based testing to debug DP generators. However, unlike existing work to test specific targets, *Firebolt* needs to handle a variety of DP generators. *Firebolt* adopts a syntax-guided approach to generate test cases and designs generic methods to verify the correctness of each test case.

## 9 Conclusion

This paper presents *Firebolt*, a blackbox testing tool designed to debug DP generators. We propose syntax-guided intent generation with semantic constraint injection and intent space pruning techniques, and program verification with automatic intent and program formalization. By evaluating three popular open-source DP generators, we show the high bug coverage and scalability of *Firebolt* compared to existing solutions.

*This work does not raise any ethical issues.*


## Acknowledgement

We sincerely thank our shepherd, Fernando Ramos, and the anonymous reviewers for their constructive comments. Ying Liu and Chen Sun are the corresponding authors. This work is supported by National Key R&D Program of China (Grant No. 2018YFB1800405), National Natural Science Foundation of China (Grant No. 61772307), and Beijing Natural Science Foundation (Grant No.4222026).

## References


[1] Barefoot Networks. Tofino. Website, 2019. https://www.barefootnetworks.com/products/brief-tofino/.

[2] Cavium. Xpliant ethernet switch product family. Website. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3):87–95, 2014.

[4] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In Proceedings of the Symposium on SDN Research, pages 164–176, 2017.

[5] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, pages 576–590, 2018.

[6] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Turboflow: Information rich flow record generation on commodity switches. In Proceedings of the Thirteenth EuroSys Conference, pages 1–16, 2018.

[7] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Deepmatch: practical deep packet inspection in the data plane using network processors. In Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies, pages 336–350, 2020.

[8] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Network anti-spoofing with sdn data plane. In IEEE INFOCOM 2017-IEEE Conference on Computer Communications, pages 1–9. IEEE, 2017.

[9] Jiamin Cao, Ying Liu, Yu Zhou, Chen Sun, Yangyang Wang, and Jun Bi. Cofilter: A high-performance switch-accelerated stateful packet filter for bare-metal servers. In 2019 28th International Conference on Computer Communication and Networks (ICCCN), pages 1–9. IEEE, 2019.

[10] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 161–176, 2019.

[11] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 863–880, 2019.

[12] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable asics. In Proceedings of the 17th ACM Workshop on Hot Topics in Networks, pages 176–183, 2018.

[13] The P4.org language consortium. P4_16 reference compiler. https://github.com/p4lang/p4c.

[14] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In Henning Schulzrinne and Vishal Misra, editors, SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020, pages 435–450. ACM, 2020.

[15] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, and Cheng Zhang. Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries. IEEE Journal on Selected Areas in Communications, 38(6):1147–1160, 2020.

[16] Vikram Nathan, Srinivas Narayana, Anirudh Sivaraman, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Demonstration of the marple system for network performance monitoring. In Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017, pages 57–59. ACM, 2017.

[17] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: query-driven streaming network telemetry. In Sergey Gorinsky and János Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 357–371. ACM, 2018.

[18] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In Proceedings of the Symposium on SDN Research, pages 76–82, 2020.

[19] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: intent-driven network traffic monitoring. In Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies, pages 295–308, 2020.

[20] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In Proceedings of the conference of the ACM special interest group on data communication, pages 99–112, 2017.

[21] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In 29th USENIX Security Symposium (USENIX Security 20), pages 595–612, 2020.

[22] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society, 2020.

[23] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In Ranjita Bhagwan and George Porter, editors, 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020, pages 701–721. USENIX Association, 2020.

[24] Jiamin Cao, Yu Zhou, Ying Liu, Mingwei Xu, and Yongkai Zhou. Turbonet: Faithfully emulating networks with programmable switches. In 2020 IEEE 28th International Conference on Network Protocols (ICNP), pages 1–11. IEEE, 2020.

[25] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable asics. In Proceedings of the 17th ACM Workshop on Hot Topics in Networks, pages 176–183, 2018.

[26] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. Hypertester: high-performance network testing driven by programmable switches. In Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies, pages 30–43, 2019.

[27] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 296–309, 2020.

[28] Miguel C. Neves, Lucas Freire, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. Verification of P4 programs in feasible time using assertions. In Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson, editors, Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018, pages 73–85. ACM, 2018.

[29] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In Sergey Gorinsky and János Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 518–532. ACM, 2018.

[30] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. p4v: practical verification for programmable data planes. In Sergey Gorinsky and János Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 490–503. ACM, 2018.

[31] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In Fernando A. Kuipers and Matthew C. Caesar, editors, ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021, pages 17–32. ACM, 2021.

[32] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free p4 programs. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 571–585, 2020.

[33] Nuno P Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. Technical Report, Tech. Rep., 2016.

[34] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. Fix with p6: Verifying programmable switches at runtime. In IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, pages 1–10, 2021.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.

[36] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, page 436–449, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pages 1–8. IEEE, 2013.

[38] Donald E Knuth. Backus normal form vs. backus naur form. Communications of the ACM, 7(12):735–736, 1964.

[39] Armin Cremers and Seymour Ginsburg. Context-free grammar forms. Journal of Computer and System Sciences, 11(1):86–117, 1975.

[40] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, volume 10196 of Lecture Notes in Computer Science, pages 262–277, 2017.

[41] Changhai Nie and Hareton Leung. A survey of combinatorial testing. ACM Comput. Surv., 43(2):11:1–11:29, 2011.

[42] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020, pages 683–699. USENIX Association, 2020.

[43] Marple code. https://github.com/performance-queries/marple.

[44] Sonata code. https://github.com/Sonata-Princeton/SONATA-DEV.

[45] Poise code. https://github.com/qiaokang92/poise.

[46] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 314–327, 2016.

[47] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.

[48] Rüdiger Birkner, Tobias Brodmann, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Metha: Network verifiers need to be correct too! In James Mickens and Renata Teixeira, editors, 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021, pages 99–113. USENIX Association, 2021.

[49] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In Proceedings of the Symposium on SDN Research, pages 1–7, 2018.

[50] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. P4fuzz: Compiler fuzzer fordependable programmable dataplanes. In International Conference on Distributed Computing and Networking 2021, pages 16–25, 2021.


# Appendix A Semantic Constraints of Advanced DP Program Generators

Table 5 lists the identified semantic constraints for Marple [16], Sonata [17], and Poise [21]. The semantic constraints can be classified into four categories, *i.e.*, banned variable redefinition, necessary variable definition, illegal variable reference to generate *compilable* intents, and other special constraints to generate *complete* intents. For each category, we give an example of how the constraint can be expressed with the formal *if-then* expressions in §4.2.

Table 5: **Summary of semantic constraints of Marple [16], Sonata [17], and Poise [21].**

| Constraints | | Description | Type | Expression |
|---|---|---|---|---|
| **Marple** | **Banned Variable Redefinition (3)** | #1: Each query has a stream name, which cannot be repeatedly defined. | Exclusion | #1 as an example:<br>$if\ \exists\ r_1\ on\ n_1, \nexists\ r_2\ on\ n_2, n_1 \leftrightarrow n_2$<br>$\langle n_1\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle streamName\rangle, *$<br>$\langle r_1\rangle: \langle streamName\rangle \rightarrow *$<br>$\langle n_2\rangle\ *, \langle prog\rangle\langle streamStmt\rangle\langle streamName\rangle, *$<br>$\langle r_2\rangle: \langle r_1\rangle$ |
| | | #2: Each aggregation function has a function name, which cannot be repeatedly defined. | | |
| | | #3: A aggregation function may define multiple aggregation states. The aggregation state names cannot be repeatedly defined. | | |
| | **Necessary Variable Definition (8)** | #4~7: Each query (*map*/*groupby*/*filter*/*zip*) operates on a stream, which should be either defined or the default input stream T. | Dependency | #4 as an example:<br>$if\ \exists\ r_1\ on\ n_1, \exists\ r_2\ on\ n_2, n_1 \rightarrow n_2$<br>$\langle n_1\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle streamQuery\rangle\langle map\rangle\langle streamName\rangle, *$<br>$\langle r_1\rangle: \langle streamName\rangle \nrightarrow T$<br>$\langle n_2\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle streamName\rangle, *$<br>$\langle r_2\rangle: \langle r_1\rangle$ |
| | | #8: *groupby* queries take an aggregation function name as input. The function should be defined. | | |
| | | #9: *groupby* queries may include self-defined variables in the aggregation key. The variables should be defined. | | |
| | | #10: *filter* queries may reference self-defined variables in its predicate. The variables should be defined. | | |
| | | #11: *map* queries may reference self-defined variables. The variables should be defined. | | |
| | **Illegal Variable Reference (2)** | #12: *map* queries assign specified or self-defined variables to computed expressions, where the variables should not be assigned repeatedly. | Exclusion | #12 as an example:<br>$if\ \exists\ r_1\ on\ n_1, \nexists\ r_2\ on\ n_2, n_1 \leftrightarrow n_2$<br>$\langle n_1\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle \text{streamQuery}\rangle\langle map\rangle\langle map_col\rangle, *$<br>$\langle r_1\rangle: \langle map_col\rangle \rightarrow *$<br>$\langle n_2\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle \text{streamQuery}\rangle\langle map\rangle\langle map_col\rangle, *$<br>$\langle r_2\rangle: \langle r_1\rangle$ |
| | | #13: *zip* queries merge fields in *different* streams. | | |
| | **Special Semantic Constraints (1)** | #14: An intent should not end with a *map* query. | Dependency | #14 as an example:<br>$if\ \exists\ r_1\ on\ n_1, \exists\ r_2\ on\ n_2, n_1 \rightarrow n_2$<br>$\langle n_1\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle streamQuery\rangle, *$<br>$\langle r_1\rangle: \langle streamQuery\rangle \rightarrow \langle map\rangle$<br>$\langle n_2\rangle: *, \langle prog\rangle\langle streamStmt\rangle\langle streamName\rangle, *$<br>$\langle r_2\rangle: \langle streamQuery\rangle \rightarrow \langle filter\rangle \mid \langle groupby\rangle \mid \langle zip\rangle$ |
| **Sonata** | **Special Semantic Constraints (2)** | #1: An intent should not end with a *map* query. | Dependency | #1 as an example:<br>$if\ \exists\ r_1\ on\ n_1, \exists\ r_2\ on\ n_2, n_1 \leftarrow n_2$<br>$\langle n_1\rangle: *, \langle prog\rangle\langle streamQuery\rangle, *$<br>$\langle r_1\rangle: \langle streamQuery\rangle \rightarrow \langle map\rangle$<br>$\langle n_2\rangle: *, \langle prog\rangle\langle streamQuery\rangle, *$<br>$\langle r_2\rangle: \langle streamQuery\rangle \rightarrow \langle filter\rangle \mid \langle reduce\rangle \mid \langle distinct\rangle$ |
| | | #2: *reduce* queries should follow *map* queries. | | |
| **Poise** | **Banned Variable Redefinition (2)** | #1: Each *list* has a name, which cannot be repeatedly defined. | Exclusion | Similar to the banned variable redefinition of Marple. |
| | | #2: Each *monitor* function has a name, which cannot be repeatedly defined. | | |
| | **Necessary Variable Definition (2)** | #3: Each *monitor* function references a *list*, which should be defined. | Dependency | Similar to the necessary variable variable definition of Marple. |
| | | #4: A *statement* may reference a *monitor* function, which should be defined. | | |