Firebolt: Finding Bugs in Programmable Data Plane Generators

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

https://www.usenix.org/conference/atc22/presentation/cao
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 $2 \times$ 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).
• **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.

#### #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.
§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.

Intent grammar formalization. Syntax-guided intent generation.

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.

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 $(N, S, \Sigma, R)$, 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 $A \rightarrow \beta$. 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 $G = (N, \mathcal{E})$ as a directed labeled graph derived from grammar $G = (N, S, \Sigma, R)$, with the nodes $N \subseteq (N \cup \Sigma)^*$ and the edges $\mathcal{E} \subseteq N \times N \times R$. Each node in $N$ 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 $A \rightarrow \beta \in R$ and derive a child node $n_2$ with a new sentential form $\alpha A \beta \gamma$. $G$ has a root node $n_0$ with sentential form $S$, and many (maybe infinite) leaf nodes.

```
<table>
<thead>
<tr>
<th>Algorithm 1 Syntax-Guided Intent Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: function SYGuG(G) {</td>
</tr>
<tr>
<td>2:    Q = Queue(G.n₀)</td>
</tr>
<tr>
<td>3: while Q.size() &gt; 0 do</td>
</tr>
<tr>
<td>4:    n = Q.front()</td>
</tr>
<tr>
<td>5:    Q.pop()</td>
</tr>
<tr>
<td>6: if n.has_nt() then</td>
</tr>
<tr>
<td>7:    A = n.first_nt</td>
</tr>
<tr>
<td>8:    for A \rightarrow \beta \in G.R[A] do</td>
</tr>
<tr>
<td>9:        n = \alpha A \gamma \rightarrow \alpha \beta \gamma \triangleright Grow graph</td>
</tr>
<tr>
<td>10:       Q.push(n₁)</td>
</tr>
<tr>
<td>11:   end for</td>
</tr>
<tr>
<td>12:   else</td>
</tr>
<tr>
<td>13:      OUTPUT(n)</td>
</tr>
<tr>
<td>14:      end if</td>
</tr>
<tr>
<td>15: end while</td>
</tr>
<tr>
<td>16: end function</td>
</tr>
</tbody>
</table>
```
Algorithm 1 depicts the procedure of growing graph \( G \) using \textit{depth first search}. We maintain a queue \( Q \) to store nodes in \( G \). First, \( Q \) is initialized with the root node \( n_1 \) (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 identify 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.

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

- \textit{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.

- \textit{Incomplete intents}. Some intents are semantically incomplete. For example, the \texttt{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 \texttt{map} primitive in the end of a query is considered incomplete and should be ruled out.

\textbf{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 (uncompilable 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.

- \textit{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:

  \[
  \text{if } \exists r_1 \text{ on } n_1, \text{ then } \nexists r_2 \text{ on } n_2
  \]

  A typical example is that one variable name (such as the name of a packet stream) cannot be defined repeatedly.

- \textit{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:

  \[
  \text{if } \exists r_1 \text{ on } n_1, \text{ then } \exists r_2 \text{ on } n_2
  \]

4.2.1 \texttt{Map} query operates on a stream that has been defined

\[
\text{if } \exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, r_2 \not\rightarrow r_1
\]

\[
\begin{align*}
(n_1) & : +, (\text{\texttt{prog}})(\text{\texttt{streamStmt}})(\text{\texttt{streamName}}), * \\
(r_1) & : (\text{\texttt{streamName}}) \rightarrow * \\
(n_2) & : +, (\text{\texttt{prog}})(\text{\texttt{streamStmt}})(\text{\texttt{streamName}}), * \\
(r_2) & : \langle r_1 \rangle
\end{align*}
\]

A typical example is that a \texttt{Map} primitive must operate on a stream that has been previously defined.

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.

\textbf{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 \((r_1, n_1, r_2, n_2)\). Next we showcase two semantic constraints in Marple [16].

Example 1 shown in Figure 4 presents an exclusion constraint. A Marple program \( \langle \text{\texttt{prog}} \rangle \) includes multiple streams \( \langle \text{\texttt{streamStmt}} \rangle \), each with name \( \langle \text{\texttt{streamName}} \rangle \). 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 \( \langle \text{\texttt{prog}} \rangle \) to the current symbol \( \langle \text{\texttt{streamName}} \rangle \). We use \( \not\rightarrow \) to indicate that the two nodes can appear in any order on the path. We use \( \rightarrow \) to match any expansion traces in \( r \) and any expansion rules in \( r \). By making \( r_2 = r_1 \), the constraint prevents \( \langle \text{\texttt{streamName}} \rangle \) from taking duplicate values of \( r_1 \) in any other node \( n_2 \).

Example 2 in Figure 4 presents a dependency constraint. If a \( \langle \text{\texttt{map}} \rangle \) 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 com-
binations 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 deriv-
a tion 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 losing intent representativeness.

**Method #1: Intra-symbol representativeness.** To handle
wide parameter range, we propose to keep representative ex-
pansion 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.

**Method #2: Inter-symbol combination representativeness.**

For cyclic symbol reference, we can think of an expansion
ring 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) the-
ory [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 gen-
erator with high efficiency, we can first extract all distinct
features, i.e., combination of non-terminal symbols, accord-
ing 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
$N = \sum_{i=1}^{k} A_k^i$ features where $A$ stands for the permutation
symbol, i.e., $A_n^m = n!/(n-m)!$. With the combination factor
$n = 2$, there exist at most $(N + A_2^2) \times s \times p$ test cases, which are finite and feasible for
testing. Our evaluation results in §6.2 show that using the

![Figure 5: Two types of intent space explosion.](image-url)
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 \(\rightarrow\) 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 \(\rightarrow\) 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:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a0(1)</td>
</tr>
<tr>
<td>2</td>
<td>a0(2)</td>
</tr>
</tbody>
</table>

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 (= t0_index 1) a0_z 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 \(\rightarrow\) 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 P416 Program: Register Reading and Writing
register(bit<32>=11); 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 P416 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
( _ BitVec 96 ) inKey
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 ⟨groupby⟩ symbol in MapReduce [16] as an example to illustrate how to construct a stateful symbol specification. An entire ⟨groupby⟩ symbol is formatted as ⟨groupby⟩ ::= groupby(⟨stream⟩, ⟨columns⟩, ⟨aggFunc⟩), 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 ⟨var⟩ in the child symbol ⟨aggFunc⟩. 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 ⟨streamName⟩. Then we get the aggregation key from the aggregation field in ⟨columns⟩. Using the aggregation key,
Firebolt constructs its specification based on its generation whether each P4 program is consistent with the corresponding (826 2022 USENIX Annual Technical Conference • f frequency, each with a different set of cally, we need to check the following three types of consis-

tion. Then, we summarize security vulnerabilities from the grams and intents formalized into Z3 assertions, we first check (5.3 Program Correctness Verification assertions, e.g. is a series of algebra expressions connected with logical oper-

erators and construct an individual Z3 expression, which

is no longer referenced by any blocks, which becomes the final output of the DP program and completes the Z3 for-

ter blocks. We iterate this process until the output 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 lat-

ter 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 for-

mula 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 vulner-

abilities 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

### 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 FCUs 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 \( f, g \) assertions that will be used to verify the DP programs.

### 5.3 Program Correctness Verification

With the Z3 formulas representing semantics of DP pro-

grams 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 lat-

ter 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 for-

mula 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 vulner-

abilities 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

```python
# (groupby) ::= groupby ((streamName), (columns), (aggFunc))
# (aggFunc) ::= def (aggFunc)((pars), (columns)); (codeblock)

DECL_FUNC() =
  for var in <aggFunc>,<pars>,<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
```


of egress_port, we assert that no branch results in the unde-
fined 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 de-
  parsed 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 P416 and P414 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 gen-
erators, including two for network telemetry, i.e., Marple [16,
43] and Sonata [17, 44], and one for security policy enforce-
ment, 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 secur-
ity 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 vi-
lations, 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 pro-
  grams 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 viola-
tions 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.

<table>
<thead>
<tr>
<th>Pruning Method</th>
<th># Detected Bugs</th>
<th># Generated Intents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Marple Poise</td>
<td>Sonata Sonata Poise</td>
</tr>
<tr>
<td>Intra-Symbol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r=1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>r=2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inter-Symbol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n=1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n=2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n=3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 (≥) 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 comparator.** Poise provides list comparator (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 ≥ and <. Besides, Poise sets a default range (0~10000) 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.

<table>
<thead>
<tr>
<th>Dp Generator Under Test</th>
<th># Intents</th>
<th># Detected Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marple</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Sonata</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Poise</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Crash bug: Firebolt finds one crash bug in Marple, while Sonata and Poise do not report any crash bugs. Marple converts the division expression \( (a/2^n) \) 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× 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 in-
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 sential 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.

**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


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>
<thead>
<tr>
<th>Constraints</th>
<th>Description</th>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Banned Variable Redefinition</strong></td>
<td>#1: Each query has a stream name, which cannot be repeatedly defined.</td>
<td>Exclusion</td>
<td>#1 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \leftrightarrow n_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(q_1) : (\text{streamName}) \to \star )(q_2) : \star \to q_1)</td>
</tr>
<tr>
<td></td>
<td>#2: Each aggregation function has a function name, which cannot be repeatedly defined.</td>
<td></td>
<td>#2 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \leftrightarrow n_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(q_1) : (\text{streamName}) \to \star )(q_2) : \star \to q_1)</td>
</tr>
<tr>
<td></td>
<td>#3: A aggregation function may define multiple aggregation states. The aggregation state names cannot be repeatedly defined.</td>
<td></td>
<td>#3 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \leftrightarrow n_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(q_1) : (\text{streamName}) \to \star )(q_2) : \star \to q_1)</td>
</tr>
<tr>
<td><strong>Necessary Variable Definition</strong></td>
<td>#4–7: Each query (map/groupby/filter/zip) operates on a stream, which should be either defined or the default input stream T.</td>
<td>Dependency</td>
<td>#4 as an example:</td>
</tr>
<tr>
<td><strong>Marple</strong></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>#8: groupby queries take an aggregation function name as input. The function should be defined.</td>
<td></td>
<td>(q_1) : (\text{streamName}) \to \star T)(q_2) : \star \to q_1)</td>
</tr>
<tr>
<td></td>
<td>#9: groupby queries may include self-defined variables in the aggregation key. The variables should be defined.</td>
<td></td>
<td>#9 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>#10: filter queries may reference self-defined variables in its predicate. The variables should be defined.</td>
<td></td>
<td>(q_1) : (\text{streamName}) \to \star )(q_2) : \star \to q_1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#10 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>#11: map queries may reference self-defined variables. The variables should be defined.</td>
<td></td>
<td>#11 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td><strong>Illegal Variable Reference</strong></td>
<td>#12: map queries assign specified or self-defined variables to computed expressions, where the variables should not be assigned repeatedly.</td>
<td>Exclusion</td>
<td>#12 as an example:</td>
</tr>
<tr>
<td><strong>(2)</strong></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>#13: zip queries merge fields in different streams.</td>
<td></td>
<td>(q_1) : (\text{map} _col) \to \star )(q_2) : \star \to q_1)</td>
</tr>
<tr>
<td><strong>Special Semantic Constraints</strong></td>
<td>#14: An intent should not end with a map query.</td>
<td>Dependency</td>
<td>#14 as an example:</td>
</tr>
<tr>
<td><strong>(1)</strong></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td><strong>Sonata</strong></td>
<td>#1: An intent should not end with a map query.</td>
<td>Dependency</td>
<td>#1 as an example:</td>
</tr>
<tr>
<td><strong>Special Semantic Constraints</strong></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td><strong>(2)</strong></td>
<td>#2: reduce queries should follow map queries.</td>
<td></td>
<td>#2 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td><strong>Poise</strong></td>
<td>Banned Variable Redefinition</td>
<td>Exclusion</td>
<td>Similar to the banned variable redefinition of Marple.</td>
</tr>
<tr>
<td></td>
<td>#1: Each list has a name, which cannot be repeatedly defined.</td>
<td></td>
<td>#1 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>#2: Each monitor function has a name, which cannot be repeatedly defined.</td>
<td></td>
<td>#2 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>Necessary Variable Definition</td>
<td>Dependency</td>
<td>Similar to the necessary variable variable definition of Marple.</td>
</tr>
<tr>
<td></td>
<td>#3: Each monitor function references a list, which should be defined.</td>
<td></td>
<td>#3 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
<tr>
<td></td>
<td>#4: A statement may reference a monitor function, which should be defined.</td>
<td></td>
<td>#4 as an example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \to n_2)</td>
</tr>
</tbody>
</table>

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