P4 Safety Verification

Mark Drobnak
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
mtd8050@cs.rit.edu

***Abstract*—P4 is a domain specific language for programming network switches in software-defined networks (SDNs). Like with most programming languages, P4 programs can have bugs and safety issues which cause undesirable behavior. Static analyzers such as p4v and bf4 attempt to detect these undesirable behaviors. This paper introduces p4-analyzer, a static analysis tool which can detect certain kinds of bugs in P4 programs. p4-analyzer uses p4v's idea of translating P4 to GCL, along with bf4's idea of creating a control flow graph with bug nodes. p4-analyzer's contributions include incorporating Petr4's formalization of P4, performing uninitialized variable analysis, and introducing a novel usage of the Z3 solver.**

Keywords—P4, Software verification, Programmable networks

I. Introduction

P4[3] is a domain specific language for programming network switches in software-defined networks (SDNs). P4 enables rapid experimentation and deployment of network protocols. Protocols programmed via P4 can operate at hardware speeds while retaining the flexibility to quickly prototype and revise their operation.

With such flexibility comes shortcomings. P4 is relatively portable across multiple targets, and the semantics of the language contain instances of target-specific behavior. Some of this behavior may result in undefined or arbitrary values entering the program, such as when an uninitialized variable is read from. P4 programs are mostly deterministic, allowing many of these undesirable behaviors to be caught statically. This has inspired the creation of a number of static analyzers for P4.

P4 static analyzers such as p4v[2] and bf4[1] have focused on detecting undesirable behaviors resulting from reading uninitialized header values. These tools often detect bugs by referring to the semantics of P4, which are specified in a document published by the P4 Language Consortium[3]. Notably, this document does not provide a formal model of P4, so reasoning about when undesirable behavior may occur is difficult. This shortcoming inspired the Petr4[4] project, which gives formal static and dynamic semantics for a large subset of the P4 language. With this formal foundation, static analysis of P4 can reason about the behavior of P4 programs more accurately.

This paper introduces p4-analyzer[1], a static analysis tool for finding bugs in P4 programs. It implements a parser and type checker for a subset of the P4 language, and performs analysis of the program via translation to a Control Flow Graph of GCL (simple variable-expression assignments). p4-analyzer is based on the formal foundations of P4 provided by Petr4, and its analysis of P4 programs is inspired by p4v and bf4. p4-analyzer currently focuses on detecting bugs caused by reading from uninitialized variables. p4-analyzer's contributions to the space of P4 static analysis include:

1. The inclusion of Petr4's formal semantics.
2. Improved uninitialized variable bug detection.
3. A novel translation of reachability predicates to Z3.

The remainder of this paper is organized as follows: First some background in P4 is provided, followed by a motivating discussion of the undesirable behavior found in some P4 programs. The related work in the area of P4 static analysis is also analyzed. The contributions of p4-analyzer are discussed, followed by an in-depth look at the analysis it performs. Some example programs are examined in the Results section, and the discussion is wrapped up with conclusions and future work.

II. Background

P4 programs implement the data plane of an SDN, at the level of individual network switches. Because of this, a P4 program goes through three major steps:

1. Parse the packet. The packet structure is examined and transformed into a representation understood by the P4 program.
2. Route/modify the packet. The main logic of the P4 program is in this step. Based on the packet's contents and the entries in match-action tables (set by the control-plane), the packet is possibly modified and routed or dropped. This is where the majority of bugs occur.
3. Emit the packet. Once the packet has been modified and routed (and not dropped), it is serialized and emitted from the switch.

A major part of the P4 language are match-action tables. These tables map values to actions. For example, a table may map the IPv4 source address of a packet to either a forward or drop action (This example is shown in Figure 1). The P4 program usually does not specify the concrete key-value pairs, or rules, in the table. The control plane will insert rules into the P4 program's tables at runtime. In this way, since a P4 program is given match-action table entries from an external control plane, tables are nondeterministic. The rest of the

[1] https://github.com/AzureMarker/p4-analyzer

language is largely deterministic and therefore many bugs can be detected statically.

The P4 language is portable across many switch architectures, but each architecture introduces slightly different capabilities and dataflows. The P4 v1.0 model, or v1model, is a commonly analyzed architecture and is used in this project.

```
control my_ingress(inout headers_t hdrs) {
  action drop_packet() { /* ... */ }
  action forward_packet(bit<32> dest) {
    /* ... */
  }

  table ip_routes {
    // Longest-Prefix Match
    key = { hdrs.source_ip: lpm; }
    actions = {
      drop_packet; forward_packet;
    }
  }

  apply {
    ip_routes.apply();
  }
}
```

Figure 1. A P4 program with a match-action table that maps source IP addresses to actions.

III. Motivation

P4 is a pragmatic language first and foremost. While it takes some measures to keep the language safe (ex. providing well-defined integer operations), there are certain ways in which unsafe or undesirable behavior can be exhibited. Some of these patterns are listed below:

1. Reading from an uninitialized variable. It is legal to define a variable in P4 without giving it a value. According to the specification, this variable has an undefined value, similar to declaring an uninitialized variable in C. Using an uninitialized value may result in undesirable or nondeterministic behavior.
2. Reading from an invalid header. Packet data is parsed into headers, which are like structs. Before a header is extracted from the input packet, the header data is uninitialized. Reading from an invalid header acts similarly to reading from an uninitialized variable, and can cause similar bugs.
3. Reading from an "out" parameter. Functions and actions in P4 assign a direction to each parameter. A parameter may either be "in" (read-only), "out" (write-only), or "inout" (read/write). "out" parameters act similarly to uninitialized variables, and can cause similar bugs.

These patterns of unsafety in P4 programs can usually be detected statically, thanks to the highly deterministic nature of P4 programs. This is the goal of P4 static analysis.

IV. Related Work

Static analysis of P4 programs is a current topic of study. The p4v[2] project analyzes P4 programs to find various safety issues and bugs. It does this by first translating the P4 program into Guarded Command Language (GCL), a language that is more well-defined and straightforward to analyze. p4v also relies on assertions inserted into the GCL code which mark when bugs may occur. Using this GCL code, p4v calculates a precondition that must be satisfied for no bugs to occur. It then runs the precondition's negation through an SMT solver (Z3). If the solver finds a satisfying assignment to the predicate, p4v then derives a model of the program's input which will trigger a bug. By performing a translation to GCL, p4v solidifies the semantics of the P4 program. Unfortunately p4v is closed source, making it difficult to determine how powerful its analysis is.

The bf4[1] project also analyzes P4 programs for bugs, focusing on bugs caused by match-action table rules inserted by the control plane. bf4 is based on the P4 reference compiler, p4c, and reuses its infrastructure in order to perform its analysis. As part of this, bf4 performs its analysis directly on P4 code instead of translating to a more well-defined language such as GCL. First, bf4 annotates the P4 code with explicit checks (ex. checking header validity). From this annotated code, bf4 creates a Control Flow Graph (CFG). To check for bugs, bf4 calculates the reachability of each node and checks if any bug nodes are reachable. Additionally, bf4 performs advanced analysis of match-action tables to determine if any rules inserted by the control plane at runtime may cause bugs. There is correspondingly a runtime component of bf4 which will enforce that no "buggy" match-action table rules are inserted.

P4 is described by an informal language specification maintained by the P4 Language Consortium[3]. This document describes the language itself and its semantics. There is also a reference compiler, p4c, which implements the specification. Unfortunately the language specification can be vague in some areas, and looking to the reference compiler is difficult due to its complexity. This has inspired the Petr4[4] project to make a specification of P4 using formal methods. A significant subset of P4 is described using formal methods, including its static and dynamic semantics. The paper gives a formal proof of type soundness and termination based on these semantics.

V. P4-Analyzer

p4-analyzer's main contributions include the inclusion of Petr4's formal semantics, improved uninitialized variable bug detection, and a novel translation of reachability predicates to Z3. These contributions are analyzed here, followed by an in-depth discussion of how p4-analyzer analyzes P4 programs.

A. Strong Theoretical Foundation

p4-analyzer is built upon the theoretical foundation of Petr4 and understands the semantics of the P4 program (as specified by Petr4) via GCL translation. Previous works were based on more experimental foundations. p4v's type checker is based on the informal language specification document and consultations with the p4c reference compiler. bf4 reuses the infrastructure of p4c and understands the semantics of the

program through informal methods. p4v created a more solid foundation via the translation of P4 to GCL, wherein the semantics of the program were made clearer. p4-analyzer combines the advantages of GCL translation with the formal static and dynamic semantics given by Petr4 to produce a stronger foundation for P4 program analysis.

With Petr4's formal semantics, it is possible to formally prove the presence of a bug. For example, Figure 2 contains the E-HdrMem and E-HdrMemUndef inference rules from Petr4's dynamic semantics. Without going into their specifics, they specify the behavior of reading a field from a header struct. This behavior depends on the validity of the header (if it has been extracted from the packet). If the header is invalid, the read will result in a "havoc" value, also known as an undefined or arbitrary value. This havoc value is highly likely to cause issues later in the program. Building up a tree of inference rule applications for a program in order to prove the presence of a havoc value (and therefore a possible bug) may be difficult, but this is the essence of static analysis based on formal methods.

B. Uninitialized Variable Bugs

bf4 analyzes the P4 program in only a few specific ways. The main checks it performs are around header validity and how headers interact with match-action tables. Notably, bf4 itself does not search for usages of uninitialized variables. Perhaps this was left out because the reference compiler that bf4 is based upon, p4c, does basic uninitialized variable analysis (marking bugs as a warning). However, p4c's analysis only detects simple cases, so there is room for improvement. p4-analyzer's main analysis, once the program passes type checking, is to check for uninitialized variable usages. Figures 3 and 4 show programs which read from an uninitialized variable. The bug in Figure 3 is caught by both p4c/bf4 and p4-analyzer, but Figure 4's bug is only caught by p4-analyzer. Note: as p4v is closed source we are unable to check how it handles uninitialized variable bugs.

p4-analyzer does not currently check the validity of headers like bf4 does due to project time constraints, but this analysis should be straightforward.

C. Novel Translation to Z3

The translation of reachability predicates to Z3 is partly novel. The predicates are formulated in terms of GCL and may refer to structs or other P4 types. Previous works (p4v and bf4) would translate complex data types into simpler ones, such as booleans and integers. Z3 supports a decent range of data types, including booleans and integers, but also algebraic data types. These previous works did not take advantage of Z3's algebraic data types functionality. p4-analyzer utilizes Z3's advanced capabilities by translating P4's complex types (e.g. structs) into Z3's type system. This allows p4-analyzer to easily translate to and from Z3. Since Z3 will also return a model of the satisfying assignment after checking a predicate, this model can be easily mapped back to the original P4 program to demonstrate the discovered bug. Z3's data types are similar to algebraic data types in functional languages like SML and Haskell. Structs can be mapped into this framework by creating a data type with a single variant. Going back to the

$$\textsc{E-HdrMem}\quad \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma', \text{header } \{\text{valid}, \overline{f : \tau = val}\}\rangle}{\langle C, \Delta, \sigma, \epsilon, exp.f_i\rangle \Downarrow \langle \sigma', val_i\rangle}$$

$$\textsc{E-HdrMemUndef}\quad \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma', \text{header } \{!\text{valid}, \overline{f : \tau = val}\}\rangle}{\langle C, \Delta, \sigma, \epsilon, exp.f_i\rangle \Downarrow \langle \sigma', \text{havoc}(\tau_i)\rangle}$$

Figure 2. E-HdrMem and E-HdrMemUndef inference rules from Petr4.

```
struct metadata_t {
  bool value;
}

control my_control(inout metadata_t meta) {
  bool var;

  apply {
    if (meta.value) {
      // Bug: 'var' is uninitialized
      meta.value = var;
    }
  }
}
```

Figure 3. A P4 program with a bug, triggered when `meta.value` equals true.

```
struct header_t { bool value; }
struct headers_t { header_t h; }

control my_ingress(inout headers_t hdrs) {
  apply {
    bool foo;
    if (!hdrs.h.value) {
      foo = true;
    }
    // Bug: foo is not always initialized
    bool bar = foo;
  }
}
```

Figure 4. A P4 program with a bug, triggered when `hdrs.h.value` equals true.

program in Figure 3, p4-analyzer creates a Z3 datatype to represent the `metadata_t` struct type:

```
(declare-datatypes () (
  (metadata_t (metadata_t (value Bool)))
))
```

The data type's single variant holds a single boolean field representing the `value` field in `metadata_t`. Now that structs are natively represented, Z3 has more precise knowledge of the source P4 program. The satisfying model that Z3 returns when it finds the bug contains the "meta" variable bound to a value of the `metadata_t` Z3 data type, where the inner field is true. This satisfying model translates directly back to P4's `metadata_t` struct type:

```
Z3: meta = (metadata_t true)
P4: meta = (metadata_t) { value = true }
```

D. Description of the Analysis Algorithm

Now the algorithm behind p4-analyzer is described. But first, it is useful to note that the flow of data through p4-analyzer is shown in Figure 5. The description of p4-analyzer's algorithm given below follows this figure.

p4-analyzer's first step, after parsing the program into an Abstract Syntax Tree (AST), is to type check the program. This analysis is based on Petr4, which defines a formal type system and calculus for a core subset of P4. Type checking can catch basic bugs, such as reading from an undefined variable or assigning a value to a field with an incompatible type. These are basic bugs and even the P4 reference compiler, p4c, detects them. However, having a well-typed program is a prerequisite for later analyses, and the type information is also core to some analyses. As part of type checking, the program is transformed from the original AST into a typed Intermediate Representation (IR). This IR embeds the type information gained during type checking and is slightly simpler than the original AST.

The second step of analysis is to transform the IR from type checking into a Control Flow Graph (CFG) of Guarded Command Language (GCL). This step combines the ideas from p4c and bf4. P4 is given more solid semantics through the translation to GCL, which is a much simpler and analysis-friendly language. The transformation from a tree-like IR to a CFG is the first step towards finding bugs in the program.

The structure of the GCL CFG is straightforward, but different from the structure used in bf4. p4-analyzer's GCL CFG is not in Single Static Assignment (SSA) form. Each node contains a sequence of straight-line GCL code. Nodes may have outgoing edges which point to other nodes. Each edge contains a predicate which must evaluate to true for the edge to be taken at runtime. This structure was chosen instead of the more traditional trailing condition + true/false edges to facilitate the nondeterministic behavior of P4's match-action tables. The GCL code itself consists of a sequence of statements. A statement either assigns an expression to a variable or adds/removes a fact about the program. Facts represent certain knowledge of the program, such as if a variable has a value. For example, once a variable has been assigned to in the original P4 program, a "HasValue" fact is added which stores the knowledge that the variable has a value. If the variable is used later, the analysis will know that it is safe to read from the variable (it is not uninitialized). This is a simple form of dataflow analysis.

As part of this GCL CFG transformation, the program is annotated with bug nodes whenever an operation might exhibit undesired behavior. For example, when an expression reads the value of a variable, it is possible that the variable is uninitialized. A "bug" node is added to the CFG and connected to the original expression node to represent the possibility that the variable is uninitialized. The edge leading to the bug node has a predicate which is true when the variable does not have a value. In other words, the bug node is reachable if the variable does not have a value at the point in the program.

The next step of analysis is computing a reachability predicate for each node in the GCL CFG. These predicates represent constraints on the incoming packet such that the node is on the execution path of the program. For example, if the "then" branch of an if statement is only taken when the first bit of the packet is a 1, then the reachability predicate of the "then" branch node will contain the requirement that the packet's first bit is a 1. These predicates are computed by traversing the CFG in topological order, propagating constraints from parent to child. For each parent, the predicate on the edge between parent and child is combined with the parent's reachability predicate using a boolean AND. If a child has multiple parents, it combines each predicate with a boolean OR. The reachability algorithm also keeps track of the variable assignments and performs simple expression folding. For example, if a := b and b := 1 + d then the analysis will replace a with 1 + d. In the case where a node has multiple incoming edges, the variables may have multiple values/expressions. This requires the analysis to keep track of a set of possible values/expressions for each variable. In this way, constraints are propagated through the graph. An example GCL CFG annotated with reachability information is shown in Figure 6, generated from the Figure 3 P4 program.

The final step of analysis is to check the satisfiability of the reachability predicates to determine if any bug nodes are reachable. The Z3 SMT solver is used for this task. The reachability predicates are translated into Z3 predicates and checked for satisfiability. If a predicate is satisfiable, Z3 returns a model of the input which makes the reachability predicate true. The reachable bug nodes and their models are output as bugs in the program, concluding p4-analyzer's analysis of the P4 program.

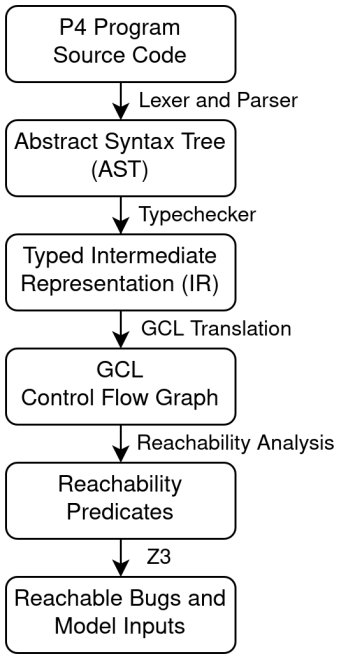

Figure 5. The flow of data through p4-analyzer

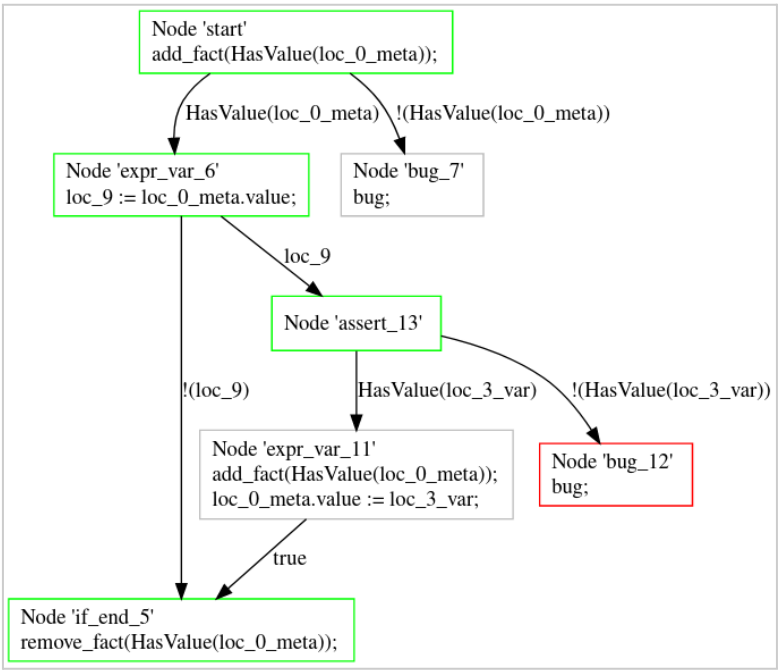

Figure 6. GCL Control Flow Graph of the Figure 3 P4 program. Green or red indicates the node is reachable.

```
control my_control() {
  apply {
    bool foo;
    bool bar = foo;
  }
}

Input Model = (empty)
```

Figure 7. A trivial example of an uninitialized variable bug, including the Z3 input model from p4-analyzer's analysis.

VI. Results

p4-analyzer was evaluated by testing against a suite of buggy programs. The programs and outputs shown here display some of the interesting results from this evaluation.

A. Simple Bugs

These programs contain simple bugs which are also caught by bf4. Figure 7 is a trivial example of reading from an uninitialized variable. The model is empty because the bug will always occur.

Figure 3 shows a slightly less trivial example of reading from an uninitialized variable. The Z3 model of the buggy input is `loc_0_meta -> (struct_0_metadata_t true)`, which translated into P4 is `meta = { value = true }`. p4-analyzer maintained the knowledge that meta is a struct all the way down to the Z3 solver.

B. Non-trivial Bugs

Figure 4 shows a program which has a non-trivial uninitialized variable bug. The variable `foo` is only initialized when `hdrs.h.value` is false. When foo is read from after the if statement, it is not guaranteed to be initialized. Running this program through p4-analyzer results in the bug being found. However, the Z3 model of the input is empty. This is caused by the inability of the reachability analysis to reason about conditional facts. In this case, the "HasValue" fact for `foo` is only available if `hdrs.h.value` is false. However, the analysis still finds the bug, and the information provided in the analysis metadata (e.g. GCL graph) is sufficient to understand the issue.

VII. Conclusions

Static analysis of P4 is an evolving topic and invites new approaches. The inclusion of formal methods opens up more new and interesting ways to analyze and verify P4 programs, and new ways of verifying the correctness of the analysis.

p4-analyzer successfully analyzes programs written in a subset of P4 to verify type safety and freedom from bugs caused by uninitialized variables. Its analysis produces human-readable artifacts to aid in bug analysis, such as the GCL control flow graph and reachability predicates. p4-analyzer introduces a novel translation of complex P4 types such as structs into Z3, simplifying the translation to and from while encoding more precise information about the program to the solver. p4-analyzer is able to detect non-trivial cases of reading from uninitialized variables. Finally, the formal foundation provided by Petr4 gives strong guarantees about the correctness of p4-analyzer's analysis.

VIII. Future work

p4-analyzer is still limited in the kinds of programs it can analyze, and there is room for improvement in its analysis.

A. Syntactic vs Semantic Analysis

p4-analyzer lays the groundwork for when bugs may occur during the translation to a GCL CFG. It is here that the semantics of P4 are codified and bug nodes are created. This step is inspired by the GCL translation done in p4c and the bug-node-annotated CFG used in bf4. Unlike bf4, the CFG contains GCL code instead of P4 code. bf4 performed

its analysis syntactically, looking at the CFG of P4 code. It is not clear if the syntactic focus of bf4 limits its potential analysis, and if the semantic focus of p4-analyzer enables more bugs to be found.

B. Full P4 Language Support

p4-analyzer currently supports a subset of P4, namely control blocks which use structs and booleans. Some important pieces of P4 including match-action tables, integers, bit strings, headers, and parsers are not implemented. Petr4 also does not support parsers in the static and dynamic semantics, so work may be needed to describe parsers in the formal specification before p4-analyzer's parser support can fully benefit from Petr4's formal foundation. Supporting these constructs and types would allow p4-analyzer to accept more programs and detect more interesting bugs.

Acknowledgments

I would like to thank Dr. Minseok Kwon for his guidance on this project as my advisor and for introducing me to the world of P4 static analysis.

References

[1] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *SIGCOMM '20* (August 2020). https://doi.org/10.1145/3387514.3405888

[2] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM '18* (August, 2018). https://doi.org/10.1145/3230543.3230582

[3] P4 Language Consortium. 2020. P4 Language Specification, Version 1.2.1. Retrieved from https://p4.org/p4-spec/docs/P4-16-v1.2.1.html

[4] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: Formal Foundations for P4 Data Planes. In *POPL '21* (January 2021). https://doi.org/10.1145/3434322