How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4

Matthias Eichholz
Technische Universität Darmstadt, Germany

Eric Campbell
Cornell University, Ithaca, NY, USA

Nate Foster
Cornell University, Ithaca, NY, USA

Guido Salvaneschi
Technische Universität Darmstadt, Germany

Mira Mezini
Technische Universität Darmstadt, Germany

Abstract

The P4 programming language offers high-level, declarative abstractions that bring the flexibility of software to the domain of networking. Unfortunately, the main abstraction used to represent packet data in P4, namely header types, lacks basic safety guarantees. Over the last few years, experience with an increasing number of programs has shown the risks of the unsafe approach, which often leads to subtle software bugs.

This paper proposes SafeP4, a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy essential safety guarantees. We equip SafeP4 with a formal semantics and a static type system that statically guarantees header validity – a common source of safety bugs according to our analysis of real-world P4 programs. Statically ensuring header validity is challenging because the set of valid headers can be modified at runtime, making it a dynamic program property. Our type system achieves static safety by using a form of path-sensitive reasoning that tracks dynamic information from conditional statements, routing tables, and the control plane. Our evaluation shows that SafeP4's type system can effectively eliminate common failures in many real-world programs.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Networks → Programming interfaces

Keywords and phrases P4, data plane programming, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.12

Related Version `https://arxiv.org/abs/1906.07223`

Funding This work has been co-funded by the German Research Foundation (DFG) as part of the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects SA 2918/2-1 and SA 2918/3-1, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the National Science Foundation under grants CNS-1413972 and CCF-1637532, and by gifts from InfoSys and Keysight.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 12; pp. 12:1–12:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

> I couldn't resist the temptation to put in a null reference [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> – Tony Hoare

Modern languages offer features such as type systems, structured control flow, objects, modules, etc. that make it possible to express rich computations in terms of high-level abstractions rather than machine-level code. Increasingly, many languages also offer fundamental safety guarantees – e.g., well-typed programs do not go wrong [23] – that make entire categories of programming errors simply impossible.

Unfortunately, although computer networks are critical infrastructure, providing the communication fabric that underpins nearly all modern systems, most networks are still programmed using low-level languages that lack basic safety guarantees. Unsurprisingly, networks are unreliable and remarkably insecure – e.g., the first step in a cyberattack often involves compromising a router or other network device [26, 19].

Over the past decade, there has been a shift to more flexible platforms in which the functionality of the network is specified in software. Early efforts related to software-defined networking (SDN) [21, 6], focused on the control plane software that computes routes, balances load, and enforces security policies, and modeled the data plane as a simple pipeline operating on a fixed set of packet formats. However, there has been recent interest in allowing the functionality of the data plane itself to be specified as a program – e.g., to implement new protocols, make more efficient use of hardware resources, or even relocate application-level functionality into the network [15, 14]. In particular, the P4 language [4] enables the functionality of a data plane to be programmed in terms of declarative abstractions such as header types, packet parsers, match-action tables, and structured control flow that a compiler maps down to an underlying target device.

Unfortunately, while a number of P4's features were clearly inspired by designs found in modern languages, the central abstraction for representing packet data – header types – lacks basic safety guarantees. To a first approximation, a P4 header type can be thought of as a record with a field for each component of the header. For example, the header type for an IPv4 packet, would have a 4-bit version field, an 8-bit time-to-live field, two 32-bit fields for the source and destination addresses, and so on.

According to the P4 language specification, an instance of a header type may either be valid or invalid: if the instance is valid, then all operations produces a defined value, but if it is invalid, then reading or writing a field yields an undefined result. In practice, programs that manipulate invalid headers can exhibit a variety of faults including dropping the packet when it should be forwarded, or even leaking information from one packet to the next. In addition, such programs are also not portable, since their behavior can vary when executed on different targets.

The choice to model the semantics of header types in an unsafe way was intended to make the language easier to implement on high-speed routers, which often have limited amounts of memory. A typical P4 program might specify behavior for several dozen different protocols, but any particular packet is likely to contain only a small handful of headers. It follows that if the compiler only needs to represent the valid headers at run-time, then memory requirements can be reduced. However, while it may have benefits for language implementers, the design is a disaster for programmers – it repeats Hoare's "mistake," and bakes an unsafe feature deep into the design of a language that has the potential to become the de-facto standard in a multi-billion-dollar industry.

This paper investigates the design of a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy basic safety guarantees. In particular, we present SafeP4, a language with a precise semantics and a static type system that can be used to obtain guarantees about the validity of all headers read or written by the program. Although the type system is mostly based on standard features, there are several aspects of its design that stand out. First, to facilitate tracking dependencies between headers – e.g. if the TCP header is valid, then the IPv4 will also be valid – SafeP4 has an expressive algebra of types that tracks validity information at a fine level of granularity. Second, to accommodate the growing collection of extant P4 programs with only modest modifications, SafeP4 uses a path-sensitive type system that incorporates information from conditional statements, forwarding tables, and the control plane to precisely track validity.

To evaluate our design for SafeP4, we formalized the language and its type system in a core calculus and proved the usual progress and preservation theorems. We also implemented the SafeP4 type system in an OCaml prototype, P4Check, and applied it to a suite of open-source programs found on GitHub such as `switch.p4`, a large P4 program that implements the features found in modern data center switches (specifically, it includes over four dozen different switching, routing, and tunneling protocols, as well as multicast, access control lists, among other features). We categorize common failures and, for programs that fail to type-check, identify the root causes and apply repairs to make them well-typed. We find that most programs can be repaired with low effort from programmers, typically by applying a modest number of simple repairs.

Overall, the main contributions of this paper are as follows:

- We propose SafeP4, a type-safe enhancement of the P4 language that eliminates all errors related to header validity.
- We formalize the syntax and semantics of SafeP4 in a core calculus and prove that the type system is sound.
- We implement our type checker in an OCaml prototype, P4Check.
- We evaluate our type system empirically on over a dozen real-world P4 programs and identify common errors and repairs.

The rest of this paper is organized as follows. Section 2 provides a more detailed introduction to P4 and elaborates on the problems this work addresses. Section 3 presents the design, operational semantics and type system of SafeP4 and reports our type safety result. The results of evaluating SafeP4 in the wild are presented in Section 4. Section 5 surveys related work and Section 6 summaries the paper and outlines topics for future work.

2 Background and Problem Statement

This section introduces the main features of P4 and highlights the problems caused by the unsafe semantics for header types.

2.1 P4 Language

P4 is a domain-specific language designed for processing packets – i.e., arbitrary sequences of bits that can be divided into (i) a set of pre-determined *headers* that determine how the packet will be forwarded through the network, and (ii) a *payload* that encodes application-level data. P4 is designed to be protocol-independent, which means it handles both packets with standard header formats (e.g., Ethernet, IP, TCP, etc.) as well as packets with custom header formats defined by the programmer. Accordingly, a P4 program first *parses* the headers in the input packet into a typed representation. Next, it uses a *match-action pipeline*

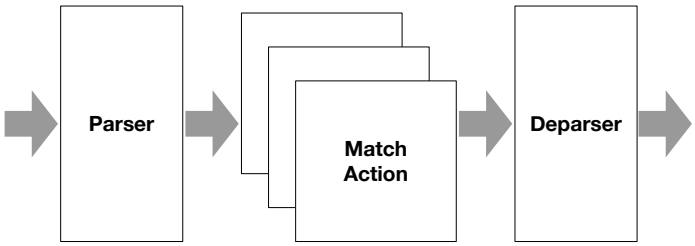


Figure 1 Abstract forwarding model.

to compute a transformation on those headers – e.g., modifying fields, adding headers, or removing them. Finally, a *deparser* serializes the headers back into into a packet, which can be output to the next device. A depiction of this abstract forwarding model is shown in Figure 1.

The match-action pipeline relies on a data structure called a *match-action table*, which encodes conditional processing. More specifically, the table first looks up the values being tested against a list of possible entries, and then executes a further snippet of code depending on which entry (if any) matched. However, unlike standard conditionals, the entries in a match-action table are not known at compile-time. Rather, they are inserted and removed at run-time by the control plane, which may be logically centralized (as in a software-defined network), or it may operate as a distributed protocol (as in a conventional network).

The rest of this section describes P4's typed representation, how the parsers, and deparsers convert between packets and this typed representation, and how control flows through the match-action pipeline.

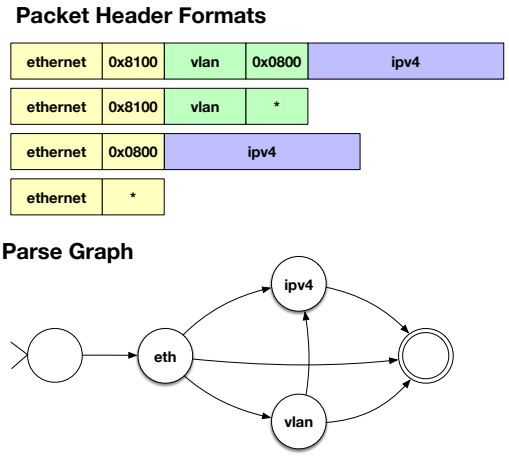
Header Types and Instances. Header types specify the internal representation of packet data within a P4 program. For example, the first few lines of the following snippet of code:

```
header_type ethernet_t {
  fields {
    dstAddr: 48;
    srcAddr: 48;
    etherType: 16;
  }
}
header ethernet_t ethernet;
header ethernet_t inner_ethernet;
```

declare a type (`ethernet_t`) for the Ethernet header with fields `dstAddr`, `srcAddr`, and `etherType`. The integer literals indicate the bit width of each field. The next two lines declare two `ethernet_t` instances (`ethernet` and `inner_ethernet`) with global scope.

Parsers. A P4 parser specifies the order in which headers are extracted from the input packet using a simple abstraction based on finite state machines. Extracting into an header instance populates its fields with the requisite bits of the input packet and marks the instance as valid. Figure 2 depicts a visual representation of a parse graph for three common headers: Ethernet, VLAN, and IPv4. The instance `ethernet` is extracted first, optionally followed by a `vlan` instance, or an `ipv4` instance, or both.

Tables and Actions. The bulk of the processing for each packet in a P4 program is performed using match-action tables that are populated by the control plane. A table (such as the one in Figure 3) is defined in terms of (i) the data it `reads` to determine a matching entry (if any), (ii) the `actions` it may execute, and (iii) an optional `default_action` it executes if no matching entry is found.




```
parser start {
  return parse_eth;
}
parser parse_eth {
 extract(ethernet);
 return select(latest.etherType){
   0x8100 : parse_vlan;
   0x0800 : parse_ipv4;
   default: ingress;
  }
}
parser parse_vlan {
  extract vlan {
  return select(latest.etherType){
    0x0800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}
```

Figure 2 (Left) Header formats and parse graph that extracts an Ethernet header optionally followed by VLAN and/or IPv4 headers. (Right) P4 code implementing the same parser.

```
table forward {
 reads {
  ipv4 : valid;
  vlan : valid;
  ipv4.dstAddr: ternary;
 }
 actions = {
  nop;
  next_hop;
  remove;
 }
 default_action : nop();
}
```

Runtime Contents of `forward`

Pattern			Action	
ipv4	vlan	ipv4.dstAddr	Name	Data
1	0	10.0.0.*	next_hop	s, d
0	1	*	remove	

Figure 3 P4 tables. `forward` reads the validity of the `ipv4` and `vlan` header instances and the `dstAddr` field of the `ipv4` header instance, and calls one of its actions: `nop`, `next_hop`, or `remove`.

The behavior of a table depends on the entries installed at run-time by the control-plane. Each table entry contains a match pattern, an action, and action data. Intuitively, the match pattern specifies the bits that should be used to match values, the action is the name of a pre-defined function (such as the ones in Figure 4), and the action data are the arguments to that function. Operationally, to process a packet, a table first scans its entries to locate the first matching entry. If such a matching entry is found, the packet is said to "hit" in the table, and the associated action is executed. Otherwise, if no matching entry is found, the packet is said to "miss" in the table, and the `default_action` (which is a no-op if unspecified) is executed.

A table also specifies the *match-kind* that describes how each header field should match with the patterns provided by the control plane. In this paper, we focus our attention on `exact`, `ternary`, and `valid` matches. An `exact` match requires the bits in the packet be exactly equivalent to the bits in the controller-installed pattern. A `ternary` match allows wildcards in arbitrary positions, so the controller-installed pattern `0*` would match bit sequences `00` and `01`. A `valid` match can only be applied to a header instance and simply checks the validity bit of that instance.

```
action next_hop(src, dst) {
 modify_field(ethernet.srcAddr, src);
 modify_field(ethernet.dstAddr, dst);
 subtract_from_field(ipv4.ttl, 1);
}
```

```
action remove() {
 modify_field(
   ethernet.etherType,
   vlan.etherType);
 remove_header(vlan);
}
```

Figure 4 P4 actions.

For example, in Figure 3, the `forward` table is shown populated with two rules. The first rule tests whether `ipv4` is valid, `vlan` is invalid, and the first 24 bits of `ipv4.srcAddr` equal `10.0.0`, and then applies `next_hop` with arguments s and d (which stand for source and destination addresses). The second rule checks that `ipv4` is invalid, then that `vlan` is valid, and skips evaluating the value of `ipv4.dstAddr` (since it is wildcarded), to finally apply the `remove` action.

Actions are functions containing sequences of primitive commands that perform operations such as adding and removing headers, assigning a value to a field, adding one field to another, etc. For example, Figure 4 depicts two actions: the `next_hop` action updates the Ethernet source and destination addresses with action data from the controller; and the `remove` action copies EtherType field from the `vlan` header instance to the `ethernet` header instance and invalidates the `vlan` header.

Control. A P4 control block can use standard control-flow constructs to execute a pipeline of match-action tables in sequence. They manage the order and conditions under which each table is executed. The `ingress` control block begins to execute as soon as the parser completes. The `apply` command executes a table and conditionals branch on a boolean expression such as the validity of a header instance.

```
control ingress {
 if(valid(ipv4) or valid(vlan)) {
  apply(forward);
 }
}
```

The above code applies the `forward` table if one of `ipv4` or `vlan` is valid.

Deparser. The deparser reassembles the final output packet, after all processing has been done by serializing each valid header instance in some order. In $P4_{14}$, the version of P4 we consider in this paper, the compiler automatically generates the deparser from the parser – i.e., for our example program, the deparser produces a packet with Ethernet, VLAN (if valid), and IPv4 (if valid), in that order.

2.2 Common Bugs in P4 Programs

Having introduced the basic features of P4, we now present five categories of bugs found in open-source programs that arise due to reading and writing invalid headers – the main problem that SafeP4 addresses. There is one category for each of the following syntactic constructs: (1) parsers, (2) controls, (3) table reads, (4) table actions, and (5) default actions.

To identify the bugs we surveyed a benchmark suite of 15 research and industrial P4 programs that are publicly available on GitHub and compile to the BMv2 [25] backend.

```
/* UNSAFE */


parser parse_ethernet {
 extract(ethernet);
 return select(ethernet.etherType)
      {
  0x0800 : parse_ipv4;
  default : ingress;

 }
}
parser parse_ipv4 {
 extract(ipv4);
 return select(ipv4.protocol) {
  6 : parse_tcp;
  default : ingress;

 }
}
```

```
/* SAFE */
parser_exception unsupported {
  parser_drop;
}
parser parse_ethernet {
 extract(ethernet);
 return select(ethernet.etherType)
      {
  0x0800 : parse_ipv4;
  default :
   parser_error unsupported;
  }
}
parser parse_ipv4 {
 extract(ipv4);
 return select(ipv4.protocol) {
 6 : parse_tcp;
 default : parser_error
    unsupported;
  }
}
```

```
parser parse_tcp {
  extract(tcp);
  return ingress;
}
```

```
control ingress {
  if(tcp.syn == 1 and ...){...}
}
```

Figure 5 Left: unsafe code in NetHCF; Right: our type-safe fix; Bottom: common code.

Later, in Section 4, we will report the number of occurrences of each of these categories in our benchmark suite detected by our approach.[1]

2.2.1 Parser Bugs

The first class of errors is due to the parser being too conservative about dropping malformed packets, which increases the set of headers that may be invalid in the control pipeline. In most programs, the parser chooses which headers to `extract` based on the fields of previously-extracted headers using P4's version of a switch statement, `select`. Programmers often fail to handle packets fall through to the `default` case of these `select` statements.

An example from the NetHCF [34, 2] codebase illustrates this bug. NetHCF is a research tool designed to combat TCP spoofing. As shown in Figure 5, the parser handles TCP packets in `parse_ipv4` and redirects all other packets to the `ingress` control. Unfortunately, the `ingress` control (bottom right) does not check whether `tcp` is valid before accessing `tcp.syn` to check whether it is equal to `1`. This is unsafe since `tcp` is not guaranteed to be valid even though it is required to be valid in the `ingress` control.

To fix this bug, we can define a parser exception, `unsupported`, with an handler that drops packets, thereby protecting the `ingress` from having to handle unexpected packets. Note however, that this fix might not be the best solution, since it alters the original behavior of the program. However, without knowing the programmer's intention, it is generally not possible to automatically repair a program with undefined behavior.

[1] We focus on $P4_{14}$ programs in this paper, but the issues we address also persist in the latest version of the language, $P4_{16}$. We did not consider $P4_{16}$ in this paper due to the smaller number of programs currently available.

```
/* UNSAFE */
control ingress {

 process_cache();
 process_value();

 apply(ipv4_route);
}
```

```
/* SAFE */
control ingress {
  if(valid(nc_hdr)) {
    process_cache();
    process_value();
  }
  apply(ipv4_route);
}
```

```
control process_cache {
    apply(check_cache_exist);
    ...
}
```

```
table check_cache_exist {
    reads { nc_hdr.key }
    actions { ... }
}
```

Figure 6 Left: unsafe code in NetCache; Right: our type-safe fix; Bottom: Common code.

2.2.2 Control Bugs

Another common bug occurs when a table is executed in a context in which the instances referenced by that table are not guaranteed to be valid. This bug can be seen in the open-source code for NetCache [13, 15], a system that uses P4 to implement a load-balancing cache. The parser for NetCache reserves a specific port (`8888`) to handle its special-purpose traffic, a condition that is built into the parser, which extracts `nc_hdr` (i.e., the NetCache-specific header) only when UDP traffic arrives from port `8888`. Otherwise, it performs standard L2 and L3 routing. Unfortunately, the `ingress` control node (Figure 6) tries to access `nc_hdr` before checking that it is valid. Specifically, the `reads` declaration for the `check_cache_exists` table, which is executed first in the `ingress` pipeline, presupposes that `nc_hdr` is valid. The invocation of the `process_value` table (not shown) contains another instance of the same bug.

To fix these bugs, we can wrap the calls to `process_cache` and `process_value` in an conditional that checks the validity of the header `nc_hdr`. This ensures that `nc_hdr` is valid when `process_cache` refers to it.

2.2.3 Table Reads Bugs

A similar bug arises in programs that contain tables that first match on the validity of certain header instances before matching on the fields of those instances. The advantage of this approach is that multiple types of packets can be processed in a single table, which saves memory. However, if implemented incorrectly, this programming pattern can lead to a bug, in which the `reads` declaration matches on bits from a header that may not be valid!

The `switch.p4` program exhibits an exemplar of this bug; it is a "realistic production switch" developed by Barefoot Networks, meant to be used "as-is, or as a starting point for more advanced switches" [18].

An archetypal example of table reads bugs is the `port_vlan_mapping` table of `switch.p4` (Figure 7). This table is invoked in a context where it is not known which of the VLAN tags is valid, despite containing references to both `vlan_tag_[0]` and `vlan_tag_[1]` in the `reads` declaration. Adroitly, the programmer has guarded the references to `vlan_tag_[`i`].vid` with keys that test the validity of `vlan_tag_[`i`]`, for $i = 1, 2$. Unfortunately, as written, it is impossible for the control plane to install a rule that will always avoid reading the value of an invalid header. The first match will check whether the `vlan_tag_[0]` instance is invalid, which is safe. However, the very next match will try to read the value of the `vlan_tag_[0].vid` field, even when the instance is invalid! This attempt to access an invalid header results in undefined behavior, and is therefore a bug.

```
/* UNSAFE */
table port_vlan_mapping {
 reads {
  vlan_tag_[0] : valid;
  vlan_tag_[0].vid : exact;
  vlan_tag_[1] : valid;
  vlan_tag_[1].vid : exact;
 } ...
}
```

```
/* SAFE */
table port_vlan_mapping {
 reads {
  vlan_tag_[0] : valid;
  vlan_tag_[0].vid : ternary;
  vlan_tag_[1] : valid;
  vlan_tag_[1].vid : ternary;
 } ...
}
```

Figure 7 Left: a table in `switch.p4` with unprotected conditional reads; Right: our type-safe fix.

It is worthy to note that this code is not actually buggy on some targets – in particular, on targets where invalid headers are initialized with `0`. However, `0`-initialization is not prescribed by the language specification, and therefore this code is not portable across other targets.

The naive solution to fix this bug is to refactor the table into four different tables (one for each combination of validity bits) and then check the validity of each header before the tables are invoked. While this fix is perfectly safe, it can result in a combinatorial blowup in the number of tables, which is clearly undesirable both for efficiency reasons and because it requires modifying the control plane.

Fortunately, rather than factoring the table into four tables, we can replace the `exact` match-kinds with `ternary` match-kinds, which permit matching with wildcards. In particular, the control plane can install rules that match invalid instances using an all-wildcard patterns, which is safe.

In order for this solution to typecheck, we need to assume that the control plane is well-behaved – i.e. that it will install wildcards for the `ternary` matches whenever the header is invalid. In our implementation, we print a warning whenever we make this kind of assumption so that the programmer can confirm that the control plane is well-behaved.

2.2.4 Table Action Bugs

Another prevalent bug, in our experience, arises when distinct actions in a table require different (and possible mutually exclusive) headers to be valid. This can lead to two problems: (i) the control plane can populate the table with unsafe match-action rules, and (ii) there may be no validity checks that we can add to the control to make all of the actions typecheck.

The `fabric_ingress_dst_lkp` table (Figure 8) in `switch.p4` provides an example of this misbehavior. The `fabric_ingress_dst_lkp` table reads the value of `fabric_hdr.dstDevice` and then invokes one of several actions: `term_cpu_packet`, `term_fabric_unicast_packet`, or `term_fabric_multicast_packet`. Respectively, these actions require the `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` (respectively) headers to be valid. Unfortunately the validity of these headers is mutually exclusive.[2]

Since `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` are mutually exclusive, there is no single context that makes this table safe. The only facility the table provides to determine which action should be called is `fabric_hdr.dstDevice`. However, the P4 program doesn't establish a relationship between the value of `fabric_hdr.dstDevice` and the validity of any of these three header instances. So, the behavior of this table is only well-defined when the input packets are well-formed, an unreasonable expectation for real switches, which may receive *any* sequence of bits "on the wire."

[2] There are other actions in the real `fabric_ingress_dst_lkp`, but these three actions demonstrate the core of the problem.

```
/* UNSAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
 }


 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

```
/* SAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
  fabric_hdr_cpu : valid;
  fabric_hdr_unicast: valid;
  fabric_hdr_multicast: valid;
 }
 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

Figure 8 Left: unsafe code in `switch.p4`; Right: our type-safe fix.

We fix this bug by including validity matches in the `reads` declaration, as shown in Figure 8. As in Section 2.2.3, this solution avoids combinatorial blowup and extensive control plane refactoring.

In order to type-check this solution, we need to make an assumption about the way the control plane will populate the table. Concretely, if an action a only typechecks if a header h is valid, and h is not necessarily valid when the table is applied, we assume that the control plane will only call a if h is matched as valid. For example, `fabric_hdr_cpu` is not known to be valid when (the fixed version of) `fabric_ingress_dst_lkp` is applied, so we assume that the control plane will only call action `term_cpu_packet` when `fabric_hdr_cpu` is matched as valid. Again, our implementation prints these assumptions as warnings to the programmer, so they can confirm that the control plane will satisfy these assumptions.

2.2.5 Default Action Bugs

Finally, the *default action* bugs occur when the programmer incorrectly assumes that a table performs some action when a packet misses. The NetCache program (described in Section 2.2.2) exhibits an example of this bug, too. The bug is shown in Figure 9, where the table `add_value_header_1` is expected to make the `nc_value_1` header valid, which is done in the `add_value_header_1_act` action. The control plane may refuse to add any rules to the table, which would cause all packets to miss, meaning that the `add_value_header_1_act` action would never be called and `nc_value_1` may not be valid. To fix this error, we simply set the default action for the table to `add_value_header_1_act`, which will force the table to remove the header no matter what rules the controller installs.

```
/* UNSAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }

}
```

```
/* SAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }
 default_action :
    add_value_header_1_act();
}
```

Figure 9 Left: unsafe code in NetCache; Right: our type-safe fix.

```
if(ethernet.etherType == 0x0800) {
 apply(ipv4_table);
} else if(ethernet.etherType == 0x086DD) {
 apply(ipv6_table);
}
```

```
if(valid(ipv4)) {
 apply(ipv4_table);
} else if(valid(ipv6)) {
 apply(ipv6_table);
}
```

Figure 10 Left: data-dependent header validation; Right: syntactic header validation.

2.3 A Typing Discipline to Eliminate Invalid References

In this paper, we propose a type system to increase the safety of P4 programs by detecting and preventing the classes of bugs defined in Section 2.2. These classes of bugs all manifest when a program attempts to access an invalid header – differentiating themselves only in their syntactic provenance. The type system that we present in the next section uses a path-sensitive analysis, coupled with occurrence typing [32], to keep track of which headers are guaranteed to be available at any program point – rejecting programs that reference headers that *might* be uninitialized – thus, preventing all references to invalid headers.

Of course, in general, the problem of deciding header-validity can depend on arbitrary data, so a simple type system cannot hope to fully determine all scenarios when an instance will be valid. Indeed, programmers often use a variety of data-dependent checks to ensure safety. For instance, the control snippet shown on the left-hand side of Figure 10 will not produce undefined behavior, given a parser that chooses between parsing an `ipv4` header when `ethernet.etherType` is `0x0800`, an `ipv6` header when `ethernet.etherType` is `0x86DD`, and throws a parser error otherwise.

While this code is safe in this very specific context, it quickly becomes unsafe when ported to other contexts. For example in `switch.p4`, which performs tunneling, the egress control node copies the `inner_ethernet` header into the `ethernet`; however the `inner_ethernet` header may not be valid at the program point where the copy is performed. This behavior is left undefined [7], a target is free to read arbitrary bits, in which case it could decide to call the `ipv4_table` despite `ipv4` being invalid.

To improve the maintainability and portability of the code, we can replace the data-dependent checks with validity checks, as illustrated by the control snippet shown on the right-hand side of Figure 10. The validity checks assert precisely the preconditions for calling each table, so that no matter what context this code snippet is called in, it is impossible for the `ipv4_table` to be called when the `ipv4` header is invalid.

In the next section, we develop a core calculus for SafeP4 with a type system that eliminates references to invalid headers, encouraging programers to replace data-dependent checks with header-validity checks.

3 SafeP4

This section discusses our design goals for SafeP4 and the choices we made to accommodate them, and formalizes the language's syntax, small-step semantics, and type system.

3.1 Design

Our primary design goal for SafeP4 is to develop a core calculus that models the main features of $P4_{14}$ and $P4_{16}$, while guaranteeing that all data from packet headers is manipulated in a safe and well-defined manner. We draw inspiration from Featherweight Java [12] – i.e., we model the essential features of P4, but prune away unnecessary complexity. The result

is a minimal calculus that is easy to reason about, but can still express a large number of real-world data plane programs. For instance, P4 and SafeP4 both achieve protocol independence by allowing the programmer to specify the types of packet headers and their order in the bit stream. Similarly, SafeP4 mimics P4's use of tables to interface with the control-plane and decide which actions to execute at run-time.

So what features does SafeP4 prune away? We omit a number of constructs that are secondary to how packets are processed – e.g., `field_list_calculations`, `parser_exceptions`, `counters`, `meters`, `action profiles`, etc. It would be relatively straightforward to add these to the calculus – indeed, most are already handled in our prototype – at the cost of making it more complicated. We also modify or distill several aspects of P4. For instance, P4 separates the parsing phase and the control phase. Rather than unnecessarily complicating the syntax of SafeP4, we allow the syntactic objects that represent parsers and controls to be freely mixed. We make a similar simplification in actions, informally enforcing which primitive commands can be invoked within actions (e.g., field modification, but not conditionals).

Another challenge arises in trying to model core behaviors of both $P4_{14}$ and $P4_{16}$, in that they each have different type systems and behaviors for evaluating expressions. Our calculus abstracts away expression typing and syntax variants by assuming that we are given a set of constants k that can represent values like `0` or `True`, or operators such as `&&` and `?:`. We also assume that these operators are assigned appropriate (i.e., sound) types. With these features in hand, one can instantiate our type system over arbitrary constants.

Another departure from P4 is related to the *add* command, which presents a complication for our expression types. The analogous `add_header` action in $P4_{14}$ simply modifies the validity bit, without initializing any of the fields. This means that accessing any of the header fields before they have been manually initialized reads a non-deterministic value. Our calculus neatly sidesteps this issue by defining the semantics of the $add(h)$ primitive to initialize each of the fields of h to a default value. We assume that along with our type constants there is a function *init* that accepts a header type η and produces a header instance of type η with all fields set to their default value. Note that we could have instead modified our type system to keep track of the definedness of header fields as well as their validity. However, for simplicity we choose to focus on header validity in this paper.

The portion of our type system that analyzes header validity, requires some way of keeping track of which headers are valid. Naively, we can keep track of a set of which headers are guaranteed to be valid on all program paths, and reject programs that reference headers not in this set. However, this coarse-grained approach would lead to a large number of false positives. For instance, the parser shown in Figure 2 parses an `ethernet` header and then either boots to `ingress` or parses an `ipv4` header and then either proceeds to the `ingress` or parses an `vlan` header. Hence, at the `ingress` node, the only header that is guaranteed to be valid is the `ethernet` header. However, it is certainly safe to write an `ingress` program that references the `vlan` header after checking it was valid. To reflect this in the type system we introduce a special construct called $valid(h)\ c_1\ else\ c_2$, which executes c_1 if h is valid and c_2 otherwise. When we type check this command, following previous work on occurrence typing [32], we check c_1 with the additional fact that h is valid, and we check c_2 with the additional fact that h is not valid.

Even with this enhancement, this type system would still be overly restrictive. To see why, let us augment the parser from Figure 2 with the ability to parse TCP and UDP packets: after parsing the `ipv4` header, the parser can optionally extract the `vlan`, `tcp`, or `udp` header and then boot control flow to ingress. Now suppose that we have a table `tcp_table` that refers to both `ipv4` and `tcp` in its `reads` declaration, and that `tcp_table`

is (unsafely) applied immediately in the `ingress`. Because the validity of `tcp` implies the validity of `ipv4`, it should be safe to check the validity of `tcp` and then apply `tcp_table`. However, using the representation of valid headers as a set, we would need to ascertain the validity of `ipv4` and of `tcp`.

To solve this problem, we enrich our type representation to keep track of dependencies between headers. More specifically, rather than representing all headers guaranteed to be valid in a set, we use a finer-grained representation – a set of sets of headers that might be valid at the current program point. For a given header reference to be safe, it must to be a member of all possible sets of headers – i.e., it must be valid on all paths through the program that reach the reference.

Overall, the combination of an expressive language of types and a simple version of occurrence typing allows us to capture dependencies between headers and perform useful static analysis of the dynamic property of header validity.

The final challenge with formally modelling P4 lies in its interface with the control-plane, which populates the tables and provides arguments to the actions. While the control-plane's only methodology for managing switch behavior is to populate the match-action tables with forwarding entries, it is perfectly capable of producing undefined behavior. However, if we assume that the controller is well-intentioned, we can prove the safety of more programs.

In our formalization, to streamline the presentation, we model the control plane as a function $\mathcal{CA}(t, H) = (a_i, \bar{v})$ that takes in a table t and the current headers H and produces the action to call a_i and the (possibly empty) action data arguments $\bar{v}$. We also use a function $\mathcal{CV}(t) = \bar{S}$ that analyzes a table t and produces a list of sets of valid headers $\bar{S}$, one set for each action, that can be safely assumed valid when the entries are populated by the control plane. From the table declaration and the header instances that can be assumed valid, based on the match-kinds, we can derive a list of match key expressions $\bar{e}$ that must be evaluated when the table is invoked. Together, these functions model the run-time interface between the switch and the controller. In order to prove progress and preservation, we assume that $\mathcal{CV}$ and $\mathcal{CA}$ satisfy three simple correctness properties: (1) the control plane can safely install table entries that never read invalid headers, (2) the action data provided by the control plane has the types expected by the action, and (3) the control plane will only assume valid headers for an action that are valid for a given packet. See technical report for details.

3.2 Syntax

The syntax of SafeP4 is shown in Figure 11. To lighten the notation, we write $\bar{x}$ as shorthand for a (possibly empty) sequence $x_1, ..., x_n$.

A SafeP4 program consists of a sequence of declarations $\bar{d}$ and a command c. The set of declarations includes header types, header instances, and tables. Header type declarations describe the format of individual headers and are defined in terms of a name and a sequence of field declarations. The notation "$f : \tau$" indicates that field f has type τ. We let η range over header types. A header instance declaration assigns a name h to a header type η. The map $\mathcal{HT}$ encodes the (global) mapping between header instances and header types. Table declarations $t(\bar{h}, (e, m), \bar{a})$, are defined in terms of a sequence of valid-match header instances $\bar{h}$, a sequence of match-key expressions $\overline{(e, m)}$ read in the table, where e is an expression and m is the match-kind used to match this expression, and a sequence of actions $\bar{a}$. The notation $t.valids$ denotes the valid-match instances, $t.reads$ denotes the expressions, and $t.actions$ denotes the actions.

Actions are written as (uncurried) λ-abstractions. An action $\lambda\bar{x}.\ c$ declares a (possibly empty) sequence of parameters, drawn from a fresh set of names, which are in scope for the command c. The run-time arguments for actions (action data) are provided by the control

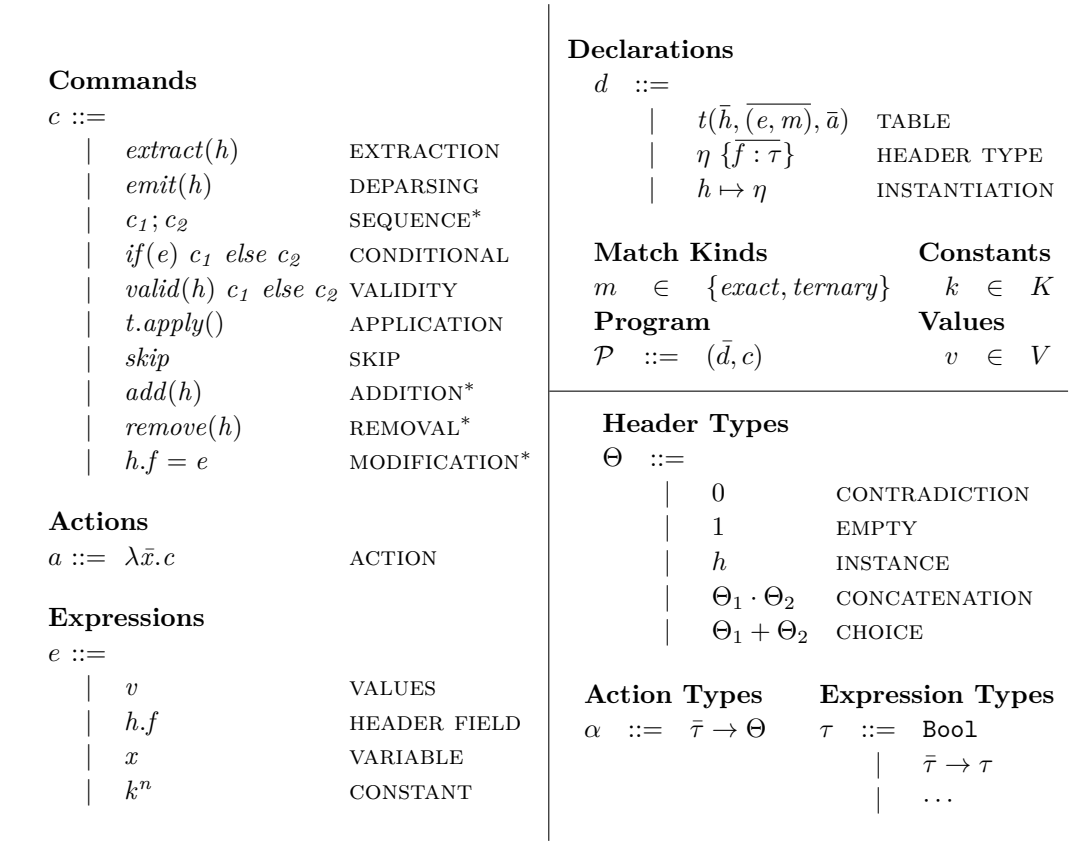


Figure 11 Syntax of SafeP4.

plane. Note that we artificially restrict the commands that can be called in the body of the action to addition, removal, modification and sequence; these actions are identified with an asterisk in Figure 11.

The calculus provides commands for extracting (*extract*), creating (*add*), removing (*remove*), and modifying ($h.f = e$) header instances. The *emit* command is used in the deparser and serializes a header instance back into a bit sequence (*emit*). The *if*-statement conditionally executes one of two commands based on the value of a boolean condition. Similarly, the *valid*-statement branches on the validity of h. Table application commands ($t.apply()$) are used to invoke a table t in the current state. The *skip* command is a no-op.

The only built-in expressions in SafeP4 are variables x and header fields, written $h.f$. We let v range over values and assume a collection of n-ary constant operators $k^n \in K$.

For simplicity, we assume that every header referenced in an expression has a corresponding instance declaration. We also assume that header instance names h, header type names η, variable names x, and table names t are drawn from disjoint sets of names H,E,V, and T respectively and that each name is declared only once.

3.3 Type System

SafeP4 provides two main kinds of types, basic types τ and header types Θ as shown in Figure 11. We assume that the set of basic types includes booleans (for conditionals) as well as tuples and function types (for actions).

$$
\begin{aligned}
[\![\Theta]\!] &\subseteq \mathcal{P}(\mathit{Header}) \\
[\![0]\!] &= \{\} \\
[\![1]\!] &= \{\{\}\} \\
[\![h]\!] &= \{\{h\}\} \\
[\![\Theta_1 \cdot \Theta_2]\!] &= [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] \\
[\![\Theta_1 + \Theta_2]\!] &= [\![\Theta_1]\!] \cup [\![\Theta_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}(h, f_i) &= \tau_i && \textit{Field lookup} \\
\mathcal{A}(a) &= \lambda \bar{x} : \bar{\tau}.\ c && \textit{Action lookup} \\
\mathcal{CA}(t, H) &= (a_i, \bar{v}) && \textit{Control-plane actions} \\
\mathcal{CV}(t) &= \bar{S} && \textit{Control-plane validity} \\
\mathcal{H}(e) &= \bar{h} && \textit{Referenced Header instances}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{maskable}(t, e, \mathit{exact}) &\triangleq \mathit{false} \\
\mathsf{maskable}(t, e, \mathit{ternary}) &\triangleq \mathcal{H}(e) \subseteq t.\mathit{valids}
\end{aligned}
$$

Figure 12 Semantics of header types (left) and auxiliary functions (right).

A header type Θ represents a set of possible co-valid header instances. The type 0 denotes the empty set. This type arises when there are unsatisfiable assumptions about which headers are valid. The type 1 denotes the singleton denoting the empty set of headers. It describes the type of the initial state of the program. The type h denotes a singleton set, $\{\{h\}\}$ – i.e., states where only h is valid. The type $\Theta_1 \cdot \Theta_1$ denotes the set obtained by combining headers from Θ_1 and Θ_2 – i.e., a product or concatenation. Finally, the type $\Theta_1 + \Theta_2$ denotes the union of Θ_1 or Θ_2, which intuitively represents an alternative.

The semantics of header types, $[\![\Theta]\!]$, is defined by the equations in Figure 12. Intuitively, each subset represents one alternative set of headers that may be valid. For example, the header type $\texttt{eth} \cdot (\texttt{ipv4} + 1)$ denotes the set $\{\{\texttt{eth}, \texttt{ipv4}\}, \{\texttt{eth}\}\}$.

To formulate the typing rules for SafeP4, we also define a set of operations on header types: `Restrict`, `NegRestrict`, `Includes`, `Remove`, and `Empty`. The restrict operator `Restrict` Θ h recursively traverses Θ and keeps only those choices in which h is contained, mapping all others to 0. Semantically this has the effect of throwing out the subsets of $[\![\Theta]\!]$ that do not contain h. Dually `NegRestrict` Θ h produces only those choices/subsets where h is invalid. `Includes` Θ h traverses Θ and checks that h is always valid. Semantically this says that h is a member of every element of $[\![\Theta]\!]$. `Remove` Θ h removes h from every path, which means, semantically that it removes h from ever element of $[\![\Theta]\!]$. Finally, `Empty` Θ checks whether Θ denotes the empty set. We can lift these operators to operate on sets of headers in the obvious way. An in-depth treatment of these operators can be found in the accompanying technical report.

3.3.1 Typing Judgement

The typing judgement has the form $\Gamma \vdash c : \Theta \Rightarrow \Theta'$, which means that in variable context Γ, if c is executed in the header context Θ, then a header instance type Θ' is assigned. Intuitively, Θ encodes the sets of headers that may be valid when type checking a command. Γ is a standard type environment which maps variables x to type τ. If there exists Θ' such that $\Gamma \vdash c : \Theta \Rightarrow \Theta'$, we say that c is well-typed in Θ.

The typing rules rely on several auxiliary definitions shown in Figure 12. The field type lookup function $\mathcal{F}(h, f_i)$ returns the type assigned to a field f_i in header h by looking it up from the global header type declarations via the header instance declarations. The action lookup function $\mathcal{A}(a)$ returns the action definition $\lambda \bar{x} : \bar{\tau}.\ c$ for action a. Finally, the function $\mathcal{CA}(t, H)$ computes the run-time actions for table t, while $\mathcal{CV}(t)$ computes t's assumptions about validity. Both of these are assumed to be instantiated by the control plane in a way that satisfies basic correctness properties – see technical report.

$$\textsc{T-Zero}\ \frac{\texttt{Empty}\ \Theta_1}{\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2} \qquad \textsc{T-Skip}\ \frac{}{\Gamma \vdash \mathit{skip} : \Theta \Rightarrow \Theta}$$

$$\textsc{T-Seq}\ \frac{\Gamma \vdash c_1 : \Theta \Rightarrow \Theta_1 \qquad \Gamma \vdash c_2 : \Theta_1 \Rightarrow \Theta_2}{\Gamma \vdash c_1; c_2 : \Theta \Rightarrow \Theta_2}$$

$$\textsc{T-If}\ \frac{\Gamma; \Theta \vdash e : \mathit{Bool} \qquad \Gamma \vdash c_1 : \Theta \Rightarrow \Theta_1 \qquad \Gamma \vdash c_2 : \Theta \Rightarrow \Theta_2}{\Gamma \vdash \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2 : \Theta \Rightarrow \Theta_1 + \Theta_2}$$

$$\textsc{T-IfValid}\ \frac{\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta\ h \Rightarrow \Theta_1 \qquad \Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta\ h \Rightarrow \Theta_2}{\Gamma \vdash \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2 : \Theta \Rightarrow \Theta_1 + \Theta_2}$$

$$\textsc{T-Mod}\ \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau_i \qquad \Gamma; \Theta \vdash e : \tau_i}{\Gamma \vdash h.f = e : \Theta \Rightarrow \Theta}$$

$$\textsc{T-Extr}\ \frac{}{\Gamma \vdash \mathit{extract}(h) : \Theta \Rightarrow \Theta \cdot h} \qquad \textsc{T-Emit}\ \frac{}{\Gamma \vdash \mathit{emit}(h) : \Theta \Rightarrow \Theta}$$

$$\textsc{T-Add}\ \frac{}{\Gamma \vdash \mathit{add}(h) : \Theta \Rightarrow \Theta \cdot h} \qquad \textsc{T-Rem}\ \frac{}{\Gamma \vdash \mathit{remove}(h) : \Theta \Rightarrow \texttt{Remove}\ \Theta\ h}$$

$$\textsc{T-Apply}\ \frac{\begin{array}{c}\mathcal{CV}(t) = \bar{S} \qquad t.\mathit{actions} = \bar{a} \qquad t.\mathit{reads} = \bar{r} \\ \bar{e} = \{e_j \mid (e_j, m_j) \in \bar{r} \wedge \neg \mathsf{maskable}(t, e_j, m_j)\} \\ \cdot; \Theta \vdash e_j : \tau_j \quad \text{for } e_j \in \bar{e} \\ \texttt{Restrict}\ \Theta\ S_i \vdash a_i : \bar{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \bar{a}\end{array}}{\Gamma \vdash t.\mathit{apply}() : \Theta \Rightarrow \left(\sum_{a_i \in \bar{a}} \Theta'_i\right)}$$

Figure 13 Command typing rules for SafeP4.

The typing rules for commands are presented in Figure 13. The rule T-Zero gives a command an arbitrary output type if the input type is empty. It is needed to prove preservation. The rules T-Skip and T-Seq are standard. The rule T-If a path-sensitive union type between the type computed for each branch. The rule T-IfValid is similar, but leverages knowledge about the validity of h. So the true branch c_1 is checked in the context `Restrict` Θ h, and the false branch c_2 is checked in the context `NegRestrict` Θ h. The top-level output type is the union of the resulting output types for c_1 and c_2. The rule T-Mod checks that h is guaranteed to be valid using the `Includes` operator, and uses the auxiliary function $\mathcal{F}$ to obtain the type assigned to $h.f$. Note that the set of valid headers does not change when evaluating an assignment, so the output and input types are identical. The rules T-Extr and T-Add assign header extractions and header additions the type $\Theta \cdot h$, reflecting the fact that h is valid after the command executes. Emitting packet headers does not change the set of valid headers, which is captured by rule T-Emit. The typing rule T-Rem uses the `Remove` operator to remove h from the input type Θ. Finally, the rule T-Apply checks table applications. To understand how it works, let us first consider a simpler, but less precise, typing rule:

$$\frac{\begin{array}{c}t.\mathit{reads} = \bar{e} \qquad \cdot; \Theta \vdash e_i : \tau_i \quad \text{for } e_i \in \bar{e} \\ t.\mathit{actions} = \bar{a} \qquad \cdot; \Theta \vdash a_i : \bar{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \bar{a}\end{array}}{\cdot \vdash t.\mathit{apply}() : \Theta \Rightarrow \left(\sum \Theta'_i\right)}$$

Intuitively, this rule says that to type check a table application, we check each expression it reads and each of its actions. The final header type is the union of the types computed for

$$\frac{\Gamma, \bar{x} : \bar{\tau} \vdash c : \Theta \Mapsto \Theta'}{\Gamma; \Theta \vdash \lambda\ \bar{x} : \bar{\tau}.c : \bar{\tau} \to \Theta'} \quad \text{(T-Action)}$$

Figure 14 Action typing rule for SafeP4.

$$\text{T-Const}\ \frac{\texttt{typeof}(k) = \bar{\tau} \to \tau' \qquad \Gamma; \Theta \vdash e_i : \tau_i}{\Gamma; \Theta \vdash k(\bar{e}) : \tau'} \qquad \text{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma; \Theta \vdash x : \tau} \qquad \text{T-Field}\ \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau}{\Gamma; \Theta \vdash h.f : \tau}$$

Figure 15 Expression typing rules for SafeP4.

the actions. To put it another way, it models table application as a non-deterministic choice between its actions. However, while this rule is sound, it is overly conservative. In particular, it does not model the fact that the control plane often uses header validity bits to control which actions are executed.

Hence, the actual typing rule, T-Apply, is parameterized on a function $\mathcal{CV}(t)$ that models the choices made by the control plane, returning for each action a_i, a set of headers S_i that can be assumed valid when type checking a_i. From the reads declarations of the table declaration, we can derive a subset of the expressions read by the table – e.g., excluding expressions that can be wildcarded when certain validity bits are false. This is captured by the function $\mathsf{maskable}(t, e, m)$ (defined in Figure 12) , which determines whether a reads expression e with match-kind m in table t can be masked using a wild-card. The maskable function is defined using $\mathcal{H}(e)$, which returns the set of header instances referenced by an expression e.

In the example from Section 2.2.3, if an action a_j is matched by the rule $(0, *, 0, *)$, both S_j and e_j are empty.

The typing judgement for actions (Figure 14) is of the form $\Gamma; \Theta \vdash a : \bar{\tau} \to \Theta$, meaning that a has type $\bar{\tau} \to \Theta$ in variable context Γ and header context Θ. Given a variable context Γ and header type Θ, an action $\lambda\bar{x}.\ c$ encodes a function of type $\bar{\tau} \to \Theta'$, so long as the body c is well-typed in the context where Γ is extended with $x_i : \tau_i$ for every i.

The typing rules for expressions are shown in Figure 15. Constants are typechecked according to rule T-Constant, as long as each expression that is passed as an argument to the constant k has the type required by the `typeof` function. The rule T-Var is standard.

3.4 Operational Semantics

We now present the small-step operational semantics of SafeP4. We define the operational semantics for commands in terms of four-tuples $\langle I, O, H, c\rangle$, where I is the input bit stream (which is assumed to be infinite for simplicity), O is the output bit stream, H is a map that associates each valid header instance with a records containing the values of each field, and c is the command to be evaluated. The reduction rules are presented in Figure 16.

The command $extract(h)$ evaluates via the rule E-Extr, which looks up the header type in $\mathcal{HT}$ and then invokes corresponding deserialization function. The deserialized header value v is added to to the map of valid header instances, H. For example, assuming the header type $\eta = \{f : bit\langle 3\rangle;\ g : bit\langle 2\rangle;\}$ has two fields f and g and $I = 11000B$ where B is the rest of the bit stream following, then $deserialize_\eta(I) = (\{f = 110;\ g = 00;\}, B)$.

E-Extr
$$\frac{\mathcal{HT}(h) = \eta \qquad deserialize_\eta(I) = (v, I')}{\langle I, O, H, extract(h)\rangle \to \langle I', O, H[h \mapsto v], skip\rangle}$$

E-Emit
$$\frac{\mathcal{HT}(h) = \eta \qquad serialize_\eta(H(h)) = \bar{B}}{\langle I, O, H, emit(h)\rangle \to \langle I, O.\bar{B}, H, skip\rangle}$$

E-EmitInvalid
$$\frac{h \notin dom(H)}{\langle I, O, H, emit(h)\rangle \to \langle I, O, H, skip\rangle}$$

E-IfValidTrue
$$\frac{h \in dom(H)}{\langle I, O, H, valid(h)\ c_1\ else\ c_2\rangle \to \langle I, O, H, c_1\rangle}$$

E-IfValidFalse
$$\frac{h \notin dom(H)}{\langle I, O, H, valid(h)\ c_1\ else\ c_2\rangle \to \langle I, O, H, c_2\rangle}$$

E-Mod
$$\frac{H(h) = r \qquad r' = \{r\ with\ f = v\}}{\langle I, O, H, h.f = v\rangle \to \langle I, O, H[h \mapsto r'], skip\rangle}$$

E-Apply
$$\frac{\mathcal{CA}(t, H) = (a_i, \bar{v}) \qquad \mathcal{A}(a_i) = \lambda\bar{x}.c_i}{\langle I, O, H, t.apply()\rangle \to \langle I, O, H, c_i[\bar{v}/\bar{x}]\rangle}$$

E-Add
$$\frac{\mathcal{HT}(h) = \eta \qquad init_\eta = v}{\langle I, O, H, add(h)\rangle \to \langle I, O, H[h \mapsto v], skip\rangle}$$

E-AddValid
$$\frac{h \in dom(H)}{\langle I, O, H, add(h)\rangle \to \langle I, O, H, skip\rangle}$$

E-Rem
$$\frac{}{\langle I, O, H, remove(h)\rangle \to \langle I, O, H \setminus h, skip\rangle}$$

Figure 16 Selected rules of the operational semantics of SafeP4; the elided rules are standard and can be found in the technical report.

E-Const
$$\frac{[\![k]\!](v_1, ..., v_n) = v}{\langle H, k(v_1, ..., v_n)\rangle \to v}$$

E-Field
$$\frac{H(h) = \{f_1 : n_1, ..., f_k : n_k\}}{\langle H, h.f_i\rangle \to n_i}$$

Figure 17 Selected rules of the operational semantics for expressions.

The rule E-Emit serializes a header instance h back into a bit stream. It first looks up the corresponding header type and header value in the header table $\mathcal{HT}$ and the map of valid headers respectively. The header value is then passed to the serialization function for the header type to produce a bit sequence that is appended to the output bit stream. Similarly, we assume that a serialization function is defined for every header type, which takes the bit values of the fields of a header value and concatenates them to produce a single bit sequence. We adopt the semantics of P4 with respect to emitting invalid headers. Emitting an invalid header instance – i.e., a header instance which has not been added or extracted – has no effect on the output bit stream (rule E-EmitInvalid). Notice also that the header remains unchanged in H.

Sequential composition reduces left to right, i.e., the left command needs to be reduced to *skip* before the right command can be reduced (rule E-Seq). The evaluation of conditionals (rules E-If, E-IfTrue, E-IfFalse) is standard. Both E-Seq, E-If, E-IfTrue and E-IfFalse are relegated to the technical report for brevity. The rules for validity checks (E-IfValidTrue, E-IfValidFalse) step to the true branch if $h \in dom(H)$ and to the false branch otherwise.

Table application commands are evaluated according to rule E-Tapply. We first invoke the control plane function $\mathcal{CA}(t, H)$ to determine an action a_i and action data v. Then we

$$\textsc{Ent-Empty}\ \frac{}{\cdot \models 1} \qquad \textsc{Ent-Inst}\ \frac{dom(H) = \{h\}}{H \models\ h} \qquad \textsc{Ent-Seq}\ \frac{H_1 \models \Theta_1 \quad H_2 \models \Theta_2}{H_1 \cup H_2 \models \Theta_1 \cdot \Theta_2}$$

$$\textsc{Ent-ChoiceL}\ \frac{H \models \Theta_1}{H \models \Theta_1 + \Theta_2} \qquad \textsc{Ent-ChoiceR}\ \frac{H \models \Theta_2}{H \models \Theta_1 + \Theta_2}$$

Figure 18 The *Entailment* relation between header instances and header instance types.

use $\mathcal{A}$ to lookup the definition of a_i, yielding $\lambda\bar{x} : \bar{\tau}.\ c_i$ and step to $c_i[\bar{v}/\bar{x}]$. Note that for simplicity, we model the evaluation of expressions read by the table using the control-plane function $\mathcal{CA}$.

The rule E-Add evaluates addition commands $add(h)$. Similar to header extraction, the $init_\eta()$ function produces a header instance v of type η with all fields set to a default value and extends the map H with $h \mapsto v$. Note that according to E-Add-Exist, if the header instance is already valid, $add(h)$ does nothing. Finally, the rule E-Rem removes the header from the map H. Again, if a header h is already invalid, removing it has no effect.

The semantics for expressions is defined in Figure 17, using tuples $\langle H, e\rangle$, where H is the same map used in the semantics of commands and e is the expression to evaluate. The rule E-Field reduces header field expressions to the value stored in the heap H for the respective field. To evaluate constants via the rule E-Const (omitting the obvious congruence rule), we assume that there is an evaluation function for constants $[\![k]\!](\bar{v}) = v$ that is well-behaved – i.e., if $\texttt{typeof}(k) = \bar{\tau} \to \tau'$ and $\overline{v : \tau}$, then $.;. \vdash [\![k]\!](\bar{v}) : \tau'$. We use these facts to prove progress and preservation.

3.5 Safety of SafeP4

We prove safety in terms of progress and preservation. Both theorems make use of the relation $H \models \Theta$ which intuitively holds if H is described by Θ. The formal definition, as given in Figure 18, satisfies $H \models \Theta$ if and only if $dom(H) \in [\![\Theta]\!]$.

We prove type safety via progress and preservation theorems. The respective proofs are mostly straightforward for our system – we highlight the unusual and nontrivial cases below an relegate the full proofs to the technical report.

▶ **Theorem 1** (Progress). *If* $\cdot \vdash c : \Theta \Rightarrow \Theta'$ *and* $H \models \Theta$*, then either,*
- $c = skip$*, or*
- $\exists \langle I', O', H', c'\rangle.\ \langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$.

Intuitively, progress says that a well-typed command is fully reduced or can take a step.

▶ **Theorem 2** (Preservation). *If* $\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2$ *and* $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$*, where* $H \models \Theta_1$*, then* $\exists \Theta_1', \Theta_2'.\ \Gamma \vdash c : \Theta_1' \Rightarrow \Theta_2'$ *where* $H' \models \Theta_1'$ *and* $\Theta_2' < \Theta_2$.

More interestingly, preservation says that if a command c is well-typed with input type Θ_1 and output type Θ_2, and c evaluates to c' in a single step, then there exists an input type Θ_1' and an output type Θ_2' that make c' well-typed. To make the inductive proof go through, we also need to prove that Θ_1' describes the same maps of header instance H as Θ_1, and Θ_2' is semantically contained in Θ_2. We define syntactic containment to be $\Theta_1 < \Theta_2 \triangleq [\![\Theta_1]\!] \subseteq [\![\Theta_2]\!]$. (These conditions are somewhat reminiscent of conditions found in languages with subtyping.)

Proof. By induction on a derivation of $\Gamma \vdash c : \Theta_1 \Mapsto \Theta_2$, with a case analysis on the last rule used. We focus on two of the most interesting cases. See technical report for the full proof.

Case T-IfValid: $c = \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Mapsto \Theta_{12}$ and $\Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Mapsto \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$.

There are two evaluation rules that apply to c, E-IfValidTrue and E-IfValidFalse

Subcase E-IfValidTrue: $c' = c_1$ and $h \in \mathit{dom}(H)$ and $H' = H$.

Let $\Theta_1' = \texttt{Restrict}\ \Theta_1\ h$ and $\Theta_2' = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta_1' \Mapsto \Theta_2'$ by assumption, we have $H \models \Theta_1'$ by a lemma formalizing the relationship between Restrict and ($\models$) (see tech report), and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union.

Subcase E-IfValidFalse: $c' = c_2$ and $h \notin \mathit{dom}(H)$ and $H' = H$.

Symmetric to the previous case.

Case T-Apply: $c = t.\mathit{apply}()$ and $\mathcal{CV}(t) = (\bar{S}, \bar{e})$ and $t.\mathit{actions} = \bar{a}$ and $\cdot; \Theta \vdash e_j : \tau_j$ for $e_j \in \bar{e}$ and $\texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau_i} \to \Theta_i'$ for $a_i \in \bar{a}$ and $\Theta_2 = \sum(\Theta_i')$

Only one evaluation rule applies to c, E-Apply. It follows that $\mathcal{CA}(t, H) = (a_i, \bar{v})$, and $c' = c_i[\bar{v}/\bar{x}]$ where $\mathcal{A}(a_i) = \lambda \bar{x}.\ c_i$. By inverting T-Action, we have $\Gamma, \bar{x} : \bar{\tau_i}; \vdash c_i : \texttt{Restrict}\ \Theta\ S_i \Mapsto \Theta_i'$. By control plane assumption (2), we have $\cdot; \cdot \vdash \bar{v} : \bar{\tau_i}$. By the substitution lemma, we have $\Gamma \vdash c_i[\bar{v}/\bar{x}] : \texttt{Restrict}\ \Theta\ S_i \Mapsto \Theta_i'$. Let $\Theta_1' = \texttt{Restrict}\ \Theta\ S_i$ and $\Theta_2' = \Theta_i'$. We have shown that $\Gamma \vdash c' : \Theta_1' \Mapsto \Theta_2'$, we have that $H' \models \Theta_1'$ by control plane assumption (3), and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union types. ◀

4 Experience (Evaluation)

We implemented our type system in a tool called P4Check that automatically checks P4 programs and reports violations of the type system presented in Figure 13. P4Check uses the front-end of `p4v` [20] and handles the full $P4_{14}$ language.[3] Our key findings, which are reported in detail below, show (i) that our type system finds bugs "in the wild" and (ii) that the programmer effort needed to repair programs to pass our type checker is modest.

4.1 Overview of Bugs in the Wild

We ran P4Check on 15 open source $P4_{14}$ programs[4] of varying sizes and complexity, ranging from 143 to 9060 lines of code. Our criteria for selecting programs was: (1) each program had to be open source, (2) available on GitHub, and (3) compile without errors, (4) and be written either by industrial teams developing production code or by researchers implementing standard or novel network functionality in P4 – i.e., we excluded programs primarily used for teaching. Out of the 15 subject programs only 4 passed our type checker, all of which were simple implementations of routers or DDoS mitigation that accepted only a small number of packet types and were relatively small (188–635 lines of code). For the remaining 11 programs (industrial and research) our checker found 418 type checking violations overall.

[3] We also have an open-source prototype implementation for $P4_{16}$ that handles the most common features of $P4_{16}$ (`https://github.com/cornell-netlab/p4check`).

[4] We chose to check $P4_{14}$ instead of $P4_{16}$, since there are currently more $P4_{14}$ programs available on GitHub.

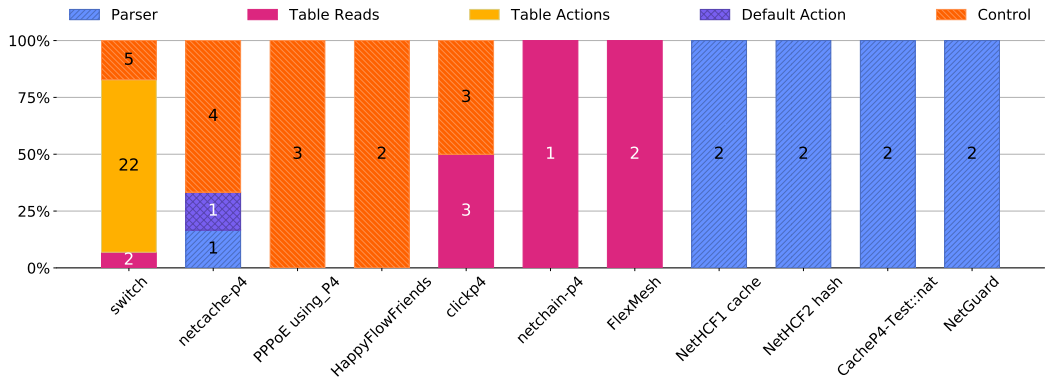


Figure 19 Proportional frequencies of each bug type per-program. The raw number of bugs for each program and category is reported at the top of each stacked bar.

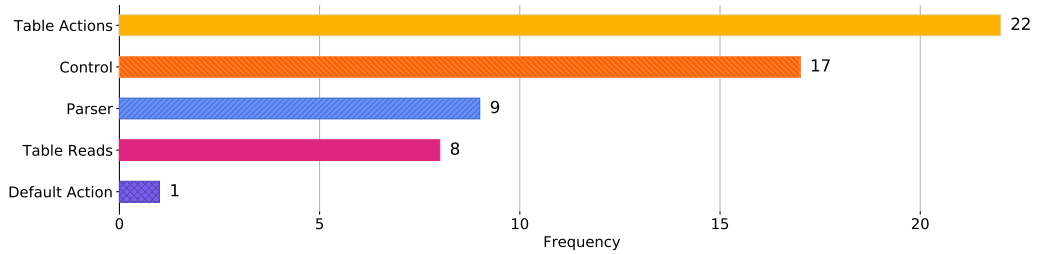


Figure 20 Frequency of each bug across all programs. The raw number of bugs in each category is reported to the right of the bar.

Frequently, multiple violations produced by P4Check have the same root cause. For example, if a single action `rewrite_ipv4` that rewrites fields `srcAddr` and `dstAddr` for an `ipv4` header is called in a context that cannot prove that `ipv4` is valid, then both references to `ipv4.srcAddr` and `ipv4.dstAddr` will be reported as violations, even though they are due to the same *control* bug (Section 2.2.2) – namely that `rewrite_ipv4` was not called in a context that could prove the validity of `ipv4`. To address this issue, we applied another metric to quantify the number of bugs (inspired by the method proposed by others [17]): we equate the number of bugs in each program with the number of bug *fixes* required to make the program in question pass our type checker. Using this metric, we counted 58 bugs.

We classified the bugs according to the classes described in Section 2.2. Figure 19 depicts the per-program breakdown of the frequency of each bug class, and Figure 20 depicts the overall frequency of each bug. Notice that even though table action bugs were the most frequent bug (with 22 occurrences), they were only found in a single program (`switch.p4`). These bugs are especially prevalent in this program because of its heavy reliance on correct control-plane configuration. Conversely, there were 9 occurrences across 5 programs for both parser bugs and table reads bugs.

Readers familiar with previous work on `p4v` [20], a recent P4 verification tool, may notice that we detected no default action bugs for the `switch.p4` program, while `p4v` reported many! The reasons for this are two-fold. First, `p4v` allows programmers to verify complex properties, which means that it can express fine-grained conditions on tables and relationships between them. In contrast, we make heuristic assumptions about P4 programs that automatically eliminate many bugs, including some default action bugs. Second, our repairs are often

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 118, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19:error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
./h.p4, line 107, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
./h.p4, line 163, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
```

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19: error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
```

Figure 21 Curated output from P4Check for the parser bug in NetHCF before (above) and after (below) modifying `parse_ethernet`.

coarse-grained and may enforce a stronger guarantee on the program than may be necessary; using first-order logic annotations, `p4v` programmers manually specify the weakest (and hence more complex) assumptions.

We make no claims about the completeness of our taxonomy. For example, we found one instance, in the HappyFlowFriends program, where the programmer had mistakenly instantiated metadata m as a header, and consequently did not parse m (since metadata is always valid) causing m to (ironically) always be invalid.

4.2 P4Check in Action

We reprise the canonical examples of each class of bugs from Section 2.2, describing how P4Check detects them and discussing ways to fix them.

4.2.1 Parser Bugfixes

Recall Figure 5, which exhibits the parser bug. The bug occurs because the parser, which extracts IPv4-TCP packets, boots unexpected packets (such as IPv6 or UDP packets) directly to `ingress`, which then assumes that both the `ipv4` and `tcp` headers are valid, even though the parser does not guarantee this fact.

In terms of our type system, the parser produces packets of type $\texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{tcp}))$; however the control only handles packets of type $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$. Hence, when typecheck this example, P4Check reports every reference to `tcp` and `ipv4` in the whole program as a violation of the type system. As shown in the top half of Figure 21, we get an error message at every reference to `ipv4` or `tcp`. The ubiquity of the reports intimates a mismatch between the parsing and the control types, which gives the programer a hint as how to fix the problem.

When we modify the `default` clause in `parse_ethernet`, as in Figure 5, and run our tool again, all of the `ipv4` violations are removed from the output, as shown in the bottom half of Figure 21. Then fixing the `parse_ipv4` parser, as in Figure 5, causes our tool to output no violations. In particular, the type upon entering the `ingress` control function is $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$, so all subsequent references to `ipv4` and `tcp` are safe.

```
port.p4, line 248, cols 8-24: warning: assuming either vlan_tag_[0]
    matched as valid or vlan_tag_[0].vid wildcarded

port.p4, line 250, cols 8-24: warning: assuming either vlan_tag_[1]
    matched as valid or vlan_tag_[1].vid wildcarded
```

```
fabric.p4 line 42, cols 41-67: warning: assuming fabric_header_cpu
    matched as valid for rules with action terminate_cpu_packet

fabric.p4, line 57, cols 17-54: warning: assuming fabric_header_unicast
     matched as valid for rules with action
    terminate_fabric_unicast_packet

fabric.p4, line 81, cols 17-56: warning: assuming
    fabric_header_multicast matched as valid for rules with action
    terminate_fabric_multicast_packet
```

Figure 22 Warnings printed after fixing `switch.p4`'s reads bug (top), and its actions bug (bottom).

4.2.2 Control Bugfixes

Recall that a control bug occurs when the incoming type presents a choice between two instances that are not handled by subsequent code. The program shown in Figure 6 uses a parser that produces the type $\Theta = \texttt{ethernet}\cdot(1+\texttt{ipv4}\cdot(1+\texttt{udp}\cdot(1+\texttt{nc\_hdr}\cdot\tau)+\texttt{tcp}))$, where τ is a type for caching operations. Note that $\texttt{Includes}\ \Theta\ \texttt{nc\_hdr}$ does not hold. However, `process_cache` and `process_value` only type check in contexts where $\texttt{Includes}\ \Theta\ \texttt{nc\_hdr}$ is true. P4Check reports type violations at every reference to `nc_hdr`. Fixing this error is simply a matter of wrapping the `process_cache()` call in a validity check as demonstrated in Figure 6. As NetCache handles TCP and UDP packets as well as its special-purpose packets, we simply apply the IPv4 routing table if the validity check for `nc_hdr` fails.[5]

4.2.3 Table Reads Bugfixes

Table reads errors, as shown in Figure 7, occur when a header h is included in the `reads` declaration of a table t with match kind k, and h is not guaranteed to be valid at the call site of t, and if $h \notin \texttt{valid\_reads}(t)$ or the match-kind of $k \neq \texttt{ternary}$.

In the case of the `port_vlan_mapping` table in Figure 7, there is a valid bit for both `vlan_tag_[0]` and `vlan_tag_[1]`, both of which are followed by `exact` matches. To solve this problem, we need to use the `ternary` match-kind instead, which allows the use of wildcard matching. When a field is matched with a wildcard, the table does not attempt to compute the value of the `reads` expression.

This fix assumes that the controller is well behaved and fills the `vlan_tag_[0].vid` with a wildcard whenever `vlan_tag_[0]` is matched as invalid (and similarly for `vlan_tag_[1]`). This also what the SafeP4 type system does, with its maskable checks in the T-Apply rule P4Check prints warnings describing these assumptions to the programmer (top of Figure 22), giving them properties against which to check their control plane implementation.

4.2.4 Table Action Bugfixes

Table actions bugs occur when at least one action cannot be safely executed in all scenarios. For example, the table `fabric_ingress_dst_lkp` shown in Figure 8 has a table action bug, which can be fixed by modifying the table's `reads` declaration. Recall that the

[5] Astute readers may detect a parser bug in this example. Hint, the `ipv4_route` table requires $\texttt{Includes}\ \Theta\ \texttt{ipv4}$ where Θ is type where it is applied.

parser will parse exactly one of the headers `fabric_hdr_cpu`, `fabric_hdr_unicast` and `fabric_hdr_multicast`, which means that when the table is applied at type Θ, exactly one of `Includes` Θ `fabric_hdr_i` for $i \in \{\texttt{cpu}, \texttt{unicast}, \texttt{multicast}\}$ will hold. Now, the action `term_cpu_packet` typechecks only with the (nonempty) type `Restrict` Θ `fabric_hdr_cpu`, and the actions `term_fabric_`i`_packet` only typecheck with the (nonempty) types `Restrict` Θ `term_fabric_`i`_packet` for $i = \texttt{unicast}, \texttt{multicast}$. P4Check suggests that this is the cause of the bug since it reports type violations for all of the references to these three headers in the control paths following from the application of `fabric_ingress_dst_lkp`.

The optimal[6] fix here is to augment the `reads` declaration to include a validity check for each contentious header. We then assume that the controller is well-behaved enough to only call actions when their required headers are valid, allowing us to typecheck each action in the appropriate type restriction. P4Check alerts the programmer whenever it makes such an assumption. We show these warnings for the fixed version of `fabric_ingress_dst_lkp` below the line in Figure 22.

4.2.5 Default Action Bugfix

Default action bugs occur when a programmer creates a wrapper table for an action that modifies the type, and forgets to force the table to call that action when the packet misses. The `add_value_header_1` table from Figure 9 wraps the action `add_value_header_1_act`, which calls the single line `add_header(nc_value_1)`.

The default action, when left unspecified, is `nop`, which means that if the pre-application type was Θ, then the post-application type is $\Theta + \Theta \cdot \texttt{nc\_value\_1}$, which does not include `nc_value_1`. Hence, P4Check reports every subsequent reference (on this code path) to `nc_header_1` to be a type violation.

To fix this bug, we need to set the default action to `add_value_1` – this makes the post-application type $\Theta \cdot \texttt{nc\_value\_1} + \Theta \cdot \texttt{nc\_value\_1} = \Theta \cdot \texttt{nc\_value\_1}$, which includes `nc_value_1`, thus allowing the subsequent code to typecheck.

4.3 Overhead

It is important to evaluate two kinds of overhead when considering a static type system: overhead on programmers and on the underlying implementation.

Typically, adding a static type system to a dynamic type system requires more work for the programmer – the field of gradual typing is devoted breaking the gargantuan task into smaller commit-sized chunks [5]. Surprisingly, in our experience, migrating real-world P4 code to pass the SafeP4 type system only required modest programmer effort.

To qualitatively evaluate the effort required to change an unsafe program into a safe one using our type system, we manually fixed all of the detected bugs. The programs that had bugs required us to edit between 0.10% and 1.4% of the lines of code. The one exception was PPPoE_using_P4, which was a 143 line program that required 6 line-edits (4%), all of which were validity checks. Conversely, `switch.p4` required 34 line edits, the greatest observed number, but this only accounted for 0.37% of the total lines of code in the program.

Each class of bugs has a simple one-to-two line fix, as described in Section 4.2: adding a validity check, adding a default action, or slightly modifying the parser. Each of these changes was straightforward to identify and simple to make.

[6] Another fix would be to refactor the single into multiple tables, each guarded by a separate validity check. However, combining this kind of logic in a single table helps conserve memory, so in striving to change the behavior of the program as little as possible, we propose modifying the table reads.

Another possible concern is that that extending tables with extra read expressions, or adding run-time validity checks to controls, might impose a heavy cost on implementations, especially on hardware. Although we have not yet performed an extensive study of the impact on compiled code, based on the size and complexity of the annotations we added, we believe the additional cost should be quite low. We were able to compile our fixed version of the `switch.p4` program to the Tofino architecture [24] with only a modest increase in resource usage. Overall, given the large number of potential bugs located by P4Check, we believe the assurance one gains about safety properties by using a static type system makes the costs well worth it.

5 Related Work

Probably the most closely related work to SafeP4 is `p4v` [20]. Unlike SafeP4, which is based on a static type system, `p4v` uses Dijkstra's approach to program verification based on predicate transformer semantics. To model the behavior of the control plane, `p4v` uses first-order annotations. SafeP4's typing rule for table application is inspired by this idea, but adopts simple heuristics – e.g., we only assume that the control plane is well-behaved – rather than requiring logical annotations.

Both `p4v` and P4Check can be used to verify safety properties of data planes modelled in P4 – e.g., that no read or write operations are possible on an invalid header. As it is often the case when comparing approaches based on types to those based on program verification, `p4v` can check more complex properties, including architectural invariants and program-specific properties – e.g., that the IPv4 time-to-live field is correctly decremented on every packet. However, in general, it requires annotating the program with formal specifications both for the correctness property itself and to model the behavior of the control plane.

McKeown et al. developed an operational semantics for P4 [22], which is translated to Datalog to verify safety properties and to check program equivalence. An operational semantics for P4 was also developed in the K framework [27], yielding a symbolic model checker and deductive verification tool [16]. Vera [30] models the semantics of P4 by translation to SymNet [31], and develops a symbolic execution engine for verifying a variety of properties, including header validity.

Compared to SafeP4, these approaches do not use their formalization of P4 as a foundation for defining a type system that addresses common bugs. To the best of our knowledge, SafeP4 is the first formal calculus for a P4-like packet processing language that provides correct-by-construction guarantees of header safety properties.

Other languages have used type systems to rule our safety problems due to null references. For example, NullAway [29] analyzes all Java programs annotated with `@Nullable` annotations, making path-sensitive deductions about which references may be null. Similar to the validity checks in SafeP4, NullAway analyses conditionals for null checks of the form `var != null` using data flow analysis.

Looking further afield, PacLang [9] is a concurrent packet-processing language that uses a linear type system to allow multiple references to a given packet within a single thread. PacLang and SafeP4 share the use of a type system for verifying safety properties but they differ in the kind of properties they address and, hence, the kind of type system they employ for this purpose. In addition, the primary focus in PacLang is on efficient compilation whereas SafeP4 is concerned with ensuring safety of header data.

Domino [28] is a domain-specific language for data plane algorithms supporting *packet transactions* – i.e., blocks of code that are guaranteed to be atomic and isolated from other transactions. In Domino, the programmer defines the operations needed for each packet

without worrying about other in-flight packets. If it succeeds, the compiler guarantees performance at the line rate supported on programmable switches. Overall, Domino focuses on transactional guarantees and concurrency rather than header safety properties.

BPF+ [3] and eEBPF [8] are packet-processing frameworks that can be used to extend the kernel networking stack with custom functionality. The modern eBPF framework is based on machine-level programming model, but it uses a virtual machine and code verifier to ensure a variety of basic safety properties. Much of the recent work on eBPF focuses on techniques such as just-in-time compilation to achieve good performance.

SNAP [1] is a language for stateful packet processing based on P4. It offers a programming model with global state registers that are distributed across many physical switches while optimizing for various criteria, such as minimizing congestion. More specifically, the compiler analyses read/write dependencies to automatically optimize the placement of state and the routing of traffic across the underlying physical topology.

While our approach to track validity is network-specific, it is similar to taint analysis [33, 10, 11], which attempts to identify secure program parts that can be safely accessed.

Of course, there is a long tradition of formal calculi that aim to capture some aspect of computation and make it amenable for mathematical reasoning. The design of SafeP4 is directly inspired by Featherweight Java [12], which stands out for its elegant formalization of a real-world language in an extensible core calculus.

6 Conclusion

P4 provides a powerful programming model for network devices based on high-level and declarative abstractions. Unfortunately, P4 lacks basic safety guarantees, which often lead to a variety of bugs in practice. This paper proposes SafeP4, a domain-specific language for programmable data planes that comes equipped with a formal semantics and a static type system which ensures that every read or write to a header at run-time will be safe. Under the hood, SafeP4 uses a rich set of types that tracks dependencies beween headers, as well as a path-sensitive analysis and domain-specific heuristics that model common idioms for programming control planes and minimize false positives. Our experiments using an OCaml prototype and a suite of open-source programs found on GitHub show that most P4 applications can be made safe with minimal programming effort. We hope that our work can help lay the foundation for future enhancements to P4 as well as the next generation of data plane languages. In the future, we plan to explore enriching SafeP4's type system to track additional properties, investigate correct-by-construction techniques for writing control-plane code, and develop a compiler for the language.

References

1 Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM. doi:10.1145/2934872.2934892.

2 Jiasong Bai, Jun Bi, Menghao Zhang, and Guanyu Li. Filtering Spoofed IP Traffic Using Switching ASICs. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 51–53. ACM, 2018.

3 Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 123–134, New York, NY, USA, 1999. ACM. doi:10.1145/316188.316214.

4 Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. doi:10.1145/2656877.2656890.

5 John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. *Proceedings of the ACM on Programming Languages*, 2(POPL):15, 2017.

6 Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37 (4), pages 1–12. ACM, 2007.

7 P4 Language Consortium. P4 Language Specification, Version 1.0.4. Technical report, Available at https://p4.org/specs/, 2017.

8 Jonathan Corbet. BPF: the universal in-kernel virtual machine, May 2014. Available at https://lwn.net/Articles/599755/,.

9 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear Types for Packet Processing. In David Schmidt, editor, *Programming Languages and Systems*, pages 204–218, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

10 William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 175–185, New York, NY, USA, 2006. ACM. doi:10.1145/1181775.1181797.

11 Wei Huang, Yao Dong, and Ana Milanova. Type-Based Taint Analysis for Java Web Applications. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 140–154, New York, NY, USA, 2014. Springer-Verlag New York, Inc. doi:10.1007/978-3-642-54804-8_10.

12 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.

13 Xin Jin. netcache-p4, March 2018. URL: https://github.com/netx-repo/netcache-p4.

14 Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-rtt coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018. Best paper award.

15 Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2017.

16 Ali Kheradmand and Grigore Roşu. P4K: A formal semantics of P4 and applications. Technical Report https://arxiv.org/abs/1804.01468, University of Illinois at Urbana-Champaign, April 2018.

17 George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

18 Chaitanya Kodeboyina. An open-source P4 switch with SAI support, June 2015. URL: https://p4.org/p4/an-open-source-p4-switch-with-sai-support.html.

19 Rahul Kumar and BB Gupta. Stepping stone detection techniques: Classification and state-of-the-art. In *Proceedings of the international conference on recent cognizance in wireless communication & image processing*, pages 523–533. Springer, 2016.

20 Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4V: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 490–503, New York, NY, USA, 2018. ACM. doi:10.1145/3230543.3230582.

21 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. doi:10.1145/1355734.1355746.

22 Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjorner, and Andrey Rybalchenko. Automatically verifying reachability and well-formedness in P4 Networks, September 2016. URL: https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/.

23 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

24 Barefoot Networks. Tofino 2. URL: https://www.barefootnetworks.com/products/brief-tofino-2/.

25 Barefoot Networks. Behavioral Model, December 2018. URL: https://github.com/p4lang/behavioral-model.

26 TJ OConnor, William Enck, W Michael Petullo, and Akash Verma. Pivotwall: SDN-based information flow control. In *Proceedings of the Symposium on SDN Research*, page 3. ACM, 2018.

27 Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

28 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM. doi:10.1145/2934872.2934900.

29 Manu Sridharan. Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android, December 2018. URL: https://eng.uber.com/nullaway/.

30 Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 Programs with Vera. In *ACM SIGCOMM*, pages 518–532, New York, NY, USA, 2018. ACM. doi:10.1145/3230543.3230548.

31 Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, pages 314–327, New York, NY, USA, 2016. ACM. doi:10.1145/2934872.2934881.

32 Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. doi:10.1145/1863543.1863561.

33 Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996. URL: http://dl.acm.org/citation.cfm?id=353629.353648.

34 Menghao Zhang. Anti-spoof, November 2018. URL: https://github.com/zhangmenghao/Anti-spoof.