Position Papers of the Federated Conference on Computer Science and Information Systems pp. 49–56
DOI: 10.15439/2019F355
ISSN 2300-5963 ACSIS, Vol. 19

A formal method to detect possible $P4_{16}$ specific errors

Gabriella Tóth
Eötvös Loránd University
Faculty of Informatics
1/C. Pázmány Péter sny, Budapest, 1117, Hungary
Orcid Id: 0000-0001-9657-7231
Email: kistoth@inf.elte.hu

Máté Tejfel
Eötvös Loránd University
Faculty of Informatics
1/C. Pázmány Péter sny, Budapest, 1117, Hungary
Orcid Id: 0000-0001-8982-1398
Email: matej@inf.elte.hu

***Abstract*—P4 is a programming language to develop data processing of networks. This kind of programs are used in network devices – like switches – to describe the way of forwarding the received packets to the proper device. Checking the correctness of these programs is not an obvious task, because they can easily hide the run time errors. We are working on a method to detect violation of P4 specific properties. The method is based on a rule system, which can detect suspicious program parts and indicate the violated property. It helps to detect and correct real errors easily. As a first step, we introduce the main idea, dealing with the access of invalid header and uninitialized fields. We also present a case study to demonstrate the applicability of the method.**

I. Introduction

P4 [1] is a domain specific programming language to develop data plane of network devices. P4 makes it possible to develop target independent, protocol independent solutions for data plane processing. Budiu and Dodd [2] describe partially the main structure and the design goals of the most recent version of the language.

Although it opened a new dimension in the network data plane, it left the safety of bound protocols. Therefore, P4 developers need to be more prudent to create correct programs, or a proper solution need to be invented to detect different source of errors.

Errors are hardly detectable, because P4 easily hides them, and we can only recognize them from the bad or suspicious behavior. For example, if the program reads a field of an invalid header then it will get an unspecified value, with which it will continue the execution and it will not stop and sign the problem.

One solution can be accurate testing, but this can be too expensive and bounded. Formal methods can be more applicable. In the near past, different verification tools were published [3], [4], [5] to work on the correctness of P4 programs, with detecting specified properties, and give the opportunity for developers to correct them.

This work has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

P4 has two main release version: $P4_{14}$ [1], [6] and $P4_{16}$ [2], [7]. There are main differences between them, for example in the structure, the syntax of the code and the deparsing phase. We work with the new one, and we would like to create firstly a method, secondly a tool to use them to check safety properties.

In this paper we introduce the first step forward a P4 specific, formal program property verification method. In this step we produce a rule system to detect different errors in the programs. Now, we work only with one possible error type, which can be caused simply by inattention and cannot be found easily. We define a property based on the validity of headers and fields, which says that, invalid headers or uninitialized fields should not be read or written, because it can cause undefined behavior. Section II shows some related works. In Section III we introduce some example for this problem. Section IV introduces the method specification and after it, we present its usage in a concrete case study in Section V.

A. Background on P4

Figure 1 contains a P4 program used by our case study. Main process of P4 programs is to get a bitstream as an input, extract the information of headers (with the parser) and modify them (with different actions) to create the new packet, which is sent forward to the network. There are some main structural units of the programs. Headers (lines 1-25) describe the handled header information about the packets. Parser (lines 27-48) extracts the data from the input bitstream to header instances. Control functions (lines 57-93) call the match-action tables, and handle the modification of headers with them. Match-action tables (lines 76-86) call actions. They work in a similar way as table lookup. They matches concrete fields of headers – named keys – with given values, and according to the result it executes an action call. The program only defines templates of the match-action tables, which contain a set of fields and a set of description of actions – their name and parameters. The concrete pairs of matching values and description

of action are coming from an external controller during run time. Actions define the modification of the values of headers. After the modification, the deparser builds the new packet from the created header data as an output bitstream and forwards it to the network.

Figure 1 shows an example, where there are two header type: $ethernet$ and $ipv4$. After the input packet arrives – as a bitstream – the parser will extract the header $ethernet$. If the value of field $ethernetType$ is $0x800$, then it will extract the header $ipv4$ too, otherwise is will not extract any other header. $MyIngress$ control function contains the modification part of the program. It has one table, which can call three different actions. One of the actions is the $ipv_4forward$, which change the $srcAddr$, $dstAddr$ and $mathitttl$ fields of the $ipv4$. Another is the action $ipv4_new$, which tries to create a new $ipv4$, with the method $setValid$, and some assignments. The last one is the action $drop$, which only drops the processed packet. The main process of the control function is a branch, which will execute the table if the header $ipv4$ is valid. After the modification of the headers, the deparser will produce the output packet – as a bitstrem – which will contain both of the mentioned headers.

II. Related Work

In the near past different P4 specific verification tools were published. We would like to highlight four of them, which work with the previous main version of P4 – named P4_14.

Assert-P4 [3] does not appoint specific properties to check, they entrust it to the developer, who can add assertions to the source code, and the tool will check their correctness. From the annotated program, it creates a C-model in which it examines the properties working based on symbolic execution – using Klee [8].

P4V [4] uses another approach of the problem. They transform the P4 code into GCL, create logical formulas from the GCL description, and check their satisfiability with Z3 [9].

Vera [5] uses also symbolic execution. It transforms the source code to SEFL, which is a modeling language to define state machines. The state machine represents every possible path of the execution, with symbolic values – using Symnet [10]. Vera works with predefined properties.

P4K [11] is a solution, which uses the $\mathbb{K}$ framework [12]. It presents the operational semantics of P4 and based on this semantics it can verify simple P4 properties using the reachability rule system of $\mathbb{K}$ [13].

All of the mentioned tools deal with validity checking, but neither of them mentioned the problem of uninitialized fields. Our solution extended the invalid header monitoring to the deparsing phase and to the usage of keys of tables. Our solution also checks the usage of uninitialized fields. Most of the proposed

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    bit<32>   srcAddr;
    bit<32>   dstAddr;
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x800: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata)

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(bit<48> dstAddr) {
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    action ipv4_new(bit<48> dstAddr, bit<48> srcAddr) {
        hdr.ipv4.setValid();
        hdr.ipv4.srcAddr = srcAddr;
        hdr.ipv4.dstAddr = dstAddr;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            ipv4_new;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

Fig. 1. Example of P4 program

tools use some type of symbolic execution, but we try to stay in a formal solution, where we need no symbolic values. Although the current version of our method can not manage properly the problem of numerous different initial and final states, we are working on a solution to reduce this problem.

III. Motivation

The first property that we introduce is the validity of headers and their fields. Reading or writing of invalid headers or uninitialized fields can cause undefined behavior. Therefore, we would like to highlight the error prone usage of them.

There are two commands, which can be used to set the validity of headers. One of them is the $setValid()$ function, which can validate them. Calling of $setValid()$ has a side effect with which every field of the header become uninitialized. The other one is the $setInValid()$ function, which set the header to invalid, and all of its fields to uninitialized too. After the usage of these commands, the fields can be initialized explicitly with assignments.

Reading fields of invalid headers and uninitialized fields results unspecified values. Writing fields of invalid headers are unnecessary, because if we would like to use their values – for example in the output packet – we need to set the header to valid, which means that their fields become unspecified. Therefore, we need to avoid these type of codes, because it can easily lead to an error.

We would like to filter every occurrence of this problem, which can be in the control functions – for example in the conditions of branches, keys of tables and assignments. Using invalid header to emit in the deparsing phase is not an error, because P4 simply will not use it during the building of the packet, although it can be suspicious. Therefore, in this paper we will consider it as a possible error. Summary, in our rule system we would like to detect the usage of uninitialized fields and invalid headers as a key of a table, part of an assignment or during deparsing.

In the following sections we will link to the initialized fields as valid and the uninitialized fields as invalid fields, for simplification.

IV. Method of property checking

The method has two main parts. First is a preprocessing phase in which the initial states, final states and core program are produced. The initial states will be created from the source of parser, the final states will be created from the source of deparser, and the program will be produced from the control functions. The second phase is the usage of the rule system, which detects the errors. The calculation examines that, the execution can reach one of the final states from every initial state. This rule system is similar to an axiomatic semantics, but it has a more complex environment structure and additional side conditions.

$$S \in State : \\ Condition \times PacketInfo \times Environment$$

Fig. 2. Type of states

A. Preprocessing – Initial state

To use the method, first we need to preproccess the P4 code to produce the initial states. This first phase collects the used headers, and creates an empty environment, where everything is invalid. Than it analyses the parsing phase and calculates the different packet information and initial environments.

Parsing is a state machine with two final states – $accept$ and $reject$. Initial states will contain those paths, which start with the $start$ state and finish in the $accept$ state. They will use the conditions of branches to describe the different input packets in the packet information.

States will be represented with a triple – it is showed by Figure 2. The first element is the collected conditions, which is $True$ in the initial state, and is changed during the usage of the rule system. The second element is the packet information, which identifies the different input packets. The third part is the used environment.

$Condition$ collects the conditions of branches from the control functions as a conjunctive formula. There will be statements for validity checking of headers.

$PacketInfo$ and $Environment$ contains information about the input packet and the created headers. Suppose their is a parsing phase, which first extracts an $ethernet$ header, and after that it branch according to the value of $ethertype$ field. If its value is $0x800$ then it will extract $ipv4$, if its value is $0x86DD$ then it will extract $ipv6$ header, otherwise it will not extract any other headers. In this case, there can be three different initial packets, which can be described with the following formulas:

First case:

$$(ethernet.ethertype = 0x800; \\ ethernet = valid,\ ipv4 = valid,\ ipv6 = invalid)$$

Second case:

$$(ethernet.ethertype = 0x86DD; \\ ethernet = valid,\ ipv4 = invalid,\ ipv6 = valid)$$

Third case:

$$(ethernet.ethertype \neq 0x800 \\ \wedge\ ethernet.ethertype \neq 0x86DD; \\ ethernet = valid,\ ipv4 = invalid,\ ipv6 = invalid)$$

Of course the environment description has concrete initial information about the fields of headers. It contains every headers and fields of the headers with their validity. In the parsing phase only total headers are parsed, therefore if a header is valid, then its fields will be valid too. The environment also contains an

additional information, the *drop* flag, which shows the packet is set to drop or not.

In the environment, now we only collect the headers, their fields and their validity information, but as future work we plan to extend this information with concrete values to make possible the analysis of more precise properties.

B. Preprocessing – Final state

Deparsing phase is implemented as a control function in P4, so it can also contains branches. During deparsing the emit commands determine which headers and in which order are added to the output packet. If this header and its fields are valid then everything is fine. If the header is invalid, then the program will not add it to the packet. It is allowed to use it, but in our case, we would like to sign any suspicious case, therefore we will handle it as a possible error source. Emitting an uninitialized field of a valid header is also wrong, because it will use an unspecified value, so we would like to prevent it.

As mentioned above, the *drop* flag is part of the environment. Therefore, there will be a final state, which is always a possible one: which contains the *drop* variable with the value *1*, and every header and field can be valid or invalid. This state describes that case, when the packet is dropped. In every other case the *drop* flag needs to be *0*, and the value of headers and fields is defined.

The conditions ($Condition$ and $Packet\ information$) of the final states are filled with $True$: $(True, (True, E)))$, because in the end of the calculation we will be able to restrict them – Rules 12 and 13 from Figure 3. There can be more final states, so there is a rule, with which the reachable one can be chosen – Rule 14.

C. Preprocessing – Core program

We need to create the core program for the rule system, so we unwrap the table applications and the action calls and concatenate the control functions – except for the deparser. We will get a sequentially program which contains every aspect of the code – for example the parameters of the actions – so we will be able to easily extend the property checking with new detection.

D. The rule system

During a deduction we prove that the program will reach one of the final state from all of the possible initial states. From the preprocessing phase we get one or more different initial states, and we need to verify the properties started from all of them. At the beginning of the method we can choose the conditions to $True$, because those will change during the code processing by adding other conditions of the branches with a conjunction.

The basic structure of the rule system is similar to the axiomatic semantics of P4. Although it is stricter than the basic behavior of the P4 programs and the used program states are extended with the above mentioned way. The rules will be inference rules in which the $PacketInfo$ part will never be changed, because it is used as an identification of the input packet – except for the case of Rule 13. Therefore, in the end of the calculation we will be able to tell more specified information about the possible errors.

Figure 3 shows the rule system of the main verification to detect errors. In the system, the S notations mean the statements, which have 3 main parts. C is a condition, P is a packet information and E is an environment. Therefore, every variants of these letters means the same type of element. On the right side of some rules there is a side condition, $S \vdash \{x\}$ – where x can be a condition of branch, a key of a table or one side of an assignment – statement. It checks the calculability of every element of the given set of statements by knowing the C conditions and E environment of the S state. Here calculability means that, every used field and header is valid.

In the rules, there are some specified expressions. Rule 1 and 2 describe the assignments. There can be two type of them: when we give value to a field (Rule 1) – in this case only the field become valid –, and when we give value to a header with a list (Rule 2) – here every field and the header are rewritten to valid. In the right side of the rule in the description of the environment there can be an expression (for example $\{S\ [E\ ||\ E[h.f \rightarrow valid]]\}$), which means that we use the S state with some modification in the E environment , especially the $h.f$ fields validity will be changed to valid. In the Rule 2, the $h.fields$ is used, which means, we use every field of header h, and in this case we set them to $valid$. In the right side of these rules, the checking of the used statements is appeared. It checks the validity of every part of the assignments, because the commands write the left side of the assignments and reads the right side.

Rules 3 and 4 process the setting of the validity of a header. The first rule sets it to valid, and its every field to invalid – because of the side effect. The other rule sets the given header and its fields to invalid.

During the process of the packet, there can be cases, when we need to drop the packet. Rule 5 describes this function, and simply set the *drop* flag to *1*.

The next three rules (Rules 6, 7 and 8) are the extended version of the common programming structures. First describes the sequence, second and third describe the branches with and without an else case. In the last two rules, it needs to check the validity of the condition of the branch.

In the last one, it uses $S_1 \wedge \neg b \Rightarrow S_2$, which can be rewritten to the following: $S_1.C \wedge toCond(S_1.E) \wedge \neg b \supset (S_2.C \wedge toCond(S_2.E))$, where $toCond(S.E)$

Let it be: $h \in Headers,\ f \in Fields$

1. $$\frac{}{\{S\}\ h.f = exp\ \{S\ [E\ ||\ E[h.f \rightarrow valid]]\}} \quad S \vdash \{exp,\ h,\ h.f\}$$

2. $$\frac{}{\{S\}\ h = list\ \{S\ [E\ ||\ E[h \rightarrow valid, h.fields \rightarrow valid]]\}} \quad S \vdash list \cup \{h\}$$

3. $$\frac{}{\{S\}\ h.setValid()\ \{S\ [E\ ||\ E[h \rightarrow valid, h.fields \rightarrow invalid]]\}}$$

4. $$\frac{}{\{S\}\ h.setInValid()\ \{S\ [\ E\ ||\ E[h \rightarrow invalid, h.fields \rightarrow invalid]]\}}$$

5. $$\frac{}{\{S\}\ mark\_to\_drop(..)\ \{S\ [E\ ||\ E[drop \rightarrow 1]]\}}$$

6. $$\frac{\{S_1\}\ Pr_1\ \{S_2\} \qquad \{S_2\}\ Pr_2\ \{S_3\}}{\{S_1\}\ Pr_1; Pr_2\ \{S_3\}}$$

7. $$\frac{\{S_1\ [C\ ||\ C \wedge b]\}\ Pr_1\ \{S_2\} \qquad \{S_1\ [C\ ||\ C \wedge \neg b]\}\ Pr_2\ \{S_2\}}{\{S_1\}\ if\ (b)\ \{Pr_1\}\ else\ \{Pr_2\}\ \{S_2\}} \quad S_1 \vdash \{b\}$$

8. $$\frac{\{S_1\ [C\ ||\ C \wedge b]\}\ Pr\ \{S_2\} \qquad S_1 \wedge \neg b \Rightarrow S_2}{\{S_1\}\ if\ (b)\ \{Pr\}\ \{S_2\}} \quad S_1 \vdash \{b\}$$

9. $$\frac{\{S_1\}\ A_1.body\ \{S_2\} \qquad \ldots \qquad \{S_1\}\ A_n.body\ \{S_2\} \qquad S_1 \Rightarrow S_2}{\{S_1\}\ table\ keys : K\ actions : \{A_1, ..., A_n\}\ \{S_2\}} \quad S_1 \vdash K$$

10. $$\frac{\{S_1\}\ A_1.body\ \{S_2\} \qquad \ldots \qquad \{S_1\}\ A_n.body\ \{S_2\} \qquad \{S_1\}\ A_{n+1}.body\ \{S_2\}}{\{S_1\}\ table\ keys : \{K\}\ actions : \{A_1, ..., A_n, A_{n+1}\}\ \{S_2\}} \quad S_1 \vdash K$$

11. $$\frac{S_1.C \supset C' \qquad \{S_1\ [C\ ||\ C']\}\ S\ \{S_2\}}{\{S_1\}\ Pr\ \{S_2\}}$$

12. $$\frac{\{S_1\}\ Pr\ \{S_2\ [C\ ||\ C']\} \qquad C' \supset S_2.C}{\{S_1\}\ Pr\ \{S_2\}}$$

13. $$\frac{\{S_1\}\ Pr\ \{S_2\ [P\ ||\ P_3]\} \qquad P_3 \supset S_2.P}{\{S_1\}\ Pr\ \{S_2\}}$$

14. $$\frac{\{S_1\}\ Pr\ \{S_i\} \qquad i \in [2..n]}{\{S_1\}\ Pr\ \{S_2, \ldots S_n\}}$$

15. $$\frac{\{S_1\}\ Pr\ \{S_m\} \qquad \ldots \qquad \{S_n\}\ Pr\ \{S_m\}}{\{S_1, \ldots, S_n\}\ Pr\ \{S_m\}}$$

Fig. 3. Rule system to verify validity

means the conversion of the environment to conditions. For example, if there is a header h, with value $valid$, it will create a condition like $h == valid$. So the expression checks that, the information of S_1 and the negated b implies the information of S_2.

The following two rules (Rules 9 and 10) describe the behavior of match-action tables, as a branch with n cases. There is validity checking of the keys in both rules. The previous one works with the tables without a default action, and the latter one works with default actions too.

The last five rules can be used to refine conditions and packet information and states. The first is the strengthen of the left side condition, the second is the weaken of the right side condition. The third one weakens the packet conditions. It is necessary during the deduction to strengthen the packet information of the final state, because at the beginning, we use $True$ to describe it, and in the end it needs to match with the concrete description. The forth one say that if their is a deduction to an S_i state then there will be a deduction to more states, where one of them is S_i. In

the deduction it means that we choose one final state. The last one can be used to separate the deduction based on the initial states.

E. Verification

The initial states will be created in preprocessing phase, which will separate the possible input packets. The final states and the core program will be produced too. We need to create a deduction from each of the initial states to one of the final states by using the inference rules. If there is any problem, the deduction will stop and we will be able to see the errors from the stuck paths. There are branches in the deduction tree. Beside a deduction has one stucked path, it can detect other errors, or it can work well in other paths too. So we can detect different errors in one proof, and determine the problems from the calculation of the incorrect conditions.

Producing a proof for every initial state, will mean that the given program is well defined, and it does not violent the examined program properties.

V. Case study

This section focuses to the P4 example, which is illustrated by Figure 1. We show the results of the verification and its phases.

A. Preprocessing – initial states

This phase of the preprocessing use the headers and the parsing phase. The environment of the initial states contains every headers and fields. There is an added flag – named $drop$ – which represents the intention of drop the input packet. The parsing phase extracts the $ethernet$ header and than the execution is branched. If condition $ethernet.ethertype = 0x800$ holds, it will extract an $ipv4$ header, on other case it will not extract other headers. After the calculation of preprocessing, two possible initial states will be produced.

The produced two possible initial states are the followings:

```
I1 = (True, ethernet.ethertype = 0x800,
      [drop = 0,
       ethernet: ( valid, {
                   dstAddr: valid,
                   srcAddr: valid,
                   ethertype: valid}),
       ipv4:     ( valid, {
                   version: valid,
                   ...
                   dstAddr: valid})])
```

Fig. 4. First initial state

B. Preprocessing – final states

The final states calculation uses the code of the deparsing phase. In the example, there is a really simple deparser, which firstly emits the $ethernet$, and secondly emits the $ipv4$ header. Therefore there will

```
I2 = (True, ethernet.ethertype ≠ 0x800,
      [drop = 0,
       ethernet: ( valid, {
                   dstAddr: valid,
                   srcAddr: valid,
                   ethertype: valid}),
       ipv4:     ( invalid, {
                   version: invalid,
                   ...
                   dstAddr: invalid})])
```

Fig. 5. Second initial state

be two reachable environments. One of them describes the case, when the packet will be dropped, and the other fields and headers validity is not important – Figure 6 represents it with the marking $others = *$. The other describes the execution when the packet is not dropped, and the $ethernet$, $ipv4$ and all of their fields are valid, because we would like to forward them.

```
F = {(True, True, [drop = 1, others = *]),
     (True, True,
      [drop = 0,
       ethernet: (valid, {
                  dstAddr: valid,
                  srcAddr: valid,
                  ethertype: valid}),
       ipv4: (valid, {
              version: valid,
              ...
              dstAddr: valid})])}
```

Fig. 6. The calculated final state

C. Preprocessing – executable program

This phase only use the code of the control functions. It concatenates the main code of the control functions, and unwrap the calls of the tables and actions.

```
Pr =
if (hdr.ipv4.isValid()) {
  table
    keys: {hdr.ipv4.dstAddr}
    actions: {
      ipv4_forward(bit<48> dstAddr) {
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
      },
      drop(){
        mark_to_drop(standard_metadata);
      },
      ipv4_new(bit<32> dstAddr, bit<32> srcAddr) {
        hdr.ipv4.setValid();
        hdr.ipv4.srcAddr = srcAddr;
        hdr.ipv4.dstAddr = dstAddr;
      }}
}
```

Fig. 7. Produced core program

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\{I_{11}\}\ hdr.ipv4.ttl = ..\ \{I_{11}\}}}{\{I_{11}\}\ hdr.ipv4.ttl = ..\ \{F_2\}}}{\vdots}}{\{I_{11}\}\ hdr.ipv4.srcAddr = ..;..\ \{F\}} \quad \dfrac{\dfrac{\checkmark}{\{I_{11}\}\ mark\_to\_drop(..)\ \{F_{11}\}}}{\{I_{11}\}\ mark\_to\_drop(..)\ \{F_1\}} \quad \dfrac{\lightning}{\{I_{12}\}\ hdr.ipv4.dstAddr = ..\ \{F_2\}}}{\{I_{11}\}\ hdr.ipv4.setValid();..\ \{F\}}$$

$$\dfrac{\dfrac{\{I_{11}\}\ tablekey :\{..\}\ actions : \{ipv4\_forward(..)..,\ drop(..)\{..\},\ ipv4\_new(..)..\}\}\{F\}}{\{I_1\}\ if(hdr.ipv4.isValid())\{...\}\ \{F\}} \quad \dfrac{\checkmark}{(I_1 \wedge \neg b \supset F)}$$

$I_{11} = (hdr.ipv4.isValid(), (ethernet.ethertype = 0x800, \{drop = 0, ethernet = valid, ethernet.all = valid, ipv4 = valid, ipv4.all = valid\}))$

$I_{12} = (True, (ethernet.ethertype = 0x800, \{drop = 0, thernet = valid, ethernet.all = valid, ipv4 = valid, ipv4.srcAddr = valid, ipv4.others = invalid\}))$

$F_{11} = (ipv4.isValid(), ethernet.ethertype = 0x800, [drop = 1, others = *])$

Fig. 8. Deduction of I_1

D. Using the system

Figure 8 contains the shorter version of the deduction from the first initial state. In the figure there are 4 main paths in the deduction. We took a short cut for the deduction. In parts where there are dots, there are branches, – because of the sequences – we just illustrate the last branch of all paths.

First path shows the effect of $ipv4\_forward$ action. It does not change the validity of the fields, so in the end it reaches the second version of the final state – the one in which the packet is not dropped.

The second path only contains the $mark\_to\_drop(..)$ statement, which set the $drop$ variable of the environment to 1, therefore it reaches the first version of the final state – which describes the packet dropping.

The third path does not reach the given final states. According to the dropping, it should reach the second version of the final state, but it is not proper because of the validity of fields of the $ipv4$ header. The $ipv4\_new$ action sets the validity of the $ipv4$ header to valid, but its side effect also sets all of its fields to invalid. After that, it only reinitializes the $dstAddr$ and the $srcAddr$, but there are others, which stay invalid, and there is no reachable final state for this environment.

The fourth path contains the execution, when the condition of the if statement was false. It is written with a logical formula, which is an implication, which has a $false$ expression in the left side, so the result of the whole formula is $true$.

$$\dfrac{\dfrac{\checkmark}{\{I_{22}\}\ table..\ \{F\}} \quad \dfrac{\lightning}{I_2 \wedge \neg ipv4.isValid() \supset F}}{\{I_2\}\ if(hdr.ipv4.isValid())\{...\}\ \{F\}}$$

$I_{22} = (hdr.ipv4.isValid(), ethernet = valid, ethernet.all = valid, ipv4 = invalid, ipv4.all = invalid)$

Fig. 9. Deduction of I_2

Figure 9 produces the deduction from the second initial state. This deduction has one right and one wrong branch. The first one has no error, because of the condition – $ipv4.isValid()$ – and the environment – where $ipv4 = invalid$ – are contradict each other. These type of paths can be accepted. This type of paths describes parts of the execution, which never runs.

The second path examines the case when the condition of the branch is false – so $\neg ipv4.isValid()$. In this case the left side of the implication is $true$, but it does not implicate any of the final states, because they do not have matching environments. It could match only with the final state, where the $drop$ field is 0, but there is no match between the environments. This result comes from the $ipv4$ header, because in the initial state it is not valid, but in the possible reachable state it is valid. Therefore, the whole formula is false. In this case, one of the final states should be reachable from the initial state, because the executed program is a $skip$ one,but this condition is not right.

VI. Future Work

A. More detailed method

First of all we will define the details of the method, for example the processing of the parser and deparser.

We will extend the method with other P4 specific properties. To work with other properties, we will need to follow the changes of fields with an environment, which will contain the values and other information – according to the checked properties – about the headers and their fields.

We will fix the disadvantage of the current method, it can not calculate all possible errors in one execution cycle. It stops in a path, when a problem is detected, and it does not continue the calculation. For example, if we use the Rule 9 in Figure 3, and the keys are not valid then we will not be able to detect the problems in the actions too, only if the first one is corrected. In the future we would like to figure out the solution

to continue the calculation, for example with some supposed assertions.

The current solution can manage only simplified structure elements of P4 programs, therefore we will need to extend the rule system, and the statements with other P4 elements. For example now we do not work with metadata headers, constants.

B. Implementation

In the near future, we will implement the introduced method. Currently we are working on representing the rule system in Coq [14]. Later we will create a tool, which will be able to analyze an arbitrary P4 code according to our method, and sign the possible error source.

C. Applicability in complex examples

The final step is to be able to detect errors in larger, more complex programs too. We would like to extract the method with the possibility of the compositional processing, therefore we could simplify the deductions. We will extend the rule system with new rules, which will allow to process the core program with divided parts.

VII. Conclusions

This paper introduces a formal method for property checking of P4 programs. For the time being, it only works with the validity of the headers and the initialization of the fields, but it will be also complemented with other, P4 specific properties. The produced method have two main parts. The first one is the preprocessing, which prepares the information for the concrete calculations – that creates the initial and final states with collecting and merging the contents of core program. The second part is the deduction, which examines that, from every initial states the program can reach one of the final states. If we can prove it with the rules then we have a well defined P4 program. If there is any problem during the deduction then we will be able to denote the error from the valuated condition or the states.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014. doi: 10.1145/2656877.2656890. [Online]. Available: http://dx.doi.org/10.1145/2656877.2656890

[2] M. Budiu and C. Dodd, "The p416 programming language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, pp. 5–14, Sep. 2017. doi: 10.1145/3139645.3139648. [Online]. Available: http://dx.doi.org/10.1145/3139645.3139648

[3] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering Bugs in P4 Programs with Assertion-based Verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018. doi: 10.1145/3185467.3185499. ISBN 978-1-4503-5664-0 pp. 4:1–4:7. [Online]. Available: http://dx.doi.org/10.1145/3185467.3185499

[4] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical Verification for Programmable Data Planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018. doi: 10.1145/3230543.3230582. ISBN 978-1-4503-5567-4 pp. 490–503. [Online]. Available: http://dx.doi.org/10.1145/3230543.3230582

[5] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 Programs with Vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018. ISBN 978-1-4503-5567-4 pp. 518–532. [Online]. Available: http://dx.doi.org/10.1145/3230543.3230548

[6] (2018) The P4 Language Specification. [Online]. Available: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf

[7] (2018) $P4_{16}$ Language Specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[9] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008. doi: 10.1007/978-3-540-78800-3-24. ISBN 3-540-78799-2, 978-3-540-78799-0 pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3-24

[10] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable Symbolic Execution for Modern Networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016. doi: 10.1145/2934872.2934881. ISBN 978-1-4503-4193-6 pp. 314–327. [Online]. Available: http://dx.doi.org/10.1145/2934872.2934881

[11] A. Kheradmand and G. Rosu, "P4K: A formal semantics of P4 and applications," *CoRR*, vol. abs/1804.01468, 2018. [Online]. Available: http://arxiv.org/abs/1804.01468

[12] G. Roşu, *𝕂: A semantic framework for programming languages and formal analysis tools*, 01 2017, pp. 186–206. [Online]. Available: http://dx.doi.org/10.3233/978-1-61499-810-5-186

[13] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based Program Verifiers for All Languages," *SIGPLAN Not.*, vol. 51, no. 10, pp. 74–91, Oct. 2016. doi: 10.1145/3022671.2984027. [Online]. Available: http://dx.doi.org/10.1145/3022671.2984027

[14] The Reference Manual of the Coq. [Online]. Available: https://coq.inria.fr/distrib/current/refman/