Annales Mathematicae et Informaticae
Accepted manuscript
DOI: https://doi.org/10.33039/ami.2023.03.002
URL: https://ami.uni-eszterhazy.hu

P4Query: Static analyser framework for P4[*]

Dániel Lukács, Gabriella Tóth, Máté Tejfel

Faculty of Informatics, ELTE, Eötvös Loránd University,
Budapest, Hungary
{dlukacs,kistoth,matej}@inf.elte.hu

Abstract. There are many important tasks in a conventional software development process which can be supported by different analysis techniques. P4 is a high level domain-specific language for describing the data plane layer of packet processing algorithms. It has several uncommon language elements and concepts that often make the analysis of P4 programs a laborious task. The paper presents P4Query, an analysis framework for the P4 language that enables the specification of different P4-related analysis methods in a generic and data-centric way. The framework uses an internal graph representation which contains the results of applied analysis methods too. In this way, the framework supports the rapid implementation of new analysis methods in a way where the results will be also easily reusable by other methods.

Keywords: P4 language, static analysis, analysis framework

AMS Subject Classification: 68N20 (Theory of compilers and interpreters)

1. Introduction

Optimization, verification and refactoring are important tasks of a software development process. All of them can be effectively supported by static functional and

[*]The research has been supported by the project "Application Domain Specific Highly Reliable IT Solutions" implemented with the support of the NRDI Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Sub programme) funding scheme.
This research is in part supported by the project no. FK_21 138949, provided by the National Research, Development and Innovation Fund of Hungary.
Supported by the ÚNKP-21-4 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

Submitted: May 31, 2022
Accepted: March 3, 2023
Published online: March 19, 2023

non-functional (e.g. execution time estimation) analysis. This analysis can be especially interesting in the case of languages having uncommon language constructs or language structures e.g. in the case of some domain-specific languages.

P4 [2] is a high level, domain-specific programming language. It is developed mainly for describing the data plane layer of packet processing algorithms of different network devices (e.g. switches, routers) in a protocol and target-independent way. Listing 1 illustrates a P4 program. The program first defines the applied header structure (in rows 1–23), then the parser part (in rows 24–35) describes how the fields of the defined headers will be set from the input bit stream (input packet). Controller parts (see rows 39–62) can modify values of fields of headers and metadata by applying lookup tables. During an application of a lookup table the program finds the appropriate row based on the keys in the table. The keys can be specific fields of the packets or some metadata. After the program finds the right row it will execute the action (usually some modifications on the packet) described by the row. It is important to note that the data plane program only defines the possible actions and describes the structure of the lookup tables, namely the keys of the table and the possible results of the lookups. However concrete data of the tables (which actions will be executed with which parameters for which key values) are defined by the control plane program, therefore it will not appear in P4. Finally, the deparse part (see rows 64–70) defines how the output bit stream (output packet) will be created from the headers.

Listing 1. P4 example.

```
 1  // Definitions
 2  typedef bit<9>  egSpec_t;
 3  typedef bit<48> macAddr_t;
 4  typedef bit<32> ip4Addr_t;
 5
 6  // Headers
 7  header ethernet_t {
 8    macAddr_t dstAddr;
 9    macAddr_t srcAddr;
10    bit<16>   etherType;
11  }
12
13  header ipv4_t {
14    bit<8>    ttl;
15    ip4Addr_t srcAddr;
16    ip4Addr_t dstAddr;...
17  }
18
19  struct headers {
20    ethernet_t   ethernet;
21    ipv4_t       ipv4;
22  }
23
24  // Parser
25  parser MyParser(...) {
26    state start { transition parse_ethernet; }
27    state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
      TYPE_IPV4: parse_ipv4;
      default: accept; } }
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept; }
}

// Control
control MyIngress(in headers hdr, ...) {
  apply {
    if (hdr.ipv4.isValid()) {
          ipv4_lpm.apply();
    }
  }
  action drop() {
    mark_to_drop(standard_metadata);
  }

  action ipv4_forward(macAddr_t dstAddr,
                      egSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
  }
  table ipv4_lpm {
    key = {  hdr.ipv4.dstAddr: lpm;  }
    actions = {
      ipv4_forward;
      drop;
      NoAction;}
    ... }
}

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

The paper describes an analysis framework for P4[1]. The framework makes possible development of different P4-related analysis methods in a generic and modular way. It uses an internal graph representation where the results of the methods are also represented as part of the graph (mainly by adding new edges to the graph). In this way, the methods can use each other's results as well. The proposed tool also allows rapid prototyping of different analytical concepts using a common toolset.

[1]The code is available from `https://github.com/P4ELTE/P4Query`.

2. Related work

Considering related work there are much research applying specific analysis techniques for P4. Most of them concentrate on error checking of P4 programs. For example, Assert-P4 [6] and Vera [15] can check the correctness of predefined properties using annotated P4 source code and Vera can also detect some common errors using custom source code without annotations. They use symbolic execution for the analysis. P4V [7] creates a formula, which describes the proper behavior of the program and checks the satisfiability of it with SMT solver. These three tools are created for an earlier version of P4, namely $P4_{14}$. There are also verification tools, which can manage the newer version ($P4_{16}$) too. BF4 [4] is created as a P4C backend, which can not only detect error possibilities, but it is able to repair them by adding new keys to the lookup tables of the program and modify the table contents. Another tool p4-data-flow [1] uses data flow analysis to detect potential bugs in P4 switch codes.

Some other tools use analytical methods for different purposes. For example, p4pktgen [11] uses symbolic execution for automatically generating test cases. Flightplan [16] can split a P4 program into a set of cooperating P4 programs and maps them to run as a distributed system formed of several, possibly heterogeneous, data planes. During this process they use several analysis techniques, to collect variables whose values need to be transferred between different data planes. SafeP4 [5] is a language which has precise semantics and a static type system that can be used to obtain guarantees about the validity of all headers which are used or modified by the program. The type checker of the language (P4Check) can also check P4 programs executing some static analysis on them.

Comparing these approaches this paper presents a generic framework that allows the implementation of several analyses methods which can use each other's results as well.

A major inspiration for this work was RefactorErl [3], a static analysis tool for Erlang. RefactorErl stores program information in a graph called the Semantic Program Graph using relation databases, and provides its own query language for exploring the stored information. Many features in our work – such as using graph databases and their built-in query languages instead of in-house solutions – can be considered to be the streamlining of best practices found in RefactorErl.

As Section 4.1 introduces, P4Query uses a Gremlin graph database as a storage backend. Recently, other works also leveraged graph databases for static analysis purposes. ProgQuery [14] is a similar static analysis tool for Java, built on Neo4j and its Cypher query language. The authors emphasize that using Neo4j yielded substantial improvements in analysis time and memory usage.

The expressive power of Gremlin is proved in another recent work [18]: the authors store C code information in Neo4j, and define recurring vulnerability patterns as Gremlin queries. Using this approach, the authors discovered 18 previously unknown vulnerabilities in the Linux kernel.

3. Motivation

P4 is a relatively new, domain-specific language having some uncommon language elements (for example match-action tables). The language makes possible the description of the data plane layer of packet processing algorithms. For a real implementation, however, in addition to the program part described in P4, a suitable control plane layer is needed. This part is more or less a black hole while we consider only the P4 source code. For these reasons, testing, verification, and generally functional and non-functional analysis of P4 programs are non-trivial tasks that sometimes require language-specific techniques.

Section 2 introduces several applications using different analysis methods for P4. The authors also have some earlier results for determining potentially erroneous code parts [17] and for predicting execution cost [9] of P4 programs. These methods usually require different analysis techniques, however these techniques often can have very similar subtasks (for example creating an abstract syntax tree or a control flow graph).

This paper presents a framework that allows the implementation of different analysis methods using a common basis. The framework applies an extensible integral graph representation where the results of the different analysis methods are represented also as part of the graph. This makes possible execution of different methods in a hierarchical order where methods can use the results (or some part of the results) of previously applied methods.

The framework supports the rapid implementation of new analysis methods applicable for P4 language in a way where results of the methods will be easily reusable.

4. Analysis framework

Traditional compilers are designed around passes: the frontend passes parse source code into an intermediate representation (IR), midend passes transform and add new information to the IR, and the backend passes create target-specific object code from the IR. Modular compilers like the P4 reference compiler P4C [12] improve this design by structuring the passes into a library: backends assemble their own frontends and midends from a catalogue of passes provided by the compiler. Moreover, P4C allows getting information from older states of the IR, as each transformation pass returns an immutable IR instance. Even here, the three-fold separation of frontend-midend-backend have to remain: in order satisfy midend-dependencies, and subsequently, backend-dependencies, the backend must sequentially execute the frontend, the midend, and finally its own passes.

One goal of the experiment we present here is to relax the three-fold structure and allow passes to reuse (depend on) each other's functionally arbitrarily, and without burdening the compiler programmer with manually finding the right sequence in which to execute the different passes.

4.1. Tool architecture

4.1.1. Description

Figure 1 depicts the four-fold design we propose as a solution for relaxing traditional compiler architecture. In principle, the components called end-user *applications* constitute the backend, i.e. the part that provides useful services to users. (See Section 5 for a few examples of applications built on top of the P4Query.) Superficially, applications provide their services by using the services provided by the *infrastructure* (see Section 4.2), also known as the frontend. In reality, both the infrastructure and applications operate on a large shared *graph* that collects all our knowledge about the program code.

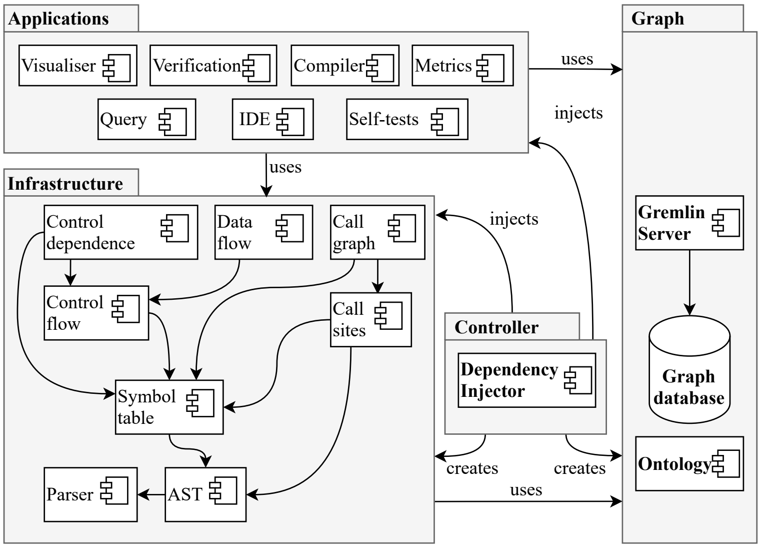

Figure 1. Architectural design of P4Query.

The infrastructure consists of a set of interdependent *analyser* components (or passes): higher-level analysers can only be executed when lower-level analysers already inserted the necessary knowledge into the graph. Similarly to analysers, applications also depend on a subset of the analysers in the infrastructure. But unlike analysers, applications are expected to only read (never modify) the graph, and consequently there are no application-dependent parts in the infrastructure. Applications must also provide a user interface (e.g. command line interface) through which their services can be accessed.

In the middle, the *controller* component ensures (via dependency injection) that all dependencies are satisfied without collisions. To achieve this, components register their provided services and their requirements to the control component, and the controller figures out in which topological order to start the analysers. The controller also guarantees that when the user executes a work-intensive application, only the minimally necessary components will be performed.

The graph is also exposed to the user (not depicted) to enable custom features (e.g. to attach external loggers, visualisers, and validators).

4.1.2. Design goals

Besides proposing a more relaxed structure of analysis passes, we had three additional goals in sight: support for different applications through uniform APIs, ease of extensibility, and data-driven programming.

In systems with *uniform APIs*, programmers have to learn only one paradigm to maintain, extend or otherwise alter the system (e.g. in the case of P4C, visitors and passes are the main concepts of such a uniform API). By supporting different applications (or backends), we mean providing a comfortable way to implement different end-user services, by reusing the same static code analysis operations. P4Query realizes uniform APIs by relying on a graph database. The information in the knowledge graph is accessed using graph queries written in the Gremlin (see 4.1.3) query language. The implication is that users, application developers, and infrastructure developers are using one, uniform data structure (the graph), and are accessing it using the same mechanism (graph queries).

Our second design goal, *ease of extensibility* is also illustrated by Figure 1. The four-fold arrangement was inspired by declarative build systems and the blackboard pattern used in distributed MI. When developers introduce new features, this arrangement enables them to think declaratively: instead of thinking about where to insert their feature inside a sequence of operations, they only have to think about their dependencies, i.e. what kind of analysers could help them.

Finally, our third design goal is *data-driven programming*. Thanks to the uniform graph API and the controller-managed dependency resolution, programmers are forced to think in terms of data instead of code: they have to look at what code knowledge is in the graph already, figure out what data they want to insert, and possibly find existing analysers that make writing queries easier for them. The information in the graph is regulated by a well-defined graph schema, and the graph topology is regulated by the well-defined requirements and services of the analysers. Moreover, since the graph instance is detached from the code analysis framework, the programmers can access it by external tools for visualising, monitoring and validating purposes. Like this, programmers can almost completely avoid understanding the existing code base, and only have to look at and interact with the data in the graph.

4.1.3. Gremlin API

In the tool architecture described in this section, we use a Gremlin graph database as a storage backend (knowledge graph). Gremlin is a compositional query language and API that is implemented by many large-scale graph databases. This makes it possible to change graph implementations with almost no modification to the P4Query code base. In earlier work [8], we also profiled a few graph backends for control flow traversals, and verified that – apart from the initial overhead – in-memory graph databases have comparable performance to built-in memory manipulation.

Another consequence is that analysers have to be implemented as graph query workflows. Since Gremlin is Turing-complete [13], theoretically all of the work can be delegated to the database, and with this, the choice of the workflow language (e.g. Java) can become negligible. Still, in our experience, coarsely granularised queries can hinder code maintainability, as these are often more difficult to read and modify (due to their of lack of common convenience features, e.g. exception handling). For this reason, we still decided to split the workload between the controller and the database.

4.2. Infrastructure

As we see earlier in Figure 1, the heavy-lifting in P4Query is done by various code analyser components, each adding new information about the P4 code to the knowledge graph using what is already there. We now introduce a few analyser modules using an example: Figure 2 depicts a small subset of the knowledge graph taken after we executed control flow analysis, call analysis, and call sites analysis on the P4 code in Listing 1 (specifically the `MyIngress` control). First, the controller finds the topological order of their dependencies, and then starts executing them in order. In this case, the first dependency executed is the parser, parsing the P4 code and filling the knowledge graph with the syntax tree nodes and edges.

Each analyser components adds new edges as an overlay graph. These overlays (domains) are separated by the `dom` edge-attributes: for example `CFG` is the domain introduced by the control flow analysis, and `CALL` is the domain of the call analysis, `SITES` is that of call site analysis. The `role` edge-attribute describes edge semantics inside their domain. For example, `calls` in `CALL` links a procedure to those procedures that it calls, such as `MyIngress` control (node 1) calling the `ipv4_lpm` table lookup (node 8). On the other hand, `calls` in `SITES` links call statements to the called procedure, such as the *direct application* of table `ipv4_lpm` (node 7) calling table `ipv4_lpm` (node 8).

At the same time, the `CFG` domain contains `flow`, `entry`, and `return` edges (among others) to denote the flow of control between various nodes of the syntax tree, and to identify entry and exit nodes. For example, by following these edges, you can see how control flows from `MyIngress` entry point (node 1) through the conditional (node 4), terminating on the call of `ipv4_lpm` (node 8). The figure also partially includes domains of other analysers, such as `SYMBOL`. This analyser creates the graph-equivalent of a symbol table by identifying which declaration declares which name, and links usages of this name in the scope of the declaration to the declaration.

We should note that topological order is, in general, not unique: the controller is free to start independent analysers in any order (even in parallel). This is not a concern as long as analysers can correctly declare their exact dependencies. Still, since all analysers work on the same shared graph, it may happen that – due to

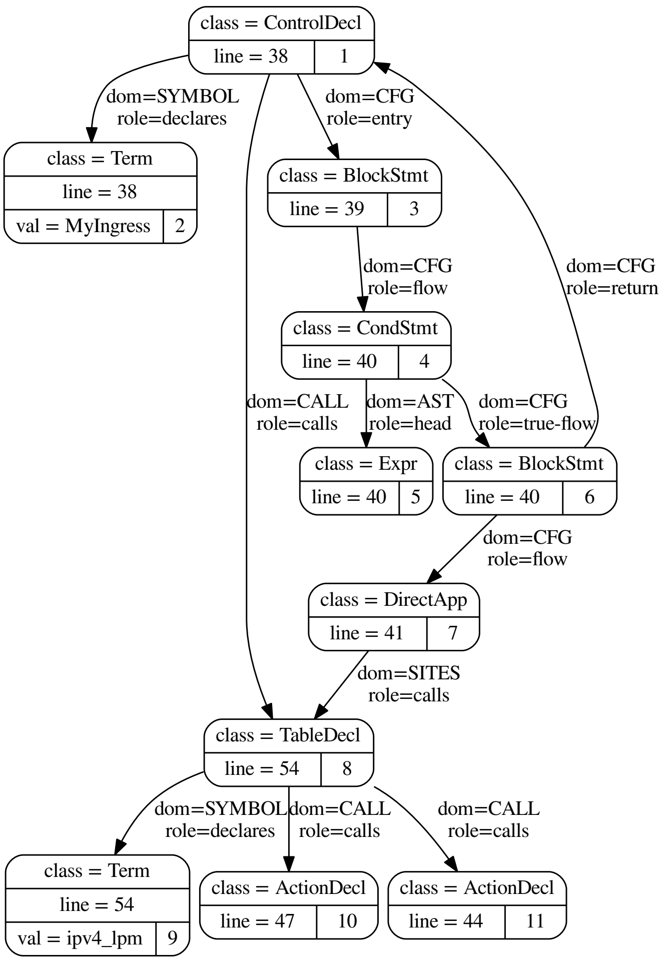

Figure 2. Knowledge graph excerpt of the Listing 1 code.

faulty implementation – an analyser have a "hidden" (unclaimed) dependency. In our experience with the aforementioned analysers, these occurrences are uncommon. Still, to avoid such bugs, we emphasize proper testing (Section 4.3) and recommend implementors to avoid writing general queries (such as selecting *all* elements) and always specify completely the elements to be selected.

4.3. Testing

Testing framework of the tool aims to achieve two main objectives: to provide the correct behaviour of the analysers and to detect possible spoils of the analysers during the development phase. To achieve these goals the tool applies unit tests and integration tests.

Unit tests need to be fast, so they work with the smallest part of the analysers, their functions. One function usually defines one query of the graph, which insert new edges into it, therefore in these cases, the tests check if the right edges are added to the graph. Using an actual P4 source to test these functions would be too costly, therefore we define the most simple graphs to check the function.

While unit tests need to be fast, integration tests can be slower, so we can use P4 files as the inputs to test the analysers. When one analyser needs to be tested, it uses the P4 file and executes every analyser, that it depends on and the tests will check the result of this running.

These tests are important for the P4Query developers, who would like to modify the predefined analysers or supplement the tool with new analysers. After the development of an analyser, the developer can insert the unit tests of the new functions and the correctness of them can be checked by these tests. For unit tests, the developer needs to define the smallest graph, which can cover most of the behaviours of the functions. If the functions are well tested, the developer can continue with integration tests and checks the correct behaviour with real P4 files.

The architecture gives the opportunity to insert this test framework as an application, which depends on all of the tested analysers. As an application, it fits into the tool as a component, which can be easily executed.

5. Case studies

In this section, we illustrate the viability of the platform by showcasing a few applications we are currently building on top of P4Query in related research.

5.1. Visualisation

Since it is the easiest to understand, the first application we introduce is graph visualisation. This application expects a list of analyser component names, executes them, and then, prints a subgraph of the knowledge graph containing only the domains of the analysers in the list. For example, to print the full version of the graph in Figure 2, we should execute P4Query with the following arguments:

```
p4query draw example.p4 –A CFG SYMBOL CALL SITES AST
```

The subcommand `draw` tells P4Query to run the visualiser on the file `example.p4`, while `-A` is a flag (defined by the visualiser UI) expects the analyser names that will be passed to the visualiser application.

A possibly interesting implementation detail here is that the visualiser technically depends on *all* the analysers defined in P4Query, since it must be able to visualise anything the user may pass. Yet, we still managed to avoid executing those that are not requested by the user (and not dependencies of the requested ones): we implemented dependency resolution in the controller using Java dependency injection (DI), and DI offers lazy initialization of the dependencies. This way we can filter the analysers and only initialize those that were requested by the user.

5.2. Verification

Verification is a possible extension of the tool, which is added to it as an application. The main focus is to detect errors and suspicious cases, which can be caused by the use of invalid header or uninitialized fields. The goal of this detection is to report these uses for the developer to avoid undefined behaviour in the programs.

The approach of the checking is defined in our previous paper [17], but in short, it calculates the pre-and post-conditions of the different blocks (i.e the control apply functions, the tables and actions), the parser and the deparser of the program, and based on these condition pairs it can detect improper use of the fields and headers. Three cases can be detected: when there are some errors in a block; when there is any inconsistency between the blocks; and when the post-/precondition of the parser/deparser is inconsistent with the pre-/postcondition of the control function.

Listing 2. Conditions of `MyIngress`.

```
MyIngress:
[
// true condition and ipv4_forward
  (Pre:
     valid:   [ipv4, ipv4.dstAddr, ipv4.ttl,
               ethernet, ethernet.dstAddr],
     invalid: [drop],
   Post:
     valid:   [ipv4, ipv4.dstAddr, ipv4.ttl,
               ethernet, ethernet.dstAddr],
     invalid: [drop]),
// true condition and drop
  (Pre:
     valid:   [ipv4, ipv4.dstAddr],
     invalid: [drop],
   Post:
     valid:   [drop, ipv4, ipv4.dstAddr],
     invalid: []),
// true condition and NoAction
  (Pre:
     valid:   [ipv4, ipv4.dstAddr],
     invalid: [ipv4, ipv4.dstAddr, drop],
   Post:
     valid:   [ipv4, ipv4.dstAddr],
     invalid: [ipv4, ipv4.dstAddr, drop]),
// false condition
  (Pre:
     valid:   [ipv4, ipv4.dstAddr],
     invalid: [ipv4, ipv4.dstAddr, drop],
   Post:
     valid:   [ipv4, ipv4.dstAddr],
     invalid: [ipv4, ipv4.dstAddr, drop]),
]
```

Listing 2 illustrates conditions calculated for control MyIngress in Listing 1. We can see 4 pairs of conditions because it has 4 possible execution paths – there are three where the condition of the branch is true, and the table executes one of the possible actions i.e. *ipv4_forward*, *drop* or *NoAction*, and one where the condition of the branch is false.

This calculation is built into the tool as an application. It uses two experts: the call graph and the control-flow graph. While traversing backwards in the call graph it can reach the applied ("called") actions and tables. Whenever it reaches a vertex like these, it starts to traverse through the proper subgraph of the control-flow graph and calculates the conditions of the actual block. Every condition is stored in the graph as a property of the called vertex of the call graph, therefore when the method reaches the actual call in the control-flow graph – for example a table is called in a control function – it can use the conditions of the called block, which have already been calculated.

5.3. Compiler

In related research [9, 10], we work on a static cost analysis tool for P4: the tool expects as input a P4 program source code together with execution environment parameters, and outputs various metrics (e.g. execution time, energy efficiency) without actually running the P4 program.

In the current paper, we will not go into details on how the cost analysis tool calculates these metrics, but the principle is that we decompose the P4 program into primitive instructions whose expected cost is constant and already known. Implementations of P4 externals such as extern calls (e.g. `packet.extract` in Listing 1) and lookup tables (e.g. `ipv4_lpm` in Listing 1) can also be passed to the tool in the form of these primitive instructions with known costs.

Listing 3. Stack machine code of `MyIngress`.

```
data:
  ...
  headers = 149
  headers.ethernet = 149    // size 114
  ...
  headers.ipv4 = 263
  headers.ipv4.valid = 263
  headers.ipv4.size = 264
  headers.ipv4.srcAddr = 265
  headers.ipv4.dstAddr = 297
  ...
code:
  ...
  // call isValid(hdr.ipv4) on line 144
  214:  load 0          // 0: local address of hdr
  215:  const 114       // 114: size of hdr.ethernet
  216:  add             // address of hdr.ipv4
  217:  invoke 144 1
  // test isValid return value
  218:  ifeq 224
```

```
// call ipv4_lpm(hdr) on line 38
219:   load 0         // 0: local address of hdr
222:   invoke 38 1
223:   pop
// terminate with status OK
224:   const 0
225:   return
```

From this, it follows that part of the static cost analysis problem reduces to a compilation-and-linking problem. As an experiment, we implemented a compiler to solve this problem as an application in P4Query. The main reason we chose P4Query, instead of the much more mature P4C compiler framework was that at first we did not know what kind of representation or code we will need to output: the control and extensibility provided by P4Query and Gremlin queries gave us tools to experiment and create quick, recyclable prototypes to help us arrive at a final vision. While P4C's safety mechanisms (e.g. C++ static type system) support developing stable software, in the case of prototyping and experimentation these same mechanisms are unused, or possibly even slowing down development.

Our current target representation for cost analysis is a sequential stack machine with an instruction set similar to JVM bytecode. Listing 3 depicts the compiler output of `MyIngress` in Listing 1. In the figure all values (bits and sizes) are represented as integers (this is a requirement by our cost analysis approach). Both `isValid` and `ipv4_lpm` have external implementation that had to be linked with the calls. While most P4 targets will not support stack machines, we chose this representation as it is relatively easy to generate, and relatively straightforward to implement. We also believe that as long as we do not count the cost of maintaining the stack, we can still make good cost estimations.

The compiler is built on top of the control flow analyser in P4Query: we traverse the CFG, and process each node by traversing the syntax tree under the node. We also use the call graph to find which label to jump to when a function is called. Thus, much of the compiler state can be delegated to the persistent knowledge graph, and only very specific data (e.g. instruction labels) and linking requires program state outside the graph.

6. Evaluation

Scalability is a very important aspect in the case of analyser tools. For investigating scalability of P4Query we have created dummy P4 programs in which the complexity of the program structure and program logic are increased continuously. In the basic case, two header type were used with one header instances each. The program first parses the two headers, then applies a table which can modify some fields of the headers, and finally it deparses them. In the second program the same structure is applied twice. The four headers are parsed (and finally deparsed) one after the other and two tables are applied sequentially. The first table uses the first header pair, and the second one the second header pair. And so on if the *complexity* of one test program said to be x then there are x header instances of both header types and x tables in the program. As a result if we increase the *complexity* of a test program, its syntax tree will be more complicated and time-consuming to process during different analysis.

Figure 3 illustrates the runtime of P4Query if we execute the CFG analyser (and its dependencies, including the syntax tree and other analysers). We highlighted the results, where the *complexity* of the program is 1, 2, 4, 8 and 16, with a fitted linear regression curve. The diagram shows that the runtime increases in linear time, so we expect P4Query to easily handle even more complex programs. Additionally, we can also inspect the runtime of individual analyses: looking at the corresponding components in each column, we see they are increasing linearly as well, which implies that it is possible to give efficient implementations of the static analysis algorithms in Gremlin.

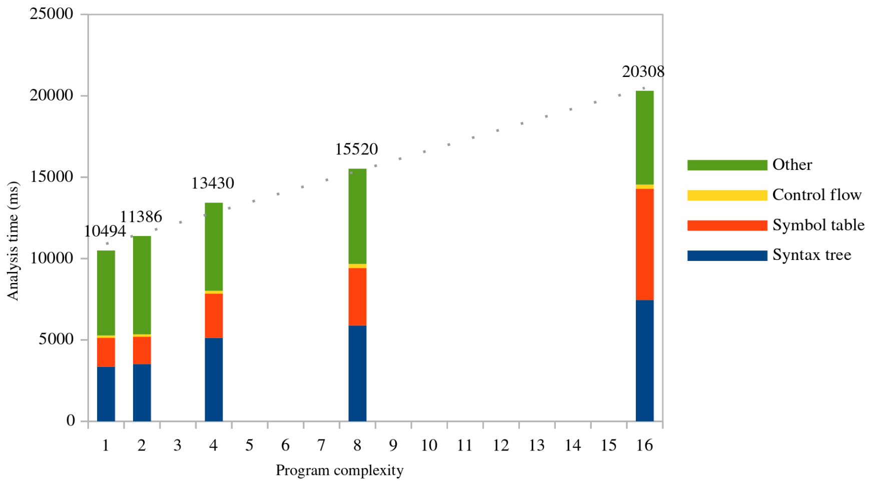

Figure 3. P4Query execution time for different program sizes.

7. Conclusion and future work

Our major purpose was to create a tool, which can facilitate and support the work of P4 developers while making the possibility to experiment with these programs. Its modular structure gives the opportunity for the user to avoid the usage of several tools for different analyses, although it makes the possibility to have all information in one place.

The framework uses a graph representation of the investigated program. All of the analysers are based on the syntax tree of the examined P4 source and they extend it with new edges while creating new subgraphs – like control-flow or call graph – or new labels to store the calculated information.

In the future, we would like to extend the tool with new analyses to give some other useful information for the developers about their P4 source. Our nearest idea is to supplement it with the dependency graph and def-use graph with which we will be able to give report, which are based on the connection between the statements.

References

[1] K. Birnfeld, D. C. da Silva, W. Cordeiro, B. B. N. de França: *P4 Switch Code Data Flow Analysis: Towards Stronger Verification of Forwarding Plane Software*, in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–8, doi: https://doi.org/10.1109/NOMS47738.2020.9110307.

[2] P. Bosshart, et. al: *P4: Programming Protocol-independent Packet Processors*, SIGCOMM Comput. Commun. Rev. 44.3 (2014), pp. 87–95, issn: 0146-4833, doi: http://doi.acm.org/10.1145/2656877.2656890.

[3] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, T. M., M. Tóth: *RefactorErl - Source Code Analysis and Refactoring in Erlang*, in: Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2, Tallin, Estonia, Oct. 2011, pp. 138–148.

[4] D. Dumitrescu, R. Stoenescu, L. Negreanu, C. Raiciu: *Bf4: Towards Bug-Free P4 Programs*, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 571–585, isbn: 9781450379557, doi: https://doi.org/10.1145/3387514.3405888.

[5] M. Eichholz, E. Campbell, N. Foster, G. Salvaneschi, M. Mezini: *How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4*, in: 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom, ed. by A. F. Donaldson, vol. 134, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 12:1–12:28, doi: https://doi.org/10.4230/LIPIcs.ECOOP.2019.12.

[6] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, M. Barcellos: *Uncovering Bugs in P4 Programs with Assertion-Based Verification*, in: Proceedings of the Symposium on SDN Research, SOSR '18, Los Angeles, CA, USA: Association for Computing Machinery, 2018, isbn: 9781450356640, doi: https://doi.org/10.1145/3185467.3185499.

[7] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, N. 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, Budapest, Hungary: ACM, 2018, pp. 490–503, isbn: 978-1-4503-5567-4, doi: http://dx.doi.org/10.1145/3230543.3230582.

[8] D. Lukács, G. Pongrácz, M. Tejfel: *Are Graph Databases Fast Enough for Static P4 Code Analysis?*, in: Proceedings of the 11th International Conference on Applied Informatics 2020, CEUR Workshop Proceedings, 2020, pp. 213–223, url: http://ceur-ws.org/Vol-2650/#paper22.

[9] D. Lukács, G. Pongrácz, M. Tejfel: *Control flow based cost analysis for P4*, Open Computer Science 11.1 (2021), pp. 70–79, doi: https://doi.org/10.1515/comp-2020-0131.

[10] D. Lukács, G. Pongrácz, M. Tejfel: *Model Checking-Based Performance Prediction for P4*, Electronics 11.14 (2022), issn: 2079-9292, doi: https://doi.org/10.3390/electronics11142117.

[11] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, P. Athanas: *P4pktgen: Automated Test Case Generation for P4 Programs*, in: Proceedings of the Symposium on SDN Research, SOSR '18, Los Angeles, CA, USA: Association for Computing Machinery, 2018, isbn: 9781450356640, doi: https://doi.org/10.1145/3185467.3185497.

[12] P4 Language Consortium: *P4C reference compiler for the $P4_{16}$ programming language*, https://github.com/p4lang/p4c, [Online; accessed 06-June-2021], 2017.

[13] M. A. Rodriguez: *The Gremlin graph traversal machine and language (invited talk)*, Proceedings of the 15th Symposium on Database Programming Languages - DBPL 2015 (2015), doi: http://dx.doi.org/10.1145/2815072.2815073.

[14] O. Rodriguez-Prieto, A. Mycroft, F. Ortin: *An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations*, IEEE Access 8 (2020), pp. 72239–72260, doi: https://doi.org/10.1109/ACCESS.2020.2987631.

[15] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, C. Raiciu: *Debugging P4 Programs with Vera*, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, Budapest, Hungary: ACM, 2018, pp. 518–532, isbn: 978-1-4503-5567-4, doi: http://dx.doi.org/10.1145/3230543.3230548.

[16] N. Sultana, et. al: *Flightplan: Dataplane Disaggregation and Placement for P4 Programs*, in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), USENIX Association, 2021, isbn: 978-1-939133-21-2, url: https://www.usenix.org/conference/nsdi21/presentation/sultana.

[17] G. Tóth, M. Tejfel: *Component-based error detection of P4 programs*, Acta Cybernetica (2021), to appear.

[18] F. Yamaguchi, N. Golde, D. Arp, K. Rieck: *Modeling and Discovering Vulnerabilities with Code Property Graphs*, in: 2014 IEEE Symposium on Security and Privacy, 2014, pp. 590–604, doi: https://doi.org/10.1109/SP.2014.44.