P4B: A Translator from P4 Programs to Boogie

Chong Ye
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
yc_thu@163.com

Fei He*
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hefei@mail.tsinghua.edu.cn

ABSTRACT

P4 is a mainstream language for Software Defined Network (SDN) data planes. P4 is designed to achieve target-independent, protocol-independent, and configurable SDN data planes. However, logic errors may occur in P4 programs, resulting in improper packet processing, which may cause serious network errors and information disclosure. In addition, P4 programs contain many branches and thus are more challenging to ensure correctness.

Formal verification is a powerful technique to verify the correctness of P4 programs. Unfortunately, current P4 verification studies lack basic toolchains, and their intermediate languages are not expressive enough. We present P4B, an efficient translator from P4 programs to Boogie, a verification-oriented intermediate representation. We provide formal translation rules to ensure the correctness of the translation process. The translated results can be verified by the toolchain of Boogie. We conducted experiments on 170 P4 programs collected from GitHub, and the experimental results demonstrate that our translator is useful and practical.

The screencast is available at https://youtu.be/8_rEj3QFQeM. The tool is available at https://github.com/Invincibleyc/P4B-Translator.

CCS CONCEPTS

• **Networks** → **Network reliability**; • **Software and its engineering** → **Formal software verification**.

KEYWORDS

Software defined networking, formal verification, P4 programming language, data plane

ACM Reference Format:
Chong Ye and Fei He. 2023. P4B: A Translator from P4 Programs to Boogie. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613091

*Fei He is the corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613091

1 INTRODUCTION

Programming protocol-independent packet processors (P4) [2] is a domain-specific language designed for the data plane of Software Defined Network (SDN). Compared with traditional networks, the main advantage of P4 is programmability. Both parameters and the function of P4 switches can be reconfigured. P4 is designed to be reconfigurable, protocol-independent, and target-independent. P4 greatly simplifies the development of new network protocols with abstraction on the forwarding model and well-designed APIs. In addition, it supports user-defined header format and actions.

The programmability provided by P4 also brings more challenges. Users may easily introduce bugs to their network devices when implementing protocols. Moreover, match-action rules are not provided at compile time, so programmers may leave corner cases unhandled. Formal methods are needed to ensure safety and correctness. However, the P4 Community currently has not provided formal semantics for the P4 language, and the details of some features are not well-defined and even contradictory [8], which makes verification and static analysis tools hard to design.

An efficient solution is translating P4 to an intermediate language with a full verification toolchain because P4 semantics can be clearly described, and the toolchain can verify the translation results. In this paper, we choose Boogie as the intermediate language. Boogie[1] is a powerful verification-oriented language designed by Microsoft Research. Based on Boogie, many verifiers have been developed, including Corral [11], VCC [4], Dafny [12], and HAVOC [3]. These tools are widely used in commercial and academic fields. Furthermore, Boogie's grammar features are similar to high-level programming languages, with built-in supports of pre-conditions, post-conditions, and assertions, making Boogie very competent being the intermediate language.

This paper presents P4B, a translator from P4 programs to Boogie. In our implementation, P4B is integrated into the P4 compiler provided by P4 Community. It supports P4 programs in both $P4_{14}$ [5] and $P4_{16}$ [6] standards and follows the P4 specifications. P4B models P4 components and abstracts P4 APIs for verification and analysis purposes. It also adds constraints on the initialization and value range of global variables. After translation, desired properties can be described as assertions and verified by the toolchain.

Recently, researchers have developed verification tools for P4. Most of them use symbolic execution, such as Vera [15] and ASSERT-P4 [10]. They also translate P4 into intermediate languages: SEFL and C. p4v [14] uses GCL to describe P4 semantics and can efficiently verify P4 programs. bf4[9] follows the idea of p4v and can fix some of the bugs found in P4 programs. However, these tools

lack translation rules, and their intermediate languages used are not designed for verification purposes. Compared with these tools, we believe our tool is competent because it provides formal translation rules to support main P4 features, and the intermediate language Boogie has a powerful toolchain for further process.

To evaluate P4B, we collected 170 P4 programs from Github and used P4B to translate them to Boogie. All the programs can be translated in under 0.5 min. And all the translated programs are grammatically correct.

2 BACKGROUNDS

In this section, we introduce the core syntax of P4 and Boogie, which is the basis of our translation rules in section 3. We also introduce the forwarding model of P4 programs.

2.1 P4 Syntax

Figure 3 shows the core syntax of P4, including types, expressions, and statements. $\overline{x}$ describes the vector of x. $X\{\overline{x}\}$ shows that X has members $\overline{x}$.

P4 basic types include bit_n and $bool$. A variable of type bit_n is a bit vector with length n. $bool$ is the boolean type. Based on basic types, P4 supports headers and structs. $header\{\overline{field}\}$ stores the protocol fields in packets. $struct\{\overline{field}\}$ is similar to structures in high-level languages. $field < \rho >$ represents the fields of headers and structs, and the field type ρ can be any P4 type. P4 also supports $headerStack$, which is a stack of headers with a constant size.

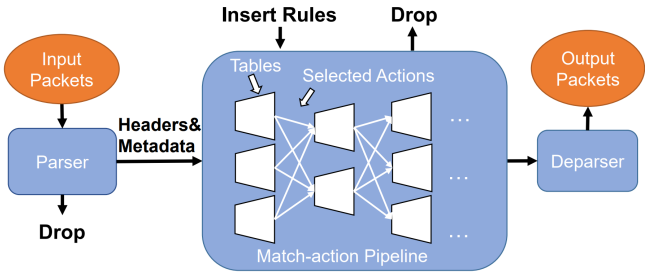

Figure 1: The forwarding model of P4

$$\rho_b = translate(\rho_p) = \begin{cases} bool, & \rho_p = bool \\ int, & \rho_p = int \\ bv_n, & \rho_p = bit_n \\ Ref, & \rho_p = header \\ Ref, & \rho_p = struct \\ [Ref]\,\rho'_b, & \rho_p = field < \rho'_p >, \\ & where\ \rho'_b = translate(\rho'_p) \\ [int]\,Ref, & \rho_p = headerStack \end{cases}$$

Figure 2: The translation rules of P4 types

As for expressions, P4 provides constants c and variables x. Common unary and binary operations are supported. P4 provides $x[e]$ for accessing arrays and stacks. For bit vectors, the expression $e[e_1 : e_2]$ extracts bits from the index e_1 to e_2. $(\rho)e$ is used to transform e to type ρ.

P4 supports multiple statement formats. Among them, assignment statements, conditional statements, and statement sequences are similar to statements in high-level languages and will not be introduced here. $transition\ parserState$ is used in the P4 parser for transition between parser states. Function call statements include calls to parsers, parser states, tables, actions, and library functions.

2.2 Boogie Syntax

Figure 4 is the core syntax of Boogie. Boogie supports bit vectors, booleans, and integers. Boogie also allows users to define mapping types $[\rho_1]\rho_2$, which maps ρ_1 to ρ_2. Constants and variables are basic expressions. Moreover, Boogie supports array accessing and bit vector slicing. Besides common statements such as assignment statements, conditional statements, and statement sequences, Boogie provides statements for verification purposes. Assumption statements and assertion statements are used to describe verification properties. The statement $havocx$ assigns x to a random value.

2.3 Forwarding Model

A typical P4 program that follows the standard execution model, V1Switch[1], declares the format of packet headers and is executed in the order of the parser, the match-action pipeline, and the deparser. The Figure 1 shows the forwarding process of P4's V1Switch model. P4 programs interact with the network environment to receive packets and forwarding rules. The parser parses P4 headers and then the match-action pipeline modifies packets based on forwarding rules. Finally, the deparser encapsulates packets and forwards them to the output port. There are also other execution models, such as Tofino. Compared to V1Switch, Tofino is a complex commercial model, but the core components are similar.

3 TRANSLATOR DESIGN

Based on the syntax of P4 and Boogie, we introduce the translation rules from P4 to Boogie in this section. In each translation rule, the translation details are indicated above the line, and the translation results are provided below the line. Translation rules use formulas such as $\vdash src \hookrightarrow dst$ to translate from P4 expression src to Boogie expression dst. Similarly, $\vdash src \rightsquigarrow dst$ represents the translation from P4 statement src to Boogie statement dst. Next, we introduce the translation of types, expressions, declarations, and statements.

Types. Figure 2 shows the translation of types. The function $\rho_b = translate(\rho_p)$ defines the translation from P4 type ρ_p to Boogie type ρ_b. Both P4 and Boogie support bit vectors, booleans, and integers, and the translation is trivial. For header and struct instances, we define the Boogie type Ref to represent the reference to an instance. And fields are translated to $[Ref]\rho'_b$, which maps an reference to its fields. We use $[int]Ref$ to represent the header stack, which maps the index to the corresponding header.

Expressions (Figure 5). We use e and ϵ to represent expressions of P4 and Boogie, respectively, and x and χ to represent variables of P4 and Boogie, respectively. When translating binary operations, unary operations, slicing, and stack accessing, we first translate the expressions that participate in the operation and then perform

[1]https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

$$\rho ::= bit_n \mid bool \mid header\{\overline{field}\} \mid field < \rho > \mid struct\{\overline{field}\} \mid headerStack \qquad headerStack ::= header[c]$$
$$e ::= c \mid x \mid e[e] \mid e\ \mathbf{op}\ e \mid \mathbf{op}\ e \mid e[e:e] \mid (\rho)e \qquad op ::= + \mid - \mid * \mid ...$$
$$s ::= x := e \mid \mathbf{if}(e)\{s\}\ \mathbf{else}\{s\} \mid s;s \mid \mathbf{transition}\ parserState \mid call\ function(\overline{e})$$
$$function ::= parser(\overline{e})\{\overline{parserState}\} \mid parserState(e)\{s\} \mid table(\overline{key}, \overline{action}) \mid action(\overline{e})\{s\} \mid control(\overline{e})\{s\} \mid externFunc$$
$$externFunc ::= extract(header) \mid mark\_to\_drop() \mid hash(e,e) \mid setValid(x) \mid setInvalid(x)$$

Figure 3: The Syntax of P4 Language

$$\rho ::= bv_n \mid bool \mid int \mid [\rho]\rho \qquad e ::= c \mid x \mid x[e] \mid e\ \boldsymbol{op}\ e \mid \boldsymbol{op}\ e \mid e[e:e] \qquad op ::= + \mid - \mid \mid \mid ...$$
$$s ::= x := e \mid assume(e) \mid assert(e) \mid havoc\ x \mid if(e)\{s\}\ else\{s\} \mid call\ procedure() \mid x := function() \mid s;s$$

Figure 4: The Syntax of Boogie Language

$$\frac{\vdash e_1 \hookrightarrow \epsilon_1 \quad \vdash e_2 \hookrightarrow \epsilon_2}{\vdash e_1\ bop\ e_2 \hookrightarrow \epsilon_1\ bop\ \epsilon_2}\ (Binary\ Operation) \qquad \frac{\vdash e \hookrightarrow \epsilon}{\vdash uop\ e \hookrightarrow uop\ \epsilon}\ (Unary\ Operation)$$

$$\frac{\vdash e \hookrightarrow \epsilon \quad \vdash e_2 \hookrightarrow \epsilon_2 \quad \vdash e_3 \hookrightarrow \epsilon_2}{\vdash e_1[e_2:e_3] \hookrightarrow \epsilon_1[\epsilon_2:\epsilon_3]}\ (Slicing) \qquad \frac{\vdash x \hookrightarrow \chi \quad \vdash e \hookrightarrow \epsilon}{\vdash x[e] \hookrightarrow \chi[\epsilon]}\ (Header\ Stack\ Accessing)$$

$$\frac{\begin{array}{l}\vdash e : bit_{n_2} \hookrightarrow \epsilon_1 : bv_{n_2} \\ if(n_1 > n_2)\ \epsilon \equiv 0bv_{n_1-n_2} + +\epsilon_1 \\ else\ \epsilon \equiv \epsilon_1[n_1-1:0]\end{array}}{\vdash (bit_{n_1})e \hookrightarrow \epsilon}\ (Bit\ Vector\ Cast) \qquad \frac{\begin{array}{c}\rho_b \equiv translate(\rho_p) \\ \vdash x : header \hookrightarrow \chi : Ref \\ \vdash f : field < \rho_p > \hookrightarrow m : [Ref]\rho_b \in GlobalVars\end{array}}{\vdash x.f \hookrightarrow m[\chi]}\ (Field\ Accessing)$$

Figure 5: The translation rules of P4 expressions

$$\frac{\rho_b \equiv translate(\rho_p)}{\vdash f : field < \rho_p > \hookrightarrow \chi : [Ref]\rho_b}\ (Field) \qquad \frac{\vdash \overline{f : field} \hookrightarrow \overline{\chi}_f \quad \overline{\chi}_f \in GlobalVars}{\vdash x : header\{\overline{f : field}\} \rightsquigarrow \chi : Ref}\ (Header)$$

$$\frac{\vdash \overline{f : field} \hookrightarrow \overline{\chi}_f \quad \overline{\chi}_f \in GlobalVars}{\vdash x : struct\{\overline{f : field}\} \rightsquigarrow \chi : Ref}\ (Struct) \qquad \frac{\vdash \overline{s} \hookrightarrow \overline{\omega}}{\vdash parserState\ p(\ )\{\overline{s}\} \rightsquigarrow procedure\ p(\ )\{\overline{\omega}\}}\ (ParserState)$$

$$\frac{\begin{array}{c}\vdash \overline{x} \hookrightarrow \overline{\chi} \quad \vdash \overline{parserState} \rightsquigarrow \overline{\omega}_{state} \\ \omega_{parser} \equiv procedure\ p(\overline{\chi})\{call\ start(\ );\}\end{array}}{\vdash parser\ p(\overline{x})\{\overline{parserState}\} \rightsquigarrow \omega_{parser};\ \overline{\omega}_{state}}\ (Parser) \qquad \frac{\begin{array}{c}action\_run_t \in GlobalVars \quad act \in \overline{action} \\ \omega_{select_{act}} \equiv if(action\_run_t == act)\ call\ act(); \\ \omega \equiv havoc\ action\_run_t;\ \overline{\omega}_{select_{act}}\end{array}}{\vdash table\ t(\overline{key}, \overline{action}) \rightsquigarrow procedure\ t(\ )\{\omega\}}\ (Table)$$

$$\frac{\vdash \overline{x} \hookrightarrow \overline{\chi} \quad \vdash \overline{s} \hookrightarrow \overline{\omega}}{\vdash action\ a(\overline{x})\{\overline{s}\} \rightsquigarrow procedure\ a(\overline{\chi})\{\overline{\omega}\}}\ (Action) \qquad \frac{\vdash \overline{x} \hookrightarrow \overline{\chi} \quad \vdash \overline{s} \hookrightarrow \overline{\omega}}{\vdash control\ c(\overline{x})\{apply\{\overline{s}\}\} \rightsquigarrow procedure\ c(\overline{\chi})\{\overline{\omega}\}}\ (Control)$$

Figure 6: The translation rules of P4 declarations

operations on the resulting expressions. The translation of casting considers the size of the source and destination types and decides to add zeros or perform slicing. Field accessing uses the reference to a header or a struct to access the value.

Declarations (Figure 6). The declaration of variable x is written as $x : type$. For header and structure declarations, we first translate each field and then add the translated variables to the global variable set *GlobalVars*. The translation of fields follows the rules of types above, and each field is translated to a new variable. P4 parsers, states, actions, and controls are similar to functions in high-level languages and are translated to Boogie procedures. Note that the parser always calls the *start* state first. The translation of P4 tables is complex because we need to simulate the matching process. Since

$$\frac{\vdash e \hookrightarrow \epsilon \quad \vdash s_1 \rightsquigarrow \sigma_1 \quad \vdash s_2 \rightsquigarrow \sigma_2}{\vdash if(e)\{s_1\}\ else\{s_2\} \rightsquigarrow if(\epsilon)\{\sigma_1\}\ else\{\sigma_2\}}\ (Condition) \qquad \frac{}{\vdash transition\ parserState \rightsquigarrow call\ parserState();}\ (Transition)$$

$$\frac{\vdash e_1 \hookrightarrow \epsilon_1 \quad \vdash e_2 \hookrightarrow \epsilon_2}{\vdash e_1 := e_2 \rightsquigarrow \epsilon_1 := \epsilon_2}\ (Assignment) \qquad \frac{drop : bool \in GlobalVars}{\vdash mark\_to\_drop(\ ) \rightsquigarrow drop := true}\ (Drop)$$

$$\frac{\vdash x \hookrightarrow \chi \quad isValid : [Ref]bool \in GlobalVars}{\vdash extract(x : header) \rightsquigarrow isValid[\chi] := true}\ (Extract) \qquad \frac{\vdash x \hookrightarrow \chi \quad isValid : [Ref]bool \in GlobalVars}{\vdash setValid(x : header) \rightsquigarrow isValid[\chi] := true}\ (SetValid)$$

$$\frac{\vdash x \hookrightarrow \chi \quad isValid : [Ref]bool \in GlobalVars}{\vdash setInvalid(x : header) \rightsquigarrow isValid[\chi] := False}\ (SetInvalid) \qquad \frac{\vdash x \hookrightarrow \chi \quad \vdash e_1 \hookrightarrow \epsilon_1 \quad \vdash e_2 \hookrightarrow \epsilon_2 \quad \omega \equiv havoc\ \chi;\ assume(\epsilon_1 \leq \chi \leq \epsilon_2)}{\vdash hash(x, e_1, e_2) \rightsquigarrow \omega}\ (Hash)$$

Figure 7: The translation rules of P4 statements

Table 1: Translation and Verification Results

P4File	loc	table	action	control	parser state	translation time (s)	total time (s)	bugs (p4b)	bugs (bf4)
07-FullTPHV2.p4	788	6	2	5	1	0.01962	0.63858	0	0
05-FullTPHV.p4	834	6	2	5	1	0.02548	0.69951	0	0
08-FullTPHV3.p4	1051	6	2	5	1	0.01508	1.05674	0	0
header-stack-ops-bmv2.p4	1424	84	168	8	3	0.02002	1.47222	0	0
block.p4	5767	110	381	70	67	1.05362	22.1443	> 50	160
switch.p4	6473	129	460	77	66	1.39436	27.2275	> 85	176
...									
total (170)	35955	678	1516	1000	446	2.808283	95.415345	> 189	511

forwarding rules are not provided, we regard the matching process as random and select an action to execute randomly.

Statements (Figure 7). We use s and ω to represent statements of P4 and Boogie, respectively. The translation of assignment and condition statements is trivial. Transitions are translated to method calls. API functions are translated to Boogie functions. *mark_to_drop* marks the packet to be dropped. *extract*, *setValid*, and *setInvalid* change the header validity. *hash* assigns the x randomly.

To improve the practicality of P4B, we automatically insert assertions for header validity[14], which is the most concerned property in current P4 verification studies. Header validity requires P4 headers to be valid before being accessed, avoiding access to uninitialized memory. We insert an assertion before each header access.

4 IMPLEMENTATION AND EVALUATION

We implemented P4B using about 2000 lines of C code. P4B works as the back-end of the compiler P4C [7]. First, we reuse the front-end of P4C, which parses a P4 program into AST. Then we follow the translation rules to translate P4 into Boogie.

We collected 170 publicly available P4 programs from GitHub and literature [10, 15]. These programs cover most of the language features of P4, including headers definition, match-action pipeline, forwarding behaviors, etc. All experiments were performed on a laptop with an Intel Core i7-7700HQ CPU and 8GB RAM.

Table 1 shows the experimental results of the six largest programs and a summary of full benchmarks. The first six columns show the basic information. The next two columns show the execution time of our translator and the total tool. Note that the total time contains the time of the front-end and the translator. The results show that most time is spent on the front-end, proving our tool's efficiency. We also check the syntax correctness, and experiments show that our translation results can be parsed and verified by the most popular Boogie back-ends, i.e., the Boogie verifier[1], Corral[11], and Symbooglix[13].

We also compared our tool with BF4 [9] for verifying header validity. We verified with Boogie back-ends and found more than 189 bugs. After manual confirmation, these bugs we reported are real bugs also found by BF4, proving the basic correctness of our translation process. The verification timeout is 4 hours. For large P4 programs such as switch.p4, Boogie back-ends cannot finish in 4 hours, so we report fewer bugs than bf4. Boogie back-ends perform general algorithms, but BF4 provides efficient algorithms for P4 verification, which is beyond the design purpose of our translator and may be studied in our future work.

5 CONCLUSION

This paper presents P4B, a tool that translates network data plane programs written in P4 to Boogie following the forwarding model. After translation, we can insert `assert` statements to describe desired properties. We currently consider action matching to be random. In future work, we plan to add abstraction and provide APIs to describe constraints for the control plane.

ACKNOWLEDGMENTS

This work was supported in part by the National Key Research and Development Program of China (No. 2018YFB1308601), and the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

REFERENCES

[1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 364–387. https://doi.org/10.1007/11804192_17

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Computer Communication Review* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

[3] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. 2007. A Reachability Predicate for Analyzing Low-Level Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 19–33. https://doi.org/10.1007/978-3-540-71209-1_4

[4] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*. Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[5] P4 Langhuage Consortium. 2018. *$P4_{14}$ language specifcation.* https://p4.org/p4-spec/p4-14/v1.1.0/tex/p4.pdf

[6] P4 Langhuage Consortium. 2022. *$P4_{16}$ language specifcation.* https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html

[7] P4 Langhuage Consortium. 2023. *P4 reference compiler.* https://github.com/p4lang/p4c

[8] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: formal foundations for p4 data planes. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434322

[9] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, Henning Schulzrinne and Vishal Misra (Eds.). ACM, 571–585. https://doi.org/10.1145/3387514.3405888

[10] Lucas Freire, Miguel C. Neves, Lucas Leal, Kirill Levchenko, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*. ACM, 4:1–4:7. https://doi.org/10.1145/3185467.3185499

[11] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. 2011. Corral: A whole-program analyzer for Boogie. In *These informal proceedings contain the papers presented at BOOGIE 2011, the First International Workshop on Intermediate Verification Languages held on 1 August 2011 in Wroc law. There were 8 submissions. Each submission was reviewed by at least 3 pro-gram committee members. The committee decided to accept 7 papers, repre.* Citeseer, 78.

[12] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[13] Daniel Liew, Cristian Cadar, and Alastair F Donaldson. 2016. Symbooglix: A symbolic execution engine for Boogie programs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 45–56.

[14] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, Sergey Gorinsky and János Tapolcai (Eds.). ACM, 490–503. https://doi.org/10.1145/3230543.3230582

[15] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, Sergey Gorinsky and János Tapolcai (Eds.). ACM, 518–532. https://doi.org/10.1145/3230543.3230548

Received 2023-05-11; accepted 2023-07-20