The effect of a corrupt program on virtualized P4 programs in HyperVDP

by

Ruben Couwenberg
Supervisors: Fernando Kuipers, Chenxing Ji

A Paper
Submitted to EEMCS faculty
Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 19, 2022

Abstract

After software defined networking (SDN) separated the control-plane from the data-plane, P4 was proposed as a solution to be able to program the data-plane. The programmable data plane (PDP) is very useful to alter the behaviour of programmable network devices. The drawback, however, is that without virtualization only one single P4 program can run at a time on the PDP. Compiler based and hypervisor based approaches can be used to virtualize the P4 data-plane to let P4 programs run alongside each other.

This increases the flexibility when compared to P4, but can potentially come with added risks. Hypervisor based approaches share resources, while compiler based approaches try to minimize the sharing of resources. This opens up hypervisor based approaches, like HyperVDP and Hyper4, to attacks from a corrupt P4 program. Because of the resource sharing, when one of the virtualized P4 programs in HyperVDP is corrupted, there potentially is a risk that the other virtualized programs also get influenced.

This paper will attempt to answer the question; can a malicious P4 program corrupt behaviour of another P4 program while running alongside each other. This will be done by laying out a method to answer this question using HyperVDP. A repository containing the updated source code of HyperVDP will also be created and provided to allow for a stable framework.

1 Introduction

In the current design of network devices, the control plane determines how the network packets should flow. The data-plane subsequently uses the control-plane rules on how to actually forward these packets. Because this design can be quite limiting in flexibility, the software defined networking (SDN) architecture was developed [8]. SDN separated this control plane of the underlying data plane allowing for greater flexibility in the control plane, especially when combined with OpenFlow [3], a control protocol for SDN. This makes it easier for programmers to reconfigure how data packets are handled on the control plane [7].

Following the advancements in SDN, the next step was to have more flexibility in the data plane as well. With the traditional design of network devices it is only possible to change the configuration of these devices in the control-plane. When functionalities are needed that the networking device does not have, only the manufacturer of the device can implement this. To convince the manufacturer to implement a new functionality for you would take a huge amount of effort and time. Consequently, to mitigate these issues, the Programmable Data Plane [9] (PDP) and the Programming Protocol-independent Packet Processors language P4 [2] were proposed.

When P4 was first introduced, it provided a state-of-the-art method to customize the functionality of data-planes, all while being open source. P4 aims to make it viable for ”programmers to be able to change the way switches process packets once they are deployed. [2, p. 1]” Thus allowing programmers themselves to implement the missing functions, drastically cutting down on the development time.

While the programmable network device does need a chip that supports P4, it is a protocol independent language. Meaning that P4 is not relying on existing packet formatting and does not have built-in support for any network protocols, e.g. IP, Ethernet, TCP. The programmer needs to specify what should happen with the data packets and make the data packets parsable in the P4 program.

Due to the recent advancements in programmable switches and the P4 language itself, the added flexibility of having multiple P4 programs running alongside each other (i.e. virtualization) in the PDP is becoming increasingly important. Virtualizing programs comes with its own difficulties however. These difficulties include the fact that the virtualized programs not only need to efficiently run alongside each other but also need to share the same physical hardware resources. HyperVDP has introduced viable methods of virtualization, while keeping the performance comparable to P4 [13].

Nonetheless, the benefits of HyperVDP could come with added risks. If programs share memory resources on the programmable network device it could potentially be possible for the programs to corrupt each other's data. To this end, this research will inspect the security vulnerabilities of virtualization when using HyperVDP. The specific question this research will attempt to answer is if a malicious P4 program could corrupt behaviour of another P4 program when they are virtualized alongside each other.

This main question will be subdivided in the following sub questions:

- Can P4 programs make use of the same data when running alongside each other?
- Can a P4 program influence data another P4 program makes use of when running alongside each other?
- Can a malicious P4 program corrupt the P4 programs that use the same data?

In the following section 2, the relevant background of P4, HyperVDP, compiler based and hypervisor based approaches will be explained more in depth. Section 3 will discuss the methodology used to answer the research question and the necessities for the testing environment are introduced. Section 4 will contain the details of the experiments performed and discuss the

results of these performed experiments. In section 5 the ethical aspects of the research will be discussed together with the thoughts that went into making the research responsible. Section 6 will discuss the results and compare this research to existing papers. Lastly, in section 7 the research conclusions will be provided together with possible improvements and new questions that arose during this research, combined with recommendations for future works.

2 Background

Additional background of HyperVDP and virtualization will be provided in the following subsections to get a better grasp of this research.

2.1 Compiler vs hypervisor based approach

Like Han et al. (2020) notes, there are two ways to approach virtualization in the P4 data-plane, compiler based and hypervisor based.

The compiler based approach focuses on separating the used resources and functionalities between the virtualized P4 programs. This is done by merging the programs into one P4 configuration file [5]. After merging the P4 programs, the configuration file is installed onto the target switch. The consequence is that if it is needed to reconfigure or delete one of the virtualized P4 programs, the configuration file has to be compiled again. Resulting in down-time of the network while this is in progress.

The other approach is hypervisor based. This approach creates a platform where the P4 programs can be installed on. By creating this platform and not focusing on the separation of resources, like the compiler based approach, there is potentially data shared between the P4 programs. HyperVDP [13] and Hyper4 [6] are two of these hypervisor based approaches. These hypervisor based approaches allow for programmers to modify the loaded P4 programs at runtime. When compared to Hyper4 (Figure 1), HyperVDP at runtime not only saves on the number of declared tables but additionally also saves on the amount of bits that are used for metadata. Zhang et al. showed that HyperVDP has on average a 2.5x performance advantage on Hyper4 in terms of bandwidth and latency while reducing the resource usage by a four fold [13, p. 2]. In this paper we will examine the hypervisor based approach HyperVDP more closely.

NFs	Native P4	Hyper4	HyperVDP
Switch	2	13	5
Firewall	3	22	8
Router	4	28	16
ARP Proxy	4	48	10

Figure 1: Table usage at runtime [13]

The fact that the compiler based approach tries to mitigate resource sharing while the hypervisor approach does not, makes it easier to potentially corrupt a P4 program that is running alongside a malicious P4 program with the hypervisor based approach. This data sharing will be used to our advantage when attempting to find a vulnerability in section 3.

2.2 HyperVDP

HyperVDP not only has a performance edge on it's hypervisor companion Hyper4, but also provides more flexibility than the native P4 language. In native P4, if a network function that is loaded on the data-plane needs to be reconfigured or removed, the table entries of the populated match-action tables get lost. This is due to the workflow of P4, in which the program needs to be recompiled and the tables repopulated after each change is processed.

HyperVDP solves this problem by creating virtual PDPs on top of the physical PDP, as is visualised in figure 2. This allows for multiple PDPs to run on the physical PDP and ensures that when a network function needs to be reconfigured, the state of the forwarding tables can be maintained. This omits the restraining nature of native P4 regarding the reconfiguring of the loaded P4 program on the switch. The reconfigurability, introduced by virtualized PDPs, at runtime is an especially welcome advantage when other network functions on the PDP need to keep running while reconfiguration is in progress.

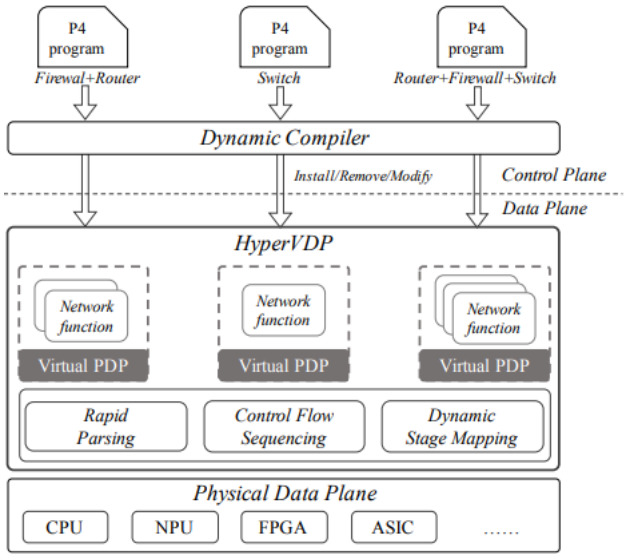

Figure 2: Design of HyperVDP [13]

3 Methodology

This research will be performed with the steps explained in the following subsections.

3.1 Test network simulation

First, to begin the attempt to answer the research questions there is a test environment needed. Mininet [4] will be used to simulate a testing network. Mininet is a tool for easy creation of testing networks and can use a pre-specified topology when creating its simulation network to ensure that the topology is thought out and works for the use case of the programmer. To this end there is a predefined topology file supplied in the repository of this research.

3.2 Behavioral model version 2

The behavioral model version 2 (BMv2) is used as a reference P4 software switch[1]. This is the software switch that would usually run the P4 program. In this research the functions of the software switch BMv2 are extended, since it should be used as the target where HyperVDP will be implemented on to virtualize the P4 programs.

3.3 Mininet using HyperVDP

The created test network needs to be instantiated using the BMv2 software switch that is capable of running HyperVDP. The source code of HyperVDP is available on Github[2]. Because the source code is now outdated, it does not work right out of the box. While effort has been put in updating the code, it still has some outdated sections, leaving some test cases that do not work. The progress of fixing and updating the source code has been uploaded[3] in a repository forked from the HyperVDP source code.

Once the simulated switch instance is created with the HyperVDP source code as intended, this switch can be used by mininet in the simulated network. This network now has a switch capable of using the virtualization that HyperVDP provides.

3.4 HyperVDP

When the test network is running multiple test network functions can be loaded onto the switch that is running in the mininet environment. These test functions are provided in the HyperVDP source code. They can be used to test the implementation of HyperVDP. Specifically this means that the match-action tables should be populated with the rules that the test network functions provide.

The source code of HyperVDP provides multiple of these test functions. However, the authors warn that these functions may not work. This warning is indeed correct as the rules of these functions throw error responses. Significant progress has been made towards a stable framework. This progress is uploaded to the Github repository of this research.

[1]https://github.com/p4lang/behavioral-model

[2]https://github.com/HyperVDP/HyperV

[3]https://github.com/2016Ruben/HyperV

3.5 Testing and probing with scapy

After the test network is set up and running, the network will be tested and probed in search of answers to the proposed research questions.

The first matter that needs to be established to answer the research questions will be if it can cause any issues when the same data is used by the virtualized programs. The main question regards a corrupted program that is running alongside other virtualized programs. Corrupted in this research will be interpreted as a program that is under (partial) control of a person trying to intentionally disrupt the other programs.

Scapy [1, 12] is used to send and receive packets on the ethernet interfaces that are created by the HyperVDP source code. These packets can be meticulously crafted using scapy to ensure the programmer has total control of the data that is sent to the switch running HyperVDP. Furthermore, scapy can also receive the responses given by the switch. Ensuring that the programmer can immediately see what has been done to the packet that was sent.

4 Experimental Setup and Results

To investigate how a malicious program, that is loaded in a virtualized data-plane by HyperVDP, can disrupt other programs, the following steps are to be taken.

To be able to use mininet in conjunction with a simulated switch that is using HyperVDP, we first need to create the file for this switch with the source code of HyperVDP. Second, we need to specify that we want HyperVDP to use the BMv2 to create the switch. Once the code from this research's repository has been cloned the project should be build using the make file. This takes the p4 source file and parses it into a JSON simple switch file.

The simple topology setup used for mininet is the following. There are 3 hosts (h1, h2, h3) bidirectionally connected to one switch (s1), see figure 3. This switch will be running with the simple_switch file previously created by HyperVDP. Using this format we can concentrate the network around one simulated HyperVDP switch to run the packets through. Then mininet can be used in conjunction with this simple custom topology and the simulated HyperVDP switch to simulate the test network.

Once this is running, the match-action tables should be populated with the test cases the source code of HyperVDP supplies. After the setup and population is complete the match-action tables will be filled with the commands of the test network function cases, thus being ready to be probed and tested for potential vulnerabilities.

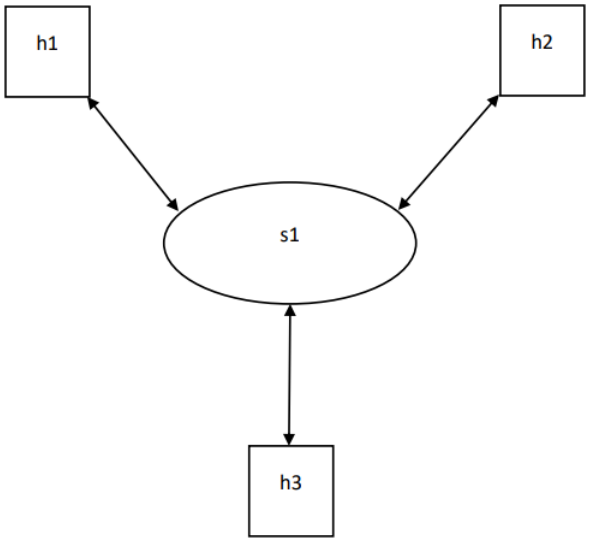

Figure 3: Simple network topology used in this research. Central switch with three bidirectionally connected hosts.

As mentioned in the previous section, using scapy creates the opportunity to generate packets that target specific parts of the virtualized PDPs. However, due to the time constraints of this paper and the tedious work that preceded the creation of the stable framework, there was insufficient time to demonstrate a significant vulnerability.

5 Responsible Research

In this research the proper amount of time has been taken to cite papers, projects and code of others to give credits to work where it is due. We acknowledge that this research could be used as a stepping stone by people with malicious intent to get a better understanding of how to corrupt virtualized P4 programs.

It is, however, stressed that this paper is more valuable when used to bring awareness to how P4 programs that are virtualized can possibly be corrupted. This kind of research is necessary to make advancements in any security related field. When this awareness increases, the solutions to such problems also grow. It is therefore necessary that this research is performed and the results shared.

To aid in the reproducibility of the methods used in this paper there has been a significant amount of time poured into writing a clear and concise explanation in methodology and experimental setup sections.

6 Discussion

Previous works concerning P4 security focused primarily on the security threat from the outside of the network [10, 11]. These papers mainly looked at how P4 programs can be created and used to prevent security threats. To the best of our knowledge no other paper has investigated and discussed the risks that occur when a malicious program is in the virtualized PDPs.

This study has limitations that should be reflected upon. The progress was more difficult than was anticipated. The consequence is that not everything has been accomplished that we set out to do at the beginning of this paper. However, significant strides to create a test environment have been taken. These strides have been taken by updating the HyperVDP source code to be usable together with fixing the test cases the source code provided.

This is where the value of this paper can be found, due to the fact that this malicious program problem has not yet been investigated. The framework that has been built during this research is a good base to seek potential vulnerabilities from.

7 Conclusions and Future Work

In this paper we laid out a method to search for possible vulnerabilities in the shared data of virtualized P4 programs. This was done by creating a stable framework for HyperVDP to use to simulate a test network with mininet. Scapy was then used to send and receive packets that can be specifically created to target the test network functions that the source code of HyperVDP provides. Due to the time constraints no viable vulnerability has been found.

An opportunity for the future would be to use this paper as a stepping stone and build upon the created framework to continue the investigation into the security of HyperVDP. The created progress is best used in finding a concrete answer to the question, if a malicious program can corrupt the virtualized P4 programs by HyperVDP that use the same data.

Two interesting possible viable test case scenarios came forward during the progression of this research. The first scenario consists of a P4 firewall program virtualized together with the malicious program. It could be possible for this malicious program to parse the packets in such a way that they will be accepted by the firewall.

Another possible scenario that came forward is when the malicious program alters the data of the packet in such a way that the other programs can not use the package. This could have far reaching consequences for the network and is therefore recommended to pursue.

HyperVDP provides a useful feature by allowing the virtualization of multiple PDP on the physical PDP. Granting the functionality of re-configuring and deleting virtualized P4 programs in runtime, without interfering with other virtualized data-planes. Yet it has no built-in security features and lacks research focused on a malicious P4 program in a virtualized PDP. Although HyperVDP is a tricky subject, it is highly recommended that continuation of this research is performed in the future.

References

[1] Philippe Biondi and the Scapy community. Scapy: Packet crafting for python2 and python3, 2022.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[3] Wolfgang Braun and Michael Menth. Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet*, 6(2):302–336, 2014.

[4] Mininet Project Contributors. Mininet: An instant virtual network on your laptop (or other pc), 2022.

[5] Sol Han, Seokwon Jang, Hongrok Choi, Hochan Lee, and Sangheon Pack. Virtualization in programmable data plane: A survey and open challenges. *IEEE Open Journal of the Communications Society*, 1:527–534, 2020.

[6] David Hancock and Jacobus van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 35–49, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Keith Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, 2013.

[8] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[9] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Comput. Surv.*, 54(4), may 2021.

[10] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi. P4 edge node enabling stateful traffic engineering and cyber security. *J. Opt. Commun. Netw.*, 11(1):A84–A95, Jan 2019.

[11] F. Paolucci, F. Cugini, and P. Castoldi. P4-based multi-layer traffic engineering encompassing cyber security. In *2018 Optical Fiber Communications Conference and Exposition (OFC)*, pages 1–3, 2018.

[12] Rohith Raj S, Rohith R, Minal Moharir, and Shobha G. Scapy- a powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5, 2018.

[13] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. Hypervdp: High-performance virtualization of the programmable data plane. *IEEE Journal on Selected Areas in Communications*, 37(3):556–569, 2019.