Tibor Schneider

Automatic Generation of Adversarial Workload for Programmable Switches

Semester Thesis SA-2019-15
March 2019 to June 2019

Tutor: Roland Meier
Tutor: Edgar Costa Molero
Supervisor: Prof. Dr. Laurent Vanbever
Abstract

The appearance of domain-specific languages such as P4 has enabled stateful applications directly in the data plane. However, the increased programmability of network devices gives rise to a new range of potential vulnerabilities. Adversarial input can influence the control flow, e.g., routing decisions, of benign flows. Existing verification approaches do not consider multiple packets at the same time, and are therefore insufficient to capture more complex vulnerabilities.

In this thesis, we present an approach to automatically verify security properties of stateful data plane programs and generate a sequence of adversarial packets impairing the behavior of the application. We leverage symbolic execution to model the entire programmable network device. We demonstrate the difficulty of designing robust data plane programs and how our tool can help programmers to achieve this goal. We evaluate our approach in several case studies and uncover critical vulnerabilities in various P4 applications proposed in the literature.
# List of Figures

3.1 System architecture .................................................. 12
3.2 Simplified Flowchart for Pipeline .................................... 13

4.1 Different states for problem with 3 attacker packets. The arrows represent the packets processed by the application changing the state of the switch .......... 16
4.2 Example for transforming a program into an SMT formula .................. 16
4.3 Example for translating a P4 Table into a Z3 formula ...................... 17
4.4 Graph with 1 resubmit and 1 recirculate, where the shaded nodes represent the blocks defined by the P4 program ........................................ 20

5.1 Pipeline stages for Count-Min Sketch .................................. 24
5.2 Time to solve problem with a given number of attacker packets ............. 25
5.3 Effect of parallel computation on different P4 Programs ..................... 25
5.4 Time to generate the Z3 formula for random applications ................... 26
5.5 Simulated attack on Blink .................................................. 29
Chapter 1

Introduction

In the last few years, there has been a paradigm shift in networking devices from only configurable ASIC network processors to both fully programmable control and data planes. A domain-specific programming language called P4 [6] has become the prevalent way of programming the data plane. This flexibility enables network operators to quickly deploy new protocols or to implement specific services directly in the data plane. However, it opens a range of new potential vulnerabilities to exploit the program running on the switch [29]. The network operator can usually not control all traffic in the network, especially for network devices connecting the internet, where the traffic must be assumed adversarial.

As an example, if a device needs to maintain state information on a per-flow basis, it is easy for an attacker to create many flows and thus exhaust the device’s memory. On the other hand, a program which adds padding to packets or generates clones can potentially be used for traffic amplification in a denial-of-service attack. Furthermore, an adversary may be able to send a particular sequence to influence the decisions for regular packets. It is challenging to develop applications for the data plane which are resilient to such attacks giving rise to the need for sophisticated tools that highlight potential vulnerabilities.

The literature on network security shows a variety of approaches for verifying security properties of control plane applications [19, 20, 30]. However, to the best of our knowledge, very few publications can be found that discuss methods for verifying security properties of data plane programs. Several publications have appeared in recent years showing promising methods for finding bugs in P4 programs [14, 25, 31] and verifying their behavior [12, 22]. Nevertheless, most of these papers do not consider multiple packets and a continually changing switch state. In contrast to this work, we do not focus on finding bugs but on finding vulnerable parts of the code that can be misused for malicious purposes.

The goal of this semester thesis is to develop a system which finds packet sequences that would trigger undesired behavior of a programmable network device. The system should be able to find vulnerabilities in a wide range of P4 programs by considering the data plane implementation, requiring little additional information about the semantics of the application. The programmer can then use this information to improve the program. An additional goal is to show the difficulty for developing stateful and tamper-resilient applications for programmable switches.

The remainder of this report is organized as follows: Section 2 summarizes important concepts of P4 and symbolic execution. In Section 3, we introduce the general approach presented in this semester thesis and explore the main design decisions. We then present the implementation in Section 4. Section 5 discusses case studies and a time complexity analysis for some sample P4 programs. Finally, we summarize the results in Section 6 and conclude in Section 7.
Chapter 2

Background

This section briefly reviews the main properties of the P4 language, as well as SMT solvers to set the stage for design and implementation. It also gives a brief overview of the reference compiler p4c and the files it generates.

2.1 The P4 Language

P4 [6] is a domain-specific language for programming network devices. It is designed around packet-processing abstractions such as headers, parsers, tables, actions, and controls. The abstract forwarding model described in [6] consists of five distinct phases: parsing, ingress processing, replication and queuing, egress processing, and deparsing. A P4 program defines the behavior of these phases, while the switch architecture (e.g., PSA [33]) describes how those phases are connected. The reference compiler p4c for the P4 programming language generates files for configuring the target as described by the program. As an example, the compiler generates a JSON file for the V1Model, a target allowing programmers to run their applications as a software switch.

Every incoming packet is first processed by the parser. The P4 program can define a set of headers, as well as a parser state machine describing how the headers should be extracted. To extract multiple elements of a header stack, P4 allows loops in the state machine. The parser is represented in the JSON file by a directional multi-graph.

Both the ingress- and egress-processing can be entirely programmed. The programmer can sequentially write the program, using familiar constructs like assignments and if-conditions, as well as match+action tables. However, some constructs (loops) and operations (floating point numbers, division, modulo) are not supported due to hardware limitations. Actions are code fragments consisting of multiple primitives and are called by the match+action tables. Actions may contain data values (parameters) which can be written by the control plane. The compiler generates a (possibly parallel) control flow of multiple connected match+action stages and conditionals, represented as a loop-free directional multi-graph, one for the ingress- and one for the egress-pipeline.

Different switch targets have different capabilities and limitations. Therefore, P4 only specifies the fundamental properties of a target. A target provides a set of extern objects and functions. As an example, the V1Model, which allows the programmer to run P4 programs as a software switch, contains, among others, the following externs:

- Register arrays are stateful objects, which can be used to store persistent data.
- Hash calculations can be used to compute a hash of some data. The V1Model supports CRC checksums (both 16 and 32 bit) and ones’ complement sum (the checksum used, e.g., for the TCP header). Additionally, the V1Model allows the user to specify a base and a maximal value for scaling the hash output to the desired range. This can be very useful for hash maps. In this document, a hash calculation refers to a specific hash primitive in the program.

In this document, a node refers to either a match+action table, a conditional, or a parser state, and a branch refers to a specific transition from one node to another in either the parser, ingress-
or egress-pipeline. A branch condition is a condition required to be true in order for the packet to take the branch (can be either the if-condition or a decision in a match+action table).

### 2.2 SMT and Symbolic Execution

The basis for Satisfiability Modulo Theories (SMT) is a Boolean Satisfiability Problem (SAT), which is the problem of determining whether there exists an interpretation of symbols which satisfies a Boolean formula. An SMT solver combines an SAT problem with theories (like arithmetic, bit-vectors, and undefined functions). As an example, assume the following formula (where \( x \) and \( y \) are both integers):

\[
x > 0 \land y = x + 1 \land y < 3
\]

An SMT solver can check this formula for satisfiability, which will return \textit{sat} for this small example. SMT solvers can also generate interpretations (also called model) of all symbols (in this case \( x = 1 \) and \( y = 2 \)) which will satisfy the formula. Moreover, SMT solvers can prove that a formula is always satisfied or can generate counterexamples in which the formula is not satisfied. In addition to Boolean expressions, the formula may contain quantifiers, the ones most used being “for all” and “there exists”.

In symbolic execution, a program is translated into an SMT formula. Instead of assuming specific values for the inputs of the program, they are treated as uninterpreted symbols. This can be used for achieving many different goals, of which a few are listed below:

- Reversing the program to find inputs which produce a given output. This can be achieved by constraining the outputs of the program.
- Verify that certain properties hold for all possible inputs.

Despite modern tools for solving SMT formulas, symbolically executing all feasible program paths does not scale to large programs, since this number usually grows exponentially with the number of branches in the code. This effect is known as path explosion.

This project uses the state-of-the-art SMT solver Z3 [10] to check a logical formula for satisfiability. The P4 Program is translated into an SMT formula, including constraints specifying the attack (see Section 3.2). This formula is then checked for satisfiability. Z3 will return \textit{sat} only if an attack is possible. In this case, the input is evaluated on the model (interpretation) returned by Z3 to obtain the adversarial packets.

### 2.3 Related Work

Software Verification has been an active topic in research for the past five decades, starting with publications from Hoare [15] and Dijkstra [11]. More recently, powerful SMT solvers like Z3 [10] have enabled automatic verification of functional correctness, with highly efficient tools such as Boogie [2] and Dafny [21]. The idea of network verification was first explored by Xie et al. [35] to statically analyze device configurations in order to check reachability properties. This approach was later applied to software-defined networking with tools like Header Space Analysis [17], VeriFlow [18] and NetKAT [1], which can be used for finding bugs in OpenFlow [24] networks. The emergence of P4, however, requires new tools for network verification. In [23], the authors present a system to automatically verify reachability and well-formedness in P4 networks. ASSERT-P4 [14] verifies general security and correctness properties of annotated P4 programs. Vera [31] uses symbolic execution to automatically uncover common bugs, including parser/parser errors, invalid memory accesses, and loops and tunneling errors. P4v [22] incorporates a symbolic control-plane interface. Other verification techniques are discussed in the literature, such as p4pktgen [25], a tool for automatically generating test cases for P4 programs, or netdiff [12], which finds bugs by checking for data plane equivalence. However, those tools are not sufficient to verify various security properties because they only consider a single packet at a time.
Chapter 3

Design

In this chapter, we give an overview of our implementation and elaborate on the main design choices. This chapter also introduces the attacker model, highlights the importance of finding hash collisions, and explains our approach to solve this challenge.

3.1 Attacker Model

Network devices connecting the Internet are exposed to a large amount of traffic, which can generally be assumed adversarial. The goal of an attacker ranges from simple denial of service attacks to sophisticated manipulations of the program flow, resulting in impaired functionality of the program. In this thesis, we assume that the attacker knows both the data plane and the control plane application running on the switch, as well as all table entries and register contents. Furthermore, he knows the entire traffic processed by the network device. The attacker can send several arbitrary packets to any external interface of the switch.

3.2 Attacking a Single Branch

In this section, we will discuss our approach for generating adversarial load impairing the program running on the switch. Finding a general definition for impaired functionality, without the user precisely specifying a set of constraints, is very difficult. Usually, it requires an understanding of the application, of the topology, and the goal of the network operator.

Our approach tries to find a series of attacker packets which manipulate the path of a victim packet through the control flow of the application. In other words, the adversary tries to change some state in the switch to force the victim packet to “reach” a specific point in the program. Some parts of the control flow are critical for the functionality of the switch, eventually affecting many different packets. We do not try to distinguish the effect of certain branches but ask the programmer to name a specific point in the application which should be attacked. Note that the conditions to reach this point must depend on at least one stateful object.

An attack consists of the victim packet, multiple attacker packets, and an initial switch state, containing table entries and register contents. Assume the victim packet processed by the switch in the initial state will take a path through the program flow, reaching the specified branch in the application. The adversary sends a sequence of attacker packets, modifying the state of the switch before the victim packet is processed. We consider this branch to be vulnerable if it is only reached by the victim packet when the switch is in the initial state, but not when the state was already modified by the attacker.

Our strategy for finding attacks relies on SMT solvers, namely Z3, which require an exact logical expression. This expression should only be satisfied if there exists a vulnerability. The properties of this expression will shape the attacker model. As an example, assume that the problem generated does not specify the header data of the victim (the packet whose path through the control flow should be modified). As a consequence, the SMT solver searches for a specific victim for which the attack is possible. We allow the user to constrain certain input fields, table entries, and other stateful objects such that the solver returns a reasonable attack.
3.3 System Architecture

The overall architecture is depicted in Figure 3.1. First, the P4 program must be compiled using the reference compiler p4c to obtain the JSON file. This file contains all information about the application needed to be synthesized and executed on a programmable network device (cf. Section 2.1). The analyzer takes this JSON file as an input to generate adversarial load. In addition to the packets necessary for the vulnerability, the analyzer also generates table entries and the register content, which can be used to simulate the attack.

The analyzer only considers the p4 application in the JSON format without any control plane application. However, a configuration file can be prepared to set the initial value for registers, table entries, CRC parameters, and mirroring sessions (used for cloning) on the switch (cf. Section 3.6).

3.4 Hash Functions

Usually, network devices need to maintain state for a vast number of flows. However, monitoring all flows is in most cases impossible due to memory limitations. A typical approach is to use probabilistic data structures like bloom filters [4, 13] and sketches [8], which are based on hash table lookups. Because of this, hash functions usually need to be considered in order to attack a data plane program.

For achieving line-rate-processing (of up to 12.8 TB/s for the Barefoot Tofino 2), the data plane can only perform simple checksum computation. The PSA [33] supports CRC checksums, both the 16-bit and the 32-bit variant, and ones’ complement summation used for IPv4, TCP, and UDP headers. Fortunately, those checksums are relatively easy to brute force or to reverse, compared with cryptographic hash functions. The input bit-width of all hash functions used in a P4 program is always known at compile time, implying that the CRC hash function can be expressed as an SMT formula. However, adding hash functions to an already complex problem will increase the complexity significantly.

Our approach is to split the problem into multiple parts. The first one is to find potential attacks by ignoring the hash collisions. The second part finds a hash collision by using the information from the first part to infer the required path of the victim and all attacker packets through the control flow of the program. This modification speeds up the solver significantly, see Section 5.1.1 for measurements.

The problem can be simplified further by precomputing several hash collisions and constraining the problem to use them. This requires specific hash calculations of multiple packets to be matched. However, this is not trivial, since the hash does not necessarily need to be computed at the same location in the program code (cf. Section 4.5). Section 3.5 introduces the pipeline which combines the ideas above.

3.5 Pipeline

This section describes the pipeline for finding a vulnerability for a single branch (transition) in the original P4 program. A simplified flowchart of the pipeline is depicted in Figure 3.2.

The pipeline mainly consists of 4 stages, which are repeated for an increasing number of attacker packets. The time to check a problem generated by the pipeline usually scales (worse than) exponentially with packets considered in one formula. Our implementation first tries to find a vulnerability with exactly one adversarial packet and increasing the number by 1 if there is no interpretation which satisfies the formula.
As described in Section 3.4, finding hash collisions can be difficult, especially for a large number of them. The pipeline splits the problem of finding an attack and finding the hash collisions into four stages described below.

**Stage 1** In the first stage, the P4 program is translated into an SMT formula by treating all calculations specified in the application as uninterpreted (symbolic) functions. This allows the solver to create arbitrary hash collisions without actually computing them. There is no need to continue to stage 2 if this first problem is not satisfied because the real hash function is one possible interpretation of the symbolic one. On the other hand, if this problem is satisfied, we have a strong indication for an attack. Hash collisions still need to be found in order to generate the adversarial packets.
Stage 2  This stage again translates the P4 application to a formula, but this time with many additional constraints. Unlike stage 1, all CRC hash functions added to the formula as Z3 macros. The path of all packets through the control flow is extracted from the interpretation of the problem already solved in stage 1. For the new SMT formula, all packets are constrained to take this precomputed path. Additionally, the strategy for finding hash collisions is applied (more on this in Section 3.4), which will constrain some inputs of the attacker packets. Those additional constraints have shown to improve the solve time by a very significant margin (cf. Section 5.1.1). Essentially, the problem in this stage only checks if the guess for the hash collision was correct or not.

Stage 3  The constraints from stage 2 might be too strong for an application, and the second stage can generate an unsatisfied problem, even if stage 1 returned sat. This third stage uses the same formula as the previous one, without constraining the header data of attacker packets. Therefore, this stage essentially computes hash collisions, but it does not experience path explosion. As for stage 2, the third problem might not be satisfied if the result of the hash function is used to decide on the path through the control flow.

Stage 4  This last stage of the pipeline generates a formula with the hash functions represented by defined macros, but without any optimizations. Thus, this problem takes a long time to solve (see measurement results in Section 5.1.1). Fortunately, this stage was not necessary for any of the P4 programs analyzed in Section 5. A program which satisfies the first stage, but not the last one has a vulnerability which is prohibited by the specific hash function used.

3.6 Control-Plane Interface

A P4 program as such does not fully determine the semantics of the data plane. Verification requires additional knowledge about the control plane. Several different approaches to solve this problem are discussed in recent publications [14, 22, 31]. Similar to ASSERT-P4 [31], our implementation reads table entries, as well as other runtime configuration like hash parameters and mirroring sessions\(^1\) from a configuration file.

For some applications, however, this might not be sufficient. First, some P4 programs, like Blink [16], use registers as a way for the control plane to configure some behavior of the data plane. Second, a network device is usually attacked during operation, and hence, registers and counters are already populated. Our implementation allows the user to specify the register entries for the initial state of the switch.

The initial state of the switch, including table entries and register contents, can be partially specified in the configuration file. Undefined tables and registers are treated symbolically (as uninterpreted functions or arrays) by the SMT solver. Consequently, everything not specified in the configuration file is treated as part of the attacker model. In some cases, this effect is desirable. In others, however, our implementation returns an attack, which is not possible during “regular” operation. The case studies in Section 5.2 shows how the configuration can be prepared to investigate a P4 program.

However, a control plane is usually not static. Instead, it might change table entries or registers to influence the behavior of the switch. Various switch targets, including the PSA, allow the data plane to forward information to the control plane using packet digest [33]. The fact that the control plane is orders of magnitudes slower than the data plane introduces an inevitable delay for any actions, which the controller might perform upon receiving a packet digest. Hence, modeling the control plane for the symbolic execution leads to a whole range of new challenges.

For this thesis, we assume the controller to be static; the state of the switch can only be changed directly within the data plane.

\(^1\)Mirroring sessions are used in the V1Model to specify the egress port for the cloned packet.
Chapter 4

Implementation

This chapter provides detailed information on how the application is translated into an SMT formula and how the results are interpreted and simulated.

4.1 P4 Architecture Model

Certain aspects of a programmable network device (e.g., packet buffer or replication engine) are not specified by the P4 language but must be considered for modeling the behavior of the switch. The V1Model is an architecture provided by the reference compiler p4c, allowing programmers to implement their P4 program as a software switch. Our implementation approximates the behavior of this architecture with some key differences:

- Packets are processed one after another, without any queue or buffer.
- The deparser and checksum computation after the egress pipeline are ignored. Those parts of the application do not contribute to the type of vulnerability described in Section 3.2.

Algorithm 1: After-ingress

\[
\text{if } \text{clone was called} \text{ then} \\
\quad \text{make a clone of the packet} \\
\text{if } \text{resubmit was called} \text{ then} \\
\qquad \text{start ingress processing over again for the original packet} \\
\text{else if } \text{egress spec } = 511 \text{ then} \\
\qquad \text{drop packet} \\
\text{else} \\
\qquad \text{continue egress processing}
\]

Algorithm 2: After-egress

\[
\text{if } \text{clone was called} \text{ then} \\
\quad \text{make a clone of the packet} \\
\text{if } \text{recirculate was called and egress spec } \neq 511 \text{ then} \\
\qquad \text{start ingress processing over again for deparsed packet} \\
\text{application end}
\]

The JSON application generated by the compiler contains three disjoint graphs, the parser, the ingress and egress pipeline. Algorithm 1 and Algorithm 2 specify how those graphs are connected. Resubmit and recirculate are described in Section 4.3.7. Note that both algorithms do not show the standard metadata fields which are set when transitioning from one pipeline to another.

4.2 Multiple Packets

As described in Section 3.2, an attack consists of multiple packets with a set of constraints. For each packet, the entire P4 Program is translated separately. In the following, the packet whose path through the program should be modified is called “victim”, and those packets used to achieve this goal are called “attackers”. Figure 4.1 illustrates the switch state changing as the sequence of packets is processed.

The victim is added twice to the formula. First, it is acting on the initial state, constrained to reach the target branch. However, the second victim, processed by the switch in the resulting state after the attacker sequence, is not allowed to take this branch. The input header (also the timestamp and the ingress port) must be equal for both packets.
4.3 Translate P4 to SMT

This section explains how our implementation translates a given P4 program into an SMT formula. As already described in Section 3.3, the JSON output from p4c is the input to the translator. This JSON file is used to synthesize and load the program on the target, or to simulate it. Both the ingress and egress pipelines are represented as a directed multigraph without loops, where the nodes represent the different tables (i.e., stages of the pipeline), and the edges represent the actions possible at a given node. The parser is also represented as a directed multigraph, but it is allowed to contain loops.

Before translating the program, the graph is modified to comply with the control flow described in Section 4.1. This alteration also ensures that the graph is connected. Our implementation uses this underlying graph to translate the program recursively into a list of assertions for the Z3 solver.

The final SMT formula consists of multiple packets. The packets need to have a different symbolic representation for all intermediate symbols to ensure that they are treated independently. Our implementation initially generates, per input packet, an input symbol for all header field declared in the P4 program.

4.3.1 Key Concepts for Symbolic Execution

An SMT formula for verifying a program is fundamentally different from the program itself. This section focuses on the basic approach for translating a program into an SMT formula for symbolic execution, without considering P4 specific properties.

**Assignments** In an SMT formula, there exists no notion of time or direction, and thus, it requires a bit of effort for encoding assignments. The basic approach is to generate a new symbol for every assignment. This behavior can be seen in the code example in Figure 4.2. Consider the first line of the original program in Figure 4.2a, which corresponds to the first assertion in Figure 4.2b, introducing a fresh symbol \(x!1\). Later references to \(x\) are using the fresh symbol \(x!1\) instead of the original \(x!0\) (see line 5 and 6 in Figure 4.2b).

**Branches** For symbolic execution, every path through the control flow of the program is considered at the same time. Z3 only offers a ternary if-operation, which is sufficient to model all
4.3 Translate P4 to SMT

branches (and the recombination) symbolically. The same idea from before can be used to recombine two branches. For every field changed in at least one of the two paths, a new symbol is created. The symbol is then asserted to be equal to a ternary if-operation, as shown on line 5 and 6 of Figure 4.2b. As a consequence, the translator needs to keep track of all recently used symbols in every branch to correctly recombine two branches.

Loops An SMT formula cannot contain any loops. They need to be either unrolled or replaced by an equivalent function. Fortunately, P4 only allows loops in the parser, and by recirculating packets, see Section 4.3.2 and Section 4.3.7.

4.3.2 Parser

The parser recognizes and extracts data from the packet headers. In P4, the parser is specified by a state machine, where loops are allowed. The parser can interact with the data directly (e.g., assigning a new value to a field).

The compiled JSON program contains a validity field for every header, which is assigned to 0 at the beginning of each parser processing. For every extract operation encountered in the parser, the respective validity field is set to 1. This behavior is reflected in our implementation.

The P4 language also allows the programmer to use header stacks. Every element of such a stack is treated as a regular header, including its own validity field. Section 4.3.4 explains the push and pop primitives.

As already explained in Section 4.3.1, loops need to be unrolled for generating the SMT formula. The directed graph of the parser has no limitations regarding cycles. However, cycles in the parser are used to extract header stacks, with a length already known at compile time. The parser will fail as soon as more elements should be extracted than specified in the declaration of the header stack. Our implementation generates a tree from the parser graph by unrolling every cycle as long as the parser would not fail.

4.3.3 Tables

The abstract forwarding model of P4 (described in [6]) consists of an ingress and an egress pipeline, which in terms are multiple match+action tables arranged in series, parallel, or a combination of both.

(a) Underlying directional multigraph

(b) P4 program

(c) Generated assertions

Figure 4.3: Example for translating a P4 Table into a Z3 formula
Every table in the underlying graph has an outgoing edge for every action which can be invoked by the table. If the P4 program uses the hit property of a table\(^1\), then our implementation adds an additional transition. The translation generates several different uninterpreted functions for every table. All table functions have the same input, i.e., the key specified in the table definition. Note that the match_kind does not need to be considered for uninterpreted functions. If the table is immutable, and the entries are already known during the analysis, then the uninterpreted functions are replaced by Z3 macros.

- The Decision determines the action identification for a given key. In the example depicted in Figure 4.3c, this function tbl\_decision can be found in line 5 and 7.
- An Action-Data function is created for every parameter of all actions which can be invoked by the table. See the first line of Figure 4.3c, where the function is called tbl\_f\_p.
- Table-Hit distinguishes between a table hit and a miss. This function is constrained such that it can only evaluate to false if the decision function for the same key returns the default action. It only occurs in the formula if the P4 program uses the hit property of the table.

The Example depicted in Figure 4.3 shows the assertions generated by our implementation. The same basic ideas from Section 4.3.1 are used. Figure 4.3a shows the underlying graph, generated by the reference compiler.

### 4.3.4 Primitives and Externs

P4 describes several different primitives allowed in an action. The compiler replaces some structures of the P4 source code with primitives (e.g., assignments in the form \(x = y\); are replaced by assign primitives). The architecture also provides a set of externs giving the programmer possibilities to interact with the switch itself. Our implementation interprets those primitives and translates them into a symbolical expression. This translation is trivial for most of them. Some exceptions are listed below:

- resubmit and recirculate are treated as an assignment of the standard metadata fields resubmit\_flag or recirculate\_flag, respectively. Similarly, clone is treated as an assignment of the clone\_spec field.
- modify\_field\_with\_hash\_based\_offset is the primitive created by the compiler when calling the extern clone(dst, base, calc, max) which is treated as:

\[
dst \leftarrow base + (calc \mod max)
\]

Our implementation treats those primitives as assignments shown above, where calc is replaced by either the uninterpreted function or the CRC macro, as explained in Section 4.3.6.

- modify\_field\_rng\_uniform is an extern provided by the V1Model to generate a random number, uniformly distributed in the given interval. Z3 (and SMT solvers in general) cannot encode “randomness”. Our implementation translates this primitive into an assignment of the destination field with a new uninterpreted symbol constrained to be in the interval. This ultimately gives the attacker full control over the random variables, because Z3 will give an interpretation satisfying the formula.

- push and pop are primitives for adding and removing elements from the header stack. This requires each field (including the valid bit) of each element to be assigned to the respective field of another element. Elements, which are added or removed in the process, must update their validity bit.

- add\_header and remove\_header only assign the “validity” header bit to zero or one.

\(^1\)This property can be used in the P4 program to check if a match+action table contains an entry which matches the packet.
4.3.5 Registers

The P4 language itself does not contain any stateful objects besides match+action tables. Instead, the architecture provides extern objects, which can be used to maintain a state between packets. The V1Model offers the usage of register arrays, counters, and meters. Our current implementation, however, ignores counters and meters. Registers in the Z3 formula are represented as Arrays. This is an SMT theory, characterized by the select-store axioms. The expression `select(A, i)` returns the value stored at position `i` of array `A`, and `store(A, i, v)` returns a new array identical to `A`, but on position `i` it contains the value `v`.

Register Arrays in the V1Model are specified by the number of elements and the bit-width of each element. In Z3 however, arrays are defined by a sort (Z3 symbol type) for the index and one for the value. There is no notion of the number of elements in the array\(^2\). Our implementation assumes that all register accesses in the application can only index the register in the valid range. Because of this assumption, the bounds of the indices are not verified.

4.3.6 Calculations

P4c creates named calculations for all checksum verification/computation, and usages of the hash primitive. Those contain the algorithm used and the data for which to generate a checksum. The V1Model offers, among other algorithms, both the 16- and 32-bit CRC checksum, where all parameters can be configured by the control plane.

In stage 1 of the pipeline, calculations are represented by uninterpreted functions, while for the other stages, the CRC hash function is added directly to the Z3 formula. Checksum calculations expressed as a Z3 formula can be used to reverse the expression or to find hash collisions. Algorithm 3 shows the bit-by-bit implementation used for CRC-32 (and also CRC-16 with some minor modifications).

---

**Algorithm 3: CRC-32 bit-by-bit algorithm**

**Data:** data, params

```plaintext
x ← params.xor_in

foreach byte in data do
  if params.reflect_in then
    byte ← reflect byte
  for i in {0, ..., 7} do
    top ← x & (1 << 31)
    if top & 0x80 >> i != 0 then
      top ← top xor (1 << 31)
    x ← x << 1
  if top != 0 then
    x ← x xor params.poly
  x ← x & (1 << 32)-1
if params.reflect_out then
  x ← reflect x
```

**Result:** `x xor params.xor_out`

4.3.7 Resubmit and Recirculate

The P4 language does not support loops in the ingress and egress match-action pipeline. The V1Model, however, allows packets to be sent back to the ingress parser after the ingress pipeline (called resubmit) and after the egress pipeline (called recirculate). This functionality can result in two nested loops. Our implementation unrolls these loops by duplicating parts of the underlying graph of the application. The user can specify the number of loops allowed to limit the complexity of the generated formula. Figure 4.4 shows how the graph is modified.

The PSA [33] specifies that a recirculated packet is first deparsed and then looped back to the ingress parser. In our implementation, this behavior is reflected by setting the validity bits of all headers back to 0 without touching the actual header fields. Resubmitted packets, however, are not deparsed, meaning that all changes in the ingress pipeline to the packet header are discarded.

\(^2\)It is possible to use a bit-vector of with \(w\) an as index which has the effect that the array has only \(2^w\) elements.
Figure 4.4: Graph with 1 resubmit and 1 recirculate, where the shaded nodes represent the blocks defined by the P4 program.

Since recirculate deparses the packet, a subsequent resubmit will reset the header data to the value from this last recirculate. This behavior is reflected in our implementation by assigning all header fields to the symbol used at the start of the last ingress parser.

4.3.8 Cloning

The PSA [33] describes the packet cloning mechanism, allowing the programmer to generate a copy of a packet and send it to a specific port. Similar to resubmit, a clone operation after the ingress pipeline will create a copy of the packet as it last entered the ingress pipeline.

Our implementation does not duplicate parts of the graph for cloning. Instead, once a clone operation is invoked, the cloned packet is symbolically prepared and then added to a queue. Once the translation of the original packet is finished, our implementation will then continue translating the egress processing for all clones in the queue one by one. Note that the resulting state depends on which clones were generated and processed.
4.4 Register Dependent Branches

As explained in Section 3.2, our implementation tries to find an attack which influences the path of a victim packet through the control flow of the data plane by adversarial packets sent before. This can only occur if the attacker changes some state on the switch. In our implementation, only registers can be modified in an attack, since (1) counters and meters are ignored, and (2) the controller is assumed to be static, resulting in immutable table entries. For a branch to be considered vulnerable by our definition from Section 3.2, the condition of the branch (or any prior branch) must somehow depend on a value read from a register array.

Our implementation builds a dependency tree from the application graph, to decide whether a branch condition depends (not necessarily directly) on a value read from a register and then only considers those as possible branches to be analyzed.

4.5 Proposing Hash Collisions

The second stage of the pipeline (cf. Section 3.5) proposes some precomputed hash collisions to the solver. The task to propose hash collisions can be split into two parts. The first matches a hash calculation of an attacker with a calculation of the victim packet, and the second part computes hash collisions and generates the required constraints for the solver.

Matching Hash Calculations An important challenge for proposing hash collisions is to find hash calculations which need to return the same result for an attacker packet and the victim. The naive approach is to assume that all distinct hash functions in a program must result in the same value. However, some attacks might only work when some hash results for the attacker and the victim are different. For other attacks, the hash output of different calculations need to be the same. (E.g., a firewall, which uses a bloom filter to store open connections, might switch the source and destination address and port for internal packets to open the connection for external replies.)

We have implemented two different strategies to match hash calculations, both of them using the model generated by the first stage in the pipeline:

- The aggressive strategy checks the output of every single hash calculation of the attacker and the victim. Two hash functions of an attacker and a victim are matched if the output (evaluated on the model) is equal. Additionally, both must have identical CRC parameters, as well as the same input bit-width.

- The deliberate strategy is similar to aggressive, except that the output needs to be used eventually as an index of the same register array for both the attacker and the victim.

Computing Hash Collisions The model of the first stage of the pipeline can be used to evaluate the inputs for the hash of the victim, which reveals the true output of the calculation. However, finding relevant hash collisions for two matched hash calculations of an attacker packet and the victim packet is not trivial:

- An attacker may have multiple hash calculations matched. Hence all matched calculations must be considered at the same time.

- The path of the attacker packet through the program flow may depend on a field, which is used for the hash calculation. This is usually the case for hashing the 5-tuple (including the protocol) identifying the flow because the parser requires the protocol field to decide on the next header.

Because of time constraints, we implemented a very simplistic method, which does neither compute hash collisions nor check if the control flow is dependent on the input fields to the calculation. Instead, our implementation simply tries to solve the problem where the inputs to the hash function are the same for an attacker and the victim.

\[\text{3}\] The compiler generates a new identical instance for every action called, which might result in some actions to be present multiple times in the compiled JSON file. An action called multiple times, which computes a hash, will therefore also result in multiple identical calculations, with a different name.
4.6 Interpreting the Model

Z3 can generate an interpretation, also called model, for all symbols used in a formula after it was checked for satisfiability. This interpretation includes all symbols of fields and registers in the formula, as well as all uninterpreted functions. This section describes how this interpretation is used to extract all packets and the initial state of the switch.

Packet Path Extracting the path of each packet through the control flow can be achieved by evaluating all path conditions for all branches. While translating a program, our implementation remembers all Z3 expression used in path conditions. After solving the problem, the underlying graph is traversed by evaluating every decision along the path.

Packet Headers The header order is extracted using the path of the packet by traversing the first ingress parser and adding all headers in the order in which they are extracted. The packet headers are then generated by evaluating and concatenating all fields of the required headers.

Initial State A Z3 array evaluated on the model returns a lambda expression with nested ternary if statements\(^4\). This expression can easily be transformed into a data structure storing the default value and all entries which deviate from it. Applying this procedure for all registers of the first attacker packet reveals the initial state required for the attack. Repeating this for all other packets shows how the states change.

Table Entries Extracting all table entries is not straightforward since the information must be collected from several different functions. Additionally, not all functions for the action data are used for individual keys, which result in the Z3 solver assigning unreasonable values to those inputs. Our implementation stores all symbolic table match operations during translation. After solving the problem, the entries are extracted by only evaluating the inputs and outputs of the functions stored during translation. Note, that those entries will suffice for simulating the attack.

\(^4\)In some cases, the instance returned by the model is a constant array surrounded by several store operations. This occurs when the initial array was specified as a constant array. However, this can be translated directly to a lambda expression.
Chapter 5

Results

In this chapter, we first try to examine the time complexity to solve a formula generated by our implementation. In Section 5.2, we analyze some real-world applications to show how our implementation can be used with real applications and which vulnerabilities can be found.

5.1 Time Complexity

Generally speaking, solving an SMT problem is NP-complete. There exist numerous different factors contributing to the problem complexity and the time for solving the problem. Most of those factors are very hard to quantify. Adding more expressions to the formula does not necessarily mean that it takes more time. This section gives intuition about the main factors influencing the time complexity of the analyzer. All the following experiments are conducted on a virtual machine with six cores ( clocked at 2.3 GHz) and 16 GB of memory.

For many measurements, we have used a heavy hitter detection in multiple different forms to be analyzed. They all have the following in common: A probabilistic data structure is used to count the approximate number of packets received by every flow. This value is then compared with a threshold \( n \), dropping all packets for which the value is larger than this threshold. The attack in all of them is trivial: increase the value in the respective registers to make the switch drop the victim packet. By adjusting the threshold, we have precise control over the number of adversarial packets necessary for this attack.

5.1.1 Hash Collisions

The pipeline, as described in Section 3.5 solves a problem multiple times with various different optimizations applied. Of course, the time for those stages heavily depends on the application. We have chosen a heavy hitter detection which uses a Count-Min Sketch [9] to approximate the number of packets of any flow already received. This data structure can be scaled to contain more hash functions.

The application consists of \( m \) CRC-32 hash functions, all using different parameters but the same input (the 5-tuple identifying the flow). Each of those hash functions is then used to index \( m \) different hash maps of length \( w \). Every incoming packet increments the respective value of all \( m \) register arrays. The number of packets already processed by the switch is approximated by the minimum of all \( m \) values. To attack the application with exactly \( n \) attacker packets, the 5-tuple for every attacker packet must result in the same index for all \( m \) register arrays as the victim packet.

Figure 5.1 shows the time to solve each stage in the pipeline for a given number of attacker packets (always equal to the threshold \( n \)) with 1 or 2 hash functions. Note that the time is shown in logarithmic scale, and both plots share the same y-axis. The graph clearly shows the effect of all optimizations applied for stages 2 and 3. Unsurprisingly, stage 2 drawn in orange scales significantly better than all other stages, because the path and the inputs are (partially) defined. On the other hand, stage 4 shown in red quickly becomes infeasible to solve, especially when complex hash collisions are necessary. The fourth stage for two hash functions was only measured up to four attacker packets. Increasing the number of hash maps also has an effect on the third stage. The plot also illustrates, that the time for the first stage (in blue) does
not increase significantly for $m = 2$ compared with $m = 1$, which is expected because all hash functions are treated as uninterpreted functions in this stage. From this result, we can conclude that the optimizations, reflected in the pipeline, have a positive impact on the execution time of the SMT solver. In fact, since stage 4 of the pipeline is not necessary for the Count-Min Sketch, our implementation can easily find solutions with ten attacker packets, while stage 4 with four packets did not finish after 24 hours of computing. This result also shows, that proposing good hash collisions for the solver in stage 2 is crucial for large applications. The problem of stage 4 is infeasible to solve when many hash functions are present. Fortunately, during evaluation, we encountered no applications where the fourth stage was required.

5.1.2 Sat vs Unsat

As described in Section 3.5, the analyzer solves the problem with an increasing number of attacker packets until the first attack is found. Finding vulnerabilities with this approach requires solving multiple formulas which are not satisfied. To evaluate the difference in time to solve satisfied (sat) and unsatisfied (unsat) problems, we developed a heavy-hitter detection which uses a simple hash map to store the data. The threshold allows control over the number of attacker packets, for which the problem returns sat. This application was analyzed with a very high threshold, thus always resulting in unsat. It was also analyzed with a threshold matching the number of packets (such that exactly this number is required to attack the program). Figure 5.2 shows the result, where the time is shown in a logarithmic scale. Note the following:

- The deviation is much larger than it appears in the logarithmic plot. As an example, the outlier of the sat problem (drawn in blue) for 9 attacker packets is 6 times larger than the median. The logarithmic scale also means that the upper ends of the whiskers are (in a linear scale) farther away from the mean than the lower end.

- There is a significant gap between the execution time for solvable and unsolvable problems.

- The plot also shows that the time scales at least exponentially with the number of attacker packets.
Figure 5.2: Time to solve problem with a given number of attacker packets.

Figure 5.3: Effect of parallel computation on different P4 Programs
5.1.3 Parallel Solver

In the early stages of development, we noticed a significant variance in time for the default Z3 sat-solver to find a solution to the given problems. Additionally, the default solver is running sequentially in one single thread. Our implementation leverages multiple cores on a system by solving the same problem multiple times in parallel, each thread with a different random seed. The effect of this modification was tested on two different applications:

- Blink [16] is a relatively large application, where most of the resources are spent on selecting active flows and on filtering data using a sliding window. Only two small hash functions are calculated (cf. Section 5.2.3). The results are shown in Figure 5.3a.

- A heavy-hitter detection using a Count-Min Sketch which relies heavily on hash functions (as described in Section 5.1.1). For this evaluation, stage 2 of the pipeline is skipped to highlight the effect for problems where complex hash collisions need to be found. In Figure 5.3b, the impact of the parallel solver can be seen very clearly.

Both figures have a different time axis and cannot be directly compared. The plot for Blink shows a low standard deviation compared to the Count-Min Sketch. Therefore, the expected improvement for Blink is approximately factor 2, while the improvement for the Count-Min Sketch is larger than 10. The large difference in standard deviation of both programs can be attributed to the fact that the Count-Min sketch contains more hash functions, for which the solver needs to find a collision.

5.1.4 Translation Time

To evaluate the time to translate a P4 program into an SMT formula, we have devised a method for randomly generating data plane programs. Such a program consists of \( n \) randomly generated and nested if conditions, and actions with random assignments. The time to generate the formula is shown in Figure 5.4. The time to translate an application does scale linearly with the number of nodes in the application. The impact of resubmit and recirculate, as shown by the graph, is as expected. (E.g., since recirculate duplicates the entire graph, we expect that it should take about twice as long as the same application without recirculate.) We can conclude that the time to generate the formula is substantially smaller than the time to solve the problem, especially for large programs.

![Figure 5.4: Time to generate the Z3 formula for random applications](image-url)
5.2 Case Studies

This section presents our findings for several applications. All attacks were simulated in a mininet topology, using precisely one switch, and sending the generated workload directly to the switch. See Appendix A for a guide on how to configure the application.

5.2.1 Flowlet Switching

Flowlet Switching is a simple load balancing scheme [34]. Flowlets are bursts of packets belonging to the same flow, sufficiently far apart in time for them to be routed via different paths, for minimizing packet reordering at the receiver. Flowlets automatically change size based on the extent of congestion on their path, decreasing the size on slow paths. This elasticity property allows Flowlets to compensate for poor load balancing decisions by shifting the traffic to less-congested paths automatically.

We have considered a simple implementation of Flowlet Switching based on ECMP routing [27]. The ECMP implementation works by hashing the 5-tuple identifying the flow to determine the next hop. Flowlet Switching introduces two register arrays (hash maps) to the application. The first stores the timestamp of the last packet of a flow, and the second is used as an offset for computing the hash for the ECMP next hop. On receiving a new packet, the switch computes the time difference since the last packet of the same flow. If this is larger than the threshold (100ms), then the switch updates the offset, resulting in a different next-hop for this and all following packets of the same flow. The apparent attack for this application is to send packets periodically, making sure that the timeout for a given flow is never reached. This attack will cause the program to behave like normal ECMP.

We have chosen the if-condition comparing the flowlet time gap to be attacked. (The body of this condition increments the offset register by one.) As a first step, we have not provided any control plane configuration and left everything, including the register values, to be chosen by the solver. Based on this configuration, our implementation finds two attacks with a single packet, one for forcing the victim to update the offset for ECMP, and the other for forcing the victim not to. This result has many different problems:

- The attack, forcing the victim to update the ECMP offset, is not possible. In this attack, the adversary tries to decrease the time difference for the victim packet, which means setting the timestamp to a lower value than it currently has. This requires the attacker to send a packet in the past and to somehow remove any other packets of the same flow as the victim. Therefore, we ignore this result.

- The timestamp for both the attacker and the victim packet, while being close to each other, are very far away from the value stored in the register (in fact, the difference is about 12 days).

- Even though the victim packet does no longer update the ECMP offset, the attacker packet does.

As a next step, we generate a control-plane configuration to get a realistic result. The tables of the application are filled with reasonable entries, with one prefix configured for ECMP with multiple next-hops. All registers are initialized to zero. The destination address of the victim is set to an address belonging to the prefix. We also require the timestamp of the victim to be at 1 second and the timestamp stored in the register to be 0. Finally, we set the option that no victim is allowed to reach this point in the program (i.e., no victim is allowed to update the ECMP offset).

This configuration results in an attack with a total of 9 attacker packets, all evenly spaced in time by precisely 0.1 seconds (equal to the threshold value). Therefore, the flowlet time gap is never larger than 100ms\(^1\). Our implementation found all 9 attacker packets in under 20 seconds, and most time is spent on generating the different formulas. A simulation shows the ECMP offset is indeed never changed, and all packets take the same route. By regularly sending packets, and updating every cell of the timestamp register, the program behaves like ECMP.

\(^1\)Note, that the implementation analyzed in this case study uses an exclusive comparison, i.e., “greater than”. If the comparison would be “greater or equal”, then a total of 10 adversarial packets would be necessary to attack the program.
5.2.2 Congestion-Aware Load Balancing

Like Flowlet Switching, this application [26] distributes flows over multiple parallel paths. However, this implementation utilizes queuing information in the data plane to detect congestion in the network.

ECMP forms the basis for Congestion-Aware Load Balancing (cf. Section 5.2.1). The ECMP offset of a specific flow is updated when a congestion notification is received. Packets in the network have an additional telemetry header, storing the highest ingress queue count of the switches along the path. Switches at the edge of the network send a congestion notification back (by cloning the packet) if this value exceeds a threshold. To send a notification, the program generates a clone with the source and destination IP address being reversed. However, a flow can generate a notification only every second, which reduces the number of notification messages in the network but requires the use of a hash map. Note, that congestion events are usually created by several flows, meaning that all of them would trigger a notification message. Moving all those flows might be suboptimal, causing congestion elsewhere in the network. Congestion-Aware Load Balancer only sends notification messages with a certain probability. Our implementation cannot handle random variables correctly. To still analyze the application, we have replaced all primitives generating a random values with a deterministic assignment.

An adversary may try to mitigate the creation of congestion notifications to make the system behave like a simple ECMP. Note, that, based on the implementation of Congestion-Aware Load Balancing, an adversary is not able to make the switch ignore a notification, because this decision does not depend on any stateful object\(^2\) (cf. Section 4.4).

For analysis, we prepare the controller configuration (table entries) to emulate a switch on the edge of the network. Running our implementation reveals that there exists a vulnerability: The attacker sends a packet at most a second before the victim packet with a large value in the telemetry header, such that a congestion notification is created. Therefore, no notification is generated for the victim. However, the switch generates a clone of the attacker packet and flips the source and destination address. The attacker packet can choose the source IP address such that the notification is either dropped by the routing table or forwarded away from the internal network. This attack can be generalized to affect all flows by repeating this for every cell of the hash map, making the whole system behave like simple ECMP. This is — against our expectations — a reasonable vulnerability of Congestion-Aware Load Balancing.

Our current implementation, however, cannot fully capture the semantics of this attack. It does not allow the user to specify that the attackers should not trigger a congestion notification. By chance, our implementation did return a valid attack, where the forwarding table drops the notification.

5.2.3 Blink

Blink [16] is a data-driven system which leverages TCP-induced signals (retransmissions) to detect local and remote failures directly in the data plane, without relying on slow converging and control-plane-driven BGP signals. Blink achieves this by selecting a set of active flows per prefix to track. The events of retransmissions are filtered by a sliding window to infer failure for a single prefix. In case of failure, Blink selects a backup path for the prefix directly in the data plane. Those backup paths are maintained and updated by the control plane at runtime, e.g., whenever a new BGP route is learned or withdrawn.

Blink relies heavily on register arrays. A flow array is assigned to every monitored prefix (can be configured by the control plane), which stores information about active flows of this prefix and is used to determine whether a new packet is an RTO-induced retransmission. The per-prefix sliding window counts the number of retransmissions in the last 0.8 seconds. Blink infers a failure for a specific prefix if the majority of the monitored flows destined to that prefix experience retransmissions during this time window. The sliding window consists (for each prefix) of 10 bins and a register to store the sum of all bins. Every 80 milliseconds, the sliding window moves to the next bin and subtracts the value stored in this bin from the total sum.

For analysis, we have chosen the branch responsible for selecting the primary or one of the backup routes. We have configured all tables such that Blink is enabled for a specific prefix. To simulate the switch at runtime, the flow array corresponding to the prefix is populated with randomly

\(^2\)Our implementation does classify this if-condition as not dependent on a register.
chosen values. Our implementation takes about 20 minutes to find an attack consisting of only three packets. The attack causes an overflow of the sliding window sum, which will trigger the entire prefix to be rerouted. To achieve this overflow, the attack requires more than 1.5 seconds to run, during which no other packets, destined to the same prefix, are allowed to reach the target switch. However, this is usually not feasible in practice, since the network operator should choose very active prefixes to be monitored. The reason for the overflow is that the sliding window cannot move by more than half of the window per packet instead of the full window\(^3\). After modifying the program accordingly, the vulnerability was no longer found by our implementation.

As the next step, we have manipulated the problem, such that the difference between the first and the last attacker packet is less than 50ms. Additionally, we have decreased the number of flows tracked per prefix from 64 down to 8, which reduces the number of retransmissions in the sliding window required for inferring a failure to 5. Our implementation finds an attack with five adversarial packets, taking less than a minute to find the attack. The reason for this improvement in solving time (even when considering more attacker packets) is that decreasing the range for all timestamps also decreases the number of possible paths through the control flow\(^4\). The attack essentially works by repeating the last packet of five different flows currently tracked by the switch. This will trick the switch into thinking that half of the tracked flows are experiencing failure. The vulnerability can easily be generalized to the unmodified Blink implementation (64 tracked flows per prefix) by repeating the last packet of the majority of tracked flows. The simulation of this generalized attack is depicted in Figure 5.5. It only shows traffic for a single prefix, which has a primary and two backup next-hops configured. At time 0, the attacker repeats the last 33 packets, causing the switch to initiate a fast reroute.

![Figure 5.5: Simulated attack on Blink](image)

For this attack to be successful, the adversary needs to guess which flows are currently being tracked by Blink. By eavesdropping on all interfaces of the switch, the attacker might be able to reduce the number of possibilities significantly. Another approach for the attacker is to repeat one packet of every flow of the prefix per evolution of the sliding window (i.e., every 800ms). Both approaches require the attacker to control powerful hardware in the network. Nevertheless, we consider this application to be vulnerable.

---

\(^3\)Note that the sliding window can only be updated while processing a packet. Therefore, Blink computes the number of cells by which the window should move based on the timestamp of the current packet.

\(^4\)This is not the case in general, but Blink uses the timestamp very often in comparisons.
Chapter 6
Discussion

This section discusses the approach described in this report and its limitations. The case studies (cf. Section 5.2) highlight the importance of verification tools capable of considering a sequence of packets. Our implementation found potential vulnerabilities in all conducted case studies, including Congestion-Aware Load Balancer, for which we expected to find no reasonable attack. The output of the analysis can be used to improve the program or to generalize the vulnerability, such that it affects multiple flows instead of the victim packet only.

The case studies highlight that it is particularly challenging to develop robust data plane programs with P4. Consider a P4 program tracking some state on a per-flow basis. If the programmer decides to use normal hash maps, an adversary can try to fill the entire register array with values to change the behavior of the switch for all flows. More sophisticated data structures can make such attacks less feasible in practice. However, they require complex logic, which significantly impacts the performance of the network device, since the data structure needs to be maintained explicitly by the programmer. P4 needs a new abstraction for storing key-value pairs, which achieve a balance between performance and robustness. Recent publications have considered new abstractions for improving the performance of stateful applications, such as FlowBlaze [28], which extends the abstraction model with state machines on a per-flow basis. However, most of them are only designed to optimize performance, without considering the robustness against adversaries.

6.1 Limitations & Future Work

Our measurements and case studies show that more extensive programs take a long time to analyze, especially with many attacker packets. This effect can be attributed to path explosion — adding additional packets dramatically increases the number of viable paths. Sophisticated techniques are required to solve this problem. Possible solutions are discussed in the literature [5, 7]. However, it is usually not necessary to run the analysis with hundreds of attacker packets at the same time. For many applications, it is enough to generate an attack for a single flow. Many programs can be adapted (i.e., by changing some constants), such that fewer packets are required for the attack. The attack can then be generalized by a human to scale up to the original program and to affect multiple flows. An example of this can be found in the case study of Blink in Section 5.2.3.

Our results show that proposing hash collisions can significantly improve the time to generate adversarial workload. However, our current implementation does only propose a very naive hash collision, i.e., constraining multiple fields of the attacker and the victim to be equal. Our implementation could find more vulnerabilities after stage 1 and 2 by proposing multiple collisions, chosen such that the control flow of the packets remains unaltered (cf. Section 4.5).

SMT-LIB 2 [3] describes several different (sub-)logics, with which the solver can identify fragments where it may apply specialized and more efficient satisfiability techniques. Our implementation relies heavily on bit-vectors, as well as uninterpreted functions and arrays. The SMT logic QF_AUFBV contains optimizations for such problems. In fact, for some applications, we could notice a significant decrease in execution time (e.g., Blink, which took around 15 minutes
to solve a specific problem with the default SMT solver, but only 9 seconds with QF_AUFBV). However, other applications show no improvement by applying QF_AUFBV. In order to speed up our tool and potentially increase the number of attacker packets, one could try to find better optimizations, as well as try to adapt the translation of a P4 program for the optimization to be effective.

The case-study for Congestion-Aware Load Balancing (cf. Section 5.2.2) points out that our current implementation is not able to capture the semantics of all possible vulnerabilities. Furthermore, our approach described in Section 3.2 does not cover all possible attacks, only those who manipulate the control flow of a victim packet. An attack which modifies a register value, such that regular packets are routed to a different egress port without changing their control flow path, cannot be detected. Our tool could be adapted to allow programmers to specify the symbolic conditions for an attack, similar to the approach presented in p4v [22].

Many applications discussed in the literature are running distributed algorithms, where multiple switches, all running the same program, communicate together. Our approach could be extended to analyze an entire P4 network, with multiple switches and links between them, allowing attacks on the distributed algorithm to be found. This would also allow more realistic attacker models to be explored, e.g., allowing the attacker to send packets only from an external network.

Some applications require an already populated initial state for the attack, as highlighted by the case study on Blink in Section 5.2.3. In our current implementation, the programmer must prepare a specific state manually, in order to model a realistic state of the switch during operation. We are interested in exploring possible approaches to generating a representative switch state. A possible approach to automatically generate a realistic switch state is to simulate the switch on real or synthetic network traces. Another idea is to use an assertion-based control plane interface, as described in p4v [22].

Our implementation only approximates the behavior of the V1Model, neglecting the input and output queues. However, recirculate, resubmit and clone makes it possible for a packet to be added to the ingress queue multiple times. This can result in other packets being processed before the recirculated packet, which possibly updates the state of the switch. Modeling the V1Model with queues could allow more sophisticated attacks exploiting queues to be found.

Our implementation could be extended to include counters, meters, and header unions, as well as queuing metadata, so that the entire P4_{16} specification [32], including the PSA [33], can be modeled. Random numbers in our current implementation are in fact not random, but chosen by the attacker. A symbolic treatment of random numbers could allow us to estimate the probability of a successful attack.

The P4 specifications [32, 32, 33] mention that the behavior of the switch for some special cases is undefined. This allows a switch target to do certain optimizations. As an example, the result of reading and writing a field of an invalid header is undefined. As discussed in [22], some switch targets might initialize those values to zero, but the language specification does not mandate this behavior, since it may have a profound impact on the performance of some network devices. The switch target, however, is allowed to return any arbitrary value, including the value used by the last packet. Such a behavior opens a new range of new possible vulnerabilities, specific to a switch target. Our current implementation does not model this behavior. There already exist tools [14, 22, 31] allowing the programmer to verify that these mistakes cannot occur in the application.
Chapter 7

Conclusion

In this report, we proposed a new system to symbolically execute a P4 program to verify numerous different properties of the program. Our design allows not only a single packet, but a sequence of adversarial packets to find more complex vulnerabilities. We designed a pipeline, which is able to split the problem into two smaller tasks, one for finding the path of all packets through the control flow, and one for computing hash collisions. Our system enables a mostly automatic way of preparing the problem for finding vulnerabilities and generating the adversarial workload, which impairs the functionality of the network device.

We have used our system to analyze a variety of different applications. Our implementation finds many plausible attacks, including vulnerabilities which have not been noticed previously, such as the vulnerability in Congestion-Aware Load Balancing. We have also conducted an extensive performance evaluation in order to quantify the time complexity of our system.

We have shown that it is very difficult to write stateful applications for the data plane without introducing a variety of vulnerabilities. The approach presented in this thesis is a promising first step towards robust P4 Programs that are free from vulnerabilities.
Appendix A

User Guide

A.1 Running the Analysis

Make sure that all python requirements are installed

```
pip install -r requirements.txt
```

Then, compile the program with p4c:

```
p4c --target bmv2 --arch v1model application.py
```

Prepare a configuration file (`config.json`) as described in Appendix A.2. Then, run the analysis with:

```
python main.py --verbose --info --config config.json application.json
```

Now, choose a transition which should be analyzed (or use the `--search_all_branches` option) and wait for the analysis to finish. If a vulnerability was found, then the results are printed. It shows the input fields of all packets (alphabetically ordered), and a list of all additional information required for simulation (e.g., table entries and initial states for the registers). If the option `--info` was used, then it also shows the path for every packet through the control flow, and the values of all register arrays in the switch state between the packets.

A.2 Configuration File

The JSON configuration file can contain the following objects (see `example_configuration.json`):

- 'num_resub', number: Numer of resubmit.
- 'num_recirc', number: Numer of recirculate.
- 'num_clone', number: Level of clones. (if set to 2, then the clones can also spawn clones).
- 'num_allowed_attacker_packets', integer: Maximal number of attacker packets allowed in an attack.
- 'min_allowed_timestamp', number (in seconds): Ingress timestamp of the first attacker packet must be larger than this value.
- 'max_allowed_timestamp', number (in seconds): Last egress timestamp of the victim packet must be smaller than this value.
- 'hash_collision_strategy': string:
  - 'none': do not consider any hash functions, only applies stage 1.
  - 'skip': skips stage 2 of the pipeline, without computing hash collisions
  - 'same_input'
  - 'same_input_aggressive'
• 'initial_state_method', string (applies to all registers not specified in 'initial_state'):
  - 'undefined': all registers (which do not occur in 'init_states') are treated as uninterpreted arrays.
  - 'zero': initialize all registers to zero.
  - 'interactive': allows the user to input an initial value for every register in the application.

• 'initial_states': array of JSON objects:
  - 'name': string, name of the register
  - 'values': JSON object, with a default entry 'default' and an entry for every index which is different from the default value. The value can either be an integer or a string with a numeral value.

• 'victim_fields', array of JSON objects:
  - 'field_name', string: Name of the field, including the header (can also be a field in standard metadata).
  - 'value': Value for the corresponding field. Can be a number, or a string having one of the following forms: '12', '0x40', '10.0.0.1', '00:01:0a:00:00:01'.

• 'table_settings', array of JSON objects for every table:
  - 'name', string: Name of the table.
  - 'default_entry', JSON object: Same json object as for 'entries', but without the property 'key'
  - 'entries', array of JSON objects:
    * 'key', array of strings: Key for the table entry, including the match type. Use '10.0.0.0/24' for LPM, '0x80 & & 0xF0' for ternary, and '1 -> 10' for range.
    * 'action', string: Name of the action.
    * 'action_data', array of strings: One entry for every parameter of the action chosen. Again, the same formatting can be used as in 'victim_fields'.

• 'mirroring_sessions', array of JSON objects:
  - 'mirroring_id', number: ID used in the clone primitive.
  - 'egress_port', number: Egress port to be assigned for the cloned packet at the start of the egress pipeline.

• 'custom_hash_params', array of JSON objects:
  - 'name', string: Name of the calculation (see the JSON generated by p4c)
  - 'polynomial', string.
  - 'xor_in', string.
  - 'xor_out', string.
  - 'reflect_in', bool.
  - 'reflect_out', bool.

• 'branch_never_allowed', bool: If set, then all attacker packets are not allowed to enter the selected branch.

• 'victim_must_reach_branch', bool: If set to false, the modified victim is allowed to take a completely different path through the program.
List of Abbreviations

BGP .......... Border Gateway Protocol
CRC .......... Cyclic Redundancy Check
ECMP .......... Equal-Cost Multi-Path Routing
IP .......... Internet Protocol
P4 .......... Programming Protocol-independent Packet Processors
PISA .......... Protocol Independent Switch Architecture
PSA .......... Portable Switch Architecture
RTO .......... Retransmission Timeout
SAT .......... Boolean Satisfiability Problem
SDN .......... Software Defined Networking
SMT .......... Satisfiability Modulo Theories
TCP .......... Transmission Control Protocol
UDP .......... User Datagram Protocol


