Automated Analysis for Correct and Efficient Execution of Software Middleboxes

Kaiyuan Zhang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Arvind Krishnamurthy, Chair

Xi Wang, Chair

Ratul Mahajan

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Automated Analysis for Correct and Efficient Execution of Software Middleboxes

Kaiyuan Zhang

Co-Chairs of the Supervisory Committee:
Professor Arvind Krishnamurthy
Computer Science and Engineering

Professor Xi Wang
Computer Science and Engineering

Middleboxes are important building blocks of modern networks. They provide a wide range of functionalities such as packet transformation (e.g., NATs), security (e.g., firewalls), and performance (e.g., load balancers). As middleboxes reside on the data path of networked applications, their correctness and performance have significant implications.

To improve the correctness guarantee of middlebox implementations, we studied the feasibility of applying automated verification techniques on existing middlebox implementations. Our study shows that, with slight code modifications, over 70% of existing Click elements could be verified using symbolic execution. Based on this result, we designed and implemented Gravel, a framework that allows developers to express RFC properties of middleboxes using its high-level specification API and verifies the correctness of middlebox implementations against those properties.

To utilize the packet processing power of programmable switches, we designed and implemented Gallium, a compiler that performs program analysis on a software middlebox implementation and automatically discovers parts of the program that could be offloaded to the programmable switch. Gallium then generates code in P4, a domain-specific language for programmable switches. Gallium ensures the functional equiva-

lence between the offloaded implementation and the original implementation and could save 21-79% of processing cycles.

To help developers build efficient and scalable middlebox implementations, we present Pebble, a programming framework and runtime system that provides transaction API. Pebble hides the low-level implementation details of concurrency control from the developers. It uses transaction chopping to utilize the pipeline parallelism in middlebox applications and uses optimistic concurrency control to boost the performance for concurrent read operations. Our evaluation shows that developers could build scalable middleboxes with Pebble without significant effort compared with building single-threaded ones.

Table of Contents

List of Figures

Glossary

ANNOTATION: A type of comment added to the source code that provides more information to aid the compiler.

COMPILER: A program that translate high-level source code into low-level form for execution or deployment.

FIREWALL: A network service that inspects and selectively blocks incoming network traffic.

FORMAL VERIFICATION: Mathematically proving that certain properties hold on a software implementation.

LOAD BALANCER: A network service that evenly distribute client requests to multiple servers.

NETWORK INTERFACE CARD: A hardware that connects a computer to network.

PROGRAMMING FRAMEWORK: A software that helps developer create one specific type of application. It hides low-level details from application developers.

MIDDLEBOX: A network device that filters, transforms, or inspects network traffic.

NETWORK ADDRESS TRANSLATION: A method of mapping one IP address space into another one by rewriting packet fields.

SCALABILITY: The capability of software implementation to get better performance with more CPU cores.

TRANSACTION: A unit of work that must either success or fail and could not be "partially complete".

Acknowledgments

Throughout my last six years in University of Washington, I have received a great amount of support from my mentors, colleagues, and families. Without their help, I would never be able to go through this journey.

I would first like to thank my advisors, Arvind Krishnamurthy and Xi Wang. Without their support and patience, this thesis would never be possible to happen. It has been a great honor to work with two outstanding computer scientists.

I would like to thank Danyang Zhuo. As a senior PhD student and co-author, his optimism in research and life had brought me out of my most depressing days.

I also want to thank all my collaborators for their insight and support on my research: Danyang Zhuo, Aditya Akella, Ang Chen, Samantha Miller, Pedro Fonseca, Luke Nelson, Helgi Sigurbjarnarson, Eddie Yan, Ming Liu, Tianyi Cui.

I wish to thank all the faculty members and students in systems' lab for providing feedback to my research: Adriana Szekeres, Ashlie Martinez, Ellis Michael, Henry Schuh, Jialin Li, Jialin Li, Kevin Zhao, Lequn Chen, Naveen Kr. Sharma, Niel Lebeck, Yuchen Jin.

Last but not least, I thank my parents, Shaoxun and Zhaohui, for their love and support.

Parts of this thesis were published in:

[91] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. "Automated verification of customizable middlebox properties with gravel". In: *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI}*

20). 2020, pp. 221–239

[92] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. "Gallium: Automated Software Middlebox Offloading to Programmable Switches". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 283–295

Chapter 1

Introduction

Middleboxes play a critical role in modern networks. They have been used to provide applications with more network features (e.g., NATs), better security (e.g, firewalls or IDS), and better performance (e.g, load balancers). Today, middleboxes are widely deployed in networks [76].

Because of middleboxes' ubiquitous deployment in computer networks, there are two key requirements for the design and implementation of middleboxes: functional correctness and high performance. As most middleboxes operate on the data plane and are processing packets used by networked applications, bugs in middleboxes can directly lead to system failure or information leaks [19, 20, 22, 21, 23]. Worse still, malformed packets can trigger some of these bugs and expose severe security vulnerabilities. Similarly, a slow middlebox implementation can compromise the responsiveness and availability of applications as it will have negative impacts on both the latency and throughput of those applications.

Recent progress in formal software verification offers a promising approach for middlebox developers to provide stronger correctness guarantees. With formal verification, developers can construct a machine-checkable proof that proves certain properties on the given implementation. Researchers have explored the feasibility of applying formal verification on middlebox implementations and have made significant progress [26, 90]. Crucially, these efforts tackle real middlebox implementations rather than abstract middlebox

models and verify non-trivial program properties.

On the other hand, with the rapid growth of network bandwidth, it becomes increasingly challenging for a single CPU core to keep up with it. Therefore researchers began to explore different approaches to bridge such performance gap. One possibility is to build specialized yet still programmable hardware. A good example of them is the programmable switches [12], which are packet processing devices that allow programmability at line rate. Recent studies have shown promising results in offloading middlebox implementation to programmable switches. For example, Silkroad [59] proposes to leverage programmable switches to build high-performance load balancers that can handle substantially more concurrent connections. New use cases, such as using programmable switches for in-network DDoS detection [39], are also emerging.

Another promising approach to improve the performance of software middleboxes is to utilize the multi-core architecture of modern CPUs. Hardware features such as receive-side scaling (RSS) allows the packet processing tasks to be evenly distributed among multiple CPU cores, leaving more time budget for processing each packet.

Given these new approaches to building correct and high-performance middlebox implementations. There are still several challenges that remain to be solved: Just as with using software verification in other areas of computer systems, this can incur a non-trivial amount of proof effort (e.g., 10:1 proof to code ratio in VigNAT [90]). Such proof effort is even more significant when reasoning about concurrent programs (e.g., the proof to code ratio reported in [14] is about 20:1) At the same time, the excessive proof effort prevents researchers from exploring verification of both high-level middlebox-specific properties (e.g., a middlebox rejects unsolicited external connection) and the concurrent processing of packets, which is common in software middleboxes. As a consequence, recent verification efforts focus either entirely on low-level code properties (e.g., free of crashes, memory safety) [26] or on proving equivalence to pseudocode-like low-level specifications [90, 89], and none of them provide formal guarantees for the concurrency processing of packets. On the performance side, offloading to programmable switches still requires middlebox

developers to manually select the components to offload and rewrite the offloaded code in P4, resulting in slower development and deployment time due to the excessive learning curves. Similarly, though RSS enables packet processing with multiple cores, the task of correctly implementing synchronization and concurrency control, which is notorious for its difficulty, is still left to application developers.

In this report, we explore the feasibility of automating the processes of (1) improving the functional correctness of middleboxes via formal verification, (2) offloading components of a middlebox to a programmable switch to improve its performance, and (3) building a concurrent software middlebox from a single-threaded implementation. Specifically, we aim to address the following questions in this report:

1. How to automatically verify high-level middlebox properties?
2. How to automatically detect offload-able middlebox components and generate the offloaded P4 code?
3. How to offer framework support for developers in building concurrent software middleboxes?

How to automatically verify high-level middlebox properties? In particular, our goal is two-fold. First, we want verification to work on real-world "almost unmodified" middlebox implementations without requiring manual proofs. Second, we want developers to be able to express and verify high-level properties directly translated from RFCs (e.g., RFC5382 [62] for NAT) without writing manual proofs towards each of these properties. To deliver on these goals, we seek to replicate the automated reasoning approach used in some recent verification projects that focus on file systems and OS system calls [77, 63]. Specifically, we would like to use symbolic execution to automatically encode a middlebox implementation and its high-level specification using satisfiability modulo theories (SMT) and then use solvers to verify that the implementation is consistent with the specification.

Our key observation regarding the suitability of this approach is that many existing

middleboxes are already designed and implemented in a modular way (e.g., Click [46]) for reusability. As they aim for high performance, the number of operations they perform on each packet is finite and small. Both characteristics place these middleboxes within reach of automated verification through symbolic execution. Thus, one goal of this paper is to identify domain-specific analyses that enable symbolic execution to exploit these characteristics and distill SMT encodings for middlebox implementations.

We begin by studying whether we can use automated verification on existing software middleboxes. We perform a systematic study on all 290 Click elements and 56 Click configurations (≈60K lines of code) in Click's official repository to test whether they are suitable for automated verification.

We find that a baseline symbolic executor can derive symbolic expressions for 45% of the elements and 16% of the configurations. We then introduce a set of domain-specific static analyses and code modifications (such as replacing element state by SMT-encoded abstract data types) to enable the symbolic execution of a more substantial fraction of Click elements. These techniques allow us to symbolically execute an additional 33% and 50% of elements and configurations, respectively.

Encouraged by the results of the empirical study, we designed and implemented Gravel, a framework for automated software verification of middleboxes written using Click [46]. Gravel provides developers with programming interfaces to specify high-level trace-based properties in Python. Gravel symbolically executes the LLVM intermediate representation compiled from an element's C++ implementation. Gravel then uses Z3 [88] to verify the correctness of the middlebox without the burden of manual proofs.

We then evaluate Gravel by porting five Click middleboxes: MazuNAT, a load balancer, a stateful firewall, a web proxy, and a learning switch. We verify their correctness against high-level specifications derived from RFCs and other sources. Only 133 out of 1687, 63 out of 1151, 63 out of 1447, 50 out of 953, and 0 out of 594 lines of code need to be modified to make them automatically verifiable. The high-level specification of the middlebox-specific properties can be expressed concisely in Gravel, using only 177, 70, 68,

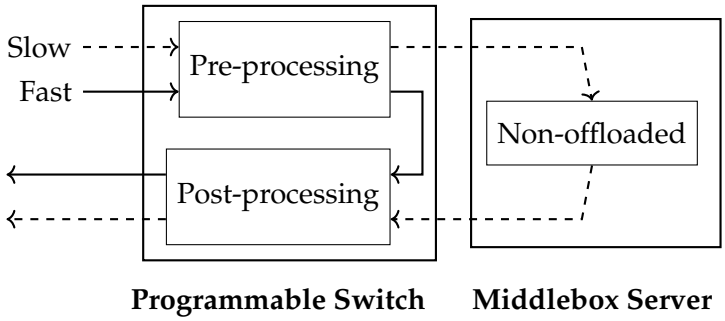


Figure 1.1: Packet flow in an offloaded middlebox.

39, and 91 lines of code. Our evaluation shows that Gravel can avoid bugs similar to those found in existing unverified middleboxes. Finally, we show that the code modifications do not degrade the performance of the ported middleboxes.

How to automatically detect offload-able middlebox components and generate the offloaded P4 code? To make this question concrete, we focus on a single type of offloading scenario shown in Figure 1.1 that uses a programmable P4 switch (e.g., a Tofino switch [82]) and a regular middlebox server.

The goal is to automatically partition the input middlebox program into three parts, i.e., a pre-processing partition, a non-offloaded partition, and a post-processing partition. The pre-processing and post-processing partitions run on the programmable switch as a single P4 program, and the non-offloaded partition executes on the middlebox server. Packets coming to the middlebox go through the pre-processing, non-offloaded, and post-processing partitions sequentially. Because a programmable switch is efficient at packet processing, this offloading scenario can improve middlebox performance if enough instructions are offloaded to the switch. Further, if the non-offloaded partition is not involved in processing a packet, the packet can simply skip the middlebox server, reducing latency and processing cycles in the server. We call this a fast path in the middlebox.

We have designed and implemented Gallium, a compiler that automatically partitions and compiles a software middlebox written in C++ to a functionally equivalent middlebox that leverages a programmable switch to achieve higher performance. Gallium has to address three main challenges: (1) ensuring the output code is functionally equivalent

to the input middlebox, (2) enforcing expressiveness constraints and resource limits on the pre- and post-processing partitions, and (3) providing the run-to-completion concurrency model even though packet processing is split across a software middlebox and a hardware switch.

Gallium partitions the input middlebox source code at a fine granularity (e.g., P4 operations and CPU instructions). To ensure functional equivalence, Gallium keeps track of each instruction's dependencies, i.e., what instructions need to run before a given instruction. For an instruction, I, to behave in the same way for the unpartitioned and partitioned versions, we need to make sure the state accessed by I is the same across the two versions. Therefore, we need to execute all the instructions that may affect I's observable state before we run I. Gallium provides a simple approach to extract all the instruction-level dependencies by comparing each instruction's read and write sets (i.e., the collection of variables an instruction accesses or modifies).

Because programmable switches only accept P4 programs—a declaration of a pipeline of match-action tables with limited ALU functions (e.g., addition, subtraction, comparison), Gallium statically analyzes the input middlebox to figure out the set of instructions to assign to the pre- and post-processing partitions. Gallium must also consider the resource constraints inside the programmable switch, e.g., the total amount of memory and the scratchpad state used to process a packet. We design and implement a partitioning algorithm that maximizes the number of offloaded operations while meeting the constraints on the programmable switch.

Finally, Gallium needs to generate deployable output middleboxes and execute them with the desired run-to-completion semantics. Gallium maps data structures and instructions in a general-purpose programming language (C++) to their P4 counterparts. Gallium synthesizes the packet formats to allow middlebox states to flow between the switch and the server. Gallium further employs state synchronization techniques typically found in primary-backup systems to provide the desired run-to-completion concurrency semantics.

We evaluate Gallium on five different middleboxes: MazuNAT, an L4 load balancer, a firewall, a network proxy, and a trojan detector, written using the Click [46] framework in C++ and totaling over 6K lines of code. Our evaluations show that Gallium is able to automatically offload middlebox functionality, save up to 79% processing cycles, reduce latency by up to 31%, and improve throughput by up to 46%.

How to offer framework support for developers in building concurrent software middleboxes? To be more specific, we want to help middlebox developers build high-performance multi-threaded middleboxes while only focusing on the core packet processing logic. That is, developers would implement a concurrent middlebox as if they are implementing it for a single-threaded environment. The task of constructing a concurrent version of it that is both correct and high performance should be handled by programming framework and runtime system.

At a conceptual level, one of our observations is that middlebox implementations often have a pipelined structure. This is because the networking protocol stack is defined as a set of layers (i.e. the seven layers in the OSI model) and packet processing logic of middleboxes often consists of the same layering structure, resulting in a "pipeline" paradigm. In fact, existing software middlebox frameworks such as Click [46] and NetBricks [66] all utilize this pipeline structure to modularize middlebox implementation. We argue that this pipelined nature of software middlebox could also be used by a programming framework to provide parallelism when executed with multiple threads.

Another observation is that multicore packet processing can benefit from existing work on concurrency control of database transactions. One scalability bottleneck of middleboxes is their statefulness, but database transactions are able to scale despite touching very stateful data. Moreover, middleboxes are often processing read-most workloads—most middlebox state is not modified for most packets. The database community has developed optimistic concurrency control mechanisms that can perform well under such workloads.

Based on these two observations, we build Pebble, a programming framework and

runtime system that provides transaction API for developers to implement software middleboxes. With Pebble, developers implement a middlebox as a set of transactions. Pebble uses a transaction chopping technique to utilize the pipeline parallelism of middlebox programs while having the runtime system provide serializable execution of chopped transactions. For each chopped transaction, Pebble adopts optimistic concurrency control (OCC) to improve transaction execution performance for read-most workloads and read-only transactions.

We implement Pebble as a Rust library with over 4K lines of code and use DPDK for efficient packet I/O. We built two concurrent middleboxes with Pebble, a NAT, and a load balancer. Our experience shows that Pebble could reduce the effort of developing middleboxes for multi-threaded environments.

1.1 Roadmap

In the rest of this report, we will first study the problem of formalizing and automatically verify high-level middlebox properties in Chapter 3. We design and implemented Gravel, a framework that lets developers formalize middlebox properties from existing RFCs and automatically verifies whether those properties hold on the implementation. Chapter 4 explores the feasibility of automated offloading of middlebox implementation to programmable switches. We present Gallium, a compiler that discovers components in the middlebox that could be offloaded to a programmable switch and generate the P4 code automatically. Chapter 5 describes our preliminary results on providing framework support for concurrent software middleboxes. We conclude in Chapter 6.

Chapter 2

Background

As middleboxes are responsible for processing massive amount of packets in the network. Improving their correctness and performance have always been the focus of middlebox developers. To provide better correctness guarantee, researchers have turned into *formal verification* methods, where machine-checkable proofs are used to verify correctness properties of software implementations. On the other hand, recent development of programmable networking devices such as the *programmable switches* allows customizable packet processing at line-rate, making offloading a promising direction of improving middlebox performance. Besides that, as the gap between networking device bandwith and CPU processing power keeps growing, it is inevitable for software middlebox implementations to utilize multiple CPU cores and provide correct and efficient *concurrency control* mechanism.

2.1 Software Verification

Formal verification is a promising approach to provide strong correctness guarantee for software implementations. With the help of theorem provers or SMT solvers, developers could produce a machine-checkable proof to verify that certain property holds on the implementation. Recent studies have successfully applied software verification to various kinds of system softwares, such as OS kernel [44, 63, 34], file systems [15, 77], and distributed systems [36, 86].

At a very high level, verification requires two inputs from developers: specification and implementation. The specification is a formal definition of correctness properties that developers want to verifiy, such as memory isolation of an OS kernel or consensus of a distributed key-value store. Implementation, on the other hand, is the actual code getting executed when the software is deployed, such as the source code or compiled binary. After taking these two inputs from developers, a verifier is used to construct a machine-checkable proof. Depending on the complexity of the verification task, the proof could either be created automatically (such as using SMT solvers) or constructed manually by developers (such as using Coq [17] or Isabelle [65]).

2.2 Programmable Switches

Programmable switches [12] are packet processing devices that allow programmability at line rate. We assume an abstract switch model, as described in [10], that provides the following functionality.

Match-action tables: The internal architecture of a programmable switch is a chain of physical match-action tables. Each match-action table can match on specific fields of the packet header and trigger a set of actions, including packet header rewriting, dropping a packet, or delivering a packet to the next table. The match operation can check for exact matches, wildcards, and longest prefix matches. As long as each packet goes through the chain only once, packet processing is at line rate.

Registers or stateful memory: A limited amount of memory can maintain state across packets, such as counters, meters, and registers. This state can be both read and updated during packet processing.

Computation primitives: The switch can perform a limited amount of processing on header fields and registers (e.g., additions, bit-shifts, and hashing).

Programmable switches support the P4 [10] programming model. A P4 program specifies the header format, the pipeline sequence, including what packet header fields each table matches on, what actions each table takes, and how the tables are interlinked. The

contents of the tables are read-only for the data plane. To modify the contents of a table, the switch's CPU (x86) can issue commands to the switching silicon. These commands issued through the control plane can be significantly slower than packet processing.

There are several restrictions imposed by programmable switches. They support only a small set of ALU operations on limited data types. Currently, only integers are supported but not floating-point numbers. The list of supported operations is limited to integer addition, subtraction, bitwise operations (i.e., AND, OR, XOR, NOT, SHIFT), and comparison. Further, a programmable switch also limits the number of sequential processing steps to the number of match-action pipeline stages (generally around 10 to 20), so the amount of computation that can be offloaded is bounded. Within a pipeline stage, rules are processed in parallel.

A key restriction is the memory limit on the programmable switch. The total amount of memory is a few tens of MBs on today's switches. Another memory constraint that requires attention is the scratchpad memory allowed for per-packet processing. A programmable switch allows some metadata to be stored in a scratchpad memory while processing a packet. This metadata is allocated when a packet arrives and is garbage-collected when the packet leaves the switch. The total amount of scratchpad memory for storing packet-level metadata is less than a few hundred bytes for processing a single packet.

Programmable switches also restrict the control flow. P4 programs cannot have loops, because the hardware realization is a sequence of physical tables, and each packet goes through the sequence only once. Another implication is that if a packet does a lookup on a particular table, the packet can no longer access the same table in a later pipeline stage.

Programmable switches can read and write packet contents that are only at the beginning of the packet (typically, the first 200 bytes of a packet). This restriction means that developers must carefully design the packet format if they want to transfer additional data using packet headers.

2.3 Transaction and Concurrency Control

Commonly used in databases and file systems, a transaction is an abstraction that defines a unit of work. One major benefit of using transaction interface is to let developers focus more on the application logic. Low-level implementation details such as use locking to prevent race condition or using logging to provide fault tolerance are all handled by the execution environment of transactions and are transparent to application developers.

2.3.1 Serializability

Serializability is an intuitive way of defining correctness of concurrently running transactions. Its formalization is a well-studied topic in databases and distributed systems. At a high level, an execution of a set of concurrent operations is considered serializable if there exists an ordering of these operations where the result of the concurrent execution and sequential execution have equivalent results.

One common formalization used to reason about serializability is the serialization graph. A serialization graph is a directed graph where each vertex represents a thread. Two vertices are connected by a directed edge if the two atomic operations from different threads conflict and the direction of the edge is determined by their order of execution. An execution trace is serializable if its corresponding serialization graph is acyclic.

2.3.2 Transaction Chopping and Runtime Pipelining

Transaction chopping [75] is a technique that increases parallelism by breaking down large, long-running transactions into smaller pieces. To avoid non-serializable execution of chopped transactions, it requires the absence of SC-cycles in the SC-graph of chopped transactions. The SC-graph is constructed by connecting chopped transaction pieces originated from the same larger transactions with S(ibling)-Edges and conflicting pieces from different transactions with C(onflict)-Edges. Shasha et al. [75] have shown that if the chopping result is SC-acyclic, the concurrent execution of chopped transactions remains serializable as long as sub-transactions originated from the same large transaction are

executed in serial order.

Runtime pipelining (RP) [87] relaxes the SC-acyclic requirement of transaction chopping with a runtime system that enforces serializable execution. Therefore, it could allow finer-grained chopping of transactions. The runtime system of RP assigns a rank number for each individual table and requires each transaction to acquire table locks according to rank order. Therefore, for a transaction T with the highest accessed rank number r, we know that T will never access tables with rank number r' where $r' < r$. RP then records the ordering of conflicting accesses from different transactions and enforces the same ordering for all future conflicting accesses: if the runtime system find that $T_i < T_j$[1] from previous conflicting accesses, it stops T_j from acquiring any table lock with rank $r' \geq r$, where r is the highest rank acquired by T_i. RP's runtime enforcement ensures serializability while offers better parallelism compared to a SC-acyclic chopping.

2.3.3 Optimistic Concurrency Control

Optimistic Concurrency Control(OCC) [47] is commonly used in in-memory databases [1, 35, 84] and transactional object libraries [37, 61]. Instead of acquiring locks for each accessed objects when executing the transaction, OCC makes all the read operations "unrestricted" and validates the result of those reads before committing the transaction. The major benefit of OCC that makes it promising in software middleboxes is that the read operations can be performed without any locking mechanism. This allows efficient processing when most of the transactions are readers, which aligns with the workload of typical middlebox applications such as NAT or load balancer.

[1]Here we use $<$ to denote "ordered before" relation

Chapter 3

Gravel: Automated Verification of Customizable Middlebox Properties

This chapter focuses on studying the feasibility of automated verification of middlebox implementation using symbolic execution. We began with an empirical study over 290 Click elements and 56 Click configurations in Click's official repository. We found that with a set of domain-specific static analyses and code modifications, we could symbolically execute 78 % of Click elements. Based on the results of the study, we introduce Gravel, a framework for verifying middleboxes written in Click [46]. Our evaluation shows that Gravel could avoid bugs while not degrade the performance of middleboxes.

3.1 Encoding Existing Software Middleboxes

3.1.1 Automating verification using symbolic execution

A well-established approach to software verification is deductive verification. In this style, a developer generates a collection of proof obligations from the software and its specifications. Proof assistants, such as Coq [17], Isabelle [65], and Dafny [48], are highly expressive, allowing mathematical reasoning in high-order logic. However, the verification process is mostly manual, requiring significant effort from the developer to convey his/her knowledge of why the software is correct to the verification system. For example, when applied to a NAT, VigNAT [90] shows a 10:1 proof-to-code ratio.

```
class CntSrc : public Element {
    // omitting constructor and destructor
    Packet *process_packet(Packet *pkt) {
        if (pkt->ip_header->saddr == target_src_)
            cnt_++;
        return pkt;
    }
    IPAddress target_src_;
    uint64_t cnt_;
}
```

Figure 3.1: A C++ implementation of a simple packet counter.

Recently, researchers have started exploring the feasibility of automating the verification process through exhaustive symbolic execution, which encodes the middlebox implementation into a symbolic expression that can be checked against a high-level specification. This style of software verification reduces the developers' manual proof effort and has already been used successfully to verify file systems [77] and operating systems [63]. However, this style is more limited than deductive verification, putting restrictions on the programming model. For example, Hyperkernel requires loops in its system call handlers to have compile-time bounds on their iteration counts.

To see an example of symbolic execution based verification, Figure 3.1 shows a simple packet counter. This code increments a counter when the source IP address of a packet matches a signature (i.e., `target_src_`). Here we model this `process_packet` function as $f : \mathbb{S} \times \mathbb{P} \mapsto \mathbb{S} \times \mathbb{P}$, where $\mathbb{S}$ is the set of all possible internal states (`target_src_` and `cnt_`) and $\mathbb{P}$ denotes the set of all possible packets. For simplicity, this formulation assumes that at most one outgoing packet is generated for each incoming packet. Symbolically

executing this code snippet generates the following symbolic expression:

$$\begin{aligned}\forall s,\ s' \in \mathbb{S}, \forall p,\ p' \in \mathbb{P},\ f(s,p) = (s',p') \Rightarrow \\ (p' = p) \wedge (s'.target\_src = s.target\_src) \\ \wedge (p.saddr = s.target\_src \Rightarrow s'.cnt = s.cnt + 1) \\ \wedge (p.saddr \neq s.target\_src \Rightarrow s'.cnt = s.cnt))\end{aligned}$$

This symbolic expression says that for all possible inputs, outputs and state transitions: (1) the input packet should be the same as the output packet; (2) the `target_src_` should not change; (3) if the packet's source IP address matches `target_src_`, the `cnt_` in the new state should be the `cnt_` in the old state plus 1; (4) if the packet's source IP address does not match `target_src_`, the `cnt_` should not change.

Symbolic execution alone is not enough for automated verification; it only ensures that we can automatically generate the above expression. To ensure automated verification, when the developer verifies the above expression against a specification using an off-the-shelf theorem solver (such as Z3 [88]), the solver needs to be able to solve it efficiently.

A program is suitable for **automated verification** if:

1. Symbolic execution of the program halts and yields a symbolic expression.
2. The resulting symbolic expression is restricted to an effectively decidable fragment of first-order logic.

Condition 1 means the program has to halt on every possible input. Condition 2 depends on which fragment of first-order logic a solver can solve efficiently. This fragment changes as solver technologies improve over time. Empirically, we know that if we can restrict the symbolic expression to only bit vectors and equality with uninterpreted functions, a solver can tackle the expression efficiently [63].

3.1.2 Baseline effectiveness of symbolic execution

We first study the feasibility of automating verification by examining middleboxes written using Click. We perform an empirical study on both Click elements and Click configu-

rations, which are datagraphs formed by composing Click elements. This study allows us to measure both the fraction of Click code and the fraction of automatically verifiable Click programs. To perform this empirical analysis, we implement a baseline symbolic executor to analyze whether an element or a configuration satisfies the conditions mentioned above. Since elements are C++ classes, the symbolic executor first analyzes all the member fields to determine whether the state could be encoded with SMT (Condition 2). It then performs symbolic execution over the compiled LLVM byte code[1] of the element to check if Condition 1 is met. However, since both conditions are undecidable, we choose to use the following two conservative criteria. (In fact, we describe in the subsequent section how we augment our baseline symbolic executor with domain-specific extensions.)

Absence of pointers in element state. When the symbolic executor analyzes each of an element's members, it checks whether the element state can be expressed solely by bit vectors and uninterpreted functions. Though one could use bit vectors to encode the entire memory into a symbolic state, it would be difficult to efficiently solve expressions containing such a symbolic state due to the sheer size of the search space. Therefore, we choose a conservative criterion, the absence of pointers in element states, as it is easy to see that elements without pointers always have bounded state. Each element in Click can only have a finite number of member variables, and each non-pointer variable can only consume a finite amount of memory. Thus, the state space of a Click element without pointers can always be expressed by constant size bit vectors. Of course, such criteria introduce false negatives, for example, using pointers to access a bounded data structure (e.g., fixed-size array).

Absence of loops and recursions. To determine whether Click elements' execution is bounded (Condition 1), the symbolic executor invokes the packet processing code using a symbolic element state and a symbolic packet content. The symbolic executor detects po-

[1]We chose to use LLVM byte code rather than C++ abstract syntax tree as the former makes it easier to reason about the control flow by eliminating C++ related complexities (e.g., function overloading and interface dispatching).

tential unbounded execution by searching for loops and recursive function calls and only performs execution on those elements that do not contain them. The symbolic executor performs the check by comparing each jump/call target with the history of executed instructions.

Table 3.1 shows the results of running this baseline symbolic executor. We found that 130 of the existing Click elements (45%) are suitable for automated verification. Among the ones that failed our test, 143 elements failed because of pointers, and 78 elements failed because of unbounded execution. 61 of the elements have both pointers and unbounded execution. A Click configuration is amenable to automated verification if and only if all the Click elements in the configuration can be automatically verified. Among the 56 configurations in the official Click repository, only 9 out of the 56 Click configurations (16%) are suitable for automated verification.

3.1.3 Enhancing symbolic execution

We now augment our baseline executor with additional techniques that aid symbolic execution. We also examine the impact of performing a small number of code modifications to make the middleboxes amenable to automated verification. Some of the techniques described below are broadly applicable but are likely more effective for middlebox programs that operate on packet data with well-defined protocol specifications. The remaining techniques are domain-specific analyses that are suitable only for packet processing code.

Code unrolling. When detecting a backward jump, the symbolic executor unrolls the loop and executes its loop body. The executor keeps count of how many times it executes the backward jump instruction and raises an error if the number goes beyond a pre-defined threshold. This technique is useful when the source code has loops with a static number of iterations or loops whose iteration count is a small symbolic value, as would be the case for code that processes protocol fields of known size.

Pointer analysis to detect immutable pointers and static arrays. In general, we can clas-

sify the use of pointers into three categories: pointers to singleton objects, pointers corresponding to arrays, and pointers used to build recursive data structures. These use cases introduce two distinct challenges in the symbolic execution of Click code with pointers. First, the symbolic executor needs to determine whether two pointers point to overlapping memory regions and update the symbolic state of elements correctly irrespective of which pointer is used for the update. Second, when pointers are used to implement recursive data structures, such as linked list or tree, the data structure access often involves loops whose iteration counts depend on the symbolic state of the elements. Our symbolic executor first identifies how pointers are used and then uses the appropriate technique for symbolic execution.

We first use an analysis pass to identify immutable pointers by checking which of the pointer fields in a Click element remain unmodified after allocation. At the same time, we determine which of the other program variables serve as possible aliases for a given pointer field. Further, for pointers pointing to an array of data items, the symbolic executor also performs a static bounds check on accesses performed using the pointers to ensure that all accesses are within allocated regions. By doing so, the symbolic executor can prove an invariant that accesses performed using the array pointer do not touch other memory regions.

After performing these analyses, the symbolic executor limits itself to handling accesses through immutable and unaliased pointers that refer to either singleton objects or arrays. For each pointer referring to a singleton object, the executor associates a corresponding symbolic value. For each pointer referring to an array of data, the symbolic executor uses an uninterpreted function in SMT to represent the contents of the array. The symbolic executor uses uninterpreted functions that map array index (64-bit integer) to bytes (8-bit integer) to model the content of the array. We choose to use this offset-to-bytes mapping as the unified representation for both array and packet content since reinterpreting a sequence of bytes in memory as a different type is a common practice in packet processing (e.g., parsing packet header, endianness conversion). We record updates to the

array as a sequence of (possibly symbolic) index/value pairs. Since the functions are "uninterpreted", they model all possible values of the array data. Compared with bit vectors, representing states with uninterpreted functions makes symbolic execution scale to larger state size [77, 13].

Our symbolic executor does not handle pointers that are used to build a recursive data structure, such as a linked list, except in the case of certain abstract data types for which we are able to provide SMT encodings (as discussed next).

SMT encodings of commonly used abstract data types. Our next technique avoids the symbolic execution of the data structure implementation by hiding the implementation under a well-defined data structure interface. This technique allows us to integrate implementations that may contain unbounded loops or recursive data structures into our analysis. When performing the symbolic execution, we can simply provide an encoding in SMT for common data structures, such as `HashSet`. Note that not all data structures can have their interfaces encoded in SMT. The key challenge here is to prevent the explosion of the state space; the size of the encoding should not depend on the actual size of the data structure. We managed to encode three commonly used data structures in Click, `Vector`, `HashSet`, `HashMap`, into SMT. (See §3.3.3.)

Replace element state with abstract data types. With the SMT encoding of common data types, another technique we could apply is to modify the element implementation by replacing its states with the data types mentioned above. This process requires the developer to inspect how the packet processing code uses a specific element state. If all the accesses performed on the state can be modeled using the interface of a data type with SMT encoding, we could replace the state with the SMT-encoded counterpart and run the symbolic execution on the modified implementation instead.

Consider the `CheckIPAddress` element (Figure 3.2). This element serves as a source IP packet filter. Before our proposed modifications, `CheckIPAddress` stores a list of bad IP addresses (`bad_src_`). A packet is dropped if the source IP address of the packet is listed

```
class CheckIPAddress : public Element {
    // omitting constructor and destructor
    Packet *process_packet(Packet *pkt) {
        auto saddr = pkt->ip_header->saddr;
-       for (size_t i = 0; i < num_bad_src_; i++)
-           if (bad_src_[i] == saddr)
+       if (bad_src_.find(saddr) != bad_src_.end())
                return NULL;
        return pkt;
    }
-    IPAddress *bad_src_;
-    size_t num_bad_src_;
+   HashSet<IPAddress> bad_src_;
}
```

Figure 3.2: Modification of `CheckIPAddress`'s implementation to remove the usage of pointers and loops.

in the bad IP address list. In this element, `bad_src_` and `num_bad_src_` together represents a fixed size array containing the bad IP addresses. To check whether the source IP address of a received packet matches any address in the array, `CheckIPAddress` uses a "for" loop to go through this array. `CheckIPAddress` is not suitable for automated verification: (1) The size of the array that `bad_src_` is pointing to is not known by the symbolic executor; thus, it may flag out-of-bound memory access. (2) If the executor tries to unroll the loop, it faces a path explosion problem as the number of iterations in the loop can be large.

To make this element meet the conditions for automated verification, we can modify its implementation, as shown in Figure 3.2. This change is based on the observation that the way `bad_src_` and `num_bad_src_` are used complies with the `HashSet` interface. The change replaces the pointer-size pair `bad_src_` and `num_bad_src_` with a `HashSet`. Besides that, the "for" loop to check whether the source IP is in the bad IP address list is also

Technique	# Elements	# Conf.
Unmodified	130 (45 %)	9 (16 %)
Code unrolling	138 (48 %)	9 (16 %)
Fix-sized array detection	185 (63 %)	9 (16 %)
SMT-encoded abstract datatype	218 (75 %)	13 (23 %)
Replacing with abstract datatype	222 (77 %)	15 (27 %)
Concretization	226 (78 %)	37 (66 %)

Table 3.1: Number of Click elements and configuration that can be symbolically executed.

replaced with a `find` method call. The code changes remove both the use of pointers and unbounded loops. Since the semantics of `HashSet` and its `find` interface is modeled with SMT, we can symbolically execute the element.

Concretization of control flow structures. Middleboxes perform packet classification based on the value of specific fields in the packet header. Packet classification is implemented using finite-state machines, and it is often optimized by statically compiling the classification rules into a state machine model that is stored in memory. When processing an incoming packet, the classifier performs state transitions using the rules until the state machine reaches one of the end states. If the values of the state transition table are abstract, then the classification process would appear to be unbounded.

We address this issue and enable the symbolic execution of the classification tasks. We load the Click configuration containing concrete classification rules and run the state machine creation code of the classification element. We then ingest the raw bytes representing the transition rules into symbols with concrete values. We use symbolic execution to verify that the contents of the memory region representing the state transition table remain unchanged during program execution. We then symbolically execute the packet classification code but replace the symbolic transition rules with the concrete values identified in the first step. The executor can thus process the packet classification code within a statically bounded number of steps.

3.1.4 Overall effectiveness of symbolic execution

We now repeat our analysis of Click elements and configurations after enhancing our symbolic executor with these additional techniques. Table 3.1 shows the result. Our techniques improve the fraction of elements that can be symbolically executed from 45% to 78%. The fraction of Click configurations that are suitable for automated verification improves from 16% to 66%.

Our symbolic executor cannot handle 22% of the Click elements. Among the 64 unsupported elements, 19 of them could not be symbolically executed because there are loops that traverse the payload of the packets (e.g., `AES` element for encryption). Another 26 elements use customized data structures that contain pointers that can not be modeled with SMT. One such example is `LookupIP6Route` element that uses a match table with longest prefix matching as opposed to a traditional exact match hash table. 11 elements contain loops whose number of iterations is based on the current (symbolic) element state. For example, the `AggregateFilter` element, which aggregates incoming packets according to their header values, has to loop over a queue to determine which aggregation group a packet should belong to. 8 elements have pointer accesses that are deeply coupled with the rest of the code that replacing with abstract data types is not feasible. For example, `IP6NDSolicitor` uses a set of linked lists to handle the response messages of the neighbor discovery protocol.

Three approaches can potentially improve Gravel's ability to verify more Click elements automatically. The first approach is to model more data structures using SMT. Currently, Gravel only supports `HashMap`, `HashSet`, and `Vector`. The second approach is to allow developers to write annotations to rule out part of the implementation that is not relevant to the specification. For example, if the developers only want to prove that the `AES` element does not change the TCP header of the packet, the symbolic executor can skip over the loop that traverses the packet payload. The third approach is to use an interactive theorem prover (e.g., Coq [17], Dafny [48]) to verify the correctness of element-level im-

plementations. These interactive theorem provers can verify higher-order logic than what SMT can verify. For example, more sophisticated data structures such as priority queues or an LRU cache could be more easily verified with the help of an interactive prover.

3.2 The Gravel Framework

Gravel is a framework for specifying and verifying Click [46]-based software middleboxes. It aims to verify high-level properties, such as a load balancer's connection persistency, against a low-level C++ implementation. Gravel uses symbolic execution to translate the C++ implementation into a symbolic expression automatically, and it uses the techniques described in the previous section to enhance the effectiveness of symbolic execution. In this section, we describe how Gravel allows developers to specify the desired high-level properties using Python code and a domain-specific library containing verification primitives. In Section 3.3, we describe how we check whether the symbolic expression derived from the implementation provides the desired properties.

3.2.1 Overview

Figure 3.3 shows the workflow of Gravel. Gravel expects three inputs from middlebox developers:

1. Click configuration, which is a directed graph of elements.
2. A set of high-level middlebox specifications.
3. Element-level specifications for all Click elements used in the configuration.[2]

Like building a normal Click middlebox, Gravel first takes as input a directed graph of Click elements. In Click, a middlebox is decomposed into smaller packet processing "elements". Each element keeps private state that is accessible only to itself and has a set of handlers for events such as incoming packet or timer events. Elements can also have many input and output ports through which elements can be connected with others and transfer packets. The directed graph from a Click configuration connects Click elements

[2]Gravel provides specifications for commonly used elements.

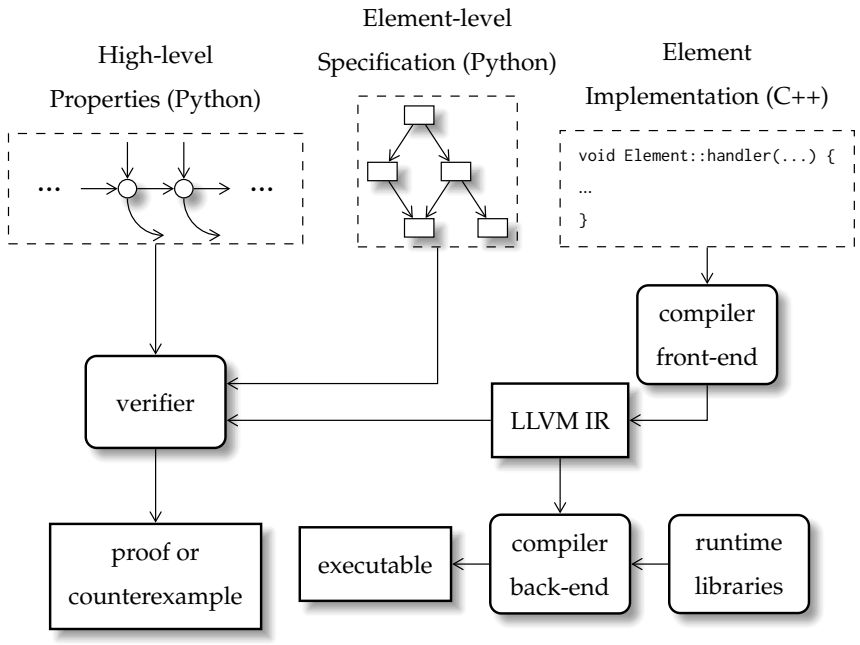


Figure 3.3: Development Flow of Gravel. Top three boxes denote inputs from middlebox developers; rounded boxes denote compilers and verifiers of Gravel; rectangular boxes denote intermediate and final outputs.

together to form the dataplane for packet processing. The topology of the directed graph remains unchanged during the execution of the middlebox.

Gravel then requires a formalization of the high-level middlebox properties. To check properties automatically with an SMT solver, they need to be expressed using first-order logic. In Gravel, properties are formalized as predicates over a trace of events. Gravel includes a Python library for developers to specify middlebox-specific properties.

Gravel also requires a specification for each Click element. The element-level specification describes each element's private state and packet processing behavior. The element-level specification provides a simplified description of an element's behavior and omits low-level details such as performance optimizations. Gravel again provides a Python library for developers to write element-level specifications.

With these three inputs, Gravel verifies the correctness of the middlebox in two steps. First, Gravel checks whether a Click configuration composed using Click elements satis-

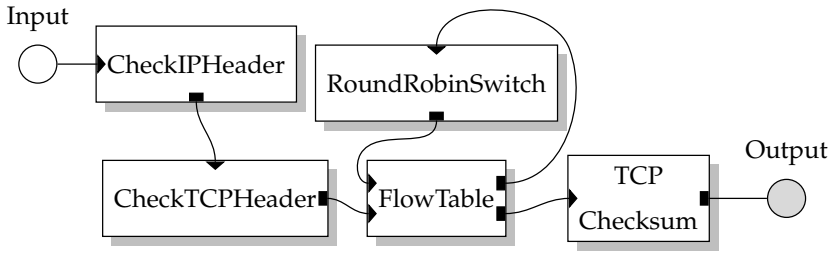


Figure 3.4: Breakdown of ToyLB's functionalities into packet-processing elements.

fies the desired high-level properties of the middlebox. A high-level property is expressed as a symbolic trace of the middlebox's behavior (in Python). Gravel verifies the high-level property by symbolically executing the datagraph of elements using element-level specifications (in Python). Then, Gravel verifies that the low-level C++ implementation of each element has equivalent behavior as the element-level specification. Gravel compiles the low-level C++ implementation into LLVM intermediate representation (LLVM IR) and then symbolically executes the LLVM IR to obtain a symbolic expression of the element. Gravel then checks whether the element-level specification holds in the element's symbolic expression. If there is any bug in the Click configuration or the implementation of the elements, Gravel outputs a counterexample that contains element states and an incoming packet that makes the middlebox violate its specification.

3.2.2 A Sample Application: ToyLB

The rest of this section describes the Gravel framework in the context of a simple running example corresponding to a Layer-3 load balancer, ToyLB. ToyLB receives packets on its incoming interface and forwards them to a pool of servers in a round-robin fashion. It steers traffic by rewriting the destination IP on the packet. ToyLB resembles popular Layer-3 load balancer designs used by large cloud providers [33, 28].

The ToyLB middlebox is decomposed into five elements, as shown in Figure 3.4. When there is an incoming packet, it first goes through two header-checking elements, `CheckIPHeader` and `CheckTCPHeader`. These two elements act like filters and discard any packet that is not a TCP packet. Then, the `FlowTable` element checks whether the packet belongs to a TCP

flow that has been seen by ToyLB earlier. If so, `FlowTable` rewrites the packet with the corresponding backend server's IP address stored in the FlowTable and sends the packet to the destination server. Otherwise, the `FlowTable` consults a `RoundRobinSwitch` scheduler element to decide which backend server should the new connection bind to. After the `RoundRobinSwitch` decides which backend server to forward the packet to, `RoundRobinSwitch` notifies the `FlowTable` of the decision. The `FlowTable` stores the decision into its internal state and also rewrites the destination address of the packet into the destination server. For further simplicity, low-level functionalities such as ARP lookup are omitted in ToyLB.

We next describe how Gravel can be used to model high-level specifications of middleboxes such as ToyLB and then outline how the element-level properties are specified. Later, in §3.3, we show how Gravel performs verification.

3.2.3 High-level Specifications

Gravel models the execution of a middlebox as a state machine. State transitions can occur in response to external events such as incoming packets or passage of time. The time event can be used to implement garbage collection for middlebox states. For each state transition, the middlebox may also send packets out.

Gravel provides a specification programming interface, embedded in Python, for developers to specify high-level properties. Developers can use the interface to describe middlebox behavior over a symbolic event sequence. (See §3.3.3.)

Packets in Gravel's high-level specification are expressed using key-value map abstraction, where the keys are the name of header fields and values are the content of the fields. This abstraction makes the specification concise and hides the implementation details that are less related to high-level properties (e.g., the position of IP addresses in the packet header).

Gravel provides three kinds of core interfaces (see §3.3.3) in its high-level specification: (1) a set of `sym_*` functions that allow developers to create symbolic representations of different types of states such as IP address, packet, or middlebox state; (2) middlebox's

event-handling functions, like `handle_packet(state, pkt)`, `handle_time(state, timestamp)`, that takes as input the current state of the middlebox and the incoming packet/time event, and returns an (optional) output packet and the resulting middlebox state after a state transition; and (3) the `verify(formula)` function call that first encodes the given logical formula in SMT and invokes the SMT solver to check if `formula` is always true. Besides that, Gravel also provides some helper functions for developers to encode high-level middlebox properties.

To make this concrete, we next describe how to encode two high-level properties of ToyLB using this specification programming interface. We describe how to encode two load balancer properties: (1) liveness and (2) connection persistence. We first consider the liveness guarantee.

PROPERTY 1 (ToyLB liveness)**.** For every TCP packet received, ToyLB always produces an encapsulated packet.

In Gravel, this can be specified as:

```
def toylb_liveness():
    # create symbolic packet and symbolic ToyLB state
    p, s0 = sym_pkt(), sym_state()
    # get the output packet after processing packet p
    o, s1 = handle_packet(s0, p)
    verify(Implies(is_tcp(p), Not(is_none(o))))
```

In this liveness formulation, we first construct a symbolic packet `p` and the symbolic state of the middlebox `s0`. Then, we let the middlebox with state `s0` process the packet `p` by invoking the `handle_packet` function. After that, the state of the middlebox changes to `s1`, and the output from the middlebox is `o`. If `o` is `None`, the middlebox has not generated an outgoing packet. This high-level specification says that, if the incoming packet is a TCP packet, the middlebox has an outgoing packet.

Note that the formulation of liveness property is abstract, given that it does not say anything about the states of the middlebox. We don't even formulate the set of data struc-

tures used by `ToyLB`. This brevity is indeed the benefit of using high-level specifications. These formulations are concise and are directly related to the desired middlebox properties.

Now, we move to a more complex load balancer property—connection persistency. This property is crucial to a load balancer as it ensures that packets from the same TCP connection are always forwarded to the same backend server.

PROPERTY 2 (ToyLB persistency)**.** If ToyLB forwards a TCP packet to a backend b at time t, subsequent packets of the same TCP connection received by ToyLB before time $t +$ *WINDOW*, where *WINDOW* is a pre-defined constant, will also be forwarded to b.

Formulation of Property 2 is more complex than the liveness property because it requires a forwarding requirement (i.e., the forwarding of packets of a certain TCP connection to b) to hold over all possible event sequences between time t and time $t + WINDOW$. This complexity means that we cannot formulate connection persistency with traces containing only a single event, but rather, we need to use induction to verify that the property holds on event traces of unbounded length.

Gravel allows us to specify Property 2 as an inductive invariant. First, we formulate the forwarding condition that should be held during the time window. The `steer_to` function defined below determines whether a packet received at time `t` will be forwarded to the backend server with address `dst_ip`. The code snippet first lets the middlebox handle a time event with timestamp `t`, followed by the handling of `pkt`. We ascertain whether the packet is forwarded to `dst_ip` by checking that the output from the packet processing is not `None` and that the resulting packet's destination address is `dst_ip`.

```
def steer_to(state, pkt, dst_ip, t):
    o0, s_n = handle_time(state, t)
    o1, s_n2 = handle_packet(s_n, pkt)
    return And(Not(is_none(o1)),
               o1.ip4.dst == dst_ip,
               payload_eq(o1, pkt))
```

Then, for the base case of induction, we specify that once ToyLB forwards a packet of a particular TCP connection to a backend, subsequent packets from the same connection received within a period *WINDOW* will be forwarded to the same backend. Similar to the formulation of the liveness property, the following code snippet first creates two symbolic packets and a symbolic middlebox state, then invokes `handle_packet` to obtain the output packet as well as the new state after packet processing. After that, the code requires the verifier to prove that if `p0` is forwarded to `dst_ip`, then a packet, `p1`, in the same connection received any time before the expiration time `ddl` is also forwarded to `dst_ip`, assuming that the middlebox state hasn't changed from state `s1`.

```
def base_case():
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    o, s1 = handle_packet(s0, p0)
    dst_ip, t0 = sym_ip(), s0.curr_time()
    t = sym_time()
    ddl = t0 + WINDOW
    verify(Implies(And(Not(is_none(o)),
                       o.ip4.dst == dst_ip,
                       from_same_flow(p0, p1)),
              ForAll([t], Implies(t <= ddl,
                     steer_to(s1, p1, dst_ip, t)))))
```

In addition to the base case invariant, the specification includes two inductive cases showing that processing an additional event (e.g., a packet from a different connection or time event) does not change the forwarding behavior. The two inductive cases specify that the invariant `steer_to(...)` holds on the middlebox states when processing packets or time events if the timestamp is before the expiration time.

```
def step_packet():
    dst_ip, p0, p1 = sym_ip(), sym_pkt(), sym_pkt()
    s0, t0, p_other = sym_state(), sym_time(), sym_pkt()
    o, s1 = handle_packet(s0, p_other)
    verify(Implies(And(steer_to(s0, p0, dst_ip, t0),
```

```
                    from_same_flow(p0, p1)),
                steer_to(s1, p1, dst_ip, t0)))

def step_time():
    dst_ip, p0, p1 = sym_ip(), sym_pkt(), sym_pkt()
    s0, t0, t1 = sym_state(), sym_time(), sym_time()
    _, s1 = handle_time(s0, t1)
    verify(Implies(And(steer_to(s0, p0, dst_ip, t0),
                    t1 < t0, from_same_flow(p0, p1)),
                steer_to(s1, p1, dst_ip, t0)))
```

3.2.4 Element-level Specifications

Verifying high-level specifications directly from low-level C++ implementations is hard because of the gap in their semantics. Similar to all the seminal work [77, 63, 36] in software verification, we break down the verification process using refinement. Gravel requires the developer to give specifications of each element. As long as the element-level specifications capture the behavior of their corresponding elements' implementation, we can simply use the element-level specifications to prove the high-level specifications. Compared to deductive verification, this incurs a lower verification effort because element-level specifications are short (§3.4). Element-level specifications can be reused across different middleboxes. The element-level specification in Gravel consists of two parts: the definition of abstract states that will be used by the element during execution, and a set of event handling behaviors in response to incoming packets and time events.

Element states. Specification of a Gravel element starts with a declaration of the state associated with the element. To ensure efficient encoding with SMT, Gravel requires the state to be bounded. More specifically, elements' state in Gravel may contain: (1) fixed-size variables including bit vectors; (2) maps from one finite set to another (e.g., a map from IP address space to 64-bit integer). For example, in ToyLB, the state of `FlowTable` is defined as:

```
class FlowTable(Element):
```

```
num_in_ports = 2
num_out_ports = 2

decisions = Map([AddrT, PortT, AddrT, PortT], AddrT)
timestamps = Map([AddrT, PortT, AddrT, PortT], TimeT)
curr_time = TimeT
...
```

This part of element-level specifications defines three components of `FlowTable`'s state:

- `decisions` maps from a TCP connection to a backend server address. `FlowTable` identifies a TCP connection by the tuple of source and destination addresses and port numbers. This map is used to store the results from the `Selector` element.
- `timestamps` stores the latest times at which packets were received for each TCP flow stored in `decision`.
- `curr_time` stores the current time.

Here the types such as `AddrT` and `TimeT` are pre-defined integers of different bit widths. Besides the state, the code also informs Gravel as to how many input/output ports the `FlowTable` element has through `num_in_ports`/`num_out_ports`.

Event handlers. Gravel requires each element to have a handler function for packets received from its input ports. This packet handler needs to be specified in the element-level specification. The specification of the packet handler describes the operations the element performs when handling packets. Besides that, an optional time event handler can also be specified. In Gravel, the two event handlers are defined as functions with the following signatures:

```
flowtable_process_packet(state, pkt, in_port) → actions
flowtable_process_time(state, timestamp) → actions
```

The return value of each event handler (`actions`) is a list of *condition-action pairs*. Each entry in the list describes the action an element should take under certain conditions. In

the python code, developers can write:

```
Action(cond, { port_i : pkt_i }, new_state)
```

to denote an action that sends `pkt_i` to output port `port_i` while also updating the element state to `new_state`. This action will be taken when condition `cond` holds. To make it concrete, let us consider the packet handler of `FlowTable`. Upon receiving a packet, `FlowTable` does one of the followings:

- If the packet is from the `CheckTCPHeader` element, and the `decisions` map contains a record for the connection, `FlowTable` rewrites the destination address and sends the packet to `TCP Checksum` element, as shown in Figure 3.5.
- If the `FlowTable` does not have a record for a packet, the packet is sent to `RoundRobinSwitch` element.
- If the packet is sent from `RoundRobinSwitch`, `FlowTable` records the destination decided by `RoundRobinSwitch` and forwards the packet to `TCP Checksum`.

Similarly, `FlowTable`'s behavior in response to time changes is also specified as *condition-action*s:

```
def flowtable_process_time(self, s, time):
    new = s.copy()
    # update the "curr_time" state
    new.curr_time = time
    # records with older timestamps should expire
    def should_expire(k, v):
        return And(s.timestamps.has_key(k),
                   time >= WINDOW + s.timestamps[k])

    new.decisions = new.decisions.filter(should_expire)
    new.timestamps = new.timestamps.filter(should_expire)
    return Action(True, {}, new)
```

When `FlowTable` is notified of a time change, it updates its `curr_time` to the given time value. Gravel offers a `filter` interface for its map object, which takes a predicate, `should_expire`,

```
def flowtable_process_packet(s, p, in_port):
    flow = p.ip4.saddr, p.tcp.sport, \
           p.ip4.daddr, p.tcp.dport
    # the case when flowtable has record of the flow
    known_flow = And(
        # packet is received from the network
        in_port == IN_TCP_FILTER,
        # flowtable has record of the flow
        flow in s.decisions)
    # construct the encapsulated packet
    fwd_pkt = p.copy()
    fwd_pkt.ip4.dst = s.decisions[flow]
    # update the timestamp of the flow with current time
    after_fwd = s.copy()
    after_fwd.timestamps[flow] = s.curr_time
    known_flow_action =
        Action(known_flow,
               {PORT_TO_EXT: fwd_pkt}, after_fwd)
```

Figure 3.5: Example of an element-level action.

and deletes all the entries that satisfy the predicate. `FlowTable` uses this to remove all the records that were inactive for a period longer than a constant `WINDOW` value.

3.2.5 ToyLB's Element-level Specification

This section gives a detailed description of the element-level specification of ToyLB. As mentioned in §3.2, element-level specification in Gravel is given as a list of "condition-action" pairs. In Gravel, developers write python functions that generates the list of possible actions for an element. For example, The `CheckIPHeader` element only forwards packets that are both IP packets and are not from a known "bad" address:

```
def checkipheader_process_packet(s, p, in_port):
    is_bad_src = p.ip.src in s.bad_src
```

```
    return [Action(And(p.ether.ether_type == 0x0800,
                       Not(is_bad_src)),
                   {0: p},
                   s)]
```

Remember that the `Action` is used to create a *condition-action* entry, which denotes an action that the element takes under certain condition (§3.2).

Similarly, `CheckTCPHeader` filters all packets that are not TCP packets.

```
def checktcpheader_process_packet(s, p, in_port):
    return [Action(p.ip.proto == 6,
                   {0: p},
                   s)]
```

`RoundRobinSwitch` not only performs address rewriting for incoming packets, it also updates packet header fields and its own state:

```
def checkipheader_process_packet(s, p, in_port):
    is_bad_src = p.ip.src in s.bad_src
    return [Action(And(p.ether.ether_type == 0x0800,
                       Not(is_bad_src)),
                   {0: p},
                   s)]
```

The `FlowTable` element have a more complex specification as it takes one of three actions based on both the content of the incoming packet and its own state:

```
def flowtable_process_packet(s, p, in_port):
    flow = p.ip4.saddr, p.tcp.sport, \
           p.ip4.daddr, p.tcp.dport
    # the case when flowtable has record of the flow
    known_flow = And(
        # packet is received from the network
        in_port == IN_TCP_FILTER,
        # flowtable has record of the flow
        flow in s.decisions)
```

```
# construct the encapsulated packet
fwd_pkt = p.copy()
fwd_pkt.ip4.dst = s.decisions[flow]
# update the timestamp of the flow with current time
after_fwd = s.copy()
after_fwd.timestamps[flow] = s.curr_time

known_flow_action =
    Action(known_flow,
           {PORT_TO_EXT: fwd_pkt}, after_fwd)

# the case when flowtable does not know the flow
consult_sched = And(
    in_port == INPORT_NET,
    Not(flow in s.decisions))
unknown_flow_action =
    Action(consult_sched, {PORT_TO_SCHED: p}, s)

# packet from the Scheduler
register_new_flow = in_port == IN_SCHED
# extract the new_flow
new_flow = p.inner_ip.saddr, p.tcp.sport, \
           p.inner_ip.daddr, p.tcp.dport
# add the record of the new_flow to FlowTable
after_register = s.copy()
after_register.decisions[new_flow] = p.ip4.daddr
after_register.timestamps[new_flow] = s.curr_time
register_action =
    Action(register_new_flow, {PORT_TO_EXT: p},
           after_register)

return [known_flow_action,
        unknown_flow_action,
```

```
register_action]
```

Summary: Overall, we presented an example of (1) how to specify high-level trace-based middlebox properties, and (2) how to write element-level specifications to make verification modular. We provide a framework for developers to articulate complex trace-based properties. These high-level properties are implementation-independent. Element-level specifications decouple verification problem into two orthogonal problems: that element-level specifications conform to the high-level properties and that elements' implementations comply with their element-level specifications.

3.3 Verifier Implementation

Gravel proves the middlebox properties with two theorems:

THEOREM 1 (Graph Composition). The element-level specifications, when composed using the given Click configuration, complies with the high-level specification of the middlebox.

THEOREM 2 (Element Refinement). The C++ implementation of Click is a refinement of that element's specification. That is, every possible state transition and packet processing action of the C++ implementation must have an equivalent counterpart in the element-level specification.

Theorem 1 verifies that the composition of element-level specifications meets the requirement in the high-level specifications. Theorem 2 verifies that Click's C++ implementation of each element meets its element-level specification.

3.3.1 Graph Composition

Gravel verifies the Graph Composition theorem (Theorem 1) in two steps. First, Gravel symbolically executes the event sequence specified in high-level specifications. Second, Gravel checks whether the high-level specifications hold on the resulting state and outgoing packets from symbolic execution.

Gravel performs symbolic execution on the directed graph. Before the symbolic execution, Gravel creates a symbolic state of the entire middlebox, which is a composition of the symbolic states of all elements in the middlebox. Remember that the high-level specification describes required middlebox behavior on an event sequence. The goal of the symbolic execution is to reproduce the sequence symbolically. For example, if the high-level specification contains an incoming packet, Gravel generates a symbolic packet at the source element of the directed graph. This symbolic packet, when processed by the first element of the graph, can trigger handlers of other downstream elements, which are symbolically executed as well. If the element-level specification contains a branch (e.g., depending on the packet header, a packet can be forward to one of the two downstream elements), Gravel performs symbolic execution in a breadth-first search manner.

After performing symbolic execution for each event type, Gravel records the updated state of each element as well as the packet produced by each output element. Gravel provides this information as the return value of the `handle_*` functions in the high-level specification. Gravel then invokes the functions defined in the high-level specification. Once the `verify` function is invoked, Gravel encodes the high-level specifications into SMT form and uses a solver to see if they always hold.

Loops in the graph. Gravel allows the directed graph of elements to contain loops in order to support bi-directional communications between elements, such as `FlowTable` and `RoundRobinSwitch` in ToyLB (§3.2). However, loops may introduce non-halting execution when we symbolically execute the datagraph. Gravel addresses this issue by setting a limit on the number of elements traversed by the symbolic executor. When the symbolic execution hits this limit, Gravel raises an alert and fails the verification. For example, in ToyLB, the `FlowTable` is hit at most twice: when `FlowTable` cannot find a record for a certain packet, the packet is sent to `RoundRobinSwitch`, which will later send the packet back to `FlowTable`; upon receiving packets from `RoundRobinSwitch`, `FlowTable` records the selected backend server into its own records and does not send the packet back to `RoundRobinSwitch`. Thus, the maximum number of elements traversed during the symbolic

execution is 6, and developers can safely set 6 as the limit for ToyLB.

The graph composition verifier is implemented with 1981 lines of Python. It exposes a similar set of interfaces as Click configuration language so that developers could port existing code into the verifier. The verifier uses the Python binding of Z3 to generate symbolic packets and element states.

3.3.2 Element Refinement

Gravel verifies the Element Refinement theorem (Theorem 2) in two steps. First, a symbolic expression of the element is generated for each event handler's compiled LLVM intermediate representation. Second, Gravel checks if the element's specification holds on the symbolic expression.

Before performing the symbolic execution, Gravel first uses the LLVM library to extract the memory layout of the C++ class of the element, along with the types of each of its member variables. The verifier can later use this information to determine which field is accessed when it encounters memory access in LLVM bytecode. As mentioned in §3.1.3, to bound the symbolic execution step and state size, abstract data structures are executed by using their abstract SMT model instead of actual code. A complete list of the data structures and interfaces replaced is given in §3.3.3.

For packet content access and modification, Gravel's symbolic executor is compatible with Click's `Packet` interface. In the LLVM bytecode, packet content accesses are compiled into memory operations over a memory buffer. To establish the relation between packet header fields and memory offsets, Gravel needs to extract the symbolic header field value for each output packet after the symbolic execution. Gravel first computes offsets for each header field. Note that these offsets are also symbolic values as they depend on the content of other packet fields. After that, Gravel extracts the value of each header field from the memory buffer of the packet. Each extracted value is then encoded into an SMT formula and compared against fields from the abstract packet using an SMT solver. Gravel concludes that the packet and the memory buffer are equivalent when values of

all fields are equivalent.

At the end of symbolic execution, the verifier gets a list of ending states, along with the packets sent out at each output port and the path conditions under which it can be reached. For each entry in the list, Gravel uses Z3 to find an equivalent counterpart in the element specification. If such a counterpart exists for all entries, the refinement of the element is proved.

Gravel's element refinement verifier is implemented in C++ using the LLVM library. The verifier invokes LLVM library's IR parser and reader to load and symbolically execute the compiled LLVM bytecode of each Click element. Besides the SMT encoding of all LLVM instructions used in the compiled Click elements, the verifier also has the SMT encoding of the abstract data types as described in §3.1. The refinement verifier and the symbolic executor consists of 10396 lines of C++.

3.3.3 Gravel Programming Interface

High-level Specification Interface

Table 3.2 gives a list of the interfaces Gravel offers to the developers. The core interfaces of Gravel includes:

- Functions that generates symbolic value (bitvectors) of different sizes (the `sym_*` API).
- Functions that performs graph composition and returns the result of packet or event processing (`handle_*`)
- The `verify` function which informs Gravel's verifier the verification task to perform.

Besides the core interfaces, Gravel also provides a set of helper functions to ease the formalization effort. These functions include functions that access header fields and functions that checks whether two packets are from the same TCP flow. Table 3.2 also lists some examples of helper functions.

Function name	Description
`Core Interfaces:`	
`sym_*() → SymValT`	Create a symbolic value of corresponding type
`handle_packet(s, pkt, in) → pkts, ns`	Handle packet, returns packets and new state
`handle_time(s, ts) → pkts, ns`	Handle time event, returns packets and new state
`verify(formula)`	Encode given formula and verify it
`Helper Functions:`	
`is_none(output) → Bool`	Check if an output is `None`
`payload_eq(p1, p2) → Bool`	Determine if two packets have the same payload
`from_same_flow(p1, p2) → Bool`	Check if two packets are from same TCP flow
`is_tcp(pkt) → Bool`	Check if a packet is TCP packet

Table 3.2: Gravel's specification programming interface.

Modeling Abstract Data Structure

As discussed in §3.3, Gravel masks the actual C++ implementation of several data structures and replace them with an SMT encoding during the symbolic execution in order to generate SMT expressions that could be efficiently reasoned about by SMT solvers. Table 3.3 lists all the interfaces that Gravel's symbolic executor masks during the verification process. This section gives more details on how Gravel generates SMT encoding for these data structure interfaces in a way that the resulting formular can be effciently solved.

Unlike bounded data such as the content of a network packet or an integer field in element state, which can be encoded as a symbolic byte sequence using the bitvector theory of SMT, these data structures have a large state space. This means that encoding them with bitvectors does not results in practically solvable expression. For example, the state of a `HashMap<IPAddress, IPAddress>` could grow up to $2^{64} - 1$ bytes. This sheer size makes it infeasible to be encoded using bitvectors.

Gravel's symbolic executor choose to use a different approach and represents data

structures as a set of uninterpreted functions. In the aforementioned `HashMap` example, Gravel represents the map as two functions:

$$f_{contain} : \{0,1\}^{32} \mapsto \{\bot, \top\}$$
$$f_{value} : \{0,1\}^{32} \mapsto \{0,1\}^{32}$$

$f_{contain}$ maps from the key space $\{0,1\}^{32}$ to boolean space and represents whether certain key is present in the `HashMap`. Similarly, f_{value} represents the mapping between hashmap keys and the corresponding values.

Each of the data structure interfaces is also modeled by Gravel as operations performed on uninterpreted functions. For the `find(K k)` interface of `HashMap`, Gravel first gets the symbolic value representing whether the key is in the map by computing $f_{contain}(k)$. Based on the result, Gravel takes different actions:

$$\text{If } f_{contain}(k) = \top, \texttt{find(k)} = f_{value}(k)$$
$$\text{If } f_{contain}(k) = \bot, \texttt{find(k)} = \bot$$

In the actual implementation, $\bot$ is represented as `HashMap::end()`.

The `intert(K k, V v)` interface performs update on the content of the `HashMap`. In Gravel, this is modeled as creating a new set of uninterpreted functions, $f'_{contain}$ and f'_{value} such that:

$$\begin{aligned}
&\forall k' \in \{0,1\}^{32}\cdot \\
&\quad f'_{contain}(k') = (f_{contain}(k') \vee (k = k')) \\
&\wedge (k \neq k') \Rightarrow f'_{value}(k') = f_{value}(k') \\
&\wedge f'_{value}(k) = v
\end{aligned}$$

Similarly, `erase(K k)` replaces $f_{contain}$ with a new function $f'_{contain}$ such that:

$$\forall k' \in \{0,1\}^{32} \cdot f'_{contain}(k') = f_{contain}(k') \wedge (k \neq k')$$

Besides modeling interfaces from existing Click code base, Gravel also adds a set of iteration interfaces that corresponds to commonly used data structure traverse paradigms. These interfaces could be used to abstract away loops in the Click implementation and making more elements feasible for automated verification.

Gravel currently provides two interfaces for `HashMap`, `map` and `filter`. for `map` interface, Gravel takes as parameter a function g and replace f_{value} with a function f'_{value} where:

$$\forall k \in \{0,1\}^{32} \cdot f'_{value}(k) = g(k, f_{value})$$

Similarly, `filter` takes a predicate p and create a function $f'_{contain}$ such that:

$$\forall k \in \{0,1\}^{32} \cdot f'_{contain}(k) = p(k, f_{value})$$

The modeling of interfaces of `Vector` and `HashSet` are similar to the modeling of `HashMap` mentioned above. The main difference are that `HashSet` only uses $f_{contain}$ function, where as `Vector` uses a symbolic integer to denote the size of the vector and does not have a $f_{contain}$ function.

3.3.4 Trusted Computing Base

The trusted computing base (TCB) of Gravel includes the verifier (used for proving Theorem 1 and Theorem 2), the high-level specifications, the tools it depends on (i.e., the Python interpreter, the LLVM compiler framework, and the Z3 solver), and Click runtime. Note that the specification of each element is not trusted.

3.4 Evaluation

This section aims to answer the following questions:

- How much effort is needed to port existing Click applications? Can Gravel scale to verify the Click applications?
- Can Gravel's verification framework prevent bugs?
- How much run-time overhead does the code modification introduce to middleboxes in order for them to be automatically verifiable by Gravel?

Function name	Description
`Vector<T>:`	
`const T& get(unsigned int)`	Get value by index
`void set(unsigned int i, T v)`	Set i-th value of vector to `v`
`void map(void(*)(T) f)`	Apply function `f` for all value in vector
`HashMap<K, V>:`	
`V &find(K k)`	Lookup by key `k`
`void insert(K k, V v)`	Insert key-value pair `k, v` into the hashmap
`void erase(K k)`	Delete key `k` from the hashmap
`void map(void(*)(K k, V v) f`	Apply function `f` to all key-value pair in hashmap
`void filter(bool(*)(K k, V v) p)`	Filter key-value pairs in the using predicate `p`
`HashSet<T>:`	
`T &find(T v)`	Check if `v` is present in hashset
`void insert(T v)`	Insert `v` into the hashset
`void erase(T v)`	Delete `v` from the hashset
`void filter(bool(*)(T v) p)`	Filter with predicate `p`

Table 3.3: Data structure interfaces supported by Gravel.

3.4.1 Case Studies

To evaluate whether Gravel can work for existing Click applications, we port five Click applications to Gravel. For each application, we choose a set of high-level middlebox-specific properties either by formalizing them directly or extracting them from existing RFCs. We use Gravel to verify that these properties hold. Gravel also verifies the low-level properties, such as memory safety and bounded execution.

MazuNAT: MazuNAT is a NAT that has been used by Mazu Networks. MazuNAT consists of 33 Click elements. MazuNAT forwards traffic between two network address

		LOC	Verif. Time (s)	LOC changed
MazuNAT	Impl	1687	–	133
	Spec (element)	443	64.60	–
	Spec (high-level)	177	3.78	–
Firewall	Impl	1151	–	63
	Spec (element)	73	32.30	–
	Spec (high-level)	70	0.67	–
Load Balancer	Impl	1447	–	63
	Spec (element)	101	10.87	–
	Spec (high-level)	68	1.48	–
Proxy	Impl	953	–	50
	Spec (element)	92	30.63	–
	Spec (high-level)	39	0.72	–
Switch	Impl	594	–	0
	Spec (element)	131	27.73	–
	Spec (high-level)	91	1.61	–

Table 3.4: Development effort and verification time of using Gravel on five Click-based middleboxes.

spaces, the internal network, and the external network. It mainly performs two types of packet rewriting:

1. For a packet whose destination address is the NAT, the NAT rewrites its destination IP address and port with the corresponding endpoint in the internal network.
2. For a packet going from the internal to the external network, NAT assigns an externally visible source IP address and port to the connection. The NAT also needs to keep track of assigned addresses and ports to guarantee persistent address rewriting for packets in the same connection.

One common property we verified for all five middleboxes is that the middlebox does not change the packets' payload:

PROPERTY 3 (Payload Preservation)**.** For any packet that is processed by the middlebox, the middlebox never modifies the payload of the packet.

For NAT-specific properties, we verified that MazuNAT meets the requirements proposed in RFC5382 [62].[3] These requirements are proposed to make NATs transparent to applications running behind them [31].

PROPERTY 4 (Endpoint-Independent Mapping)**.** For packets p_1 and p_2 from the same internal IP, port $(X : x)$, where

- p_1 targets external endpoint $(Y_1 : y_1)$ and gets its source address and port translated to $(X_1' : x_1')$
- p_2 targets external endpoint $(Y_2 : y_2)$ and gets its source address and port translated to $(X_2' : x_2')$

the NAT should guarantee that $(X_1' : x_1') = (X_2' : x_2')$.

PROPERTY 5 (Endpoint-Independent Filtering)**.** Consider external endpoints $(Y_1 : y_1)$ and $(Y_2 : y_2)$. If the NAT allows connections from $(Y_1 : y_1)$, then it should also allow connections from $(Y_2 : y_2)$ to pass through.

PROPERTY 6 (Hairpinning)**.** If the NAT currently maps internal address and port $(X_1 : x_1)$ to $(X_1' : x_1')$, a packet p originated from the internal network whose destination is $(X_1' : x_1')$ should be forwarded to the internal endpoint $(X_1 : x_1)$. Furthermore, the NAT also needs to create an address mapping for p's source address and rewrite its source address accordingly.

These properties are essential to ensure the transparency of the NAT and are required for TCP hole punching in peer-to-peer communications.

---

[3] We omit the set of requirements related to ICMP becuase MazuNAT does not support ICMP.

Middlebox	Bug ID	Description	Can prevent?	Why/Why not?
Load Balancer	bug #12	Packet corruption	✓	high-level specification
	bug #11	Integer underflow	✓	element refinement
	bug #10	Hash function imbalanced	✗	not formalized in specification
	bug #6	throughput imbalanced	✗	not formalized in specification
Firewall	bug #822	Counter value underflow	✓	element refinement
	bug #691	Segfault	✓	element refinement
	bug #1085	Crash	✗	Gravel assumes correct init
NAT	bug #658	Invalid packet can bypass	✓	element refinement
	bug #227	Stale entries	✓	high-level specification
	bug #148	Infinite loop	✓	element refinement

Table 3.5: Bugs from real-world software middleboxes.

We also prove that the MazuNAT preserves the address mapping for a constant amount of time:

PROPERTY 7 (Connection Memorization)**.** If at time t, the NAT forwards a packet from a certain connection c, then for all states s' reachable before time $t + THRESHOLD$, where *THRESHOLD* is a predefined constant value, packets in c are still forwarded to the same destination.

Property 7 guarantees that the NAT can translate the address of all packets from a TCP connection consistently. The constant *THRESHOLD* defines a time window where the TCP connection should be memorized by the NAT. The NAT has the freedom to recycle the resources used for storing connection information after the time window expires.

Load Balancer: Besides the round-robin load balancer mentioned in §3.2, we also verified a load balancer using Maglev's hashing algorithm [28]. Its element graph looks exactly the same as in Figure 3.4. The only difference is that the `RoundRobinSwitch` element is replaced by a hashing element that uses consistent hashing. The load balancer steers packets by

rewriting the destination IP address.

We verified connection persistency for both of the load balancers. The goal of connection persistency is to make load-balancing transparent to the clients.

PROPERTY 8 (Load Balance Persistence)**.** For all packets p_1 and p_2 from connection c, if the load balancer steers p_1 to a backend server, then the load balancer steers p_2 to the same backend server before c is closed.

Stateful Firewall: The stateful firewall is adapted from the firewall example in the Click paper [46]. Besides performing static traffic filtering, it also keeps track of connection states between the internal network and the external network. The firewall updates connection states when processing TCP control packets (e.g., *SYN*, *RST*, and *FIN* packets), and removes records for connections that are finished or disconnected.

We prove that the stateful firewall can prevent packets from unsolicited connections [60]. Also, the firewall should garbage collect finished connections.

PROPERTY 9 (Firewall Blocks Unsolicited Connection)**.** For any connection c, no packet in c from the external network is allowed until a *SYN* packet has been sent out for c.

PROPERTY 10 (Firewall Garbage-collects Records)**.** For any connection c, no packet in c from the external network is allowed after the firewall sees a *FIN* or *RST* packet for c.

Web Proxy: The Web proxy transparently forwards all web requests to a dedicated proxy server. When the middlebox receives a packet, it first identifies if it is a web request by checking the TCP destination port. For web request packets, the proxy rewrites the packet header to redirect them to the proxy server. The proxy also memorizes the sender of the web request to forward the reply messages back to the sender.

We prove that the web proxy middlebox forwards packets in both directions.

PROPERTY 11 (Web Proxy Bi-directional)**.** For any web request packet p with the 5-tuple `(SA,SP,DA,DP,PROTO)`, if the middlebox forwards p to the proxy server and rewrites the 5-tuple to `(SA',SP',DA',DP',PROTO)`, then a packet from the reply flow with 5-tuple `(DA',DP',SA',SP',PROTO)`

should be forwarded back to the sender.

Learning Switch: The Learning switch implements the basic functionality of forwarding Ethernet frames and MAC learning. The switch learns how to send to an Ethernet address A by watching which interface packets with source Ethernet address A arrives. If the switch has not learned how to send to an Ethernet address, it broadcasts the packet to all its interfaces.

We prove the following properties about the switch.

PROPERTY 12 (Forwarding Non-interference)**.** For any Ethernet address A, the behavior of how the switch forwards packets targeting A is not be affected by packets whose source Ethernet address is not A.

PROPERTY 13 (Broadcasting until Learnt)**.** For any address A, if the switch broadcasts packets targeting A, it keeps broadcasting until a packet from A is received by the switch.

3.4.2 Verification Cost

To understand the cost of middlebox verification on Gravel, we evaluate the amount of development effort and the verification time. Table 3.4 shows the result.

Development effort. We find that porting existing Click applications to Gravel requires little effort and that writing specifications with Gravel are also easy. We only modified 133 lines of code in MazuNAT to make it compatible with Gravel. The firewall and load balancer required only 63 lines of code modifications. Our proxy required 50 lines of code to be changed, and the switch requires no modification. Most of the required code changes come from the `IPRewriter` element. We had to remove the priority queue that is used for flow expiration and instead use a linear scan to expire old mappings. Other code changes include removing pointers to other elements in `FTPPortMapper`, replacing `ARPTable` in `ARPQuerier` with hashmaps, and the change of `CheckIPHeader` mentioned in §3.1. The specifications are concise. The high-level specification is below 200 lines of code and the element-level specifications are less than 450 lines of code for all five middleboxes. The

associated developer effort is also small. For the web proxy and learning switch, it took less than one person-day for both the high-level properties and the element specifications. The load balancer and the stateful firewall each required a full day's effort in order to port them to Gravel and verify their correctness. The most complicated middlebox in our case study, MazuNAT, took about 5 person-days to port and specify. Five elements (`Classifier`, `IPClassifier`, `IPRewriter`, `CheckIPHeader`, and `EtherEncap`) are reused across these middleboxes, and thus we reuse their element-level specifications.

Verification time. With Gravel's two-step verification process, Gravel's verifier can efficiently prove that the middlebox applications provide the desired properties. Most of the verification time is spent on proving the equivalence of the C++ implementation of each element and its element-level specification. Verification of the high-level specifications from the element-level specifications took less than 4 seconds for the different applications. Overall, even for MazuNAT, the overall verification time is just over a minute.

3.4.3 Bug prevention

When verifying MazuNAT with Gravel, we found that the original MazuNAT implementation did not possess the endpoint independent mapping property (Property 4). MazuNAT uses a 5-tuple as the key to memorize rewritten flows. This means that when MazuNAT forwards a packet coming from the external network, the packet's source IP address and source port affects the forwarding behavior, violating Property 4. To fix this, we changed the `IPRewriter` element to use only a part of the 5-tuple when memorizing flows.

To evaluate the effectiveness of Gravel at a broader scope, we manually analyze bugs from several open-source middlebox implementations. We wanted to understand whether these bugs can happen if the middlebox is built using Gravel. We examine bug trackers of software middleboxes with similar functionalities as those in our case studies (i.e., NAT, load balancer, firewall) and search the CVE list for related vulnerabilities. We inspect bug reports from the NAT and firewall of the netfilter project [64], and the Balance load balancer [5]. Since the netfilter project contains components other than the NAT and the

firewall, we use the bug tracker's search functionality to find bugs relevant only to its NAT and firewall components. We inspect the most recent 10 bugs for all three kinds of middleboxes and list the result in Table 3.5.

Of the 30 bugs we inspected, we exclude 10 bugs for features that are not supported in our middlebox implementations, 3 bugs related to documentation issues, 5 bugs on command-line interface, and 2 bugs on performance.

From the remaining 10 bugs, Gravel's verifier is able to catch 7 of them. Among these bugs, *Bug #12* in the load balancer and *bug #227* in the NAT can be captured by the verification of the high-level specification as they lead to the violation of Property 3 and Property 7 respectively. Other bugs involving integer underflow or invalid memory access can be captured by the C verifier. Note that there are still three bugs Gravel cannot capture, such as incorrect initialization of the system and properties that are not in our high-level specifications (e.g., unbalanced hashing).

3.4.4 Run-time Performance

To examine the run-time overhead introduced by the code modifications we made, we compare the performance of the middleboxes before and after the code modifications. We run these Click middleboxes on DPDK [24].

Our testbed consists of two machines each with Intel Xeon E5-2680 (12 physical cores, 2.5 GHz), running Linux (v4.4) and has a 40 Gbps Mellanox ConnectX-3 NIC. The two machines are directly connected via a 40 Gbps link. We run the middlebox application with DPDK on one machine and use the other machine as both the client and the server.

The code modification to make these Click applications compatible with Gravel has minimal run-time overhead. We measure the throughput of 5 concurrent TCP connections using *iperf*, and use *NPtcp* for measuring latency (round trip time of a 200-byte TCP message). Table 3.6 shows the results. The code modifications introduce negligible overheads in terms of throughput and latency.

		Throughput (Gbps)	Latency (μs)
NAT	Unverified	37.39 ($\pm$ 0.03)	14.43 ($\pm$ 0.19)
	Gravel	37.41 ($\pm$ 0.04)	15.14 ($\pm$ 0.22)
LB	Unverified	37.38 ($\pm$ 0.04)	14.82 ($\pm$ 0.23)
	Gravel	37.37 ($\pm$ 0.04)	14.86 ($\pm$ 0.20)
Firewall	Unverified	37.37 ($\pm$ 0.05)	15.21 ($\pm$ 0.20)
	Gravel	37.38 ($\pm$ 0.04)	15.11 ($\pm$ 0.24)
Proxy	Unverified	37.36 ($\pm$ 0.05)	14.54 ($\pm$ 0.19)
	Gravel	37.35 ($\pm$ 0.06)	14.35 ($\pm$ 0.18)
Switch	Unverified	37.36 ($\pm$ 0.05)	15.02 ($\pm$ 0.19)
	Gravel	37.39 ($\pm$ 0.07)	14.96 ($\pm$ 0.29)

Table 3.6: Performance of verified middleboxes, compared to their unmodified counterparts.

3.5 Related Work

Middlebox verification. Verifying the correctness of middleboxes is not a new idea. Software dataplane verification [26] uses symbolic execution to catch low-level programming errors in existing Click elements [46]. Our work is also based on Click, but we target high-level middlebox-specific properties, such as load balancer's connection persistency. In addition, we show that 78% of existing Click elements are amenable for automated verification with slight code modifications. VigNAT [90] proves a NAT with a low-level pseudocode specification. Vigor [89] generalizes VigNAT to a broader class of middleboxes and verifies the underlying OS network stack and the packet-processing framework. We believe it is non-trivial to extend VigNAT and Vigor to specify and verify the set of high-level trace-based NAT properties (e.g., hairpinning, endpoint-independence) Gravel can verify.

We note though that specifying the correctness of programs is a fundamentally hard

problem. Gravel chooses to let developers specify high-level specifications on a symbolic trace of packets. We find specifications using Gravel's specification interface to be more abstract than psuedo-code like NAT specification in VigNAT [90]. However, even with Gravel, writing specifications is still hard. For example, specifying the connection persistency property for ToyLB requires the usage of induction (§3.2.3). Empirically, we find that a trace-based specification is flexible enough to express the correctness of middleboxes in the RFCs we examined.

Network verification. In the broader scope of network verification, most existing work [4, 8, 42, 43, 67, 58, 81, 57, 27, 3] targets verifying network-wide objectives (e.g., no routing loop) assuming an abstract network operation model. Gravel, along with other middlebox verification work [26, 90, 89], aims to verify the low-level C++ implementation of a single middlebox's implementation. As switches become programmable [11], researchers have built tools to debug [80], verify [53, 32] P4 programs. Similar to Gravel, this line of work relies heavily on symbolic execution. Our work targets "almost unmodified" middleboxes written in C++.

Currently, Gravel only supports verification of middleboxes implemented with Click. However, since our key observation on Click middleboxes, that the number of operations performed processing each packet is finite and small, may also hold on non-Click middleboxes, we believe that Gravel's verification techniques can also be applied on other middleboxes. For example, the eXpress Data Path (XDP) in the Linux kernel also constrains the packet processing code to be loop-free. It also only allows a limited set of data structures for maintaining global states. These properties make it seem plausible that one could apply Gravel's verification techniques to it.

SMT-based automated verification. Automated software verification using symbolic execution has recently become popular. This technique has been used to successfully verify file systems [77], operating systems [63], and information flow control systems [78]. However, this technique usually requires a complete re-implementation of the target application because of the restricted programming model. We conduct a systematic study

on (§3.1) whether unmodified Click-based software middleboxes can be automatically verified.

3.6 Summary

Verifying middlebox implementations has long been an attractive approach to obtain network reliability. We explore the feasibility of verifying "almost unmodified" software middleboxes. Our empirical study on existing Click-based middleboxes shows that existing Click-based middleboxes, with small modifications, are suitable for automated verification using symbolic execution. Based on this, we have designed and implemented a software middlebox verification framework, Gravel. Gravel allows verifying high-level trace-based middlebox properties of "almost unmodified" Click applications. We ported five Click applications to Gravel. Our evaluation shows that Gravel can avoid bugs found in existing middleboxes with small proof effort. Our evaluation also shows that the modifications required for automated verification incur negligible performance overheads. Gravel's source code is available at https://github.com/Kaiyuan-Zhang/Gravel-public.

Chapter 4

Gallium: Automated Software Middlebox Offloading to Programmable Switches

This chapter presents Gallium, a compiler that automatically partitions and compiles a software middlebox written in C++ to a functionally equivalent middlebox that leverages a programmable switch to achieve higher performance. Gallium provides the following guarantees: (1) functionally equivalence between input program and the generated offloaded implementation, (2) complying the expressiveness and resource constraints of the programmable switch, and (3) providing a run-to-completion semantic for the concurrent execution of packet processing across software middlebox and hardware switch.

We evaluate Gallium on five different middleboxes: MazuNAT, an L4 load balancer, a firewall, a network proxy, and a trojan detector, written using the Click [46] framework in C++ and totaling over 6K lines of code. Our evaluations show that Gallium is able to automatically offload middlebox functionality, save up to 79% processing cycles, reduce latency by up to 31%, and improve throughput by up to 46%.

4.1 Gallium Overview

Gallium is a compiler that transforms an input middlebox program written in a general-purpose programming language (C++) using Click APIs [46] into a functionally equivalent middlebox implementation that has two parts: (1) a P4 program that runs on a pro-

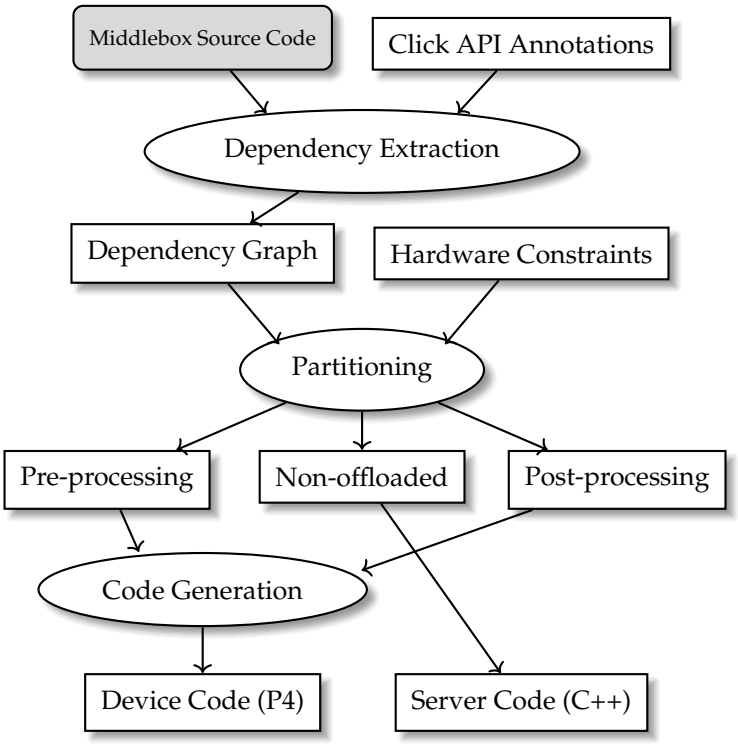


Figure 4.1: Workflow of Gallium. The shaded box is the input program written by middlebox programmers. Other rectangular boxes are annotation and configuration inputs given to Gallium, intermediate representations, and final outputs.

grammable switch, and (2) a C++ program that runs on a traditional middlebox server.

4.1.1 Goals

Gallium has to realize three goals with the generated code.

- *Functional equivalence:* the combined effect of the two parts (i.e., the P4 program and the C++ code) should be functionally equivalent to the input middlebox program.
- *Conform to constraints:* the generated P4 program should satisfy the expressiveness and resource constraints of programmable switches.
- *Concurrency-safe execution:* the output middlebox must provide per-packet run-to-completion semantics even though the code on the switch executes concurrently with the server program.

We now elaborate on the third goal. Runtime systems for middlebox dataplanes typically guarantee the run-to-completion semantics for packets, wherein all protocol processing for a packet is done before moving on to the next packet. In our setting, where we are concurrently executing portions of the middlebox logic on the switch and the server, we run the risk of a later packet observing none or only a subset of the state updates made by a previous packet.

We, therefore, formalize the desired *run-to-completion* semantics. If a packet p_j is received by the middlebox[1] after p_i and if p_j is causally dependent on p_i's receipt (e.g., p_i is a SYN packet and p_j is the corresponding SYN-ACK packet), then we require p_j to observe all of the state updates made by p_i. If p_j is not causally dependent on p_i's receipt (i.e., there is no end-to-end constraint that the middlebox has processed p_i before p_j), then p_j should observe either all or none of the state updates made by p_i. Note that, if the later packet p_j observes none of the state updates, then the resulting execution is considered to be equivalent to the setting where p_i was delayed in the network and received after p_j.

The correctness criteria described above embody both a condition on handling causal dependencies (i.e., packets observe the state updates of other packets that causally preceded them) and an atomicity requirement (i.e., packets either observe all or none of the state updates of previous packets). It is worth noting that this is similar to the correctness requirements imposed on multi-threaded middlebox programs, wherein in-flight packets are processed atomically by the middlebox in any order.

4.1.2 Compilation Process and Execution

We outline below how Gallium is designed to meet the desired goals. We overview the partitioning and compilation process and then describe how the runtime system ensures the correct execution of the generated code. Figure 4.1 shows the workflow of Gallium.

To ensure functional-equivalence, we need to keep the instruction-level dependencies

[1] In the context of Gallium, a middlebox refers to the aggregated entity comprising of a server and a switch.

equivalent between the source program and the generated program. (See §4.2.1.) Let us consider a code statement S in the source program, and let Gallium assign this statement to one of two partitions (i.e., server code or switch program). We need to ensure that S has the same effect in the generated middlebox as in the input middlebox. This requirement means that, before S runs, the program state that S can access should be the same in the input and the generated program. To capture this notion, we define a "depends on" relation that combines both data and control dependencies that exist in the program. A statement S_2 depends on S_1, if and only if the observable variables in S_2 are affected by S_1's execution. Any time two statements access the same state, with at least one of them being a write, or if a statement determines whether or not some other statement will be executed, we will tag a dependency between the two statements. In the generated program, we ensure that, for any statement S, all statements that S depends on have been executed before S.

The second step is to partition the input program's source code into the desired three partitions (i.e., a pre-processing partition, a non-offloaded partition, and a post-processing partition) while maintaining the dependencies. (See §4.2.2.) At the same time, Gallium must ensure that the generated P4 program, which contains the pre- and post-processing partitions, conforms to the expressiveness and the hardware resource constraints.

We develop a label-removing algorithm that partitions the source program and preserves dependencies while meeting the switch constraints. Our algorithm first finds a partitioning that maximizes the number of statements offloaded to the programmable switch by only considering the restrictions imposed by the program dependencies and P4's expressiveness constraints. Our algorithm then refines the partitioning result by gradually moving statements from the pre- and post-processing partition to the non-offloaded partition until it satisfies all the resource constraints.

The final step is to generate code that provides the desired concurrency semantics. (See §4.2.3.) The partitioning process assumes that the middlebox state is synchronously replicated across the switch and the server and that the packets are processed one at a

time. The code generation step, along with the runtime system, ensures that the generated program provides a per-packet run-to-completion semantics even as the switch and the server concurrently consume packets. Crucially, Gallium identifies what program state has to be replicated and includes distinct mechanisms for handling replicated and non-replicated state in order to provide run-to-completion semantics.

For state replicated across the programmable switch and the middlebox server, the concurrent processing of packets should exhibit behaviors that would have been obtained if the packets had been processed sequentially. As a concrete example, consider a NAT that maintains a bidirectional address mapping using two connection tables: one which maps an internal address and port to an externally visible port, and the other which maps the externally visible port to an internal address and port. When the NAT receives a SYN packet from a new TCP connection, it updates both of these connection tables to handle subsequent packets from both sides properly. If this computation is performed on the server, then the updates would have to be consistently and atomically replicated on the switch. Therefore, Gallium provides efficient mechanisms for state synchronization.

Gallium also provides mechanisms to communicate non-replicated state between the server and the switch. Because the non-offloaded partition may require additional information from the pre-processing partition (e.g., a temporary variable computed by the pre-processing partition), Gallium has to synthesize a packet format where the additional information can be delivered using the packet header. A similar mechanism also has to exist for delivering information from the non-offloaded partition to the post-processing partition. Gallium uses static analysis to identify the set of variables that need to be transferred and allocates packet header fields to store these variables. Finally, Gallium maps all the data structures and instructions to their P4 counterparts, e.g., from a HashMap lookup to a P4 table lookup. Gallium requires a middlebox developer to annotate the maximum size for each data structure stored in the programmable switch.

At the end of the compilation, Gallium outputs: (1) a deployable P4 program that contains both the pre- and post-processing partitions; and (2) a C++ server program that

corresponds to the non-offloaded partition.

4.2 Design

This section describes the details of dependency extraction, partitioning, code generation, and runtime execution. We use a simple load balancer, `MiniLB`, as a running example. `MiniLB` uses consistent hashing over the source and destination IP addresses to assign incoming TCP connections to a list of server backends. `MiniLB` steers packets by rewriting the destination IP address of the packet. To ensure that packets in a given connection are sent to the same backend server even when the list of backends changes, `MiniLB` stores the mapping from existing connections to backends and steers packets using this mapping. For simplicity, `MiniLB` does not garbage collect completed connections. `MiniLB` contains a single Click element, and its C++ source code is shown below.

```
class MiniLB {
  HashMap<uint16_t, uint32_t> map;
  Vector<uint32_t> backends;
  void process(Packet *pkt) {
    iphdr *ip = pkt->network_header();
    uint32_t hash32 = ip->saddr ^ ip->daddr;
    uint16_t key = (uint16_t)(hash32 & 0xFFFF);
    uint32_t *bk_addr = map.find(&key);
    if (bk_addr != NULL) {
      ip_hdr->daddr = *bk_addr;
      pkt->send();
    } else {
      uint32_t idx = hash32 % backends.size();
      uint32_t bk_addr = backends[idx];
      ip_hdr->daddr = bk_addr;
      map.insert(&key, &bk_addr);
      pkt->send();
    }
  }
```

```
};
```

4.2.1 Dependency Extraction

The first step is to extract the statement-level dependencies in the source program. When we create the partitions (pre-processing, non-offloaded code, post-processing), we want to move as many statements as possible to the pre-processing and post-processing partitions to maximize offloading. The statement-level dependencies determine whether it is possible to move a particular statement to other locations in the source program that are conducive to offloading.

We define the dependency relation "S_2 depends on S_1" to represent the constraint that "S_2" must run after "S_1". This dependency could exist due to one of many reasons, e.g., both statements write to the same memory location. To formally define the dependency condition, we first define a "can happen after" relation. "S_2 can happen after S_1" means that, for all possible program executions of the input program, there is at least one execution trace where S_2 is performed after S_1. This "can happen after" relation denotes a possible dependency of S_2 on S_1. On the contrary, if S_2 cannot happen after S_1, it is impossible for S_2 to depend on S_1.

Extracting "can happen after" relations is straightforward. Gallium builds a control-flow graph of the source program. Whether S_2 can happen after S_1 is simply whether S_2 is reachable from S_1 in the control-flow graph.

After we have extracted all the "can happen after" relations, we need to pick the real dependencies inside this set. There are three types of dependencies that we consider, as in a program dependence graph [30].

- Data Dependency: S_1 modifies the state that S_2 reads from or writes to (i.e., read after write and write after write).
- Reverse Data Dependency: S_1 reads some variable or state modified by S_2 (i.e., write after read).

- Control Dependency: S_1 modifies a condition variable used to determine whether S_2 should be executed.

Note that, for programs with loops, S "can happen after" itself and, thus, can also "depend" on itself.

When performing the dependency analysis, Gallium needs to understand what variables and data structures a program statement might access. Specifically, Gallium has to identify a read set, including all the locations a statement may read, and a write set, including all the locations a statement may modify. For simple operations, the source code itself contains the information. For operations invoked through abstract data structure APIs, we need to know the state accessed for each API invocation. This knowledge comes from annotations on the APIs.

In Gallium, we require annotations for both data structure APIs (such as `HashMap` and `Vector`) and APIs used to access packet headers. In particular, we need two types of annotations for the Click APIs: (a) the data read and modified when calling into the API and (b) if the API returns a pointer, the data referred to by the pointer.

In `MiniLB`, we have the following annotations:

- The methods `network_header()` and `transport_header()` return pointers to the IP and TCP headers of the packet, respectively. Further, a read/write using the returned pointers is also a read/write of the corresponding headers.
- The method `HashMap::find()` performs reads on both the input parameter (e.g., `key` pointer) and the `HashMap` data structure (e.g., `map`).
- The `[]` operator of `Vector` class reads the parameter (e.g., `idx`) and the `Vector` (e.g., `backends`).
- The method `HashMap::insert()` reads the two input parameters (e.g., `key` and `bk_addr`) and modifies the HashMap (e.g., `map`).

With these annotations, Gallium constructs the read and write sets for each statement. For statements without method calls, Gallium directly constructs the read and write sets

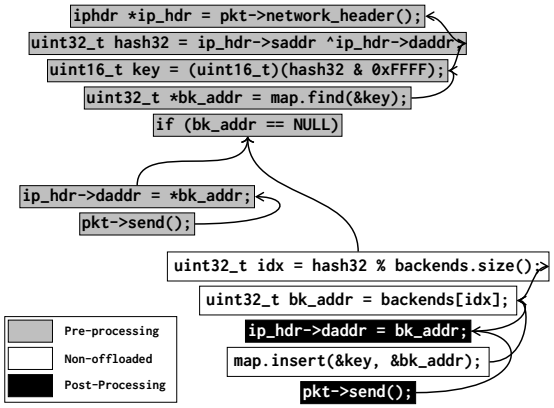
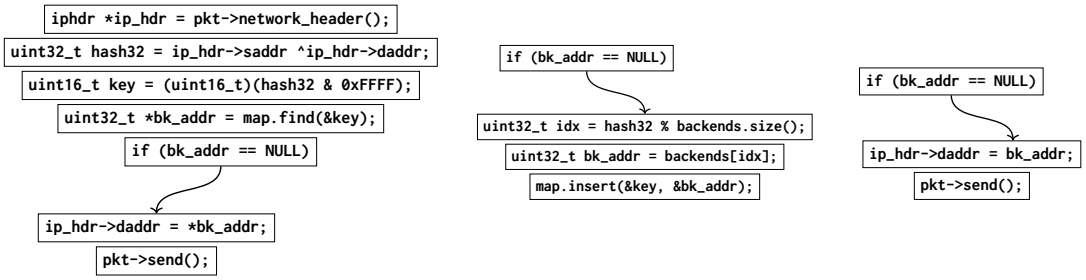


Figure 4.2: The dependency graph of `MiniLB` with partitions.

because the variables are explicit. For a Click API invocation, Gallium uses the corresponding annotation to build the read and write set. When the source program uses a pointer dereference, Gallium performs pointer analysis to determine the variable referred to by the pointer. For example, in `MiniLB`, when dereferencing the pointer `ip_hdr`, Gallium traces the origin of the pointer and uses the annotation of `network_header()` API to determine that this is an access to the packet's IP header. Gallium inlines all other function calls before constructing the read and write sets.

Finally, Gallium builds a directed dependency graph from the per-instruction read and write sets. Vertices in the graph are statements in the program, and edges denote the dependencies between statements. For each pair S_1 and S_2, Gallium creates an edge from S_1 to S_2 if S_2 depends on S_1 by checking whether one of the three dependency conditions hold. Figure 4.2 is the extracted dependency graph for `MiniLB`.

In our examples, we represent the dependency graph using statements in C++. Our implementation, however, creates the dependency graph on LLVM Intermediate Representation (IR) of the source code because LLVM's syntax is simpler than C++. We also ensure that a statement in the LLVM IR can be mapped to a corresponding switch pipeline statement if the programmable switch supports the execution of the operations performed in the statement. This step is relatively straightforward, given that our switch target supports only a limited number of operations, all of which are available as primitives in the



(a) pre-proccessing (b) non-offloaded (c) post-processing

Figure 4.3: Control-flow graphs for pre-processing, non-offloaded, and post-processing partitions of `MiniLB`.

LLVM IR. As we expand Gallium to target other, more expressive execution platforms, we might need more flexible intermediate representations.

4.2.2 Partitioning

Given the dependency information, Gallium then partitions the middlebox program into three pieces, pre-processing, non-offloaded, and post-processing segments, each corresponding to a packet processing step.

There are two problems to solve in partitioning the source program: (1) we need to consider the expressiveness of P4, and (2) we need to consider resource constraints. The latter includes issues such as how much memory is consumed by the pre- and post-processing partitions, how much packet header space we need to transfer information between programmable switches and servers, and can pre- and post-processing partitions fit onto the limited number of processing pipeline stages in the programmable switch.

We choose to deal with these two problems separately. The first step is to determine how to partition the program if we have an unbounded amount of resources (e.g., switch memory, switch processing pipelines, and packet header space). In this step, we only consider P4's expressiveness and the source program's dependencies. The result of the first step is three partitions where we put as many statements as possible in the pre- and post-processing partitions, while maintaining functional equivalence to the source

program. For the next step, we refine the partition by carefully moving statements from the pre- and post-processing partitions to the non-offloaded partition so that the final partitions meet the resource constraints.

Assigning Execution Labels to Statements

We use a label-removing algorithm to solve the first problem. The basic idea is to assign a set of labels (pre, non_off, $post$) to each statement to denote whether a statement can belong to a specific partition. We begin with each statement having all possible labels and then gradually remove labels based on a set of label-removing rules that use the computed dependencies.

We define $L(S)$ to be the set of labels for S. We initialize labels in the following way:

$$L(S) = \begin{cases} \{pre, post, non\_off\} & \text{if } S \text{ is supported by P4} \\ \{non\_off\} & \text{otherwise} \end{cases}$$

This initialization means that any statement not supported by P4 should be in the non-offloaded partition. A statement S is supported by P4 if and only if these conditions hold:

1. S involves only those operations that P4 supports, e.g., integer addition, subtraction, and comparison.
2. S's access of the packet, if any, is only to the packet header fields and not packet payloads.
3. S is a Click API invocation with a P4 implementation, e.g., a hash table lookup that can be replaced with a P4 table lookup.

Gallium applies a set of label-removing rules over all the statements repetitively. The updating rules are given as constraints between the labels of statement S and all its neighboring statements in the statement dependency graph. Here, we use $S_1 \rightsquigarrow S_2$ to denote the fact that "S_2 depends on S_1", and $\rightsquigarrow^*$ to denote the transitive closure of $\rightsquigarrow$.

Gallium uses the following label-removing rules during the labeling process:

$$(1) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge \mathit{post} \notin L(S)) \Rightarrow \mathit{post} \notin L(S')$$

$$(2) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge \mathit{pre} \notin L(S')) \Rightarrow \mathit{pre} \notin L(S)$$

$$(3) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge S, S' \textit{ access same global state} \wedge \mathit{pre} \in L(S')) \Rightarrow \mathit{pre} \notin L(S)$$

$$(4) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge S, S' \textit{ access same global state} \wedge \mathit{post} \in L(S)) \Rightarrow \mathit{post} \notin L(S')$$

$$(5) \quad \forall S, S \rightsquigarrow^* S \Rightarrow L(S) = \{\mathit{non\_off}\}$$

The first two rules ensure that the partitions are consistent with the dependencies. The third and fourth rules are required due to the P4 language's limitation that a global state can only be accessed once inside the switch pipeline. In particular, if there is some global state accessed in multiple different program locations, then at most one of the statements can be executed on the switch.[2] The last rule prevents the offloading of statements that appear in a loop body as P4 does not support loops. Gallium repeatedly applies these rules to eliminate labels until no label can be further removed. This algorithm always converges as the total number of labels monotonically decreases and terminates when the constraints are satisfied for every pair of statements in the dependency graph.

Satisfying Resource Constraints

The labels assigned to each statement indicate the partitions a statement can be assigned to, given the dependencies but without considering the resource constraints. We consider several types of resource constraints. We notice that there are two types of state a middlebox needs to store: per-packet state and global state. The per-packet state is pieces of information whose lifetimes only last during the processing of a single packet, such as the `hash32` variable in `MiniLB`. The global state has to be maintained across all packets, such as the `map` variable in `MiniLB`.

[2]The third and fourth rules can be relaxed if the switch supports a disaggregated RMT architecture [16].

We need to enforce the following resource constraints:

- *Constraint 1:* The total size of the global state maintained by the switch does not exceed the size of the switch memory.
- *Constraint 2:* The length of the longest dependency chain in the offloaded code cannot exceed the switch's pipeline depth.[3]
- *Constraint 3:* Each element of the global state maintained on the switch can only be accessed once during packet processing.
- *Constraint 4:* The total size of the per-packet state does not exceed the maximum allowed per-packet metadata.
- *Constraint 5:* The additional per-packet information transferred between the server and the switch is bounded.

The first, second, and fourth constraints are due to the limited memory and computational resources on a programmable switch. The third one reflects the limited expressiveness of the pipeline architecture in traditional P4 devices; match action tables can only be accessed once during packet processing to ensure efficient packet processing at line rate. The last one is because the Ethernet frame size is limited, and we want to use a large fraction of the frame size to deliver actual packet content. We set this constraint to be 20 bytes.

We can meet all of the five constraints by moving more of the code to the non-offloaded partition. (Note that executing all of the code on the server trivially satisfies the constraints.) Gallium first deals with Constraints 1, 2, and 3. Because once Constraints 1, 2, and 3 are met, moving more code to the non-offloaded partition never violates them again. For (1) and (2), Gallium first computes the following dependency distance metric of various statements from both the beginning and the end of the program. The dependency distance between two program points is the length of the longest dependency chain

[3]We use a conservative constraint on the length, as the actual number of operations that could be performed on the switch is embedded in the P4 compiler and is not publicly available. We choose a conservative value based on empirical experiments.

connecting the two points. Given a programmable pipeline with k stages, it removes "pre" labels from all statements that are at a dependency distance of greater than k from the program's entry point and "post" labels from all statements that are at a dependency distance of greater than k from the program's exit point. This transformation helps us satisfy Constraint 2. Gallium then gradually removes "pre" labels from statements in the reverse source program order and "post" labels from statements in the source program order until Constraint 1 is satisfied.

For Constraint 3, Gallium uses an exhaustive search to find the placement that maximizes the number of statements on the programmable switch. For each global state, Gallium first locates all the accesses to the state that are labeled with *pre* (or *post*). It then enumerates all possible placements of those accesses where only one of them is executed on the switch. Gallium computes the number of statements that could be put on a programmable switch in each case and choose the placement with the maximum number of statements.

Gallium then moves more code to the non-offloaded partition, performing a variable liveness test after each removal to determine the amount of program state that needs to be transferred across partition boundaries, and repeats this until the partitioned code satisfies Constraints 4 and 5. Each time a statement is moved, Gallium runs the label-removing algorithm to ensure that the dependency constraints are met.

However, note that moving code from the offloaded partition does not always reduce the data that has to be transferred to the non-offloaded partition. For example, moving an integer addition from the switch to the server requires the offloaded code to send two integers to the middlebox server instead of one). Finding the minimal set of code that, when moved to the server, could satisfy Constraint 5, requires enumerating all possible combinations of code movements. Gallium chooses to use a greedy approach: It tries to move code to the non-offloaded partition based on a fixed topological order of the data dependencies. This heuristic will give us a sub-optimal result when there are multiple branches in the offloaded code. For instance, the greedy approach only checks the second

branch after the first branch is all removed, instead of considering all possible orders of moving code fragments. Nevertheless, it only requires one linear scan of the offloaded code, and therefore, it can find a code partition that satisfies Constraints 4 and 5 in a reasonable amount of time.

Gallium assigns a partition for each statement by looking at the labels. When a statement has both "pre" and "post" labels, Gallium assigns the statement to the pre-processing partition. When a statement has the "post" label but not the "pre" label, Gallium assigns it to the post-processing partition. The rest of the code is assigned to the non-offloaded partition. Gallium then splits the input program's control flow graph (CFG) into separate CFGs that correspond to the three partitions. Figure 4.3 shows the resulting CFGs for `MiniLB`.

4.2.3 Code Generation and Runtime Execution

We now describe the code generation process and the mechanisms used for communicating state between the switch and the server. First, we explain how we transform the CFGs of the pre- and post-processing partitions to a P4 program. For the non-offloaded partition, Gallium needs to convert the corresponding CFG back to C++. Transforming from a CFG to C++ is easy because C++ is very expressive, and we omit the details for this transformation. Second, we describe how the temporary state can be communicated in-band by customizing the packet format used between the switch and the server. Third, we detail the runtime mechanisms for synchronizing global state and how we can provide run-to-completion semantics for concurrent packet processing.

Mapping CFG to P4

We now describe how we map a CFG to a P4 program. Figure 4.5 shows how we map different states and instructions in the CFG to corresponding P4 primitives.

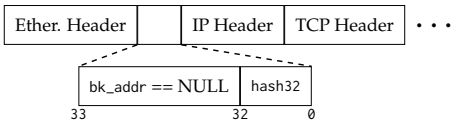
The per-packet state, such as temporary variables, are mapped to metadata fields in the scratchpad memory. Since the amount of metadata that can be allocated is less than 100 bytes to conserve on scratchpad memory, Gallium records when temporary variables

are first and last used. Gallium reuses the memory consumed by variables that are no longer useful. In `MiniLB`, `key` and `bk_addr` are temporary variables.

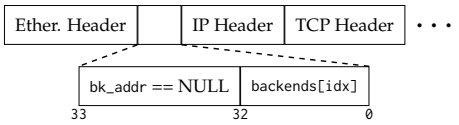
For the global state, Gallium chooses different representations based on the access pattern. States that are accessed exclusively by the switch will be maintained on the switch if there is a P4 realization of the state. Gallium supports two types of global state on the programmable switch: global variables and maps. Gallium maps these two types of switch state into P4 match-action tables and P4 registers, respectively. Note that Gallium choose to use P4 registers for global variables only if the switch alone accesses the state. Similarly, states that are exclusively accessed by the server are represented with the same data structure as the original program. For the program state accessed by both the switch and the server, Gallium maintains a copy of the state on both devices. State replication enables faster access but complicates updates, as we will discuss later. In `MiniLB`, `map` is a HashMap in the input middlebox, and it is offloaded as a P4 match-action table. Because a C++ `HashMap` can be unbounded in size, but programmable switches have a limited amount of memory, Gallium requires a middlebox developer to annotate a maximum size for each `HashMap` that the developer wishes to offload.

Once states are mapped to their P4 counterparts, Gallium start to maps each instruction to P4. Branches, packet header accesses, and ALU operations are directly mapped to their P4 counterparts. Map lookups in the input middlebox program are mapped to P4 match-action table lookups. In `MiniLB`, `map.find(&key)` is mapped to a P4 table lookup.

We also combine the pre- and post-processing partitions into a single P4 program. Gallium includes all the match-action table and register definitions from the two partitions. The instructions from the two partitions are also placed in the combined P4 program. To determine which partition should execute when receiving a packet, Gallium creates a match-action table that matches on the ingress interface of the packet at the beginning of the processing pipeline. If the packet is coming from the interface connected to the middlebox server, Gallium invokes the post-processing partition. Otherwise, the pre-processing partition handles the packet.



(a) Packet from the pre-processing to the offloaded partition



(b) Packet from the non-offloaded to the post-processing partition

Figure 4.4: The packet format used between programmable switch and middlebox server in `MiniLB`.

Communicating temporary state between the switch and the server

Some temporary state has to be communicated between the switch and the server. Gallium transmits this state along with the packet data. We determine the packet format for packets going from the pre-processing partition to the non-offloaded partition and from the non-offloaded partition to the post-processing partition. Gallium does a variable liveness test on the partition boundary to decide what variables need to be transferred across partition boundaries. [4] Gallium allocates space in the packet header to store these variables and generates the corresponding packet header parser specification in P4.

We insert these additional packet header fields between the Ethernet header and the IP header. The Ethernet header is expected by the receiving NIC and ensures that the packet is delivered over the wire to the middlebox server's NIC. We don't need to put the additional header after the IP header because we expect the programmable switch to have a direct connection to the middlebox server; thus, a destination Ethernet address is sufficient for routing. To accommodate this additional header, we use a slightly larger maximum transmission unit (MTU) for the link between the programmable switch and the middlebox server compared with the rest of the network.

Figure 4.4 shows the packet format for `MiniLB`. The condition for branching, `bk_addr==NULL`, has to be shared across all three partitions. Gallium allocates 1 bit in the packet header for

[4] In the partitioning step, Gallium has already verified that the additional packet header space needed for transferring these variables is below 20 bytes.

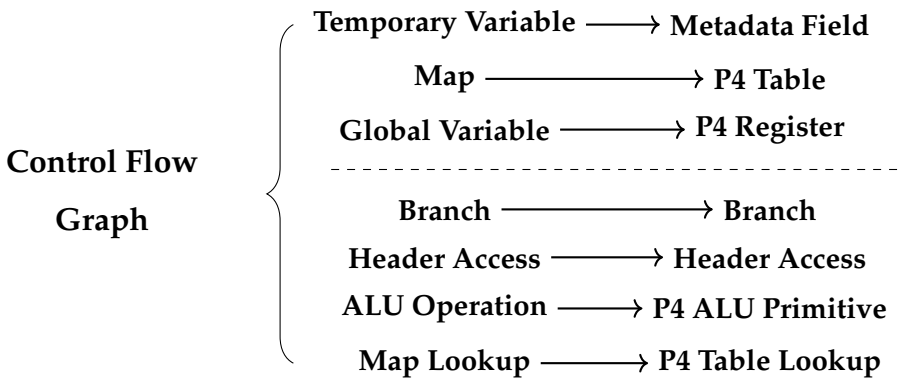


Figure 4.5: Mapping a control-flow graph's states and instructions to their P4 counterparts.

this information. Also, `hash32` is communicated from the pre-processing partition to the non-offloaded partition. `backends[idx]` is communicated from the non-offloaded partition to the post-processing partition. Gallium allocates 32 bits in the packet header for storing these variables.

Synchronizing global state

We now describe the state synchronization techniques required to provide the desired run-to-completion semantics. Recall that this execution semantics requires a packet's state updates to be atomic, i.e., other packets observe all or none of the state updates performed by a packet. Further, a packet is required to observe all of the state updates performed by causally antecedent packets.

It is worth noting some aspects of the desired run-to-completion semantics, which we had described earlier in §4.1. The correctness criteria does not impose execution restrictions on packets that are traversing the network at about the same time, since those packets could be processed by a middlebox server in any order. Instead, a packet P's causally antecedent packets are precisely those packets that have already been received by one of the end-hosts before P is transmitted. We just need to ensure that the state updates of these causally antecedent packets have been consistently replicated across the

switch and the server.

Further, the correctness criteria does not provide ordering guarantees. That is, packet P that is not causally dependent on packet Q might observe the state changes performed by packet Q, but might still be transmitted by the middlebox (i.e., the server-switch pair) before packet Q. We believe that this is not unreasonable as this is equivalent to packet reordering introduced on multi-threaded middlebox implementations or even by the network itself.

Distributed state management: We first note that our partitioning and code generation techniques ensure that, when a global variable or map is replicated across the switch and the server, any updates will only be made by the server. Given this placement and access restriction, the required run-to-completion semantics is equivalent to supporting transactional (or serializable) operations on the state, with the switch operations being limited to read-only accesses. Given this observation, we borrow two techniques from the distributed systems literature on primary-backup systems to enforce the desired semantics. *Atomic update* ensures that the server updates on the shared state are atomically reflected on the switch. *Output commit* ensures that the packet causing the updates is buffered on the server until the updates are reflected on the switch. When used together, the techniques will guarantee the desired run-to-completion semantics.

Atomic update: We now describe the implementation of the atomic update in Gallium. Similar to journaling in file systems, Gallium's runtime system first puts all the modifications into a dedicated switch memory and then uses a single (atomic) write operation to make them visible to subsequent packets. For each match table stored on the programmable switch, a smaller-sized "write-back" table is created. Besides that, a single bit is also added to the switch state, indicating whether the write-back table should be used during table lookup. When the P4 program performs a lookup to the table and observes that the bit is set to true, it first reads the write-back table. If there is a matching entry, it will be used as the result of the table lookup. Otherwise, the main match table

will be used. The middlebox server performs switch state updates in three steps. First, the server uses the switch control plane API to insert entries to the write-back tables. (A special value indicates table entry deletion.) Then, the server flips the bit by performing an additional control plane operation. This operation makes the entries in the write-back tables visible to subsequent packets. Finally, the middlebox server writes the updates to the main tables and toggles the bit after all updates are performed.

Analysis: We now discuss the correctness and performance implications of the scheme described above. The combined use of transactional (or atomic) updates and the output commit protocols ensure that the switch-server pair exhibits the same consistency properties as a chain-replicated, primary-backup system. Switch operations are restricted to read-only operations on replicated state, just as with the tail of a chain-replicated system [71]. A packet that performs updates to replicated state is only released after the switch has performed the updates. Subsequent packets generated after an end-host has received would see the updated state even when processed on the switch.

A Gallium middlebox could reorder packets sent through it, and this reordering could result in performance issues for TCP flows. Consider, for instance, a packet that is forwarded to the server for slow-path processing. If it were to make updates to the replicated state, then it would be buffered until the updates are propagated to the switch, and subsequent packets could be processed and transmitted by the switch before the server releases the slow-path packet. Fortunately, for most middleboxes, the slow-path processing is often invoked on a small number of packets or just control packets, such as SYN, SYN-ACK, FIN, and RST packets, and this reduces the occurrence of reordered data packets. It is also worth noting that features providing dataplane management of the traffic manager, expected in upcoming Tofino switches [74], can help eliminate these reordering issues. With this hardware support, flows with outstanding packets requiring slow-path processing can be buffered on separate queues and then released by the packet transmitted by the server. We leave the exploration of this mechanism for future work.

4.3 Implementation

We implement Gallium using 5712 lines of C++. Gallium uses *Clang* (version 6.00) to generate an LLVM Intermediate Representation (LLVM IR) of the input program. All the functionalities described in §4.2 are implemented as analysis passes on the LLVM IR as it has a simpler syntax than C++. Also, because LLVM IR itself is in a Static Single Assignment (SSA) form, it eases the tracking of when variables are assigned and used. We use the *llvm-dev* library to extract a CFG of the input program. The generated P4 code is compiled and deployed using the SDK provided by the Barefoot Tofino switch. The non-offloaded partition (C++) is compiled and linked with the DPDK library [24] and is deployed as a DPDK application.

As we mentioned in §4.2, Gallium has a set of annotations on Click APIs in order to perform dependency extraction. We have manually annotated the Click APIs to access data structures, including `Vector` and `HashMap`, and the APIs to access packet headers. Gallium models the read and write set for each LLVM instruction. For ALU instructions, the read set of the instruction consists of all the instruction operands, and the write set is simply the destination register. For memory access instructions—load/store—the read/write set of the instruction is the data the pointer points to. Gallium leverages LLVM IR's type metadata to determine the type and size of the dereferenced pointer.

4.4 Evaluation

We aim to answer the following questions in this section:

- Can Gallium enable automated software middlebox offloading to programmable switches?

- How much performance benefits do the offloaded middleboxes provide?

Middlebox	Input (C++)	Output (P4)	Output (C++)
MazuNAT	1687	516	579
Load Balancer	1447	522	602
Firewall	1151	506	403
Proxy	953	292	279
Trojan Detector	882	571	418

Table 4.1: Comparison of lines of code for Click-based middleboxes before and after Gallium compiles them.

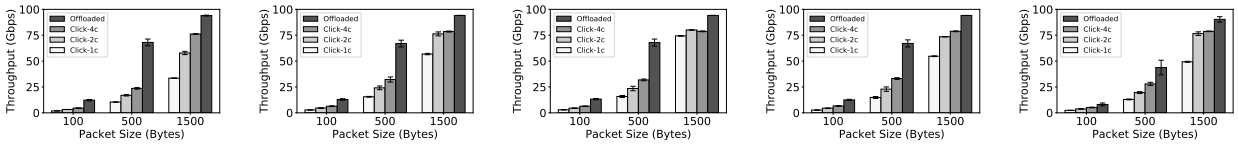
4.4.1 Case Study

We use five Click-based middleboxes to evaluate Gallium: (1) MazuNAT, (2) an L4 load balancer, (3) a firewall, (4) a transparent proxy, and (5) a Trojan detector.

MazuNAT. MazuNAT is a NAT implementation used by Mazu networks. At a high level, MazuNAT is a gateway middlebox that separates two network spaces, an internal network and an external network. For traffic going from the internal to the external network, MazuNAT allocates a new port and rewrites the packet header, and the flow itself appears to be sourced by MazuNAT. The port allocation is performed using a monotonically increasing counter. MazuNAT memorizes the mapping from addresses to ports for existing connections and enforces the mapping for subsequent packets of existing connections.

When MazuNAT receives a packet from the external network, MazuNAT checks if there is a corresponding mapping created by connections from the internal network. If not, MazuNAT drops the packets from the external network. If a corresponding mapping is found, MazuNAT rewrites the packet according to the mapping and forwards it into the internal network to reach its destination.

L4 Load Balancer. The load balancer application is similar to the `MiniLB` example show



(a) MazuNAT (b) L4 Load Balancer (c) Firewall (d) Proxy (e) Trojan Detector

Figure 4.6: Throughput comparison between Gallium middleboxes and their FastClick counterparts. Gallium middleboxes only use a single core in the middlebox server. FastClick versions of the middleboxes use 1, 2, and 4 cores, respectively. Error bars denote standard deviations.

in §4.2. Similar to `MiniLB`, the load balancer assigns incoming TCP and UDP traffic to a list of backend servers. It uses the hash value of the five-tuple (i.e., source IP address/port, destination IP address/port, transport protocol) to determine the backend server to which the connection is assigned. It also uses a map to keep track of the assigned flows to ensure that packets in the same connection are steered to the same backend server. In addition to the functionalities in `MiniLB`, the L4 load balancer garbage-collects finished connections by intercepting TCP control packets, such as RST (reset) and FIN (finish). The L4 load balancer also has a time-out mechanism: idle connections are garbage-collected after 5 minutes without the FIN packet.

Firewall. The firewall is adapted from an example middlebox in the Click paper [46]. It filters packets using a whitelist mechanism. Each entry specifies a five-tuple that is allowed to go through the firewall. When a packet arrives, it is dropped if its five-tuple cannot be found in the whitelist.

Transparent Proxy. The transparent proxy is also adapted from an example in the Click paper [46]. The transparent proxy redirects traffic to a web proxy based on the TCP destination port. The proxy internally keeps a list of TCP destination ports. Upon receiving a packet, the proxy checks whether the TCP destination port is in the list. If the destination port is in the list, instead of forwarding the packet, the proxy rewrites the

packet header to steer the packet to a designated web proxy.

Trojan Detector. The Trojan detector [25] keeps track of TCP connection states of each endhost. It identifies an endhost as a Trojan if the following sequence of events is observed: (1) The endhost first creates an SSH connection. (2) It then downloads a HTML file from a web server, or a `.zip` or `.exe` file from a FTP server. (3) Finally, it generates Internet Relay Chat (IRC) traffic.

4.4.2 What's offloaded?

To evaluate how much middlebox functionality can be offloaded, we examine the lines of code before and after the five middleboxes are compiled by Gallium. Table 4.1 shows the result. Here, the total lines of code do not include Click data structure implementations. After compilation, the generated code (i.e., the combination of the P4 and C++ code) has fewer instructions than the input program, as P4 abstracts away several types of packet processing. For example, the `IPClassifier` element, which performs generic packet classification, could be abstracted using a single match action table in P4.

After compilation, MazuNAT's address translation tables—the maps that store the five-tuple rewriting rules for both internal and external TCP flows—are offloaded to the programmable switch. Besides that, the counter used for port allocation is also offloaded to the switch as a P4 register. When new rewriting rules have to be added to the address translation table, the pre-processing code will pack the current counter value into the packet header and send it to the middlebox server, where the table update is performed.

When applying Gallium to MazuNAT, we added the annotation that the address translation mapping would not have more than 65536 (2^{16}) entries, since each port number can have at most one entry in the map. This annotation allows Gallium to place the address translation maps on the switch.

Similar to the offloaded version of `MiniLB`, Gallium produces an offloaded version of the load balancer where the connection consistency map is stored in the switch. New incoming connections and packets with TCP control flags (RST and FIN) will be forwarded

to the middlebox server, as handling those packets requires an update to the map.

The P4 program generated for the firewall middlebox contains two match-action tables to filter the traffic from both directions. These tables offload the functionality performed by the `IPClassifier` elements used by the firewall. The 403 lines of code in the non-offloaded part are mainly for constructing and inserting the firewall rules.

For the proxy, the pre-processing code contains one match-action table that checks the incoming TCP packets' destination port. A packet rewriting action is also included in the P4 program to rewrite the TCP packet's destination to the web proxy server.

Gallium places Trojan detector's TCP flow state table on the programmable switch. TCP control packets, such as SYN or SYNACK, triggers a table update. These packets are forwarded to the middlebox server. Besides that, HTTP requests from an endhost that have received SSH traffic before are also be processed by the middlebox server to determine the type of requested file. Most of the TCP data packets that do not require deep packet inspection are handled solely by the programmable switch.

4.4.3 Performance

We first microbenchmark, for each of our five middleboxes, the throughput, latency, and CPU overheads. We then evaluate the performance overhead introduced by performing state synchronization when updating the state replicated on the switch. After that, we evaluate the five middleboxes using realistic workloads.

Experiment Setup. Our testbed consists of three servers and a Barefoot Tofino switch. Each server has an Intel Xeon E5-2680 CPU (2.5GHz, 12 cores) and a Mellanox ConnectX-4 100 Gbps NIC. Servers run Ubuntu 18.04 with Linux kernel version 4.15. All the three servers are connected to a Barefoot Tofino switch via 100 Gbps links. We dedicate one server to be the middlebox server. The middlebox server runs DPDK version 17.11. These two servers use traditional Linux networking stacks to generate and receive packets.

To compare performance with non-offloaded middleboxes, we use FastClick [6] to run non-offloaded middleboxes in the middlebox server and configure the routing table in the

Middlebox	FastClick	Gallium
MazuNAT	$23.16 \pm 0.53\,\mu s$	$15.98 \pm 0.21\,\mu s$
Load balancer	$23.09 \pm 0.31\,\mu s$	$15.96 \pm 0.20\,\mu s$
Firewall	$22.45 \pm 0.27\,\mu s$	$15.96 \pm 0.20\,\mu s$
Proxy	$22.72 \pm 0.87,\mu s$	$15.64 \pm 0.85\,\mu s$
Trojan Detector	$22.58 \pm 0.74,\mu s$	$14.80 \pm 0.43\,\mu s$

Table 4.2: Latency comparison of Gallium middleboxes and their FastClick counterparts. The numbers after $\pm$ denote the standard deviations.

switch to ensure all packets go through the server.

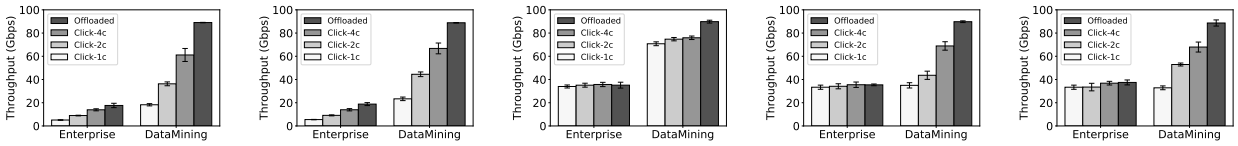
TCP Microbenchmark. We generate ten parallel TCP connections using `iperf` to test the maximum achievable throughput of the middleboxes. When we run Gallium middleboxes, we restrict the usage of the processing in the middlebox server to be on a single core. For FastClick, we test the middleboxes with 1, 2, and 4 cores. We also test different packet sizes (e.g., 100, 500, and 1500 bytes). We test the throughput ten times and measure the average and standard deviation.

Gallium substantially improves middlebox throughput and reduces CPU overheads on the middlebox server. Figure 4.6 compares the maximum achievable throughput of Gallium middleboxes and their FastClick-based counterparts. Overall, compared with FastClick running on 4 cores, Gallium with a single core outperforms by 20-187%. If we constrain the throughput to be identical, Gallium saves processing cycles by 21-79%. These performance benefits are because the non-offloaded partitions in the Gallium middleboxes are rarely used. In the Gallium version of MazuNAT and load balancer, only 0.1% of the packets in TCP flows are processed by the middlebox server. For the firewall and the proxy, all packet processing happens in the programmable switch.

Gallium reduces latency by 31%. We use `Nptcp` to test TCP packet latency. Table 4.2 shows the result. Like the reduction in processing overheads, the latency reduction comes

# tables	Insert	Modify	Delete
1	$135.2 \pm 22.0\,\mu s$	$128.6 \pm 23.6\,\mu s$	$131.3 \pm 18.8\,\mu s$
2	$270.1 \pm 33.0\,\mu s$	$258.3 \pm 34.9\,\mu s$	$262.7 \pm 29.8\,\mu s$
4	$371.0 \pm 39.2\,\mu s$	$363.0 \pm 37.3\,\mu s$	$366.1 \pm 37.7\,\mu s$

Table 4.3: Latency of updating offloaded P4 tables from middlebox server.



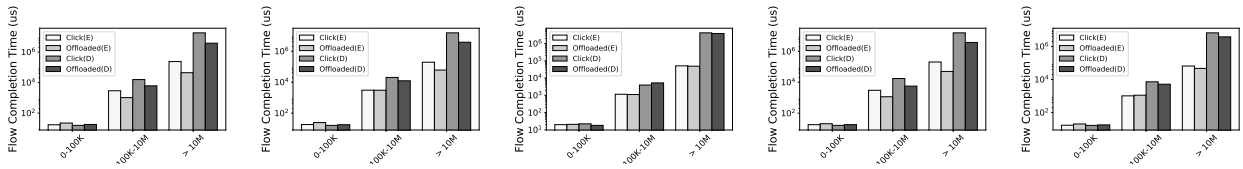
(a) MazuNAT (b) L4 Load Balancer (c) Firewall (d) Proxy (e) Trojan Detector

Figure 4.7: Throughput comparison of Gallium and FastClick on the enterprise workload and the data-mining workload.

from the fact that most packets do not go through the server in Gallium.

State Synchronization Overhead. We create programs with different table sizes and different numbers of tables to test the latency required to update offloaded tables from middlebox servers. We measure the latency of updating 1, 2, and 4 tables. Table 4.3 shows the latency of insertion, modification, and deletion for offloaded tables. A single table update is about 5x the end-to-end latency of a packet sent through a software middlebox. Gallium can provide overall performance benefits as long as the slow path and corresponding state synchronization operations are invoked infrequently. We consider this next using realistic traffic traces through the generated middleboxes.

Realistic Workloads. We evaluate two realistic workloads: an enterprise workload and a data-mining workload. The workloads (i.e., flow size distributions) are drawn from the CONGA work on datacenter traffic load balancing [2]. These workloads have both short flows and long flows. The majority of flows in both the enterprise and the data-mining workload are small; 90% of the flows in both workloads contain less than ten



(a) MazuNAT (b) Load Balancer (c) Firewall (d) Proxy (e) Trojan Detector

Figure 4.8: Flow completion time comparison of Gallium and FastClick on the enterprise(E) and data-mining(D) workload.

packets. We draw 100000 flow sizes from the flow size distributions and use 100 threads to send traffic. A thread sends a single connection at a time and starts a new connection when the current connection finishes.

To evaluate Gallium's throughput and CPU benefits, we fix the number of cores used in the middlebox server and use the RX counter of the receiver side's NIC to compute the average throughput during the experiment.

Gallium improves the overall throughput and reduces CPU overheads on both workloads. Figure 4.7 shows the result. Compared to the 4-core version of FastClick, Gallium using one core can achieve 1-35% more throughput on the enterprise workload and 18-46% more throughput on the data-mining workload. If we constrain the throughput to be the same as what 4-core FastClick can achieve, this means we can save 3-81% processing cycles or 0.03-4.39 server cores. We do better on the data-mining workload because the long flows are longer than that in the enterprise workloads.

Which connections do Gallium speed up in these realistic workloads? We measure the flow completion time for connections of different sizes. We bin flows based on their sizes and compute an average flow completion time for each bin. Figure 4.8 provides the comparison between Gallium and FastClick. We see that the reduction in flow completion time is concentrated on the long flows. This behavior is because long flows will have the majority of their packets handled by the programmable switch instead of the server.

4.5 Discussion

Reducing memory usage of programmable switches. One optimization to reduce memory usage of programmable switches is to let the programmable switch store only a fraction of any table, e.g., `MiniLB`'s `map` that stores the mapping from IP addresses to ports. For any packet that the programmable switch does not know how to handle, the middlebox server handles it instead. This means we need to handle the additional challenge of synchronizing the table in the software middlebox and its "cache" table in the programmable switch. We leave it to future work.

Extra functionalities in programmable switches. The P4 program generated by Gallium only uses registers and match-action tables with exact matches. Besides these functionalities, the P4 language provides other useful abstractions for packet processing, such as the longest prefix matching when matching on packet header fields, such as IP addresses. Programmable devices may also provide platform-specific P4 primitives such as advanced ALU operations (e.g., swapping the lower- and higher-32 bits of a 64-bit integer) or hardware-accelerated hash functions. Currently, these functionalities are not used by Gallium as some abstractions, such as LPM, do not exist in software middleboxes.

Dealing with complex data structures. Currently, Gallium can only handle two data structures, `HashMap` and `Vector`. This is because we know how to map them to P4 native primitives (§4.2.3). We find that these two data structures are the most commonly used ones in Click. Middleboxes that use complex data structures (e.g., linked list, tree) have to be processed as part of the non-offloaded partition or require code modifications to enable offloading.

Cost model of offloading. Currently, Gallium tries to offload as many lines of code (LLVM instructions) as possible to the programmable switch. This approach simplifies the algorithm design, as discussed in §4.2.2. This model does not consider that the offloading of different operations will give the middlebox different performance benefits. For example, offloading a table lookup to the programmable switch will provide more

performance benefits than offloading an integer addition operation. Therefore, Gallium's partitioning algorithm may produce sub-optimal partitions as the algorithm may choose to offload an integer addition over a table lookup. One possible solution to this is to assign a weight to each operation, which would be a measure of its associated performance benefit when executed on the switch. We would then try to maximize the weight of instructions offloaded to the switch as we move operations into the non-offloaded partition in the algorithm described in item 4.2.2. We leave it to future work.

4.6 Related Work

Offloading to programmable network hardware. Hardware offloading is a well-known technique to improve performance. There have been many studies that use programmable network hardware to accelerate specific applications. For example, KV-Direct [49] is a key-value store on FPGA NICs. IncBricks [56] and NetCache [41] use programmable network hardware to build an in-network cache. NOPaxos [52], Eris [51], and NetChain [40] use programmable switches to do in-network coordination. DAIET [73] aggregates data along network paths using programmable switches. Silkroad [59] leverages a programmable switch's match action table to scale up the number of concurrent connections for load balancers. The current offloading approaches need manual effort in partitioning the source program, while Gallium provides a compiler-oriented approach to partition the source program automatically.

Frameworks for programmable network hardware. Our work is in the category of helping developers move their applications to programmable network devices. Floem [68] allows developers to explore different offloading methods for programmable NICs. iPipe [54] is an actor-based framework for offloading distributed applications onto programmable NICs. Both Floem and iPipe require manually partitioning applications, and they assume the smart NIC can accept programs written in C. ClickNP [50] is another framework using FPGA-based smart NICs for offloading network functions. Similar to Gallium, ClickNP uses the Click [46] dataflow programming model. Different from all

this work, our focus is on P4, a hardware-independent target for a class of programmable networking hardware. P4's expressiveness is more restricted than C and FPGA, and Gallium's goal is to partition middlebox source code and deal with resource constraints of P4-based programmable hardware. Domino [79] provides a DSL for developers to write packet processing programs that could be compiled to run on programmable line-rate switches. A major difference between Domino and Gallium comes from the choice of deployment model. While Domino deploys the entire program onto the switch and rejects programs that could not be deployed, Gallium aims at partitioning the program and makes a portion of it deploy-able on the switch.

Program slicing. The compilation techniques to partition a source program based on the dependency relations and control-flow graphs are similar to program slicing [38, 72, 85]. However, program slicing aims to abstract the program: extract a minimal program whose logic is similar to the source program for a subset of the source program's variables. Our goal is to partition the source program into multiple partitions, where the cumulative effect is the same as the source program.

4.7 Summary

Offloading software middleboxes to programmable switches can yield orders-of-magnitude performance gains; however, manually rewriting software middleboxes is hard and time-consuming. In this chapter, we have designed and implemented Gallium. Gallium uses program partitioning and compilation to transform software middleboxes to their functionally-equivalent versions, which leverage programmable switches for high performance. Our evaluations show that Gallium can save 21-79% of processing cycles and reduce latency by about 31% across various types of software middleboxes. Gallium's source code is available at https://github.com/Kaiyuan-Zhang/Gallium-public.

Chapter 5

Pebble: Transactional Packet Processing on Multi-Core Servers

This chapter focuses on automating the process of building concurrent, scalable software middleboxes. We present Pebble, a programming framework and runtime system that provides a transaction abstraction for middlebox developers. Pebble uses the transaction chopping technique to break down the packet processing logic to utilize the pipeline parallelism of middleboxes. For each chopped transaction piece, Pebble uses optimistic concurrency control (OCC) to make them scalable when running concurrently. Pebble guarantees the serializability of chopped transactions by tracking the data dependencies at runtime and avoiding cycles in the serialization graph. Our experience shows that Pebble could reduce the effort of developing middleboxes for multi-threaded environments.

5.1 Pebble

Pebble is a programming framework plus runtime system that provides a transaction API for implementing software middleboxes. Developers could express the packet processing logic of middleboxes as a set of transactions. Pebble executes these transactions on a multi-core server while providing serializability guarantees.

5.1.1 Design Goals

As mentioned earlier, we make two observations of software middleboxes. The first observation is that middleboxes can often be divided into a sequence of processing stages due to the layered nature of the networking protocol stack. For example, the logic of maintaining a flow table for TCP is often independent of the ARP protocol address lookup logic. Another observation is that middleboxes are often under a "read-most" workload: modification of middlebox state is much rarer compared with reading those states.

Based on these observations, we propose the following design goals for Pebble:

GOAL 1. Pebble should utilize the pipeline structure of middlebox logic and allow parallel execution of different components of the same piece of code.

GOAL 2. Pebble's locking and concurrency control should minimize its runtime overhead for read operations.

Besides these two goals derived from the observations, we also add a third design goal where Pebble should guarantee the correctness of concurrent execution. We choose providing serializability as our design goal as it is intuitive to application developers:

GOAL 3. The parallel processing of packets in Pebble should result in a serializable execution.

We next describe Pebble's programming model, followed by the description of transaction chopping and OCC techniques used by Pebble.

5.1.2 Programming Model of Pebble

To implement a software middlebox, Pebble takes two inputs from the developer: a list of global states maintained by the middlebox, and a set of transactions. Pebble supports both packet transactions and periodically running transactions.

As a concrete example, let's consider a simple middlebox, `MiniLB`. `MiniLB` is a load balancer that uses a round-robin approach to pick from a pool of backend servers.[1]

[1]Note that `MiniLB` is designed to demonstrate techniques of Pebble, its design may not be optimal

To build MiniLB with Pebble, developers need to give as input both the statement for middlebox states and a set of transactions. As shown in 5.1b, MiniLB maintains three global states: a hash map (flow_table) that keeps track of backend servers for established flows, an array of backend servers' IP addresses (backends), and an integer number (next_backend) that stores the index of the next backend server that MiniLB should use when new flow arrives.

Transactions in Pebble are represented as functions, each transaction takes as input a reference to the middlebox state. Transactions that handle input packets also take the packet as an extra argument, as shown in 5.1a. When accessing middlebox states, Pebble requires the transaction code to use read() and write() methods, which internally handle locking and concurrency control.

```
fn packet_tx(s: &State, pkt: PktT) {
  let flow_id = get_flow_id(&pkt);
  if s.flow_table.read().contains(&flow_id) {
    /* -- <rewrite packet> -- */
  } else {
    let nb = s.next_backend.write();
    let backend_addr = s.backends.read()[*nb];
    *nb = (*nb + 1) % N_BACKEND;
    /* -- <add new entry to flow table> -- */
    /* -- <rewrite packet> -- */
  }
}
```

(a) Packet processing transaction

```
struct State {
  flow_table:   HashMap<FlowID, u32>,
  backends:     RwLock<Vec<u32>>,
  next_backend: RwLock<usize>,
}
```

(b) Middlebox State

```
fn expire_flow(s: &State, time: TimeT) {
  for i in 0..s.flow_table.size() {
    let e = s.flow_tablee[i].write();
    if e.last_access + EXP_TIME < time {
      *e = None;
    }
  }
}
```

(c) Periodically executed transaction

Figure 5.1: Developer's input to Pebble for MiniLB

5.1.3 Optimistic Concurrency Control

Pebble uses OCC as its concurrency control algorithm to guarantee serializable execution of transactions. Pebble choose to use OCC for the following reasons:

- In OCC, all the locking operations of shared objects happens at the commit phase, which is after the execution of the application transaction code. This allows the framework to perform locking automatically without code change from the developer.

- As the locking is delayed to the commit phase, deadlocks could also be avoided without developers' interference by always acquire locks in the same order.

- The OCC commit protocol could use version number validation instead of reader-writer locks for read operations, therefore removing the implicit write operations for read accesses.

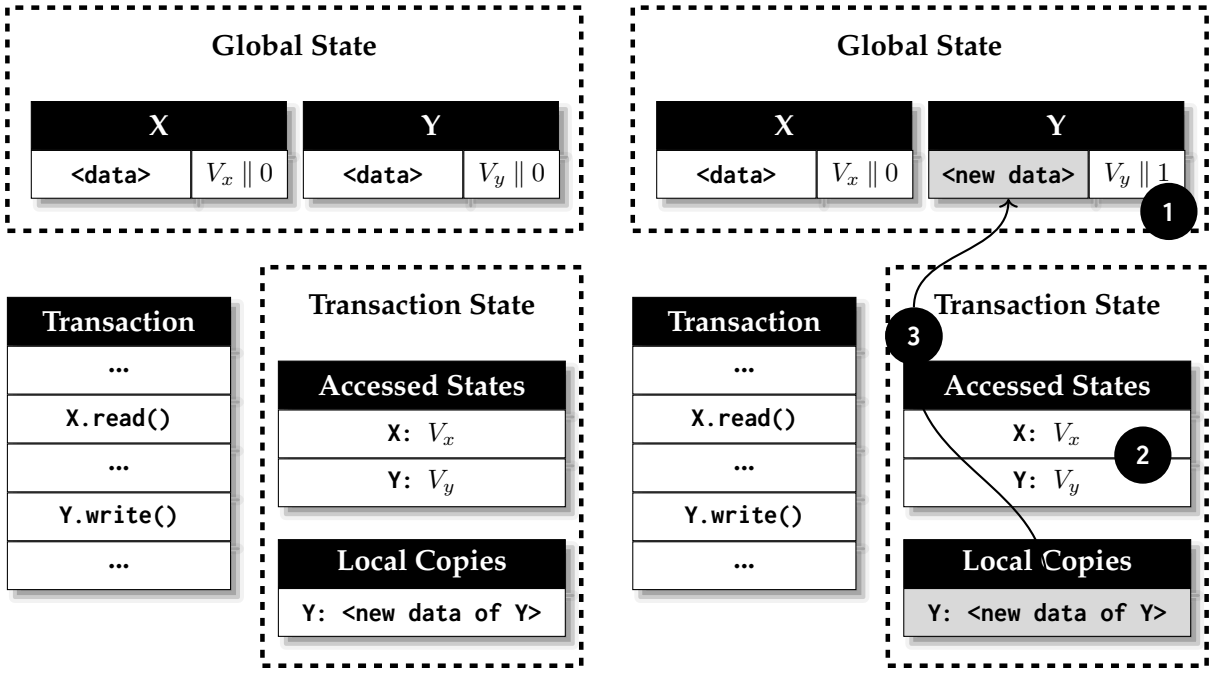
With OCC, developers using Pebble implement the application as a set of transactions. These transactions will get executed when packets are received from the device. Developers could also ask Pebble to execute a transaction once every fixed time interval (such as every 50ms).

Executing Transactions

When executing a transaction, Pebble keeps track of all the global objects accessed by the transaction body. Each global object contains the following metadata information:

- `version number`: The current version of the object. The last bit of this number is used to indicate if the object is locked by a writer. A writer thread increases the version number of an object after it finishes modifying its content.

- `lock rank`: A rank number used to determine the order each object is locked during the commit phase of OCC. This number is uniquely chosen during the initialization



(a) Before Commit (b) During Commit

Figure 5.2: State snapshot of Pebble's OCC before and during commit

of the object.

For read access, Pebble reads the `version number` of the object, if it is locked (version number is odd), Pebble blocks until the writer unlocks it. After that, Pebble records the address of the object in its read set, along with its current version number. For write access, besides waiting for the object to get unlocked and record the version number in the write set, Pebble also needs to create a local copy of the object, this local copy is used to record all the changes made to the object by the current transaction. As a concrete example, Figure 5.2a shows the states of Pebble after executing an example transaction. In this transaction, two global states, `X` and `Y` are accessed. As shown in the figure, when the transaction code uses `read()` and `write()` method to access global states, the current version number V_x and V_y will be loaded from the global memory and stored in a thread-local data structure (i.e. the "Accessed States" box shown in the figure). Besides that,

because the access to `Y` is a write operation, Pebble will create a local copy of the global object `Y` and let the transaction code perform all its modifications on this local copy.

Commit Protocol

Upon finishing the execution of the transaction code, Pebble goes to the commit phase where it tries to commit the transaction and writes all its changes back to the global object. Pebble's commit protocol follows the same structure as Silo [83]. At a high level, there are three major steps of the commit protocol. As shown in Figure 5.2b, Pebble first locks all the global states that the current transaction is going to modify by setting the last bit of version number to 1 (step ❶). After that, Pebble reads the version number of each accessed object again to make sure that no other transaction has committed changes to them (step ❷). Finally, the modification made by the transaction is copied from the thread-local memory to the global memory (step ❸). The detailed commit protocol works as follows.

1. Pebble sorts all the objects in its write set based on the `lock rank` of the object.

2. For each object in the (sorted) write set, Pebble locks the object by setting the last bit of its `version number` from 0 to 1. Pebble performs the version number update using atomic compare and exchange operation to make sure that the version number of the object does not change while locking the object.

3. For each object in the read set of the transaction, Pebble loads its version number again and compares it against the recorded value in the read set.

4. If any of the objects have a mismatched version number, Pebble aborts the transaction: It discards all the local copies in the write set and unlocks all objects in the write set.

5. If all the version numbers are unchanged, the transaction is committed: Pebble copies the modified value of each object from the write set to their global memory, increases the version number of each modified object by 1, and unlocks all objects.

Note that with Pebble's OCC algorithm, no writes are required for objects that are read but not modified by the transaction. Thus for a read-only transaction, Pebble does not introduce any extra writes to shared states, making the application more scalable under a read-most workload.

5.1.4 Breaking Down Transactions for Better Parallelism

While OCC guarantees serializable execution, the memory footprint of the transaction itself could also have an impact on the performance of the system. For most applications such as a NAT or a load balancer, the amount of memory accessed during the processing of a single packet is usually small. However, some of the periodically executed logic, such as expiring old entries in a flow table, requires accessing a considerable amount of global states. The long execution time and large coverage of accessed states make it much more likely that some other transactions have conflict state modifications with this one. Therefore making transaction aborts much more likely to happen.

To reduce the chance of conflicts, Pebble allows the developers to break down large transactions into smaller pieces. Each of the smaller pieces (later referred to as "mini transactions") is executed with a separate OCC. This allows earlier commit of modifications made by the transaction and therefore reduces the likelihood of conflicts and aborts. However, as pointed out in [75], simply breaking down large transactions and executing them without restrictions may lead to a non-serializable result. To prevent this, Pebble tracks the ordering dependencies among the running transactions and avoids forming cycles in the serialization graph.

Additional Metadata for Dependency Tracking

To track ordering dependencies of transactions. Pebble adds the following additional metadata to each global objects:

- `readers`: A list of un-committed transactions that have read the current object.
- `writer`: The last transaction that has modified the object.

Note that since Pebble pins each transaction to a single worker thread, the length of `readers` never exceeds the number of worker threads. In the actual implementation, Pebble represents `readers` with an array, and stores the last transaction number on that thread that has accessed the object in the corresponding slot.

Besides additional per-object metadata, Pebble also needs to add extra per-transaction metadata:

- `dependencies`: A list of transactions that should be ordered before the current transaction in the serialization order.

- `transaction ID`: A monotonically increasing ID that is assigned at the initialization time of each transaction. In Pebble, the tuple $(threadID, transactionID)$ is used to uniquely name a transaction. Mini transactions shares the same $transactionID$.

- `max lock rank`: The maximum rank number of all the objects that have been accessed by the committed mini transactions of the current transaction.

All of the metadata above could only be modified by the thread executing the transaction. However, the `transaction ID` and `max lock rank` field will be read by other concurrent threads. Note that since each worker thread could only execute one transaction at a time, Pebble could check if a transaction $(tid, txid)$ is committed simply by checking if the `transaction ID` field of thread tid is larger than $txid$. Therefore, the information that a transaction is committed could be broadcasted to other threads by incrementing the `transaction ID` field by 1.

Avoiding Cycles in Serialization Graph

Pebble tracks the dependency and delays the execution and commit of mini transactions in order to guarantee serializability. To track the ordering dependency among transactions, Pebble reads the `readers` and `writer` field of each accessed global object before the commit phase starts and adds all the conflicting transactions to the dependency set. These

dependencies include read-write, write-read, and write-write conflicts. The `readers` and `writer` set of each object is updated after the version number validate and before the write back happens (between step 3 and 4 in Figure 5.1.3).

With the dependency information, Pebble insert delays to prevent cycles in the serialization graph. In Pebble, the delays could be inserted at two points.

The first type of delay is added before Pebble acquires all the write locks during the commit phase. For each object that was read or modified by the current transaction or mini transaction, Pebble checks if all the transactions in its `dependencies` set are either committed or have a `max lock rank` higher than the rank of the accessed object. Pebble will stall the current transaction until the condition is met.

The second type of delay is added before a committed transaction marks itself as committed. Pebble loads the transaction ID field of all the other running threads. It then checks if all of them have a larger value than transactions in the `dependencies` set of the current transaction. Pebble waits until this condition is met before letting other threads know that the current transaction is committed.

With the two types of delay, before a mini transaction trying to access object o, Pebble forces it to wait until it knows that all the predecessors in the serialization graph will never access a shared object. This prevents cycles in the serialization graph as there will be no edges pointing back to a predecessor of any vertex, which in turn guarantees serializability.

5.2 Evaluation

This section evaluates how well Pebble could help developers build scalable software middleboxes. To be more specific, we aim to answer the following questions:

- How much effort does it take to implement middlebox with Pebble compared with implementing a single-threaded version?

- Do middleboxes built with Pebble scale with the number of threads?

5.2.1 Building Middleboxes with Pebble

To understand how much extra effort is added by Pebble compared to implementing a single-threaded version. We implement two middlebox applications using Pebble and compared them with their single-threaded counterpart.

NAT

Our first middlebox application is a network address translator(NAT), the NAT implementation maintains two hashmaps for the bijective address mapping between external and internal network spaces. Besides the packet handling transaction, the NAT periodically executes a transaction that iterates through the hashmaps and expires entries that are inactive for a pre-defined time threshold.

Compared with the single-thread implementation of NAT. The packet processing transaction remains the same. On the other hand, we did break down the record-expiring transaction so that access to a single entry becomes an individual "mini transaction". This breaking down is done by adding 5 lines of source code.

Load balancer with backend statistics

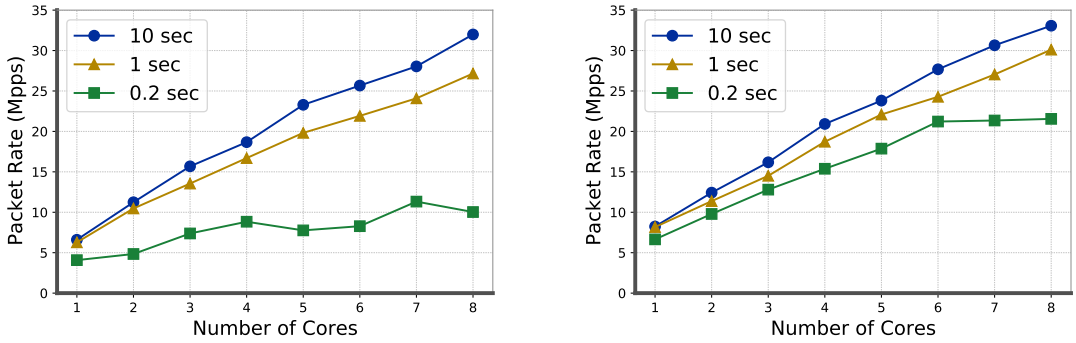
The next middlebox we built with Pebble is a load balancer. For each incoming TCP flow, the load balancer uses a hash function to determine which backend server the flow should be redirected to. The chosen backend server is recorded in a flow table to provide connection persistency. The load balancer also keeps track of the bandwidth to each of the backend servers. Like the NAT application, the load balancer also expires inactive mappings from its flow table.

When implementing it on Pebble, we also divided the flow expiration transaction into smaller pieces. Besides that, for the statistics of bandwidth to each backend server, we changed the implementation so that each thread keeps a thread-local version of the statistics and created another transaction that periodically collects those data and updates the global state. Compared with the single-threaded implementation, we changed a total of

16 lines of code.

Our experience of building the two applications above shows that Pebble's transaction API could let developers build middleboxes without much extra effort. Though for the load balancer, we added thread-local states for better performance, all the concurrency control logic are handled by Pebble, making the modification easy to implement. Each of the middleboxes took us less than 3 person-days to develop.

5.2.2 Performance of Middleboxes built with Pebble



(a) NAT

(b) Load balancer

Figure 5.3: Packet processing rate of middleboxes built with Pebble, using different intervals for expiring inactive flows.

To understand how well could applications built with Pebble scale with the number of worker threads, we measure the maximum packet processing rate for the aforementioned two applications. Our evaluation consists of three server machines running Ubuntu Linux with kernel version 4.15. Both our benchmark program and middlebox applications are using DPDK 17.11 library with 40 Gbps Mellanox ConnectX-3 NICs. Our benchmark program generates 64-byte UDP packets between the sender and the receiver and all of the packets are being processed by the middlebox. Besides measuring the change of maximum packet rate when increasing the number of threads, we also evaluate the effect of increasing amount of write operations by invoking the flow expiration transaction, which

is write-intensive, more frequently.

As shown in Figure 5.3, both applications built with Pebble could scale well with the number of working threads: Compared to single-core performance, the two applications could achieve maximum speedup of $4.8\times$ and $4.0\times$ with 8 CPU cores. Our evaluation also shows the impact of intensive write operations on the scalability of Pebble. For both applications, the packet processing rate decreases when increasing the frequency of flow-expiration transactions. This is because the chance of conflict accesses to global states increases with the rate of write operations. We found that our NAT application is more sensitive to the frequency of flow-expiration transactions due to the fact that NAT needs to maintain a bijective mapping (i.e. two hashmaps).

5.3 Related work

Packet processing frameworks. There has been several previous studies that provides a programming framework to help developers implement packet processing applications at line-rate or close to line-rate. The Click [45] modular router allows developers to construct applications by composing pre-defined modules (called elements). This improves the velocity of application development while at the same time makes it easier for application developers to create high-performance programs as the elements are optimized by framework developers. Recent studies such as [7] and [29] further optimize and reduce the per-packet processing overhead to get better packet processing throughput. Pebble, on the other hand, focuses on improving the performance of multi-threaded packet processing where state-sharing among different CPU cores are present. Therefore, our work is orthogonal to these works above and could potentially be complementary to them.

Packet processing with programmable hardware. Another line of research on improving packet processing performance is offloading packet processing logic to programmable hardware such as FPGAs [50] or smartNICs [69, 55]. Though Pebble's solution can only applied on software implementations and therefore could not be applied on FPGA implementations. SmartNICs are often equipped with a multi-core ARM or MIPS CPU, which

means that Pebble's concurrency control mechanisms could potentially improve packet processing on those smartNICs.

Packet processing with safe languages. The idea of using safe languages in systems software to reduce the programming effort or providing stronger correctness and safety guarantees has been studies by many previous researches [70, 9, 18]. In the domain of packet processing, the XDP system from the Linux kernel allows developers to install packet filters into the kernel. It uses extended Berkeley packet filter (eBPF) to ensure the packet filter code is safe to be executed inside the kernel. Another example of using safe languages for packet processing is NetBricks [66]. It utilizes the type system of Rust to provide inter-NF isolation without introducing any runtime overhead to packet processing. Pebble also choose to use Rust programming language, because its mutability information of references type and its lifetime system could be used to enforce correct locking in application code.

5.4 Summary

The rapidly increasing bandwidth of networking devices have made it challenging for software middleboxes to achieve line-rate. We argue that instead of letting middlebox developers worry about multi-core performance, the software middlebox runtime should provide support to automate the processing and let developers focus more on implementing packet processing logic. We have designed and implemented Pebble, a programming framework and runtime system that allow developers implement software middleboxes using a transaction API. Pebble uses optimistic concurrency control and transaction chopping to scale middlebox implementations to multiple CPU cores while providing serializability guarantee.

Chapter 6

Conclusion

Software middleboxes play a critical role in modern computer networks for improving safety, availability, and performance of networked services. Today, they suffer from correctness bugs, and large amounts of manual programming effort have to be spent in leveraging multi-core CPUs or programmable switches to speed up these software middleboxes. This thesis provides an automated solution for correct and efficiency software middleboxes using automated program analysis and runtime support. We introduce three systems, Gravel, Gallium, and Pebble.

Gravel explores the feasibility of formally verify high-level middlebox properties using symbolic execution. Our empirical study shows that with some code modification and correct abstraction, a symbolic executor could cover about 78% of Click [46] elements. Based on this, we built Gravel to allow verification of high-level middlebox properties of "almost unmodified" Click middleboxes.

Gallium is a compilation tool that uses static analysis to partition a given middlebox implementation so that part of it could be offloaded to a programmable switch. During the compilation, Gallium guarantees the functional equivalence of the input and output program and provides a run-to-completion semantic for the concurrent processing of programmable switch and software middlebox. Our evaluation shows that Gallium could save 21-79% of processing cycles and reduce latency by about 31%.

Finally, we introduce Pebble, a programming framework and runtime system that

provides a transaction API to middlebox developers. Pebble uses transaction chopping technique to improve the parallelism of input transactions, and uses OCC to guarantee serializable execution of chopped transactions. Pebble allows developers to build multi-core middlebox implementation while focusing on the core packet processing logic.

To further pushing automation in improving correctness and efficiency of middleboxes, we find the following research directions worth exploring:

Automatic transaction chopping with program analysis. As mentioned in Chapter 5, the current implementation of Pebble requires middlebox developers to manually divide a large transaction into smaller pieces. Though Pebble's runtime system provides correct and efficient execution of chopped transactions, the task of identifying "choppable" transactions is still left to the application developers. On the other hand, several previous researches [75, 87] on database and transaction processing systems have proposed static analysis algorithms that could automatically break down large database transactions into smaller snippets. Therefore, one natural question that arises here is whether we could design a static analysis algorithm that chops packet processing transactions.

Automated verification of concurrent middlebox implementation. One major limitation of Gravel is that it assumes a serializable execution of packet processing code. This implies that the correctness guarantee provided by Gravel's formal verification does not automatically hold when executed with multiple CPU cores. With the growing gap between network bandwidth and single-core performance, the necessity of providing formal guarantees for concurrent middlebox implementation also arises. On the other hand, our experience in developing Pebble shows that a transaction API that provides serializability could potentially reduce the problem of verifying concurrent implementation back to formal verification of sequential code. Therefore, one question that we find worth exploring is whether we could provide a formally verified transaction API that extends Gravel's correctness guarantee to multi-threaded middlebox implementation.

Bibliography

[1] Divyakant Agrawal, Arthur J Bernstein, Pankaj Gupta, and Soumitra Sengupta. "Distributed optimistic concurrency control with reduced rollback". In: *Distributed Computing* 2.1 (1987), pp. 45–59.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 503–514.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. "NetKAT: Semantic Foundations for Networks". In: *POPL*. 2014.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. "SNAP: Stateful Network-Wide Abstractions for Packet Processing". In: *SIGCOMM*. 2016.

[5] *Balance, Inlab Networks*.

[6] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast Userspace Packet Processing". In: *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '15. Oakland, California, USA: IEEE Computer Society, 2015, pp. 5–16. ISBN: 978-1-4673-6632-8. URL: http://dl.acm.org/citation.cfm?id=2772722.2772727.

[7] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast userspace packet processing". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2015, pp. 5–16.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification". In: *SIGCOMM*. 2017.

[9] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. "Extensibility safety and performance in the SPIN operating system". In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. 1995, pp. 267–283.

[10] 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". In: *SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. URL: http://doi.acm.org/10.1145/2656877.2656890.

[11] 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". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: http://doi.acm.org/10.1145/2656877.2656890.

[12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 99–110. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486011. URL: http://doi.acm.org/10.1145/2486001.2486011.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. 2008.

[14] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. "Verifying concurrent software using movers in {CSPEC}". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 306–322.

[15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare logic for certifying the FSCQ file system". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 18–37.

[16] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. "DRMT: Disaggregated Programmable Switching". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017.

[17] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. http://coq.inria.fr/distrib/current/refman/. INRIA. July 2016.

[18] Cody Cutler, M Frans Kaashoek, and Robert T Morris. "The benefits and costs of writing a {POSIX} kernel in a high-level language". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 89–105.

[19] *CVE-2013-1138*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1138.

[20] *CVE-2014-3817*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3817.

[21] *CVE-2014-9715*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9715.

[22] *CVE-2015-6271*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6271.

[23] *CVE-2017-7928*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7928.

[24] *Data Plane Development Kit*. https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html.

[25] Lorenzo De Carli, Robin Sommer, and Somesh Jha. "Beyond pattern matching: A concurrency model for stateful deep packet inspection". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 1378–1390.

[26] Mihai Dobrescu and Katerina Argyraki. "Software Dataplane Verification". In: *NSDI*. 2014.

[27] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "Dataplane Equivalence and Its Applications". In: *NSDI*. 2019.

[28] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *NSDI*. 2016.

[29] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostic. "PacketMill: Toward Per-Core 100-Gbps Networking". In: *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21), April 19–23, 2021, Virtual, USA*. ACM Digital Library. 2021.

[30] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925.

[31] Bryan Ford, Pyda Srisuresh, and Dan Kegel. “Peer-to-Peer Communication Across Network Address Translators.” In: *USENIX ATC*. 2005.

[32] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. “Uncovering Bugs in P4 Programs with Assertion-based Verification”. In: *SOSR*. 2018.

[33] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. “Duet: Cloud Scale Load Balancing with Hardware and Software”. In: *SIGCOMM*. 2014.

[34] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. “Certified concurrent abstraction layers”. In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 646–661.

[35] Theo Härder. “Observations on optimistic concurrency control schemes”. In: *Information Systems* 9.2 (1984), pp. 111–120.

[36] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. “IronFleet: Proving Practical Distributed Systems Correct”. In: *SOSP*. 2015.

[37] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. “Type-aware transactions for faster concurrent code”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. 2016, pp. 1–16.

[38] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural slicing using dependence graphs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990), pp. 26–60.

[39] *In-Network DDoS Detection*. https://barefootnetworks.com/use-cases/in-nw-DDoS-detection/.

[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "Netchain: Scale-Free Sub-RTT Coordination". In: *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*. NSDI'18. Renton, WA, USA: USENIX Association, 2018, pp. 35–49. ISBN: 9781931971430.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.

[42] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks." In: *NSDI*. 2012.

[43] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *NSDI*. 2013.

[44] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[45] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[46] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The Click Modular Router". In: *TOCS* (2000).

[47] Hsiang-Tsung Kung and John T Robinson. "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226.

[48] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3-642-17510-4, 978-3-642-17510-7. URL: http://dl.acm.org/citation.cfm?id=1939141.1939161.

[49] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.

[50] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016.

[51] Jialin Li, Ellis Michael, and Dan R. K. Ports. "Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 104–120. ISBN: 9781450350853. DOI: 10.1145/3132747.3132751. URL: https://doi.org/10.1145/3132747.3132751.

[52] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. "Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 467–483. ISBN: 9781931971331.

[53] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. "p4v: Practical Verification for Programmable Data Planes". In: *SIGCOMM*. 2018.

[54] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. "Offloading Distributed Applications onto SmartNICs Using IPipe". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Beijing, China: Association for Computing Machinery, 2019, pp. 318–333. ISBN: 9781450359566. DOI: 10.1145/3341302.3342079. URL: https://doi.org/10.1145/3341302.3342079.

[55] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. "Offloading distributed applications onto smartnics using ipipe". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 318–333.

[56] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. "IncBricks: Toward In-Network Computation with an In-Network Cache". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 2017.

[57] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. "Checking Beliefs in Dynamic Networks". In: *NSDI*. 2015.

[58] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. "Debugging the Data Plane with Anteater". In: *SIGCOMM*. 2011.

[59] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 15–28. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098824. URL: http://doi.acm.org/10.1145/3098822.3098824.

[60] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. "Flow-level State Transition as a New Switch Primitive for SDN". In: *HotSDN*. 2014.

[61] Shuai Mu, Sebastian Angel, and Dennis Shasha. "Deferred runtime pipelining for contentious multicore software transactions". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[62] *NAT Behavioral Requirements for TCP*.

[63] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *SOSP*. 2017.

[64] *The netfilter.org Project*.

[65] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

[66] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of {NFV}". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 203–216.

[67] Aurojit Panda, Ori Lahav, Katerina J Argyraki, Mooly Sagiv, and Scott Shenker. "Verifying Reachability in Networks with Mutable Datapaths." In: *NSDI*. 2017.

[68] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. "Floem: A Programming System for NIC-Accelerated Network Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 663–679. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/phothilimthana.

[69] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. "Floem: A programming system for NIC-accelerated network applications". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 663–679.

[70] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. "Pilot: An operating system for a personal computer". In: *Communications of the ACM* 23.2 (1980), pp. 81–92.

[71] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. 2004.

[72] Thomas Reps and Genevieve Rosay. "Precise interprocedural chopping". In: *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*. 1995, pp. 41–52.

[73] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. "In-network computation is a dumb idea whose time has come". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 2017.

[74] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G. Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. "Programmable Calendar Queues for High-speed Packet Scheduling". In: *17th USENIX Symposium on Networked Systems Design and Implementation*. Ed. by Ranjita Bhagwan and George Porter. 2020.

[75] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. "Transaction chopping: Algorithms and performance studies". In: *ACM Transactions on Database Systems (TODS)* 20.3 (1995), pp. 325–363.

[76] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. "Making middleboxes someone else's problem: network processing as a cloud service". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24.

[77] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. "Push-button Verification of File Systems via Crash Refinement". In: *OSDI*. 2016.

[78] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. "Nickel: A Framework for Design and Verification of Information Flow Control Systems". In: *OSDI*. 2018.

[79] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. "Packet Transactions: High-Level Programming for Line-Rate Switches". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 15–28. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934900. URL: http://doi.acm.org/10.1145/2934872.2934900.

[80] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "Debugging P4 programs with Vera". In: *SIGCOMM*. 2018.

[81] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "SymNet: Scalable Symbolic Execution for Modern Networks". In: *SIGCOMM*. 2016.

[82] *Tofino: World's fastest P4-programmable Ethernet switch ASICs*. https://barefootnetworks.com/products/brief-tofino/.

[83] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. "Speedy transactions in multicore in-memory databases". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 18–32.

[84] Rainer Unland. *Optimistic concurrency control revisited*. Tech. rep. Arbeitsberichte des Instituts für Wirtschaftsinformatik, 1994.

[85] Mark Weiser. "Program slicing". In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357.

[86] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015, pp. 357–368.

[87] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. "High-performance ACID via modular concurrency control". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 279–294.

[88] *The Z3 Theorem Prover*. https://github.com/Z3Prover/z3.

[89] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. "Verifying Software Network Functions with No Verifcation Expertise". In: *SOSP*. 2019.

[90] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. "A Formally Verified NAT". In: *SIGCOMM*. 2017.

[91] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. "Automated verification of customizable middlebox properties with gravel". In: *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 2020, pp. 221–239.

[92] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. "Gallium: Automated Software Middlebox Offloading to Programmable Switches". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 283–295.