Thèse n° 10 169

Verification of Software Network Functions with No Verification Expertise

Présentée le 30 septembre 2020

à la Faculté informatique et communications
Laboratoire d'architecture des réseaux
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Arseniy ZAOSTROVNYKH

Acceptée sur proposition du jury

École
polytechnique
fédérale
de Lausanne

2020

Abstract

Software network functions (NFs), such as a network address translator, load balancer, or proxy, promise to bring flexibility and rapid innovation to computer networks and to reduce operational costs [46]. However, continuous updates and flexibility typically come with the risk of bugs [42]. As networks are a critical component of modern computing (e.g., the Internet, cloud infrastructure), both users and operators demand that they be as reliable as economically possible [108].

One way to guarantee reliability is by formal verification. To prove the absence of bugs, formal verification applies rigorous mathematical logic.

State-of-the-art formal verification methods either require substantial effort and verification expertise on the part of the practitioner or lack completeness. In particular, theorem proving [147], to guide a mechanical proof checker, requires a human to annotate the code they verify with logical steps, and this annotation is something for which NF developers typically lack the time or expertise to do. Whereas, symbolic execution (SE) [124] can analyze the code automatically, hence requiring significantly less effort; but it has difficulties with the so-called path explosion problem [24].

Fortunately, software NFs have a special structure:

- They feature a single infinite event-processing loop, and most of their code does not pose serious challenge to SE.
- The parts that do pose a challenge to SE typically implement well-defined data structures that are reused across NFs.

In this dissertation, we present Vigor, a new method and tool for verifying software NFs; it exploits this special structure of NFs. We make two contributions:

First, we show that it is possible to develop software NFs that are guaranteed to satisfy a formal specification and at the same time exhibit competitive performance. We develop a practical toolchain for developing NFs in C by using a standard fast packet-processing framework (DPDK). Vigor provides a library of formally verified data structures (libVig); this library replaces the parts posing a challenge to SE. The Vigor toolchain applies exhaustive SE to the NF code that uses libVig. Vigor introduces a new technique, which we call “lazy proofs”, for automatically stitching the proofs together and, ultimately, for certifying the whole NF correct relative to its functional specification.

Second, we show that *practical* high-level semantic (functional) verification of software NFs is possible. By practical, we mean a verification that (a) requires no verification input other than the specification (*push-button*), (b) guarantees correctness of the entire runtime software stack, i.e., operating system, driver, framework, and NF (*full-stack*), and (c) does not require writing a comprehensive specification to verify the first line of code (*pay-as-you-go*).

In summary, Vigor provides push-button, full-stack, pay-as-you-go formal verification of software NFs with no reduction in NF performance or developer productivity.

To substantiate our claims we implement five NFs—a NAT, a dynamic load balancer, a MAC-learning bridge, a firewall, and a traffic policer—prove their correctness, and show that their performance is as good as the performance of the fastest non-verified software NFs. All of our NFs achieve a median latency of 4.1±0.2 μs and over 4 millions of packets per second throughput with 10 Gbps network interfaces. In terms of developer productivity, an average partial property takes 6.4 lines of Python specification, and the verification of the full stack (i.e., NF, packet-processing framework, operating system, driver) takes from a few hours to one day. The Vigor toolchain is open-source and is available, along with tutorials and example NF implementations, at https://vigor.epfl.ch.

Résumé

Les fonctions réseau logicielles, telles qu'un traducteur d'adresse réseau (NAT), répartiteur de charge, ou proxy, promettent d'apporter de la flexibilité, d'accélérer l'innovation et de réduire les coûts dans le domaine des réseaux informatiques [46]. Cependant, les mises à jour fréquentes et la flexibilité viennent souvent avec un risque accru de problèmes [42]. Les réseaux tel qu'Internet étant un composant critique de l'informatique moderne, les utilisateurs et opérateurs demandent le plus de fiabilité possible dans leurs contraintes financières [108].

Une manière de garantir la fiabilité est la vérification formelle. Pour prouver l'absence de problèmes, la vérification formelle applique de la logique mathématique rigoureuse.

L'état de l'art en vérification formelle consiste en des méthodes qui soit demandent une expertise et un effort substantiels de la part du programmeur, soit ne vérifient pas complètement les programmes. En particulier, la preuve assistée par ordinateur [147], technique avec laquelle un développeur guide un assistant de preuve vers le résultat désiré, demande au programmeur d'annoter le code avec les étapes logiques de la preuve, ce qui est au-delà de l'expertise des développeurs de fonctions réseau. Une autre technique est l'exécution symbolique [124], qui peut analyser le code automatiquement sans effort du développeur mais ne fonctionne que sur des programmes peu complexes [24].

Heureusement, les fonctions réseau logicielles ont une structure particulière :

- Elles contiennent une seule boucle infinie, dont le code est pour la plupart analysable avec de l'exécution symbolique.
- Le reste du code, qui ne peut être analysé avec l'exécution symbolique, correspond à des structures de données qui sont réutilisées par plusieurs fonctions réseau.

Dans cette dissertation, nous présentons Vigor, une nouvelle méthode implémentée par un outil pour vérifier les fonctions réseau logicielles ; Vigor exploite cette structure particulière. Nous faisons deux contributions :

Premièrement, nous démontrons qu'il est possible de développer des fonctions réseau logicielles qui satisfont une spécification formelle tout en affichant des performances compétitives. Nous développons une série d'outils pour développer de telles fonctions en C via une librairie standard de gestion de paquets réseau (DPDK). Vigor fournit une librairie de structures de données vérifiées formellement; cette librairie remplace les parties du code qui ne sont pas analysables avec de l'exécution symbolique. Vigor utilise l'exécution symbolique exhaustive sur le code des fonctions réseau. Vigor contient une nouvelle technique, que nous appelons "preuves paresseuses", pour automatiquement combiner le résultat de l'exécution symbolique et les preuves des structures de données en une preuve que les fonctions réseau implémentent correctement leur spécification.

Deuxièmement, nous démontrons que la vérification *pratique* sémantique (fonctionnelle) de haut niveau de fonctions réseau logicielles est possible. Par pratique, nous entendons une vérification qui (a) ne demande rien d'autre que la spécification, (b) garantit que l'intégralité de la chaîne logicielle est correcte, et (c) permet aux développeurs de vérifier des spécifications partielles s'ils le désirent.

En résumé, Vigor permet la vérification intégrale de fonctions réseau logicielles, avec spécifications complètes ou partielles, en appuyant simplement sur un bouton, sans réduire la performance des fonctions ou la productivité des développeurs.

Pour justifier nos allégations, nous implémentons cinq fonctions réseau, prouvons qu'elles sont correctes, et démontrons que leur performance est aussi bonne que celle des fonctions réseau non vérifiées les plus rapides. Toutes nos fonctions réseau obtiennent une latence médiane de 4,1±0,2 microsecondes et plus de 4 millions de paquets par seconde de débit sur des interfaces réseau de 10 Gigabits par seconde. En matière de productivité de développeurs, une preuve partielle moyenne prend 6.4 lignes de code utilisant le langage de programmation Python, et la vérification de l'intégralité de la chaîne logicielle prend jusqu'à une journée. Vigor est open source et disponible, avec des tutoriels et exemples, sur https://vigor.epfl.ch.

Acknowledgements

To the two people who initiated and shaped the direction of this long and exciting journey, Katerina Argyraki and George Candea, my terrific advisors, thank you. You have taught me many things, bettered me as a researcher, and provided much-needed guidance throughout my Ph.D.

I was lucky to work with many fellow Ph.D. students that made the Vigor project possible. Thank you, Mihai Dobrescu and Volodymyr Kuznetsov, for igniting my research and sustaining it in its infancy. Solal, my second pilot, thank you for being a reliable colleague who can come up with a solution to any problem and implement it. Thank you for your significant contribution that enabled a great deal of this dissertation. And thank both you and Betty for the fun you brought with the numerous board games and HC^2. Rishabh, your research domain flexibility, unmatched versatility, and guidance in the inter-person communication helped me a immensely both in the scientific work and in navigating the social structure of the academic world. I am grateful for our emotional bond and friendship, for your personal support and dedication, for the run exercises and research sprints, fun games, and purpose-in-life discussions. Luis Pedrosa, thank you for your contribution to both the design and the implementation levels of the project and for your professional and personal involvement.

Master's students, Matteo Rizzo, Lucas Ramirez, and Martin Vassor, made an instrumental contribution to the Vigor project. For this, I am extremely grateful.

I thank Jonas Fietz, Sam Whitlock, Mia Primorac, Georgia Fragkouli, Zeinab Shmeis, Yotam Harchol, Lei Yan, and Passindu Tannage for their invaluable input and good company.

To the senior generation, Jonas Wagner, Baris Kasikci, Stefan Bucur, Kristian Zamfir, Radu Banabic, Silviu Andrica, Vitaly Chipounov, Pavlos Nikolopoulos, and Dimitri Melissovas, thank you for setting the spirit, the collaboration dynamics, and for your support up to the very end. I am grateful to my colleagues, Amer Chamseddine, Loïc Gardiol, Adrian Herrera, Bin Zhang, Peter Zankov, Alexandru Copot, Ana Sima, who shared with me the silent hours of chores and loud moments of fun.

I had the support of four resourceful ladies: Céline Brzak, Nicoletta Isaac, Isabelle Coke, Céline Dupuy. I am grateful to then for alleviating the burden of the Swiss red tape and for making my Ph.D. paperwork, travel, and conference attendance feel like smooth sailing.

I am thankful to my thesis committee members for allocating time in their busy schedule to read my thesis and to attend my private defense: Viktor Kunčak, who found an elusive bug in an example code snippet in this dissertation, Mathias Payer, who ran the defense smoothly despite the extraordinary circumstances, Michael Tautschnig, especially for the deep and detailed feedback, Peter O'Hearn, who also helped to shape the Vigor paper [233], Jon Howell, who attended the lengthy private defense, despite the OSDI submission deadline the next day, and my advisors, Katerina Argyraki and George Candea.

I am grateful to Holly Cogliati-Bauereis for proofreading my thesis and teaching me the nuances of English grammar and shades of meaning.

I thank EPFL, the Swiss National Science Foundation, and Intel for funding my research.

Finally, I want to extend my gratitude to my family and especially to my supportive parents Yury and Olga; and to my close friends from MIPT for keeping my spirit upright: Dmitrii Ustiugov, Pavel Zaichenkov, Pavel Yudin, Plotnik Nikolay, and Rustem Feyzkhanov.

Table of Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
DNS	**D**omain **N**ame **S**ervice
ESE	**E**xhaustive **S**ymbolic **E**xecution
FN	**F**alse **N**egative
FP	**F**alse **P**ositive
FW	**F**ire**W**all
IHL	**I**nternet **H**eader **L**ength
IP	**I**nternet **P**rotocol
LB	**L**oad **B**alancer
LOC	**L**ine **O**f **C**ode
LPM	**L**ongest **P**refix **M**atch
MAC	**M**edia **A**ccess **C**ontrol
NAT	**N**etwork **A**ddress(-Port) **T**ranslator(-ion)
NF	(Software) **N**etwork **F**unction
NFV	**N**etwork **F**unction **V**irtualization
NIC	**N**etwork **I**nterface **C**ard
NOP	**N**o **OP**eration
POD	**P**lain **O**ld **D**ata
RFC	**R**equest **F**or **C**omments
SE	**S**ymbolic **E**xecution
SEE	**S**ymbolic **E**xecution **E**ngine
TCB	**T**rusted **C**omputing **B**ase
TCP	**T**ransmission **C**ontrol **P**rotocol
TN	**T**rue **N**egative
TP	**T**rue **P**ositive

Introduction

In this introductory part, we provide the context for the Vigor project described in this thesis. Vigor proposes a solution to the problem outlined in Chapter 1 (Problem Statement) that emerged as software started playing a dominant role in the design of network devices. We then position this thesis with respect to prior and concurrent research in Chapter 2 (Related Work). Finally, we list the ways in which this thesis advances the state of the art in Chapter 3 (Contributions).

Chapter 1

Problem Statement

1.1 ISPs and Enterprises Need and Use In-Network Processing

So-called network middleboxes [41] perform many essential functions in modern computer networks. Firewalls, intrusion prevention systems, and intrusion detection systems improve the security of the network by acting in-line on every packet and by performing threat analysis and deep packet-inspection. Network port/address translators (NAT, NAPT) and load balancers (LB) enable service multiplexing, redundancy, and scaling. WAN optimizers improve bandwidth consumption and perceived latency between endpoints. VPN endpoints provide secure and confidential access to private networks. Protocol converters (e.g., IPv4 to IPv6) enable incremental deployment of new protocols. Usage monitors enable fine-grained billing for the clients.

Although the argument about in-network deployment of such functions is ongoing [203], ISPs and enterprises have been widely deploying middleboxes for at least a decade [69, 93]. Middleboxes have been accused of having multiple drawbacks that stem primarily from the violation of the end-to-end principle [194, 205]. A middlebox can hinder the deployment of new protocols [173, 206], e.g., running HTTP 2.0 across a web proxy designed for HTTP 1.1. Keeping a state related to a connection in a network middlebox introduces an additional point of failure and complicates debugging [41]. Traffic inspection in the network is also susceptible to the usual privacy and censorship concerns [81]. Nevertheless, the security, performance, and scalability benefits seem to outweigh drawbacks [217]. And, presently, middleboxes are ubiquitous in ISP [131, 221, 225] and enterprise [199] networks and even home routers [34, 253].

1.2 Software Network Functions Enable Innovation and Flexibility

Over the past decade, there has been a migration from ASIC-based to software-based implementations of middleboxes, which brings both benefits and drawbacks. The main benefits are shorter development cycles and the flexibility to deploy network functionality on demand [186], which has enabled the vision of network function virtualization [47, 95]. The conventional path of hardware implementation and deployment on custom ASICs incurs a long development cycle [7]. Hardware development involves the development of an expensive lithography matrix that requires high production volumes to offset its cost. The common middle ground, FPGA [7], accelerates middlebox development but comes at a high cost per unit, as compared to ASICs and commodity hardware, and it mandates a development model that is not familiar to middlebox developers. We see the main drawback of software-based implementations as the lack of reliability, which is the characteristic of today's software: for example, bugs, unpredictable behavior, and security vulnerabilities.

We expect software middleboxes to become a standard component of IT infrastructure because the flexibility of software enables rapid innovation [44, 83, 196]. Researchers can develop software middlebox prototypes to support, showcase, and evaluate their ideas at costs lower than baking a new chip. Network-appliance vendors can promptly react to changes in functionality demands. Network operators can rapidly apply security patches and experiment with custom protocols and architectures. When the development of a HW appliance is required, compared to when it is not, all the above takes substantially longer and costs orders of magnitude more.

In our work, we focus on the data plane of software middleboxes i.e., the performance-critical code that implements the core packet-processing functionality of the middlebox; we refer to such code as a "network function" (NF).

Data plane (NF) is the part that needs verification the most and that is the easiest to verify in a middlebox. A middlebox typically comprises other components, such as a control plane and a web configuration interface. The data plane is concerned directly with processing or forwarding the network traffic, whereas the control plane and the manual configuration interface configure the data plane to implement the desired policy. Data plane acts in real-time and pays, with latency and throughput observed by the users, for every instruction it executes. As a result, it cannot afford complex algorithms or unbounded iteration, except for the main loop. The other parts of the middlebox benefit from more lax time constraints yet serve a wider range of functions. These circumstances bring these other parts close to general-purpose software, in terms of complexity and implementation variability. Furthermore, operators tolerate many defects in the web configuration interface because they often do not affect end users. Conversely, the data plane is sufficiently restricted to make automatic verification possible and is very important to get right, as a failure in the data plane often corrupts, destroys, or leaks the user traffic.

1.3 Frequent Software Updates Undermine Reliability

It is generally accepted that software updates bring the risk of introducing bugs [9, 92, 121, 138, 157]. Indeed, about 25% of Linux kernel and PostgreSQL changes [76] and an average of 30% of the changes in a dozen popular open-source projects [123] are considered buggy. Furthermore, 15–25% bug-fixes introduce new bugs [228].

Software evolves constantly, often in ad-hoc ways, hence it is difficult to keep correct. Higher-level abstractions provided by programming languages help developers cope with the rising complexity of the systems they develop but often come with performance penalties unacceptable in the NF domain.

Software updates tend to introduce bugs. Even in such a well-known and respected project as Linux (in particular Debian GNU/Linux), the flow of new critical bugs never dwindles [49, 170, 239].

A shorter development cycle, a lower expertise threshold, and a larger design space, as compared to hardware implementations, all contribute to the proliferation of bugs in software. Software middleboxes are particularly known to cause communication services disruption [182].

However, software-quality-assurance research has developed a large arsenal of tools and methods for code reliability. In the next chapter, we describe the state of the art of such methods by focusing primarily on formal verification.

Chapter 2

Related Work

In this chapter, we briefly discuss software testing in Section 2.1 and introduce the software verification approaches that are used by or compete with Vigor and the main works in the domain. We introduce, in particular, symbolic execution in Section 2.2.2 and theorem proving in Section 2.2.3: we describe their strengths and explain why they do not meet the goals of this thesis. We then describe the major advances in system software verification (Section 2.3) in general, NF verification (Section 2.4) in particular, and network verification (Section 2.5), which is different from NF verification. We conclude with Section 2.6 by explaining the context for this thesis.

2.1 Testing

In this section, we introduce testing, discuss its variants, and show why it is not sufficient for the goal of guaranteeing the correctness of NFs. We explore black-box and white-box testing, with user-supplied and automatically-generated input, and provided and inferred specification. We cover fuzz testing that gained traction recently and various static analyses that enhance testing effectiveness. We conclude by showing that testing is not sufficient to guarantee the absence of bugs in such sophisticated programs as NFs.

A traditional approach to improve software quality is testing. Testing consists of running the system-under-test for some input and checking the produced output. Most non-trivial software projects test their code as a part of their workflow [127]. Because it is usually the approach of choice when it comes to quality assurance, many flavors have been developed.

Depending on how much is known of the interworkings of the system, it can be tested as a *black box* (know nothing) or a *white box* (know everything).

In black-box testing, the test is designed for an abstract specification of the system. For example, an HTTP server is tested against different HTTP messages that comply with the abstract specification of the protocol; an image converter receives a set of tasks to convert images of the supported formats encoded according to the file format specification; and a *map* data structure is tested against the generic sequences of update/lookup operations.

White-box fully testing explores the detailed knowledge of the system implementation. The test can take advantage of this knowledge by exercising particular inputs that engage a specific implementation feature, e.g., filling a map implemented as a growable hash-table up to a point where it reallocates to reduce the load. The test can also use extra channels to fine-tune the system state and configuration, e.g., giving an optional “hint” argument for a map lookup call. Finally, the test can use extra channels to learn more about the execution than just the output. For example, it can instrument the application to check some safety properties or to gather the coverage information.

The next differentiator of testing approaches is the provenance of the test inputs. A test needs the input it runs the system on. Either a human supplies the input, or a machine generates it algorithmically.

Tests with manually created inputs are typically more expensive yet effective at detecting and localizing a bug, more than with automatically generated ones. On the one hand, manually crafted test inputs often require a nontrivial investment of valuable human effort. Such tests usually accompany the code and evolve with it. They can be created by the developers, together with the code, or they can be extracted from customers’ bug reports or feature requests. Automatic testing, on the other hand, consumes mostly machine effort but might fail to explore important behavior cases. Also, a failed test with automatically generated input provides very little information to help a developer to localize the problem.

Fuzzing [22, 71, 150] is testing with automatically-generated randomized inputs. It began as testing on purely random sequences but quickly evolved various strategies to steer the input generation by using the information available. A white-box-style fuzzing can use branch-coverage information to favor inputs that exercise new branches or it can use static analysis to determine pivotal points in the input space.

Fuzzers, such as AFL [231] and libFuzzer [198], usually implement an evolutionary algorithm [151] to select the inputs covering as much of the code as possible based on the feedback coming from the code instrumentation. To avoid instrumentation or gain a better understanding of the structure of the input space, a whole line of fuzzers integrate with symbolic execution (see the next section) [43, 143, 164, 208] and with other kinds of static analysis, e.g., a taint analysis [220]. A black-box-style fuzzing, in order to be able to exercise the core logic of the tested program, might use a certain grammar for the input format to pass the parsing stage (e.g., CSmith [227], LangFuzz [104]), which is hard to do using purely random inputs.

The running environment is a part of the input that might be neglected at first glance. It is often a challenge to test a system that depends on its environment. The environment includes the hardware running the software we test, the operating system and other programs that run in parallel, the computer network if the software uses it, and the deployment topology if we test a distributed system.

Emulating the environment enables us to create an otherwise unlikely situation in order to make sure the system reacts appropriately, e.g., to various faults. *Fault injection* exercises this power by introducing faults into the emulated environment: for example, MEFISTO [113] tests the hardware tolerance to faults in its logical gates, FERRARI [116] and FTAPE [215] flip bits in the CPU registers and memory, Ju et al. [114] inject faults between OpenStack services, FATE [94] forces to fail some of the system calls. All these systems, no matter the input and the environment, need a way to decide whether the resulting behavior of the system is correct.

Validation of the output is as important as the test input to the test effectiveness. As the purpose of testing is to check if the system behaves correctly, we need a way to define the correct behavior.

Writing a correctness specification consumes valuable human resource, hence researchers have been actively looking for alternatives. One way to avoid specifying correctness for a system is to use a weak generic property that is a necessary condition for most systems. For example, crash freedom requires a program to never crash (raise a hardware exception), i.e., to never generate an exception such as the ones due to accessing an invalid memory address or dividing an integer by 0. An alternative is to deduce the specification from the available information such as the documentation (e.g., [213, 218]), available manually-written test cases [192], comparable implementations, previous versions of the system (i.e., differential testing [148]), relevant standards (e.g., [23]), or, for a sub-system (e.g., a function or a class), and its usage patterns (e.g., [192]).

Nevertheless, manual specification is worth the effort more often than it might seem. Specifying the system correctness, besides enabling the testing, forces developers or customers to formalize their understanding of the functionality. This enables multiple parties to agree on what they work on, as well as to explore multiple corner cases that do not come to mind in an informal discussion. According to Boris Beizer, “More than the act of testing, the act of designing tests is one of the best bug preventers known.” [19]

With the combinatorial growth of the input space, it is important to check entire classes of inputs instead of single points. Researchers propose static and dynamic analyses that check the properties of multiple behaviors in a single pass. Notable examples, discussed in the next section, are symbolic execution and model checking.

Testing usually improves the reliability of a system but, alas, cannot guarantee its correctness. When it is possible to exhaust the input space for some systems, e.g., basic logical devices or security-protocol message sequences, the testing becomes a mechanical formal verification. However, in most cases, exploring the whole input space is not feasible, even if we cover entire classes of inputs at a time. As Edsger W. Dijkstra said, “Program testing can be used to show the presence of bugs, but never to show their absence!” [66]

To achieve a complete assurance in the correctness of a system, we need formal verification.

2.2 Formal Verification

Formal verification stems from mathematical reasoning and provides formal guarantees for the correctness of a system. Unlike the testing discussed above, formal verification takes a radically different view of program correctness and proves the absence of bugs rather than their presence. Formal verification methods reason about the properties of all executions at once, without necessarily having to enumerate each one of them explicitly.

2.2.1 Background

The word "formal" stems from "formula", the diminutive of "form", and "verification" comes from "verifico" (lat.) = verus ("true") + facio ("do, make"). Hence, formal verification produces true forms. In mathematical reasoning, we interpret the word "form" as "abstraction". A form abstracts some properties of an object. Formal verification then reasons about these properties, ignoring the remainder ("content"). We are free, however, of the constraints of the physical world and can define the "form" as necessary for a problem at hand.

To illustrate the notion of "form", the essential concept for understanding formal verification, consider for example an ideal gas. The ideal gas law formula $PV = nRT$ essentially deals with only a few properties of the gas: P (pressure), V (volume), n (amount), and T (temperature). It ignores the "contents", all the individual molecules (not even speaking about the elementary particles and their complex interactions) that constitute the gas. Positions, velocities, compositions of these molecules are also known as micro-parameters. Instead, the "form" (pressure, volume, amount, and temperature) summarizes all the properties relevant to our thermodynamic problem and renders it tractable, enabling us to formulate the law [51] expressed in the formula. These four variables are the macro-parameters: Unlike the micro-parameters, they describe the gas as a whole, leaving out information about any of its constituents.

Software formal verification deals with abstractions of computer programs and their state.

The abstract state is the state of the system, relevant to the analyzed program and the considered property. The concrete state of the system is usually more detailed, such as the bit state at each gate, but the abstraction removes unnecessary complexity and makes the problem tractable. In the ideal gas example, the concrete state of the system is the coordinates, velocities, and chemical composition of all the gas molecules; and the abstract state is simply the set of 4 parameters defined above: {P, V, n, T}.

Once the target property and the appropriate abstraction are set, a formal verification practitioner carries out a proof to establish the truth (i.e., Does this program indeed have the property or not?)

We call "proof" an argument that is checked mechanically. We can write the proof on paper using a semi-formal notation that could persuade another human, or we can code the proof in a formal language that a mechanical proof checker would endorse. Although both mechanically and manually checked arguments are proofs, the former is more practical for two reasons. First, proof on paper requires a human expert to spend a prohibitive amount of their time to check the proof for each application anew. Instead, we seek to eliminate the human verification effort. Second, it provides a lesser assurance than mechanical proof because reasoning about software tends to involve a large number of subtle details. In our experience, humans tend to be less reliable and consistent than machines in dealing with detail-oriented tasks.

Considering real-world applications, as is often the case in engineering, we need to be prepared to discuss different forms of approximations that practice, in general, introduces.

According to Rice's theorem [191], a complete and precise analysis for a non-trivial property is not possible. The practitioners have to compromise on the precision or on the completeness of an analysis. The approximation can either introduce impossible behaviors or miss possible ones. If the approximation introduces impossible behaviors we say it "overapproximates" the program. If it misses possible behaviors, it "underapproximates" the program.

To characterize the analysis predictions with respect to the ground truth, software testing researchers established terminology that is summarized in Table 2.1. A testing framework looks for bugs in the programs. A bug is any violation of the specified correctness property such as correct memory usage, well-defined program behavior, isolation between different components, etc. For a testing framework, a positive outcome means a piece of evidence that the property does not hold. It is a bug, or violation of the property, e.g., an incorrect memory access, an undefined behavior, a leak.

In software verification, a positive result should mean a proof that a correctness property holds. However, we keep the terminology inherited from the software testing field. That is why by a negative result we mean that the property holds and no bugs can be found and by a positive result we mean that the property does not hold, i.e., there are bugs.

Table 2.1: The established framework for describing analysis relation to reality.

Analysis Conclusion: "Property..."	"... Holds"	"... Does **Not** Hold"
Property Actually Holds	True Negative (**TN**)	False Positive (**FP**)
Property Actually Does **Not** Hold	False Negative (**FN**)	True Positive (**TP**)

Table 2.1 systematizes the relation between the analysis result and the ground truth. The positivity adjective describes the analysis conclusion. As the property usually asserts the absence of a certain class of bugs, a negative conclusion means the analysis reported no bugs and a positive conclusion refers to a possible bug. The truth adjective is whether the conclusion correctly describes reality.

To be on the safe side, if an analysis does not terminate in the allotted time, we consider it a positive conclusion, i.e., the property does *not* hold for example on one of the unexplored behaviors.

Different analysis goals are incompatible with different kinds of approximations. A *sound* analysis admits no false negatives (FN). Such an analysis is suitable for formal verification, as it is guaranteed to never mistakingly accept an incorrect program. A *complete* analysis admits no false positives (FP). This sort of analysis is safe to include as an automatic judge filtering a repository of third-party applications, as it would never turn down a correct program.

Sometimes, the analysis has to be sound in only certain circumstances and complete in others. For a negated property, a sound analysis becomes complete. If a sound analysis conclusion depends on a negation of a property P, the subordinate analysis for P must be complete. Godefroid et al. demonstrate this practically in [89].

Both soundness and completeness are important, however, for the utility of an analysis. An analysis that always concludes that the property can be violated is sound but not useful. An analysis that always concludes the property is never violated is complete but not useful either.

Finally, the same analysis can be unsound when applied to some program and incomplete when applied to some other program. Such an analysis provides no guarantees, as it is neither sound nor complete. It can still be useful to either find bugs or gain partial assurance in the program correctness, depending on the rate of FNs and FPs.

Now, we turn to abstract interpretation: a class of static analysis techniques that enable formal verification of computer programs such as software NFs. Abstract interpretation describes several static analysis techniques that have been successful in finding bugs and proving their absence, including symbolic execution (Section 2.2.2) and theorem proving (Section 2.2.3).

Abstract interpretation is a computation of the transitions of the abstract state that is conditioned by a program. It was first proposed by Cousot and Cousot [60]. Abstract interpretation follows the control flow of the program (the usual sequence of operations taking into account branching in the code). The control flow often contains loops. To render the analysis finite, Cousot and Cousot propose to iteratively (re)compute the abstract state at each point of the loop until a fixed point is reached. At such a point the recomputed abstract state that does not change, even after yet another iteration of the loop in the given abstraction. To guarantee the existence of the fixed point, abstract interpretation draws ever more generic instances of the abstract state from a partially ordered set with an upper bound—bounded semilattice.

If we forgo the termination guarantee and allow the abstract state of the program to be drawn from any set and not only from bounded semilattices, we obtain symbolic execution [27, 107, 124, 188]. Symbolic execution is automatic. Though invented more than 40 years ago, it has been useful ever since for formal verification [29, 67, 96] and for test generation [5, 10, 36, 38, 87, 132, 195, 197]. Symbolic execution does, however, often fail to terminate in a reasonable amount of time. And this is why practitioners, in most cases, do not run it to completion. The latter compromises both soundness and completeness. Hence, they use it to find bugs in the behaviors of the program that it explores in the limited time it is run for.

Type theories are a well-developed formal verification tool that enables mechanical theorem proving that is sound, complete, and takes a practical amount of machine time (usually seconds). Most programmers are acquainted with types in their basic form, which requires little to no annotations. Taken to the extreme, types can encode almost any property, from secure computation [156, 216] to program termination [223] and complexity bounds [219]. Theorem proving [147] takes advantage of this expressive power and enables users to formulate sophisticated theorems and, by using type-checking algorithms, to prove them in a machine-readable and mechanically checked language [15, 90, 162, 163, 169].

Not all theorem provers are type checkers. For instance, Mizar [13], the longest continuously running proof-assistant project [85], uses rewrite rules at its core. VeriFast [55, 110], a C- and Java-oriented theorem prover, is a symbolic-execution engine at its core. It trades off the automation against expressive power and a much richer abstraction language. Boogie [14], a program verification back-end behind VCC [55], Dafny [136], SMACK [187], and others, uses weakest precondition generation [65]. See [85] for an overview of theorem provers.

Theorem proving is powerful but expensive. It has been instrumental in certifying the correctness of a range of systems [45, 98, 99, 126]. However, theorem proving is usually prohibitively expensive to use for most software developments [39].

Model checking is a verification and a bug-finding method. A model checker verifies that the model of a system refines the provided logical specification that is often formulated in a temporal logic capable of capturing a sequence of facts. The model of the system is a finite state machine. The model checker traverses the state transitions in the model and compares them with the specification. The first flavor of model checking, explicit-state model checking, was started with an objective to verify concurrent systems. Traditional model checking is better suited to verify hardware designs than software, as the former usually have more structured and smaller state space (i.e., micro parameters).

Consider an example code: `int8_t b = input(); int8_t a = (0 < b) ? b : -b;` and a specification `0 <= a`. The total state of the system is 3 bytes: the input and two variables `a` and `b`. An explicit-state model checker explores 2^8 possible inputs and verifies that on every input the specification holds, e.g., it checks `b = 42` $\Rightarrow$ `0 <= 42` and `b = -42` $\Rightarrow$ `0 <= -(-42)`.

Explicit-state model checking [12, 54, 105, 155, 185], although it is inexpensive, practical, sound, and complete, works on systems with a state-space many orders of magnitude smaller than that of a typical software. Explicit-state model checking enumerates all concrete states of the system, i.e., all possible values of all relevant registers, memory, and input sequences. The behavior of stateful NFs, by the means of persistent memory, depends on unbounded sequences of forwarded packets. Such persistent memory is often large and explicit-state model checking tries to enumerate an untractable number of distinct states in the finite-state machine and cannot exhaust it, which is called a "state explosion" problem.

Symbolic model checking [35, 149] comes closer to symbolic execution in generality. In one go, it coalesces the system states using symbolic representation and explores entire state subspaces. Symbolic model checking tools usually represent the system state as ordered binary-decision diagrams (OBDDs) [135] or SMT formulas same as symbolic execution. SMT formulas are logical formulas for which an SMT solver can automatically find a satisfying assignment (see more about SMT in the next section). So in the example above, an SMT-based symbolic model checker considers only two classes of inputs: `0 < b` and `!(0 < b)`. The transitions it needs to verify are therefore `(0 < b)` $\Rightarrow$ `0 <= b` and `!(0 < b)` $\Rightarrow$ `0 <= -b`. It discards both of them using an SMT solver. Symbolic model checking has seen many successes [21, 53, 97, 229]. Unfortunately, it carries over the same practical limitation as symbolic execution (see the next section), i.e., it can only handle automatically the formulas supported by an SMT solver, and we failed to express the abstract state and semantic properties of network functions in such formulas. So symbolic model checking cannot verify stateful NFs.

Considering the advantages and limitations of the listed approaches, we combine symbolic execution and theorem proving in the Vigor approach to achieve both low cost and high power in the domain of software NFs. Vigor applies each approach to the part of the code it is best suited for and combine their proofs.

Even though symbolic model checking could replace the symbolic-execution stage for the stateless part of an NF, it is challenging to integrate with the theorem proving stage of Vigor workflow. As we explain below, the symbolic-execution stage in the Vigor workflow reports the sequences of the stateful library calls for all possible execution paths through the stateless part of the NF. These function-call sequences are essential for checking high-level semantic correctness. Symbolic model checking, unlike symbolic execution, does not provide an abstraction of an execution path necessary for our approach.

2.2.2 Symbolic Execution

An abstraction representing the program state is essential to a program analysis. The right abstraction must capture all the properties of interest. Yet, as the ideal gas metaphor illustrates, the more details the abstraction includes, the harder it is to use.

Naturally, the abstraction best suited for reasoning about a program is specific to the program and the property of interest. For example, a transaction concept could be relevant for database systems but inappropriate for a stream transcoder. The number of instructions executed could be enough for a worst-case execution analysis but insufficient to check array-access bounds.

It takes a software analysis specialist to come up with an abstraction that would best capture the important aspects of the program and enable efficient abstract interpretation.

However, one set of abstractions proves to be particularly useful for a very large set of programs. It comprises a symbolic formula for each individual value used in the program.

Henceforth, we use SE for symbolic execution and SEE for symbolic execution engine, e.g., KLEE [36] or KeY [1].

The formulas have to belong to a specific set of theories that enable automatic solving. A theory describes a language (a set of formulas) and the inference rules over this language. Examples of theories amenable for automatic solving in most practical cases are Boolean arithmetic, bit vector manipulations, integer arithmetic, uninterpreted functions.

Although leaving behind the termination guarantees, this abstraction allows for automatic analysis of relatively large programs. On the minus side, exhaustive enumeration of the program behavior with SE is often infeasible because SE often fails to terminate. On the plus side, it is easy to use. A non-specialist can apply symbolic execution tools to their software, without having to worry about the abstraction choice or adding any, except very primitive, annotations. Such annotations mark the varying input values and express the desired property formulated like another output in the same programming language. Both kinds of annotations can be defaulted to `argc`/`argv` arguments to the `main` function or to contents of the input file and non-crash conditions, such as, e.g., absence of division by 0 and null-pointer dereference.

The automatic satisfiability (SAT, SMT) solvers were developed to accompany SE. SAT (short from "Boolean SATisfiability") solvers are specialized software for the search for a variable assignment that would make a Boolean formula true. SMT (satisfiability modulo theories) solvers take the idea one step further and enable embedding into the Boolean formulas different "theories" (a theory describes the formula language and defines its semantics with inference rules), such as integer arithmetic, bit-vector arithmetic, uninterpreted functions, and quantifiers. The supported theories must allow for automatic reasoning for all or a big number of formulas. For example, Cooper [58] and Hodes [103] present procedures for reasoning about Presburger arithmetic [112, 183].

Proving (un)satisfiability is the first problem to be shown as (co-)NP-complete [57]. However, solvers [16, 63, 72, 82, 153] have made significant progress and continue to evolve the search heuristics. The reader can follow the progress on the online competition platform: http://www.satcompetition.org. There have been 22 competitions since 1992. 55 solvers participated in the competition of 2019 [100].

SMT solvers do all the heavy lifting in symbolic execution: deciding whether a particular judgment is true for all the program outputs that correspond to an admissible input sequence and generating a counterexample for the judgments that do not hold in some cases. Here, we do not further discuss SAT/SMT, as Vigor uses it as is and Yurichev published a thorough hands-on introduction [230].

To get an idea of how SE works, let us consider an example program:

```c
void minmax(int *x, int* y) {
  if (*y < *x) {
    *x = *x + *y;
    *y = *x - *y;
    *x = *x - *y;
  }
}

int main() {
  int a = symbolic(), b = symbolic();
  minmax(&a, &b);
  assert(a <= b);
  return 0;
}
```

Listing 2.1: A simple example sorting two numbers (pointed to by `*x`, `*y`) in-place, and a simple "test harness" that sets up the context for SE.

A symbolic-execution engine interprets this program just like it would run its compiled version, except that some values are *symbolic*. The example comprises two functions: `minmax` that we intend to analyze, and `main` that sets up the context for the SE of `minmax` and checks the outcome. Figure 2.1 demonstrates how SE unwinds the execution tree of the example program.

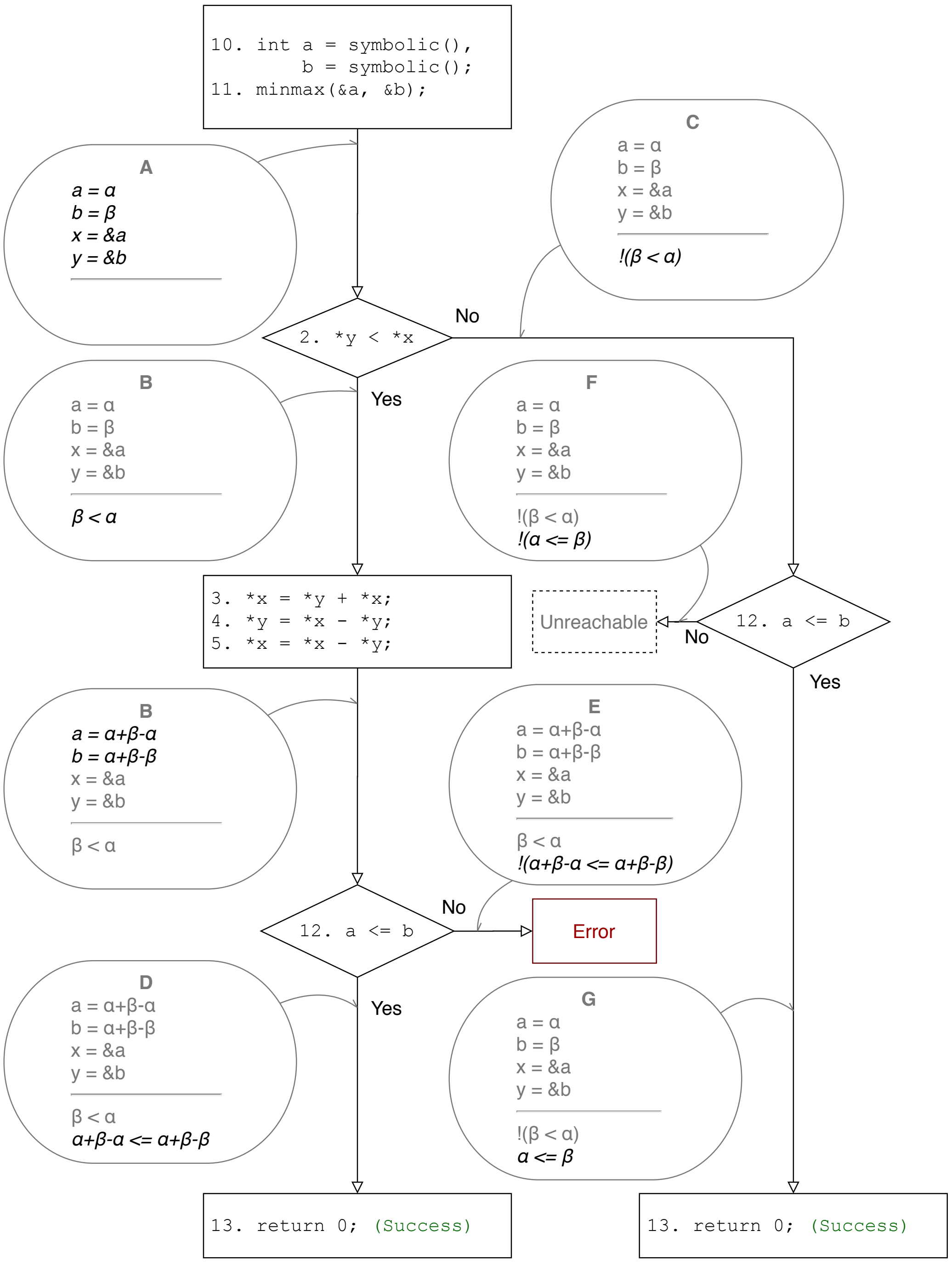


Figure 2.1: Symbolic-execution tree of the MinMax example

According to the ANSI C and later C standards, execution starts from the `main` function (line 9). First, it allocates arguments for the subsequent call. `symbolic()` is an intrinsic function that returns a fresh symbol for each call. As the concept of a symbol is external for the C language, the SEE extends C with an intrinsic function `symbolic()` that enables the generation of symbols when executed by the SEE. Line 10 initializes two variables `a` and `b` with fresh unconstrained symbols α and β. Then it proceeds to call `minmax` with the pointers to these two local variables.

The execution context (also known as symbolic state store) is the full machine state, including the global and local variables, and the registers (including the next instruction pointer, IP). We follow only parts of the context important for this walkthrough. The important parts in the execution context at this point (the context A) are the variables: two symbolic: `a =` α, `b =` β; and two concrete: `x = &a, y = &b`.

`minmax` starts with a branching point steering the execution depending on a symbolic (i.e., involving symbols) condition. In this case, it is $\beta < \alpha$. The SEE replicates the execution context A through the action called *forking*. In one copy, let's call it B, the SEE assumes the condition to be true, adds it into the assumptions heap, and jumps into the `then` branch. This condition together with the accumulated assumptions heap is called a *path condition* (PC) [124]. Here PC is "$\beta < \alpha$". In the other copy, let's call it C, the SEE assumes the condition to be false, adds it into the assumptions heap, and jumps into the `else` branch. Hence, in the alternative copy, the PC here is "`not (`$\beta < \alpha$`)`". The `else` is empty and omitted here, thus the execution finishes `minmax` function and continues in `main` with the assertion. The SEE then repeats the analysis for each copy in the same way.

For B, the execution continues from line 3, where the programmer intended to swap `a` and `b` not using temporary variable. At the end of the block, B has `x =` $\alpha + \beta - \alpha$, `y =` $\alpha + \beta - \beta$.

The second branching point raises from the assertion on line 12. The SEE transforms the assertion into a branch that leads to a special `error`-inducing operation. This branch forks the execution context into D and E.

D explores the execution where the assertion is not triggered. It inherits B and adds another statement to the path condition: "`not (`$\alpha + \beta - \alpha <= \alpha + \beta - \alpha$`)`". It then reaches the `return` instruction on line 13 and terminates.

E explores the error possibility. The SEE adds "$\alpha + \beta - \alpha = \alpha + \beta - \alpha$" to the inherited path condition $\beta < \alpha$. It then invokes an SMT solver to check the feasibility of the branch.

Depending on the SMT solver and the formulas encoding it can detect an overflow in $\alpha + \beta$. Although in the two's-complement representation $\alpha + \beta - \alpha = \alpha$, in the C standard it is undefined behavior if the sum overflows. The SMT returns a counterexample that triggers an overflow: $\alpha = 2^{31} - 1$, $\beta = 1$.

We left the last execution context C at the assertion on line 12. As for B above, the SEE forks the C into G and F, adding "$\beta <= \alpha$" and "`not` ($\beta <= \alpha$)" statements to their respective path conditions. Note how the same line of code now produces the different formulas because in this execution context variables have different values.

Once the SEE checks the feasibility of F, the SMT solver returns UNSAT, meaning that the two assumptions in the path condition: (`not` ($\alpha < \beta$) and $\alpha <= \beta$) are contradictory. As a result, the SEE drops F as unfeasible.

G safely reaches the `return` on line 13 and completes the analysis.

The major drawback of SE is the rapid growth of the number of paths with the size of the program. In our small example, we explored 4 paths total (1 unfeasible, 1 error, 2 successful terminations), but this number grows combinatorially with each branch and each loop iteration, which often leads to a large or infinite number of paths.

For this reason, traditionally practitioners run SE for a limited time to try to find as many bugs as possible. The SEE uses heuristics to choose the most promising paths. A better heuristic can improve the SEE efficiency and increase the assurance in the correctness of the analyzed software, but it does not enable the SEE to exhaustively run all the paths. In the networking domain, SE serves to find susceptibility to manipulation attacks [130] and susceptibility to performance attacks [175]. To automate testing OpenFlow applications, Canini et al. [40] combine SE with model checking.

Exhaustive SE is still possible, but in a limited domain that does not allow unbounded loops and requires a relatively small number of branches. These conditions are often found in operating system kernels [161, 201], security monitors [160], file systems [200], and kernel-bypass NFs (Vigor).

Exhaustive SE enables verification. SE is complete by design, as long as the underlying SMT solver provides sound counterexamples. Running it exhaustively makes SE sound because it enumerates all the possible behaviors of the software. Having checked properties over each behavior enables SE to guarantee the correctness, regardless of the behavior that ends up playing out. In other words, exhaustive SE verifies the software with respect to the properties of interest.

Interaction with the environment is essential for most realistic software but poses a challenge for SE. We call the environment everything that is outside of the system being analyzed—the outer world. All programs that interact with the user or a device, or involve multiple actors, are connected to the outer world. Yet, SE effectively executes multiple runs in one go, whereas, the environment normally supports only one interaction at a time. For example, a network interface card (NIC) does not understand a symbolic packet, and users are confused if they see or need to enter symbolic formulas instead of concrete numbers. Regular environment agents, such as a NIC, terminal, or a disc drive, do not support symbolic values that capture multiple runs.

Most realistic programs interact with the environment they run in by calling the corresponding I/O functions, or by accessing the memory mapped into hardware registers: for example, reading system time with `clock()`, reading or writing a file with `fread`/`fwrite`, or setting a network card mode to `promiscuous`, by writing `1` to the corresponding memory address. When the SEE simultaneously simulates multiple runs of the software, it has to handle such calls and memory accesses.

One strategy is to make the analysis underapproximate the set of behaviors by concretizing the symbols involved. When the SEE encounters a system function call, it performs a query to the solver to obtain feasible concrete values for each symbol involved in the arguments and forwards the call to the host system. The approach was introduced in the DART [87] project and named *concolic* execution (*conc*rete + symb*olic*) in CUTE [197]. EXE [37] and SAGE [88] made further advances, where the latter crossed symbolic execution with fuzzing [22, 71, 150].

This strategy has the advantage of universality and no overhead: It enables SEE to handle any call, as long as the host system supports it. It demands no additional effort from the SE practitioners. It preserves the valid concrete state; hence, it produces false positives only if the *sequence* of system calls is infeasible given the concrete arguments to those calls.

However, there are serious drawbacks:

- Underapproximation: by concretizing the symbols, SEE explores only one possible behavior out of many. Although it might suit bug finding, which is best-effort and can not guarantee that the program is bug-free, it is unacceptable for verification, that would miss possible behaviors.

- Side effects: most system calls affect the host system. If the SEE uses a single-host environment to handle system calls, there would be three consequences:

  (a) The software under SE could corrupt the host system. This can be solved by running the SEE in a VM or another sandbox.
  (b) The side effects could interleave or accumulate in an undesired sequence that was never intended by the software under SE because the SEE explores multiple behaviors simultaneously and out of order and it is not aware of implicit dependencies between the system calls. The most often occurring case, rather innocuous but demonstrative, is terminal output (`cout`) that turns into an unreadable hodgepodge.
  (c) The system call could require input from an external source such as a user, a cryptographic random number generator, a network, or a sensor. In this case, SE would likely stall, if no precaution is taken. If the SEE wants to continue, it has to invent a valid input value and that requires an understanding of the context.

S2E [48] addresses the issues (a) and (b) through the whole-system simulation. Conceptually, it treats the operating system, together with the target program, as the target, and it keeps a snapshot of the entire machine state and executes an independent instance of a VM for each path/execution context. In practice, S2E explores that the differences between VM states are small compared to their full state and makes a SE of the whole-system feasible.

Symbolic hardware [132, 189] partially addresses (c) by extending S2E with models of hardware devices. Symbolic hardware enables SE of device drivers, the critical and hard-to-get-right domain of programs. Unfortunately, symbolic hardware restricts the analysis to exploring only the device behavior patterns envisioned and expressed by the developer of the symbolic model. As symbolic models operate in the language of symbolic formulas supported by the SEE, they can express limited behavior patterns and have to approximate it.

The alternative strategy is to model all the interactions with the environment [36]. A SEE developer needs to write a library of models for each such potential interaction. A model of a function F is a function that approximates F. The model is typically simpler than F and SEE can symbolically execute the model when the analyzed program calls it. For example, a model of user input could return a fresh unconstrained symbol. A set of models can accurately account for the context and implicit dependencies between system calls, can work with symbolic arguments, thus eliminating the need in concretization, and can evince multiple behaviors, by forking the symbolic state and returning symbolic values. Therefore, if all symbolic models do not underapproximate the behavior of their respective system calls, i.e., they explore all possibilities of the real execution, the SEE can exhaustively explore behaviors of the software and provide verification guarantees.

However, due to the wide variety of system calls, this strategy is labor-intensive and error prone [11] in the general case. For the code that is not amenable to SE, Vigor uses approximate best-effort models that are easier to write and then compensates for their inaccuracy with automatic validation (see Part A). Vigor uses symbolic hardware for environment interaction.

Challenges in Applying Symbolic Execution to Vanilla Code

Applying symbolic execution to the NF code poses three challenges: dealing with unbounded loops, handling state manipulating code, and formulating the semantic property.

Consider an example of an iteration:

```
#define MAX_N (1000)
int sqrt(int N) {
  bool occurred[MAX_N] = {false};
  int root = N/2;
  do {
    occurred[root] = true;
    root = (root + (N/root))/2;
  } while (!occurred[root])
  return root;
}
```

Listing 2.2: A simple unbounded loop that causes path explosion in SE.

The unbounded loop engenders potentially an infinite number of paths to explore (each with one iteration more to explore), thus rendering SE potentially infinite. Naïve SE tries to enumerate all paths hence might never terminate. Such growth in the number of paths, which makes SE run for long or infinite time, is called *path explosion* [24] and is analogous to state explosion in model checking.

Adding computational power to combat path explosion improves the bug-finding performance but hardly helps verification. Bucur et al. in Cloud9 [33] enabled parallel SE, they made it possible to scale SE horizontally and to increase the rate at which it explores the paths by adding worker machines. As the SEE explores more potentially buggy paths, it is a big improvement for finding bugs. However, it does not enable verification of a program that contains even a single unbounded loop.

State merging is another possible way to address path explosion, by leading either to an approximation of the program state or to state explosion. Compositional SE [86] and state merging [133] merge execution paths that converge to the same program point. After merging two paths, the SEE makes a decision about the combined state. It can either keep the two different versions and endure the handling overhead that this larger representation incurs or it can generalize multiple versions with a shorter representation, thus approximating both behaviors.

Loop havoc-ing [137] relaxes state abstraction and renders the analysis finite. Transforming a loop by using loop invariants [59] enables the analysis to model all the iterations of the loop in a single one. Havoc-ing replaces all the variables that might change (such as `root` and contents of `occurred`) in the loop by unconstrained symbols at the beginning of the iteration. Note, that it does not replace `N` in Listing 2.2, as `N` stays unmodified throughout the loop.

However, even the single-pass SE of the loop body takes significantly longer than executing the loop body once (which would be an underapproximation). This slow down is due to another classic SE challenge, "symbolic indexing" [24, 88, 197]. When indexing an array with a symbolic variable (or when dereferencing a symbolic pointer), the constraints on the program state become complex [32, 204, 214] because the symbolic pointer or index could be referencing many different locations. The issue is alleviated for symbolic indices of a reasonably sized array or when a type information helps to reduce the number of possible targets for symbolic pointers. In a low-level untyped representation, the possible target for a symbolic pointer is any memory location. The complexity of the constraints overwhelms the SMT solver, and the SEE slows down or stops. The alternative approach, forking SE for possible index values, blows up the number of states. In general, SE is not practical for the code with symbolic pointers or offsets. Alas, a stateful C code almost always has some of these. Also for verification purposes, we cannot afford concretization as in SAGE [88], BitBlaze [204], or MAYHEM [43] to preserve soundness.

The reader might have noticed a potential aliasing issue in Listing 2.1. If `minmax` is invoked with the same pointer as both arguments (`x = y`), it results in nullifying the value `*x`, which is presumably not the desired outcome.

Two circumstances prevent the exhaustive SE to detect this issue: the underspecification (the assertion on line 12 allows `a == 0 && b == 0`) and the practical difficulty of aliasing handling in SE. Aliasing is a (potentially undesired) overlap in the memory referenced by two pointers. SE, on its own, usually, does not detect potential aliasing because doing this involves symbolic pointers and is very expensive. The user has to be watchful for the places where the issue might arise and check the cases of aliasing separately.

The NF code, however, uses pointers sparingly and only for two purposes: state manipulation and packet manipulation. This characteristic enables us to implement crude memory ownership [52] mechanism to protect the few pointer usages in the verified NFs.

Another inconvenience of SE is the limit on the properties it can reason about, which must be expressible in the theories supported by the underlying SMT solver. Classic SE fully relies on the underlying SMT solver for dealing with formulas. This constrains the range of NF properties it can prove. For example, as far as we know, there is no support for a longest-prefix-match theory in the modern SMT solvers, hence SE cannot express the semantics of an IP router. Whereas, theorem proving enables the user to define their theories best suited to express NF semantics.

In summary, SE is a method that can analyze large codebases with a high degree of automation, yet exhaustive SE suitable for verification is challenging for many common programming idioms. Stemming from concrete interpretation, SE can handle realistic applications and is an excellent bug-finding tool. To prove the absence of bugs we need to run SE exhaustively. The following constructs pose challenges for exhaustive SE: loops without static bound, input-dependent array indices, and complicated arithmetic that defies modern SMT solvers.

In the next section, we consider another abstraction for the program state, the one of theorem proving. This abstraction brings about a different point in the tradeoff between automation and versatility. Theorem proving imposes fewer restrictions on the program state representation. The theorem proving practitioner pays for this liberty with the effort necessary to guide the proof checker.

2.2.3 Theorem Proving

In this section, we consider software verification by automated theorem proving: a program analysis that is less autonomous yet more versatile than SE. We use it to handle the code that is difficult for SE. Theorem proving applies powerful mathematical reasoning to the software. Note, that unlike SMT solvers, which derive efficient heuristics to find a solution for a set of equations, theorem proving seeks mathematical rigor in the proofs it produces, even at the cost of proof complexity. Often, reasoning about software involves a large number of details and corner cases; theorem proving practitioners use mechanical proof checkers to rigorously verify proofs of a range of properties. Proof assistants and other analysis tools help practitioners to carry out the meticulous proofs and, sometimes, to generate them completely automatically.

In this dissertation, by the term "theorem proving" we mean software verification using automated theorem proving in first-order separation logic [168], an extension of the Hoare logic [102].

Many successful proof checkers [15, 162] are based on the concept of a type. A type is a set of values with a common property. A type system is another example of abstraction for the program state. It enables the developer to specify the domain for each result, i.e., the set of possible values each result can take on.

Here, by "result" we mean the value computed after evaluating an expression.

For example, for this function: `float div(int a, int b) { return a/b; }` the type signature suggests that the function takes two values that range over integer numbers and returns a floating-point number.

`div` is trickier than it looks. The function seems intuitive at first glance, however, its result might surprise inexperienced C programmers, as it actually always returns integer values of a `float` type.

The trick here is that C compiler silently performs typecasts that are not always intuitive. As this occurs automatically, it can mismatch the programmer's intent.

For our example, there are two options:

```
//(A)
float div(int a, int b) { return (float)(a/b); }

//(B)
float div(int a, int b) { return ((float)a)/((float)b); }
```

Although option B might seem more appropriate for this context, the compiler chooses option A: this option performs approximate integer division, discards the non-integer part of the result, then converts it into a real number.

More strongly typed languages chose a different strategy. In every ambiguous case, they require the developer to supply annotations to clarify their intention. For example in OCaml the developer might write[1]:

```
(* compile-error *)
let div (a : int) (b : int) : float = a / b

(* compile-error *)
let div (a : int) (b : int) : float = a /. b

(* (A) *)
let div (a : int) (b : int) : float = float_of_int(a / b)

(* (B) *)
let div (a : int) (b : int) : float = float_of_int(a) /. float_of_int(b)
```

Notice, how the explicit use of annotations forces the developer to think about integer-to-float conversion and to express precisely whether they want to perform a floating-point division or an integer division.

Considering the types abstraction, we can call the type of a function (or its signature) its abstract definition. For some purposes, all we might need is to know whether we can provide integer arguments to the function and what kind of value it returns, and we can ignore the function implementation details.

Let us say we want to statically verify the correctness of a `printf` expression:

```
printf("The quotient of %d / %d = %f", a, b, foo(a, b));
```

After parsing the static string, the compiler can now check that the types of the arguments correspond to the format specifiers.

In this abstraction, the specification of `div` is `int, int -> float`. However, even the `B` implementations do not comply with this specification because they break on `b = 0` (undefined behavior in C or throw an exception in OCaml), whereas the specification does not admit undefined behavior nor exception.

A more expressive type system lets us prohibit the `0` value for the second argument, e.g., `int, int\0 -> float`.

Modern type systems have developed enough expressivity to encode statements and reason in higher-order logic.

[1]OCaml uses "`/`" for integer division and "`/.`" for a floating-point division.

In the interpretation of Brouwer-Heyting-Kolmogorov (BHK) [30, 101, 129], a constructive proof of a logical formula is an example of a value of the type that corresponds to the formula.

Usually, theorems require a set of preconditions and guarantee a set of postconditions. In BHK interpretation, a proof of a theorem is a function that takes values that represent proofs of preconditions and returns a value of the type that corresponds to the postcondition.

The BHK interpretation (also expressed as Curry-Howard correspondence [61, 62, 106]) enables mechanical proof-checking through following simple type-rules that correspond to a particular first or higher-order logic [15, 162].

Now, if a verification practitioner adds the following two steps, she can scale proof checking to claims about large computer programs.

1. Modularize the proofs. The developer verifies each function/procedure/routine once, against an abstract specification, also called a contract. Whenever the proof checker meets a function call, it uses only the specification of the callee function, instead of interpreting its body again.
2. Separate memory into disjoint regions. Potentially, any routine can depend on and affect any memory location. To avoid specifying the entire memory of the machine in each contract, we introduce access tokens. A routine is permitted to read from or to write to a particular domain, only if it has acquired the corresponding token either as a contract precondition or as a result of another call. This flavor of logic, tailored for verification of computer programs, is called separation logic [167, 190]. It enables concise contracts that specify only the memory related to the routine.

To reduce the human effort required, many automated theorem provers sacrifice the strength of the provided guarantee by involving SMT solvers [55, 110, 125, 136]. These provers trade the low annotation burden for transparency, the low trusted computing base, and, potentially, the expressiveness. If an SMT solver (whose effectiveness is limited theoretically [57]) fails to see the satisfiability of a formula, the user can only guess how to help it. The SMT solvers here are trusted agents, as the client prover trusts their result. The client prover's language might be limited to the theories supported by the underlying SMT solver.

The past decade has been especially fruitful for theorem proving applied to large systems. Beringer et al. [20] and Zinzindohoué et al. [236] used it to prove functional correctness and cryptographic properties of OpenSSL implementation and a library of cryptographic primitives HACL*. Various theorem-proving tools back the correctness of multiple low-level systems, such as seL4 [126] kernel, Hyperkernel [161], and Komodo [78] security monitor. Hawblitzel et al. [98, 99] used theorem proving to verify network applications written in Dafny [136], a high-level programming language with a builtin specification syntax and a proof checker.

Challenges in Applying Theorem Proving to Low-Level Code

To certify source code with theorem proving the verification specialist:

1. formulates the preconditions (predicate that is required to hold before the code starts executing),
2. formulates the postconditions (predicate that is guaranteed to hold after the code execution finishes),
3. formulates loop invariants (predicate that holds on before and after every iteration), and
4. inserts the special annotations that help the proof checker to track the abstract state transformations across the code.

Once the necessary annotations are in place, an automatic proof checker fills the remaining gaps and endorses the proof (the certificate). A proof assistant facilitates the interaction between the verification specialist and the proof checker. It reports the proof steps that are too large for the proof checker to cover automatically or incorrect. For each proof step, the proof checker needs to explore a limited search space, which might be done with SE.

Theorem proving traditionally requires uncommon expertise and an extra effort from developers. To use the traditional theorem-proving techniques, the developer has to perform the four steps listed above, in addition to developing the application. All of these steps require an understanding of the first-order separation logic and experience in reasoning about program properties and interacting with a proof assistant. Moreover, the extra effort exerted in these four steps is typically larger than the development of the application, and it grows faster [145]. For example, the seL4 [126] correctness proof is 20 times larger than its implementation. Ironclad [99] report 4.8 proof-to-code ratio in Dafny [136] language, due to their proof automation with the Z3 [63] SMT solver. A functionally certified keyboard driver [176] has the same ratio (4.8) in VeriFast, that uses the in-house SMT solver, Redux, derived from Simplify [64]. Finally, our library libVig, also verified correct with VeriFast, consists of roughly 12.5 times more proof annotations than executable code.

Typically NF developers do not possess the required formal methods expertise. The cost of the correctness proof, which is many times over the development code, is prohibitively high for the software NF domain.

2.3 Verifying Systems Software

Systems-software verification is characterized by its scale. To capture the entire system, it encompasses a multitude of moving parts: either high-level actors, as in distributed systems, or hardware components and subcomponents as in operating systems and supervisors.

Steady progress in formal methods and software verification has enabled system verification, as the systems referenced below demonstrate. The typical tradeoff in verification is the strength of the property vs. ease of use.

On the ease-of-use end of the tradeoff are verification efforts that provide low-level properties such as crash freedom (the program never crashes) and memory safety (the program uses only the allocated memory when it is allocated) for NFs. Dobrescu et al. [68] use exhaustive symbolic execution to prove these properties in simplified or stateless NFs. Panda et al. [171] ensure such properties by using a safe language, Rust [146]. Vigor can be viewed as a generalization of this prior work. It enables higher-level semantic properties to be verified, makes it more accessible to developers, and covers the entire NF stack.

seL4 [126] (certified operating system kernel), Cogent [3], FSCQ [45] (certified FS), and CompCert [209] (certified compiler) prioritize the strength of the proven property. All systems are proven correct with respect to a complete semantics. However, they all require the use of high-level (sometimes esoteric) programming languages and deep expertise in verification, which we consider a high barrier to adoption.

As pioneers in the domain, these systems enabled the formal mechanical verification of complex software systems. Now, we need to make this verification practical and to enable broad adoption. Our motivation for Vigor is to make the verification of NFs accessible to most (ideally all) NF developers.

Our goal with this work is to enable the development of software NFs that are guaranteed to be semantically correct and that offer competitive performance and to preserve developer productivity.

Our work falls in the general area of "data-plane verification." This term is typically used to denote two different types of approaches: One category of work treats the combination of the *configured* data planes of network devices in a network and their interconnection as one large data plane and reasons about network properties (reachability, loops, etc.); e.g., VeriFlow [122], Header Space Analysis [119], or FlowChecker [2], we refer to this as "network verification." An orthogonal category reasons about properties of the data-plane *software* running on individual devices, and reasons about software properties (e.g., crash freedom, bounded execution time, memory safety [68], or functional correctness [140]); we refer to this as "NF verification."

Verification of the Bottom of the Stack

As a subdomain of systems verification, a bottom-of-the-stack verification involves reasoning about the details usually abstracted for most developers, as well as verification experts. Such details include the CPU state, virtual memory organization, call stack layout, IO bus protocols, and peripheral device interfaces.

Much work has been devoted to verifying the lower layers of the stack, including verified operating systems [126, 161], drivers [12, 132, 180, 189], and file systems [200]. Our work builds on some of these techniques to fully verify the entire NF stack.

2.4 Verification of Software Network Functions

In NF verification, the goal is to prove that a particular property (e.g., there exists no input packet that can trigger a buffer overflow in the NF) holds for all networks and workloads, i.e., regardless of how the NF is configured, connected, and used.

Note, that *NF verification* is different from what is traditionally understood by *network verification*. As network verification concerns the properties of entire computer networks, it often simplifies the representation of their individual nodes.

The success of *network verification* depends on the success of *NF verification*: Network verification relies on models of the NFs that compose the network, whether these models are informally captured in an RFC or more formally in a SEFL model [211], NICE model [40], etc.

A model-based proof that a packet always reaches a destination is trivially invalidated by an implementation bug in a middlebox. Such bug can, for instance, cause that packet to be dropped, in violation of the model. There are ways of testing whether such a model is faithful to a given implementation [211], but there is a large gap between testing and verification: A successfully tested model can still miss behaviors that occur in the implementation. When used in conjunction with network verification, NF verification can, however, ensure that an NF implementation deployed in the real network is indeed faithful to the model used for verifying the network, and therefore the conclusions drawn by network verification are valid.

In contrast to network (configuration) verification, there is much less work on NF verification [68, 140, 210].

The closest work to ours is that of Dobrescu et al. [68], they verify NFs written in Click [128], including a NAT. They prove the low-level properties of crash freedom and bounded execution for these NFs. To reason about entire Click pipelines, their approach relies on exhaustive SE of individual Click elements and an on-demand composition of the resulting analysis summaries. Like Vigor, their approach puts all the state in special data structures, however, it does not verify the data structures themselves or that the NF uses them correctly. For this reason [68] cannot prove semantic properties. The step forward that enables such proofs is the "lazy proofs" technique we describe in the next part.

Gravel [235] is a project with goals similar to ours and is being developed in parallel. They provide a way for verifying, in an almost fully automated manner, NFs written in Click. Out of the Click elements analyzed by Zhang et al., 45% can be automatically verified, with additional 33% being verifiable after minor modifications. Similarly to Vigor, Gravel symbolically executes the stateless part of the NF.

The fundamental difference between Gravel and Vigor is what happens with code that cannot be symbolically executed. In Vigor, such code resides in libVig, and we *verify* it with VeriFast. In Gravel, such code is *assumed correct* and modeled with specifications encoded in satisfiability modulo theories using uninterpreted function theory. This means that no amount of verification can provide a full guarantee of correctness. A further consequence is that Gravel cannot perform full-stack verification. Gravel's trusted computing base (TCB) includes their data structures, the Click runtime, DPDK, and the operating system, whereas in Vigor these components are all formally verified or absent (Click runtime).

Click modules provide a level of abstraction higher than Vigor data structures, which can make writing NFs easier. However, these modules offer limited reuse (see Figure B6.3 in Part B), and NF developers commonly need to implement new Click modules when implementing new NF functionality [171]. To use Gravel, the developer must not only ensure that this new module is amenable to SE but also write a full-blown specification for the module; this can often be a showstopper. Another difference between Vigor and Gravel is the language used to formulate the abstract semantics of data structures. Gravel formulates the data-structure semantics in the language that an SMT solver understands directly. The use of this language enables more-automated reasoning but might prove incapable of expressing certain data structures (e.g., an LPM lookup table). Vigor, for the same purpose, uses first-order logic, which is more expressive and less automated, and overcomes the automation gap by utilizing domain-specific heuristics implemented in our custom-designed tool—called Validator.

Specification of Software Network Functions

An NF can be said to be correct only relative to a specification. Semantic verification establishes refinement from the NF specification to the NF implementation. In other words, the proof of correctness demonstrates how the given specification abstracts the given implementation.

For verification to be useful for increasing the trustworthiness of the implementation, the specification needs to be simpler than the implementation. The specification should omit the implementation details and retain the properties essential for the NF semantics. In other words, the specification should be more abstract than the implementation.

We argue that an NF specification language must be expressive, concise, familiar, and formal. First, the language has to be *expressive* to cover a range of NFs. A part of the expressivity is extensibility: With the growth of the framework applicability to more kinds of NFs, the language should organically include new primitives to express those NFs. Second, a *concise* language enables a quicker inspection of and, therefore, a higher level of trust in the NF specification. Third, the NF community is more likely to adopt a *familiar* language for an easier transition to it. Finally, to perform mechanical verification, a machine needs to parse and reason precisely about the specification constructs. This is only possible if it is written in a *formal* language.

Vigor proposes a specification language, a Python dialect, for expressing NF semantic properties. Others have explored the benefits of rigorous specifications for Internet standards: Musuvathi et al. [154] model-checked the Linux TCP implementation against a formal specification. They used C to encode the TCP reference model. Although expressive and familiar, C is arguably not concise and not entirely formal, as it permits undefined behavior. Bishop et al. [23] rigorously specified the sockets API and the TCP/UDP protocol stacks for specification-driven testing. They used a higher-order logic provided by the HOL [91] system. HOL language is expressive, concise, and formal, but not familiar. NF developers are usually unfamiliar with logical languages.

Another possible source of the specifications is the set of models used for network verification. For example, Stoenescu et al. [211] focus on network verification, nevertheless, they rely on detailed NF models more than other work in this area, and they test but do not verify, their models for faithfulness to the corresponding NF implementations. Their tool, SymNet, introduces a new domain-specific language SEFL designed to model the NFs for the purpose of network verification. SEFL is concise and formal. Although it is not familiar, it includes under 20 instructions, hence an NF developer will have no problem learning it. However, SEFL's expressiveness is rather limited. In particular, it fails to encode any dynamic NF state persistent between packets. It is also unclear how easy it is to extend SEFL. Nevertheless, once translated into our Python dialect, such models can serve as a semantic specification for a Vigor NF, thus eliminating the need for trust in the faithfulness of the former and flawlessness of the latter.

Once the NFs are proven to be functionally equivalent to their specification, network verification can use them to establish properties about entire networks.

2.5 Network (Configuration) Verification

In network verification, the goal is to demonstrate that a particular property (e.g., that a packet with certain header features always reaches a given destination) holds in a specific network with particular NFs that are configured and connected in a particular way.

For example, given the configuration in Figure 2.2 a network verification tool can determine whether a TCP packet with port 80 from Host A can reach Host B.

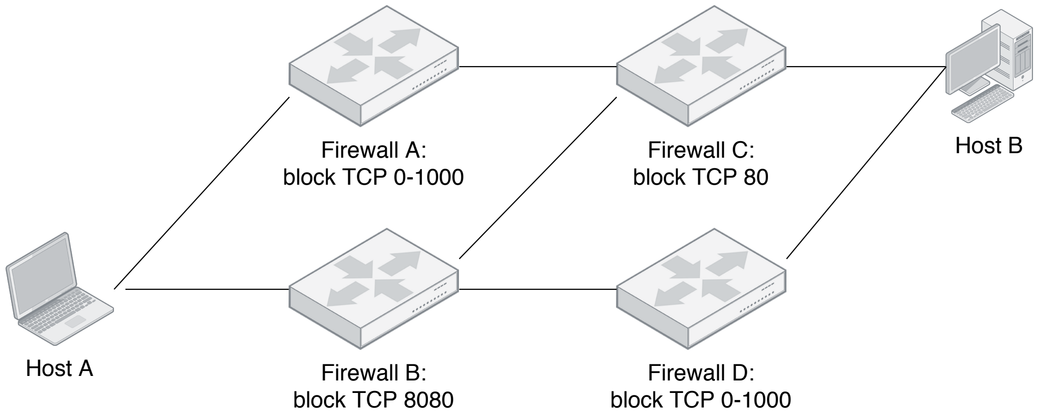


Figure 2.2: Example network topology

Even in such a simple topology, the answer is not obvious. Realistic networks involve hundreds of thousands of nodes: hosts, as well as intermediate hops, such as routers, firewalls, tunnel endpoints, load balancers, NATs, network accelerators, etc. Moreover, these nodes constantly go online and offline and change configuration and connectivity.

For such a scale, network operators depend on tools. The tools must be able to promptly answer queries for any existing network or to ensure certain properties for network configurations and topologies that are about to be deployed.

There is a rich body of work on network verification [6, 17, 18, 80, 84, 172, 179, 222, 226].

Often, network verification guarantees that a certain network behaves correctly, given its topology, configuration, and NF *models* [77, 79, 118, 119, 122, 141, 142, 193, 212, 224].

By automatically building models for NFs, recent work [152], developed concurrently with this thesis, attempts to bridge the gap between network and NF verification. These models are then used to prove network-wide properties.

An intermediate line of research focuses on the properties of NFs in a particular context. Such properties include protocol interoperability [174], implementation equivalence [31, 134], and performance contracts [109].

2.6 Our Target Application Domain

In our view, practicality consists of the simultaneous achievement of two design goals: competitive performance and low verification effort. The latter has three components: writing the code in a way that can be verified, writing the proof, and verifying the proof. Vigor supports C, hence it does not impose an undue burden on the writing of the NF code. Therefore, we focus on devising a technique for productively writing and for verifying realistic NFs.

Dobrescu et al. [68] verified a stateful NAT, but they proved only the low-level properties (crash freedom and bounded execution) and assumed memory safety, as a result, they did not encounter some of the harder challenges of stateful NFs, such as handling low-level memory manipulation and proving semantic correctness of the state evolution. We resolve these challenges, without placing on operators the burden of writing or adapting models, and keeping the NF implementations' performance in the same ballpark as that of non-verified NFs.

As we want to offer competitive performance and preserve developer productivity, we consider NFs that are implemented in C on top of a standard I/O framework, such as the Data Plane Development Kit (DPDK) [249], because this is the way high-performance NFs are developed today.

We have described well-known verification approaches that could be relevant to this task, such as whole-program theorem proving or per-path/per-state techniques such as symbolic execution and model checking. With the former, verifying a property proof is relatively fast, but writing the proof is a slow, often manual, job. With the latter, verifying a property in a real NF can take long, even forever, due to path/state explosion, but it is easy to automate. To verify a stateful NF, neither approach seems practical on its own.

Nevertheless the impressive advances in systems verification hint that it is possible to achieve the challenging goals incumbent upon Vigor.

In the next chapter, we formulate our contributions and the thesis they support.

Chapter 3

Contributions

In this thesis we make the following contributions:

- A composite verification approach that enables push-button, full-stack, and pay-as-you-go verification of high-performance NFs.
- A library of verified data structures for the NF domain that enable verification of stateful NFs and a framework implementing the proposed approach.

Thesis

Data-plane network functions can be written in a way that enables regular software developers to maintain competitive performance and to formally verify them in a practical fashion: full-stack (no trusted software at run time), pay-as-you-go, and push-button (no additional effort other than formulating the correctness condition).

Part A

VigNAT: Verifying a Stateful Network Function

Our goal is to build a practical NF verification framework, which requires that we simultaneously achieve both rigorous verification and high performance. We propose an approach for how to achieve this. Before developing the idea into a complete toolchain, we first perform an initial experiment by applying our proposed approach to a Network Address Translator (NAT), a prototypical stateful NF. In this part, we describe this experiment, find that it is indeed possible to simultaneously achieve both verification and high performance. Then, in the next part, we describe the full-blown approach. The essence of the ideas and results described in this part were published in our VigNAT paper [234] that is a stepping stone towards the full-blown Vigor [233], described in Part B.

VigNAT is a NAT implemented in the Vigor framework and proven correct with respect to our formalization of RFC 3022.

Chapter 1

Overview

The rationale behind our approach is that different verification techniques are best suited for different types of code.

Symbolic execution (Section 2.2.2 of the introduction) on the one hand is easy to use but fails to scale to realistic software systems and to most useful semantic properties. The beauty of symbolic execution lies in its ease of use: it enables automatic code analysis, hence it can be used by developers without verification expertise. As shown in the introduction, the challenge with symbolic execution is its notorious lack of scalability: applying it to real C code typically leads to path explosion [68, 211]. The part of the real NF code that typically leads to an unmanageable path explosion is the one that manipulates the state.

Theorem proving (Section 2.2.3 of the introduction) on the other hand, scales up to software systems of realistic size and complexity, but requires substantial human effort and expertise. Examples of seL4 [126], Ironclad [99], CompCert [209] and others show that theorem proving is mature enough to handle realistic systems. However, these same examples demonstrate the amount of verification expertise and effort, which we find unacceptable for regular NF developers.

In our approach, we decompose the proof into parts and proceed on each part with whichever technique is suited for that part; after this, we stitch the proofs together. We posit that most NFs consist of one part that is common across many NFs (hence making it worthwhile to invest manual effort in proving its correctness) and another part that is different in each NF (and thus its verification should be automatic). We also posit that, over time, NFs will converge to using a stable, common set of data types to encapsulate NF state, and that the difference between

NFs will result primarily from how their SE-friendly code employs these data types. Therefore, we split the NF code into two parts:

- a library of data structures that keep all the "difficult" code that we then formally prove to be correct using theorem proving, which takes time and formal methods expertise but can be amortized if the library is re-used across multiple NFs, and
- a SE-friendly code that uses the library that we automatically and quickly verify using symbolic execution.

The challenge is stitching together the results of the two verification techniques. We developed *lazy proofs*, a way to automatically interface symbolic execution to a proof checker based on separation logic. We built a Validator that implements this technique and glues sub-proofs together into the final proof that VigNAT implements the NAT RFC [207].

We envision the software development process with Vigor that will involve four distinct developer roles with a clear separation of concerns:

- libVig developers who implement, optimize, and verify the components used in NFs
- standards developers who write rigorous specifications in our dialect of Python of the public standards
- NF developers who implement these standards with verified NFs
- NF operators who deploy the NFs in their data centers, run Vigor to verify the NFs they deploy satisfy the standards and, possibly, formulate and verify some deployment-specific properties.

The first role requires expertise in software verification and formal methods, but the time and effort investment can be amortized across the many NFs that share common components. The second role generally demands a specification process more rigorous than the one we observe in the usual standardization bodies, such as IEEE and IETF. The additional effort can be offset by a quicker convergence due to a more formal language and is amortized by many implementations of the same standards. Developers and operators in the two latter roles, however, should need little to no expertise in verification. They are the true beneficiaries of Vigor, as they can now write code that they can prove correct with relative ease.

1.1 Illustration of the Workflow

We illustrate the use and functioning of Vigor with a trivially simple NF that implements the Discard Protocol [181]: an infinite loop receives packets from one interface, discards the ones sent to port 9, and forwards the rest through another interface.

Listing 1.1 shows the original naïve implementation of the protocol that uses a ring queue.

```c
#define CAP 512

int main() {
  struct packet packets[CAP] = {};
  int begin = 0, queue_len = 0;

  while(1) {
    if (queue_len < CAP) {
      int end = (begin + queue_len)%CAP;
      if (receive(&packets[end]) && packets[end].port != 9)
        queue_len += 1;
    }
    if (0 < queue_len && can_send()) {
      send(&packets[begin]);
      queue_len -= 1;
      begin = (begin + 1)%CAP;
    }
  }
  return 0;
}
```

Listing 1.1: Naïve discard ring implementation.

Network interaction occurs via three functions:

- `receive` non-blockingly reads an inbound packet and stores it in the output argument, returning success or failure;
- `can_send` checks if a new packet can be sent; and
- `send` sends the packet pointed to by its argument.

Vigor proves that this NF never crashes (raises a hardware exception; an example of a low-level property) and that it never yields a packet with target port 9 (an example of a semantic property). For the latter, the gist of the proof is to show that the code never pushes onto the ring packets with target port 9, and that the ring never alters the stored packets; these two properties imply that a popped packet can never have a target port 9.

1.1.1 Preparation of the Code

To verify that this NF does not crash, we could use the approach of Dobrescu et al. [68] and run exhaustive symbolic execution. The symbolic execution (SE) technique entails using a symbolic execution engine (SEE) to explore all paths through the code and to check whether any failure conditions hold; in our case, crash conditions.

We describe the SE technique in Section 2.2.2 of the previous part. To recap, the inputs to the program are marked "symbolic," meaning that they subsume all possible input values, then the SEE simulates the execution of the program while propagating the symbolic inputs along the dataflow paths into the program's variables. When facing a branch (e.g., an `if` or a `while` loop), the SEE clones the program state and takes separately each feasible path. Branches impose constraints on variable values (e.g., an `if` statement imposes restrictions on the values of the condition variable along the `else` branch); the SEE collects these constraints along each path. When evaluating a branch condition, the SEE asks an SMT constraint solver which of the branches is feasible, given the constraints collected so far on that path prefix.

We could use a SEE such as KLEE [36] to exhaustively symbolically execute the original application code (Listing 1.1); and once all paths are found to be crash-free, the crash-freedom proof is complete. To perform SE, it is necessary to provide accurate symbolic models for the three network calls. Simple models suffice, such as `receive` generating a new symbolic packet in `dst`; `can_send` forking the program state into two states for the interface being busy and idle, respectively; and `send` doing nothing.

Alas, naïve symbolic execution will never terminate due to the path explosion. Path explosion results from the combination of outcomes of all branch conditions: Every branch condition doubles the number of paths, and every loop multiplies it by a factor; in our case, the infinite loop gives rise to an infinite number of paths.

Vigor applies loop havoc-ing, and thus it considers the loop body only once. For this, it determines the set of the variables that might change. Vigor computes a fixed point of an iterative analysis of the loop body where it compares all the values at the beginning and at the end of a loop body and grows the set of variables to havoc (and the degree of approximation of the loop) until it covers all the variables that might change. Havoc-ing replaces all the variables that might change in the loop by unconstrained symbols at the beginning of the iteration. In our example, this would be the variables `begin` and `queue_len` and the contents of `packets[0.. CAP]`; but not the value of the `packets` pointer itself because it never changes. Step 1 enables Vigor to reduce each problematic loop to a single generalized iteration that enables reasoning in one pass.

This change makes SE feasible, albeit impractical: for `CAP=5`, it takes 10 seconds to exhaustively SE the code, for `CAP=30` it takes 15 minutes, for `CAP=60` it takes 3 hours, and so on. Once we replace the infinite loop with a single iteration, we have to assume (due to havoc-ing) the `begin` and `begin + queue_len` indexes could point anywhere in the `ring` array. Consequently, the code in the loop iteration accesses memory using symbolic indexes, which causes a combinatorial explosion in SE.

In Vigor, we circumvent the problem of symbolic indexing by encapsulating the program state into well-defined data structures and by putting them into a "library of components" (e.g., the `ring` in Step 2 (Section 1.1.3)). This works well for NF development because the NF code uses a relatively small set of standard structures, such as ring buffers, hash tables, and longest-prefix-match tables. We provide the SEE with symbolic models of these abstracted components that "hide" symbolic indexes and pointers from the SEE. After the substitution in Step 2 (Section 1.1.3), the exhaustive SE takes less than 100 milliseconds regardless of the value of `CAP`.

SE is a first step on the way of a complete proof. After the exhaustive SE finds no crashes in the application code, we need to verify for each component (e.g., for the ring component) that (1) the actual component implementation does not crash itself and follows the declared API specification, and that (2) the models used during SE faithfully approximated the same specification.

Our verification process requires loop invariants to reason about the infinite event processing loop that is responsible for the processing of each packet or a timer tick.

In our experience, these loop invariants are of three kinds:

- a constant value in a specific memory location, which the initialization code sets and no loop iteration ever changes,
- data-structure invariants, inherent to libVig data structures, and
- data-structure entry constraints. These constraints are either the lower and upper bounds or a set of forbidden values for a specific integer field of the entry.

We exploit this domain property in order to automatically compose the complete loop invariant. Vigor automatically infers the first kind of invariants. libVig developers provide the second kind of invariants for each data structure as part of its specification. We consider the third kind of invariants as an extended type declaration for the data-structure entries.

For the discard-protocol implementation, the state specification looks like Listing 1.2. Vigor uses this macro to instantiate the libVig data structures and supply the parameters to their invariants. The macro takes the following arguments:

- the name of the pointer used to access the data structure: `r`
- the type of the data structure: `ring`
- the capacity of the data structure: `CAP`
- the C-struct holding the data structure entry: `packet`
- the constrained field of the entry: `port`
- the forbidden value for the field: 9

We could imagine extracting the state configuration automatically from the code using existing techniques [75, 177].

```
NF_STATE(r, ring, CAP, packet, port, 9);
```

Listing 1.2: State specification for the discard protocol implementation.

Vigor expands the `NF_STATE` macro into the two formulations of the loop invariants in formal logic and in C (Listing 1.3).

```
/*@
  fixpoint bool packet_constraints_fp(packet p) {
    switch(p) { case packet(port): return port != 9; }
  }
  @*/

static bool packet_constraints(struct packet* p) {
  return p->port != 9;
}
```

Listing 1.3: Loop invariant in C and VeriFast languages, generated from the macro on Listing 1.2.

The NF developer does two extra actions relative to writing standard code: She annotates the event processing loop and encapsulates state in libVig data structures that Vigor can reason about. Listing 1.4 shows the refactored implementation, which uses a designated ring buffer (`r`) that is accessed through four API calls. It also features the annotated event loop (`VIGOR_LOOP`).

```c
#define CAP 512

NF_STATE(r, ring, CAP, packet, port, 9);

int main() {
  struct packet p;
  struct ring *r = ring_create(CAP);
  if (!r) return 1;
  while(VIGOR_LOOP(1))
  {
    loop_iteration_begin(&r);
    if (!ring_full(r))
      if (receive(&p) && p.port != 9)
        ring_push_back(r, &p);
    if (!ring_empty(r) && can_send()) {
      ring_pop_front(r, &p);
      send(&p);
    }
    loop_iteration_end(&r);
  }
  return 0;
}
```

Listing 1.4: Vigor-ready version of the discard ring implementation.

Verification consists of three steps:

1. Writing contracts for non-SE-friendly code and proving them
2. Exhaustive symbolic execution of the SE-friendly code
3. Lazy model validation and a check against the semantic specification.

1.1.2 Step 1: Function Contracts and Proofs

For each method of a libVig data type[1], the libVig developer writes a contract, i.e., a formal specification of what the method guarantees. She also writes a formal proof that the implementation of that method satisfies the contract. This is a significant undertaking but can be amortized across the potentially many NFs that use the same data type. Listing 1.5 shows the contract and the implementation of the `ring_pop_front` function that removes the packet at the front of the

[1]We interchangeably use the terms "data structure" and "data type" with the understanding that the data structure state is encapsulated behind a well-defined interface.

ring[2]. The contract says that this function never damages the ring, removes the packet at the front of the ring, and honors certain constraints that hold for all packets in the ring, as long as the ring was in a good state and honored these constraints before the function was called. In the contract, `packet_constraints_fp` is an abstract function, i.e., the contract says that `ring_pop_front` honors *any* packet constraints, as long as these hold before it is called. The NF developer can provide desired constraints when using libVig through the type specification in the `NF_STATE` declaration; the provided constraint (Listing 1.2, expanded into Listing 1.3) conveniently serves as a loop invariant too.

```
void ring_pop_front(struct ring* r, struct packet* p)
/*@ requires ringp(r, ?packet_constraints_fp, ?lst, ?cap) &*&
             lst != nil &*& packetp(p, _); @*/
/*@ ensures ringp(r, packet_constraints_fp, tail(lst), cap) &*&
            packetp(p, head(lst)) &*&
            true == packet_constraints_fp(head(lst)); @*/
{
  //@ extract_first(r);
  struct packet* src_pkt = r->array + r->begin;
  p->port = src_pkt->port;
  r->len = r->len - 1;
  r->begin = r->begin + 1;
  if (r->cap <= r->begin) {
    r->begin = 0;
    //@ stitch_with_empty_overflow(r);
  } else {
    //@ stitch_with_empty(r);
  }
}
```

Listing 1.5: Excerpt from the implementation of `ring_pop_front()` and its formal contract. `packet_constraints_fp` parameter (the invariant condition on all the packets in the ring) is defined in the Listing 1.3.

1.1.3 Step 2: Exhaustive Symbolic Execution

Next comes the automatic analysis of the SE-friendly code:

1. Vigor replaces all function calls that access state or interact with the network with calls to a symbolic model. For example, the symbolic model for `ring_pop_front` (model (a) in Listing 1.7) returns a packet with fully symbolic content (i.e., a packet whose content could be anything whatsoever) constrained via `packet_constraints` to have its target port different from 9.

[2]This implementation is only an illustrative example. In our verified NAT, and most real implementations, we would not copy packets field by field (Listing 1.5) but rather return a pointer to the packet.

Despite its simplicity, this model captures all the behavior of `ring_pop_front` that matters in our context; specifically, it never yields a packet with target port 9.

2. Once all function calls have been replaced with calls to symbolic models, Vigor symbolically executes the resulting code.

Even though Vigor symbolically executes real C code, this step terminates quite quickly because the models are SE-friendly and have few branching points (like the one in Listing 1.7), and because the loop annotations help prevent unnecessary unrolling. This exhaustive symbolic execution, in case of success, has two outcomes, both assume the symbolic model is valid: First, it proves that the target low-level property (i.e., that the NF cannot crash for any input) holds. Second, it produces all the feasible function-call sequences that could result from running the code, along with the constraints on the program state that hold after each call. Listing 1.6 shows one such call sequence (generated by Validator) that results from an execution in which the ring is full.

Note that the `ring_pop_front` model does not even check whether it is called with an empty ring or not. While this is an underapproximation in a general case, the example application makes sure to call pop only if the ring is not empty. Therefore the check is not necessary, and the underapproximating behavior is not triggered. The next step makes sure no underapproximating behavior is triggered for the particular NF.

Also, this model clearly overapproximates a “ring”, e.g., `ring_empty` might return true even right after `ring_push`. This means, it might have caused false positives; but, as long as it does not, there is no need to complicate it. For the sake of the example conciseness, we hardcoded the packet structure into the ring component itself. In a real component, this detail is abstracted and supplied as a parameter to `NF_STATE`.

Now an SEE can analyze the code in an instant. As the reader can see, the model does not maintain any state. It only `ASSERT`s that any pushed packet satisfies the invariant, and it `ASSUME`s that every popped packet does.

1.1.4 Step 3: Lazy Model Validation and Specification Check

Finally, Vigor closes the proof gaps introduced by using loose models in the previous step and checks that every function call sequence complies with the semantic specification.

The verification process in Step 2 is speculative on the validity of the data-structure models for the given NF. This validity means that the output of the model is a superset (in the sense of constrained symbolic state) of the output that the actual implementation could produce.

Vigor runs Validator for each function-call sequence and checks two speculative assumptions:

1. Each function call is legit. For each call in the given sequence, Validator inserts the ancillary lemmas, such as `open loop_invariant(_)` on line 5 of Listing 1.6. A proof checker verifies that precondition for each function call is satisfied, e.g., in the `ring_full(arg1)` call on lint 6, the ring pointed to by `arg1` is a valid ring or the ring pointed to by `arg1` is not empty in the `ring_pop_front(arg1, &(arg2))` call on line 13. The proof checker reports a possible violation if it cannot prove that the precondition of the contract is satisfied for a certain function call.
2. The model executed for each function call correctly overapproximated all the possible outcomes of the function. In other words, the symbolic model used to produce speculative proof via symbolic execution in Step 2 was, in retrospect, valid *for that call.* For example, consider the call to `ring_pop_front` (Listing 1.6 line 13): Vigor extracts the constraints on the symbolic program state that held directly after the model was symbolically executed in Step 2, it inserts, right after the call, an assertion for this path condition, and it asks a proof checker to verify that this assertion is compatible with `ring_pop_front`'s contract. The proof checker concludes that it is, i.e., that the output of the model (a packet whose target port can be anything but 9) is a superset of the output specified by the function's contract hence also of the function's implementation (as Step 1 proved that the implementation satisfies the contract).

Vigor also checks a property required for semantic correctness of the NF as a whole:

3. The *complete* function-call sequences, i.e., sequences representing a complete iteration of the packet-processing loop, satisfy the semantic specification. Vigor verifies that the observable action, such as packet forward (recorded from the `send()` call) with particular header modifications or packet drop,

and the NF state update, both conform to the user-provided specification. In the drop-protocol example, the specification requires all the packets with a target port 9 are dropped and all the other packets are forwarded unmodified. To verify this, Validator translates the Python specification into the proof-checker annotation language, using the state declaration to connect the specification state and the implementation state and detects packet send/receive calls to match the corresponding events in the specification. This check is not shown in the example, as it represents an incomplete function-call sequence.

```c
struct ring* arg1;
struct packet arg2;

loop_invariant_produce(&(arg1));
//@ open loop_invariant(_);
bool ret1 = ring_full(arg1);
//@ assume(ret1 == true);
bool ret2 = ring_empty(arg1);
//@ assume(ret2 == false);
bool ret3 = can_send();
//@ assume(ret3 == true);
//@ close packetp(&(arg2), packet((&(arg2))->port));
ring_pop_front(arg1, &(arg2));
//@ open packetp(&(arg2), _);

//@ assert(arg2.port != 9);
```

Listing 1.6: Example function-call sequence that results from Step 2, annotated by Vigor with an assertion of a path constraint.

```c
// Model (a)
void ring_pop_front(struct ring* r, struct packet* p) {
  FILL_SYMBOLIC(p, sizeof(struct packet), "popped_packet");
  ASSUME(packet_constraints(p));
}

// Model (b)
void ring_pop_front(struct ring* r, struct packet* p) {
  FILL_SYMBOLIC(p, sizeof(struct packet), "popped_packet");
  // No constraint on the packet's target port.
}

// Model (c)
void ring_pop_front(struct ring* r, struct packet* p) {
  p->port = 0;
}
```

Listing 1.7: Symbolic models of `ring_pop_front`.

1.1.5 Dealing with Invalid Models

An invalid model causes either Step 2 or Step 3 to fail, but it never leads to an incorrect proof. An invalid model either underapproximates the behavior, i.e., it is unsound or overapproximates the behavior to the point, where it prevents the tool from proving the semantic property that holds for a more precise model (incomplete). For example, model (b) in Listing 1.7 is too abstract for our purpose: it returns a packet whose content could be anything at all, including having a target port 9. This is an "over-approximating" model in verification speak. If Vigor uses this model, Step 2 might fail, if the NF has an operation that depends on the port is different from 9 (e.g., send checks it with an assert). Another consequence of an over-approximating model is that SEE explores more executions than necessary, as it could enable some infeasible executions. Although, in our simple example this does not occur. Conversely, model (c) in Listing 1.7 is too specific for our purpose: It always returns a packet with target port `0`, i.e., it is an "under-approximating" model. If Vigor uses this model in Step 2, then Step 3a fails: Recall that, for each call, Vigor obtains the path constraints that held right after the model was symbolically executed in Step 2, and inserts, right after the call, an assertion for these path constraints. With this model, the assertion would be `//@ assert(arg2.port == 0)`. The proof checker cannot confirm that this assertion is always true because `ring_pop_front`'s contract (Listing 1.5) specifies a wider range for `arg2.port` than just `{0}`.

1.1.6 Proving Conformance to RFC Semantics

The Vigor toolchain and approach enable us to verify rich semantics derived, for example, from RFCs. In our case, we prove that our NAT box conforms to the NAT RFC.

Further properties of interest can also be derived during the specification process. Developers for standards bodies can, for example, prove that compliance with the standard's contract guarantees immunity from a specific attack vector. With such a proof in place, any resulting RFC conformance certificate for specific NFs would also transitively imply that the NF is immune to the said attack vector.

Let us now follow the `port != 9` condition across the framework. During exhaustive SE, SEE first learns the constraint in the `if` condition (Listing 1.4). Here, SEE only records it in the call sequence for model validation, and, because our model of `ring_push_back` discards the packet being pushed, it does not use it further in the SE. However, the call to `ring_pop_front` reintroduces this constraint formulated in `packet_constraints` (Listing 1.3), according to the model on Listing 1.7 (a).

We enable SE to verify the NF code as described above while verifying the library of components with separation logic: two different verification techniques for two different parts of the complete application. Doing the library verification requires deep expertise in program verification, but our effort is amortized across all NFs that use the library.

Vigor makes it possible for NF developers to verify their code without having to understand verification. Note that the invariants generation for data-plane software turns out to be automatic and to require only the advanced specification of the data structure entry types.

Theorem proving serves two purposes in the framework:

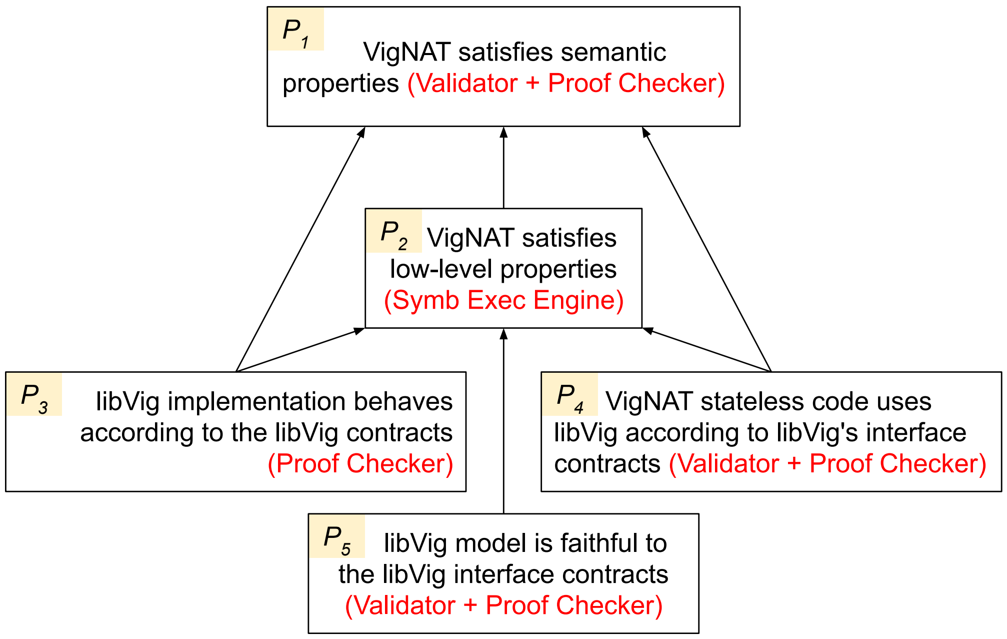
- Verification of the ring implementation. During this part, it proves that `ring_pop_front` guarantees the popped packet to satisfy the `packet_constraints_fp` (Listing 1.5).
- Validation of the call sequence. It uses the component contracts to validate that the models explored all possible outcomes of each call into libVig API and that each call sequence matches the user-provided NF specification, either the complete or partial (Listing 1.6).

The libVig component developer produces several symbolic models that capture the component semantics in more or fewer details for different use cases. For a particular use case, an NF, a particular model might be inappropriate. An inappropriate model cannot compromise the verification result because Vigor detects both over- and under-approximations. Vigor enumerates the available models to find the one suitable for the particular NF.

1.2 General Proof Structure

The proof of VigNAT correctness consists of five sub-proofs, shown in Figure A1.1. The top-level proof objective P_1 is to show that VigNAT exhibits correct NAT semantics. The proof of P_1 makes three assumptions:

- First, the code must work properly in a basic sense, such as not crashing and not having overflows (P_2).
- Second, the implementations of the library data structures must work as specified in their interface contracts (P_3), e.g., looking up a just-added flow should return that flow.
- Third, the SE-friendly part of the NF must use the data structures in a way that is consistent with their interfaces (P_4)—e.g., a pointer to the flow table is never mistakenly passed in as a pointer to a flow entry. Assuming $P_2 \wedge P_3 \wedge P_4$, the Validator produces a proof of P_1 that is mechanically verified by the proof checker.



P1 VigNAT satisfies semantic properties (Validator + Proof Checker)
P2 VigNAT satisfies low-level properties (Symb Exec Engine)
P3 libVig implementation behaves according to the libVig contracts (Proof Checker)
P4 VigNAT stateless code uses libVig according to libVig's interface contracts (Validator + Proof Checker)
P5 libVig model is faithful to the libVig interface contracts (Validator + Proof Checker)

Figure A1.1: Structure of the VigNAT correctness proof. $P_i(X) \leftarrow P_j$ symbolizes that the proof of property P_i is done by X under the assumption that P_j holds.

These three assumptions must, of course, be proven. To prove that VigNAT satisfies low-level properties—P_2 in Figure A1.1—Vigor symbolically executes the SE-friendly code and checks that the properties hold along each execution path. For this to scale, we employ abstract symbolic models of the data structures from the library. Therefore, the proof of P_2 must assume that these models are correct (P_5), that the data structure implementations satisfy their interfaces (P_3), and that the SE-friendly code correctly uses the non-SE-friendly data structures (P_4). If any of these three assumptions are missing, the proof will not work[3]. To prove P_3,

[3]It might seem strange that assumptions P_3 and P_4 are needed both for the proof of low-level

we employ relatively straightforward (but still tedious) theorem proving to show that the library implementation satisfies the contracts that define its interface.

It is in the proof of P_4 and P_5(in Figure A1.1) that we find a second scalability benefit of the lazy-proofs technique: Not only does it enable us to stitch together proofs done with different tools (thereby enabling us to employ for each sub-proof whichever tool offers the optimal benefit-to-effort ratio) but it also makes it possible to get away with proving weaker properties. For example, instead of proving that P_5 is universally true and then using this proof to further prove P_2, we first prove P_2 *assuming* P_5 and afterward prove only that P_5 holds for the *specific way* in which the proof of P_2 relies on P_5. This use-case-specific proof of P_5 is easier than proving P_5 for all possible use cases, moreover, P_5 might be false for some cases, not exercised by the given NF, and this is sufficient for the purposes of P_2. Hence we do not require the correctness of the models.

Note, that the models are proven valid on a per-case basis, the framework might fail to automatically prove the correctness of a particular NF due to a lack of a suitable symbolic model. In such a case, the NF developer will need to either write the symbolic model or request a libVig developer to write one.

1.3 Proven Properties

Listing 1.8 shows the full VigNAT specification, our formalization of the NAT RFC [207]. The original specification proven in [234] consists of 300 lines of VeriFast code. The current version, presented here, expresses the same semantics in our Python dialect [254] and it is almost six times shorter. We describe the Python dialect in detail in Appendix B.

Listing 1.8 starts by declaring the NAT state: flow table `flow_emap`, and some constants on the first 4 lines. Next, for each received packet, it expires all the flows from the table that were inactive for the past `EXP_TIME`. On lines 9–11 it unpacks the protocol headers (in the reverse order). `on_mismatch` is the outcome VigNAT is expected to produce if the received packet does not have the requested protocol header. `([],[])` prescribes to drop the packet (forward to an empty list—`[]`—of devices, with an empty list of headers). Next, it checks that the packet belongs to TCP or UDP. Line 15 branches on whether the packet comes from the

properties and that of semantic properties, but this is because “satisfying interface contracts” relates both to high-level interface semantics and to basics such as proper data encapsulation.

external or an internal network. Lines 16–32 handle the packet from the external network: search for the corresponding flow (line 17) and either drop the packet (line 23) or forward it (lines 25–30). The forwarding specification has the form of a tuple `(<devices to forward to>, <modified headers>)`. Lines 34–53 handle the packet from an internal network. The handling is similar to the handling of an external packet but it tries to allocate a new flow (lines 44–53) if the packet does not correspond to any flow in the table.

In addition to proving NAT semantics, we prove that VigNAT is free of the following undesired behaviors: buffer over/underflow, invalid pointer dereferences, misaligned pointers, out-of-bounds array indexing, accessing memory that is not owned by the accessor, use after free, double free, division by zero, problematic bit shifts, integer over/underflow, and type conversions that would overflow the destination. Proving these properties is essentially proving that a set of assertions introduced in the VigNAT code—either by default in the KLEE SEE [36] or using the LLVM undefined behavior sanitizers [158, 159, 237]—always holds.

Finally, as VigNAT maintains its state in libVig data structures, we also prove that it uses these data structures correctly, i.e., that the data structures' pre-conditions are satisfied.

```
from state import flow_emap
EXP_TIME = 10 * 1000
EXT_IP_ADDR = ext_ip
EXT_PORT = 1

if a_packet_received:
    flow_emap.expire_all(now - EXP_TIME)

h3 = pop_header(tcpudp, on_mismatch=([],[]))
h2 = pop_header(ipv4, on_mismatch=([],[]))
h1 = pop_header(ether, on_mismatch=([],[]))
assert a_packet_received
assert h1.type == 8 # big-endian 0x0800 -> IPv4
assert h2.npid == 6 or h2.npid == 17 # 6/17 -> TCP/UDP
if received_on_port == EXT_PORT:
    flow_indx = h3.dst_port - start_port
    if flow_emap.has_idx(flow_indx): # Flow is present in the table
        internal_flow = flow_emap.get_key(flow_indx)
        flow_emap.refresh_idx(flow_indx, now)
        if (internal_flow.dip != h2.saddr or
            internal_flow.dp != h3.src_port or
            internal_flow.prot != h2.npid):
            return ([],[])
        else:
            return ([internal_flow.idev],
                    [ether(h1, saddr=..., daddr=...),
                     ipv4(h2, cksum=..., saddr=internal_flow.dip,
                                         daddr=internal_flow.sip),
                     tcpudp(src_port=internal_flow.dp,
                            dst_port=internal_flow.sp)])
    else:
        return ([],[])
else: # packet from the internal network
    internal_flow_id = FlowIdc(h3.src_port, h3.dst_port,
                               h2.saddr, h2.daddr,
                               received_on_port, h2.npid)
    if flow_emap.has(internal_flow_id): # flow present in the table
        idx = flow_emap.get(internal_flow_id)
        flow_emap.refresh_idx(idx, now)
        return ([EXT_PORT],
                [ether(h1, saddr=..., daddr=...),
                 ipv4(h2, cksum=..., saddr=EXT_IP_ADDR),
                 tcpudp(h3, src_port=idx + start_port)])
    else: # No flow in the table
        if flow_emap.full(): # flowtable overflow
            return ([],[])
        else:
            idx = the_index_allocated
            flow_emap.add(internal_flow_id, idx, now)
            return ([EXT_PORT],
                    [ether(h1, saddr=..., daddr=...),
                     ipv4(h2, cksum=..., saddr=EXT_IP_ADDR),
                     tcpudp(h3, src_port=idx + start_port)])
```

Listing 1.8: Complete VigNAT specification.

Chapter 2

Verifying Non-SE-Friendly (Stateful) NF Code with Theorem Proving

VigNAT consists of SE-friendly application logic that manipulates the state stored in data structures, such as hash tables and arrays, provided by the Vigor library (libVig). For example, in Section 1.1, we placed incoming packets into a `ring` data structure from libVig. Generally speaking, the SE-friendly NF code should be free of any dynamically allocated state and complex data structures. It can retain the basic program state, such as statically allocated scalar variables as well as `struct`s of scalars, or even arrays indexed by concrete and statically predictable values.

Dealing with the explicit state in the verification of imperative, non-typesafe programs is hard, mainly due to the difficulty of tracking memory ownership and type information, as well as disentangling pointer aliases. For example, the question of which memory a `void*` pointer could ever reference is often undecidable. Functional, type-safe languages (Haskell, ML, etc.) are appealing for verification, but to us, it was paramount to both support C (preliminary evidence [232] suggests C be widely popular among NF developers) and enable verification of a fully stateful NF. We accomplish both by encapsulating the NF state behind libVig's interface and adopting a disciplined use of pointers. While this approach is not compatible with all software, we believe it is a good match for NFs.

Besides data types for NFs, libVig also provides a formal interface specification that defines the behavior of these data types, along with a proof that the libVig implementation obeys the specification. To enable the symbolic execution of SE-friendly NF code, libVig also provides symbolic models of its data types. We verify libVig once, and the proof carries over to any NF that uses the library.

2.1 libVig Implementation

Competitive performance is an important design goal, and libVig is a good place to optimize for performance. A key design decision we made is to *pre*allocate all of libVig's memory. While this lacks the flexibility of dynamic allocation at runtime, it offers control over memory layout, making it possible to control cache placement and save the run-time memory management overhead. The cost of the preallocation is negligible (e.g., VigNAT's peak resident set size is 27MB during our experiments), and we believe preallocation is fully compatible with how real NFs use state. Static allocation allows for a simpler check for memory safety, as the memory is shared between parts of the code that are handled by different verification approaches. The static layout allows for simpler checks, alleviates reasoning about possible dynamic allocation failures, and enables the trivial memory management scheme we use in our operating system—see next part.

For VigNAT, libVig provides several basic data structures that we needed to develop VigNAT: a *network flow* abstraction, a *flow table*, implemented as a double-keyed hash map, an *ID manager* to keep track of allocated ports, and an *expirator* abstraction for tracking and expiring flows. libVig also provides an *nf_time* abstraction for accessing system time and a *DPDK* layer on top of the DPDK framework. None of the libVig data structures are thread safe because Vigor works only with single-threaded NFs.

- The libVig flow table is implemented as a non-owning open-addressing hash map. We implemented the hash map as a fixed-size ring array of tuples `<bool busy-bit, int chain-count, int key-hash, void* key-ptr, int value>`. The *chain-count* field keeps the number of chains crossing the array cell. A chain is a possibly wrapped-around array segment connecting the starting point (defined by the key hash) and the actual key-value location. As the hash map fills up and collisions start to occur, multiple such chains can overlap, and we track those overlaps with this counter. *key-hash* is the hash of the lookup key, *key-ptr* is a pointer to the full key (for precise comparison), and *value* is the integer value corresponding to the key. To get the full flow-ID from the integer value, one needs to use it to index the array of values kept as a separate vector.

This separation of concerns enables better modularity. For example, we use the integer value also as a proxy for the external port number.

To insert a new entry, the libVig flow table searches linearly for a free cell (i.e., one with `busy-bit=0`) starting from an index equal to the hash of the lookup key. It increases *chain-count* for every cell on the way to thread in a new chain. Flow search and removal also starts from the hash-computed index in order to find a cell with the same hash, and then it confirms the match via a comparison of the stored key to the search key. In a relatively empty flow table, a search procedure most often terminates early once it finds that no chain goes beyond the current cell, i.e., `chain-count=0`. Otherwise, it has to scan through the whole array. Note, that this property bears little impact on the performance as we expect low collision rates and short scans. To combat a high collision rate, the NF developer can preallocate a larger hashmap thus reducing the occupancy and the collision probability.

- The libVig *ID manager* keeps track of the allocated IDs and their expiration. (In VigNAT the ID is a proxy for the external TCP/UDP port number, and an index to the array of the flow-IDs serving as keys for the low table). It is implemented as two linked lists—*busy* list and *free* list—intertwined within a single array. The array-cell index represents the unique ID, which can be busy or free, depending on which list the cell belongs to. The *free* list is a single-linked list and initially, it threads through the whole array. The *busy* list is a double-linked list. An allocation of an ID boils down to detaching the corresponding cell from the *busy* list and attaching it to the *free* list. A de-allocation is the reverse process. The least recently accessed IDs accumulate at the tail of the *busy* list. Refreshing an ID (to reflect that the ID was recently accessed and should not be expired soon) is done by moving the corresponding cell to the head of the busy list.

2.2 Using Abstraction and Contracts to Formally Specify libVig Semantics

We specify the semantics of libVig data types in terms of abstract state that the data types' methods operate on. The pre-conditions and post-conditions for each method form the *contracts* that define what each data type is supposed to do.

```
int dmap_get_by_first_key /*@ <K1,K2,V> @*/
       (struct DoubleMap* map, void* key, int* index)
/*@ requires dmappingp<K1,K2,V>(map, ?kp1, ?kp2, ?hsh1, ?hsh2,
                                ?fvp, ?bvp, ?rof, ?vsz,
                                ?vk1, ?vk2, ?rp1, ?rp2, ?m)
               &*& kp1(key, ?k1) &*& *index |-> ?i; @*/
/*@ ensures dmappingp<K1,K2,V>(map, kp1, kp2, hsh1, hsh2,
                               fvp, bvp, rof, vsz,
                               vk1, vk2, rp1, rp2, m) &*&
               kp1(key, k1) &*& (dmap_has_k1_fp(m, k1) ?
                   (result == 1 &*& *index |-> ?ind  &*&
                       ind == dmap_get_k1_fp(m, k1) &*&
                       true == rp1(k1, ind)) :
                   (result == 0 &*& *index |-> i)); @*/
{
  /*@ open dmappingp(map, kp1, kp2, hsh1, hsh2,
                     fvp, bvp, rof, vsz,
                     vk1, vk2, rp1, rp2, m); @*/
  map_key_hash *hsh_a = map->hsh_a;
  //@ map_key_hash *hsh_b = map->hsh_b;
  //@ assert [?x]is_map_key_hash(hsh_b, kp2, hsh2);
  //@ close [x]hide_map_key_hash(map->hsh_b, kp2, hsh2);
  int hash = hsh_a(key);
  //@ open [x]hide_map_key_hash(map->hsh_b, kp2, hsh2);
  int res = map_get(map->bbs_a, map->kps_a, map->khs_a,
                    map->inds_a, key, map->eq_a,
                    hash, index, map->keys_capacity);
  /*@ close dmappingp(map, kp1, kp2, hsh1, hsh2,
                      fvp, bvp, rof, vsz,
                      vk1, vk2, rp1, rp2, m); @*/
  return res;
}
```

Listing 2.1: Top-level get method `dmap_get_by_first_key` in the libVig flow table data type. `index` is an output parameter for the index of the entry whose first key matches `key`. The method returns 1 if the entry is found, 0 otherwise. See [111] for the VeriFast syntax used in this contract.

Listing 2.1 shows a snippet of a `get` method for the libVig flow table. The pre- and post-conditions immediately follow the function signature. The contracts are meant for the Validator's and proof checker's consumption, but they can also serve as documentation, when a natural-language description is ambiguous or when reading the source code falls short.

Each "`requires`" pre-condition states the requirements for the function to run: a relationship between its arguments and the abstract state, or a memory ownership token for a pointer. Each "`ensures`" post-condition specifies what holds after the completion of the method: a relationship between the arguments and the return value, the updated value at a certain memory location, or a memory ownership token.

We adopt a "sanitary" policy for the use of pointers: the SE-friendly code can pass/receive pointers across the libVig interface, but the libVig data structure remains opaque to the caller. The SE-friendly code can copy pointers, assign them, and compare them for equality, but it can only dereference the pointers to the explicitly declared entry types, such as the *flow-ID* struct. The SE-friendly code is not allowed to dereference the pointer to any of the libVig data structures. Vigor automatically checks that the SE-friendly code obeys this discipline.

2.3 Verifying libVig Correctness

Once the formalization of the interface is complete, we write the proof that the implementation of each function satisfies its interface contract, i.e., we annotate the code with assertions, loop invariants, etc., and we define lemmas for the intermediate steps of the proof.

The proof checker begins by assuming the pre-condition and steps through every code statement, developing its set of assumptions along the way. When it encounters a branch condition, it explores both branches. Inlined annotations help the checker to understand the transformations of abstract state, and it verifies that they indeed correspond to the transformations of the concrete machine-state. On each memory-access, the proof checker checks the validity of the address and the memory ownership token. For method calls, it checks the pre-condition of the called method and then assumes its post-condition, in essence, replacing the call with an assumption of the callee's post-condition (it verifies separately that the post-condition indeed holds whenever the callee returns). When reaching a return point, the proof checker checks the post-condition.

In Vigor, we use the VeriFast proof checker [110], which works for C programs annotated with pre-conditions and post-conditions written in separation logic [190]. Annotating code is not an easy task, especially for non-experts. However, separation logic is relatively friendly: it is an extension of classic Hoare logic that is designed for low-level imperative programs that use shared mutable data structures. It has a good notion of memory ownership, which makes it easy to express the transfer of ownership through pointers. Separation logic supports local reasoning [168] in that specifications and proofs of a method refer only to the memory used by that method, not to the entire global state.

Chapter 3

Verifying SE-Friendly (Stateless) NF Code with Symbolic Execution

When symbolically executing the VigNAT SE-friendly code, which calls into libVig, we do not seek to also symbolically execute the libVig implementation because this would lead to path explosion. Therefore, we abstract libVig with a *symbolic model* that simulates the effect of calling into libVig and keeps track of the side effects in a per-execution-path manner. The symbolic model differs from the formal contracts in two ways: it is executable code, and it might miss some possible behaviors of the libVig implementation.

The challenge lies in combining the results of these two verification techniques, and for this, we developed a technique we call "lazy proofs". A lazy proof consists of lemmas structured in a way that top-level proofs proceed *assuming* lower-level properties, and the latter are proven lazily a posteriori. For example, symbolic execution requires the use of valid models, i.e., models that strictly over-approximate the actual implementation; we first do the symbolic execution and only afterward validate automatically the models with respect to the exercised use cases.

Lazy proofs exempt Vigor developers from proving that our models are *universally* valid, but instead, developers only prove that the models are valid for the specific NF and for the specific properties they verified earlier with SE. First, this approach exempts the developers from creating perfect models, approximations of the data structures that are valid in every possible context, which might be impractical because of the mismatch in expressive power between the language used to describe the model and the language of the contracts. Second, lazy proofs also narrow Vigor developers' proof efforts of the validity of any particular model down to only the specific cases encountered in VigNAT and obviate the need to handle corner cases that never occur in VigNAT. This is much easier.

Not only do lazy proofs enable us to use the right tool for each desired property but they also resolve the modeling challenge: writing a symbolic model of libVig requires reconciling two conflicting objectives. On the one hand, the model must remove enough details, i.e., be abstract enough to make symbolic execution terminate in useful time. After all, it is the abstraction that reduces the number of paths to explore symbolically. On the other hand, the model must be detailed enough to capture enough libVig behaviors in order to be faithful to the libVig implementation in the context of the properties being verified. The quality of the model depends directly on which details are relevant to the proof and which are not. This, in turn, depends both on the properties to be proven and on the code that uses the model. Hence, devising a good model is often an iterative process that converges after multiple attempts on a good model customized to the code and the property to be proven. Spending time proving the faithfulness (P_5) of draft models before actually knowing that they are fit for proving P_2 would be wasteful. With lazy proofs, we can now *first* attempt the proof of P_2, assuming the model is OK and, if the model indeed helps prove the desired property, only then invest in validating the model (P_5). From a practical standpoint, this approach makes it inexpensive to write models because we do not need to spend time ironing out the very last bugs; instead, we rely on Vigor to expose these bugs over time. Moreover, lazy proofs focus our verification effort on the aspects of the model behavior that matter in the context.

In other words, lazy proofs exploit the fact that an application typically uses only a subset of the semantics offered by its libraries. Consequently, instead of proving that the libVig model accurately captures all of libVig's semantics, we only prove that it does so for the semantics used by VigNAT.

We now describe the proof of P_2 in Figure A1.1—VigNAT satisfies low-level properties.

Proving That VigNAT Satisfies Low-Level Properties

Low-level coding mistakes, such as the misuse of memory, can cause a program to crash or behave erratically, hence proving the absence of such mistakes is essential to proving higher-level semantic properties. The desired low-level properties refer to the absence of bugs such as buffer over/underflow, out-of-bounds memory accesses, double free, and arithmetic over/underflow.

To prove the absence of such bugs, Vigor performs *exhaustive* symbolic execution (ESE), by using a SEE to enumerate all execution paths through the SE-friendly part of the NF. The SEE explores all feasible branches at conditional statements, therefore ESE is fully precise (assuming it uses sound, over-approximate models): It enumerates only feasible paths, i.e., paths for which there exists a set of inputs that takes the program down that path, and does not miss any feasible paths. Low-level properties are stated as `assert`s, and for each feasible execution path the SEE reasons symbolically about whether there exists an input that could violate the `assert`. In order to make this approach feasible, Vigor first replaces all calls to libVig with calls to the libVig models; this abstracts away all state handling code, thereby removing almost all constructs that lead to path explosion, such as loosely constrained symbolic pointers. Next, the NF developer makes the SEE aware of loop bounds by marking the loop guard of the infinite loop with `VIGOR_LOOP` and Vigor constructs loop invariants (using loop havoc-ing and the `NF_STATE` arguments) so that the SEE can transform the loops to avoid unnecessary loop unrolling. This eliminates the last source of path explosion in VigNAT.

If the `assert` for each low-level property holds on *every* feasible path during ESE, then we have a proof that the SE-friendly code is free of low-level bugs, as ESE reasons about all possible inputs without enumerating those inputs. The proof that libVig behaves according to its interface contracts guarantees that libVig too is free of low-level bugs. This means that all VigNAT code satisfies the low-level properties. If this was not stateless code but a stateful program, ESE would likely not complete. Yet, in our case, the SEE checks all 108 paths through VigNAT's SE-friendly code in less than one minute.

Of course, the proof makes certain basic assumptions, such compiler correctness, and most importantly it assumes that the libVig models are correct (P_5) and that the SE-friendly code uses libVig data structures correctly (P_4). Verifying these assumptions a posteriori requires our Validator and function-call sequences, which we describe in the next chapter.

Implementation Details

In Vigor, we use the KLEE SEE [36]. It checks out-of-the-box several low-level properties, and we add the checks from LLVM's undefined behavior sanitizers [158, 159, 237]. We modified KLEE in several ways: First, we added loop invariant support and enabled KLEE to automatically find the variables that might change inside a loop and havoc [137] them. The libVig symbolic models provide loop invariants related to their bookkeeping state, which allows KLEE to avoid enumerating unnecessary paths. Second, we added dynamic pointer-access control by providing primitives that enable libVig developers to enable/disable the dereferenceability of a pointer between libVig calls. Third, we added the ability to record function-call sequences, described next. And, to ensure that KLEE either explores all paths or fails explicitly, we disabled its default termination of execution paths that are too long or take too much memory. Our custom version of KLEE is a part [238] of our public repository [254].

Chapter 4

Tying Together the Verification of SE-friendly and Non-SE-Friendly Code by Using Lazy Proofs

Our proposed *lazy proofs* technique glues together a symbolic execution engine (SEE) with a proof-checker in order to produce proofs of NF properties that were previously out of reach. Together with the separation to SE-friendly and non-SE-friendly described earlier, this constitutes the cornerstone of how we verify VigNAT.

The main idea is to use a SEE to enumerate all execution paths through the SE-friendly NF code, and

- to record for each path a function-call sequence of how the SE-friendly code interacted with the outside world and with libVig; and
- to verify that P_2 (low-level properties) holds on each path.

The Validator then transforms the function-call sequences (i.e., a representation of all possible observable behaviors of VigNAT's SE-friendly code) into mechanically checkable proofs that P_4, P_5, and ultimately P_1 (NAT semantics) hold.

4.1 Proving That VigNAT Satisfies NAT RFC Semantics

We think of the formalized NAT semantics as "function-call-sequence properties": Given a function-call sequence of the interaction between an NF and the outside world, what must hold of the function-call sequence for it to have been generated by a correct NAT NF? More specifically, the NAT properties are in the form of pre- and post-conditions for actions triggered by the arrival of a packet. The preconditions, expressed on the abstract NAT state and the incoming packet, select which action applies. The corresponding post-condition states what must hold, after the action is completed, of the abstract state and the potential outgoing packet.

In order to verify that the desired properties hold, Vigor collects from the SEE a trace of function calls for each explored execution path. This trace summarizes how the VigNAT code interacted, during symbolic execution, with (a model of) the outside world, be it the libVig library or the DPDK framework. As the traces have common prefixes, they form a tree, which is a subgraph of the NF's execution tree. In the context of this section, a *function-call sequence* is a path from the root of the traces tree to a node in the tree, be it an internal or a leaf node. In other words, the set of function-call sequences considered by Vigor consists of all *complete* function-call sequences (reaching the last call in the event-loop iteration) and all their prefixes.

Each function-call sequence has two parts: a sequence of calls that were made across the traced interface, and a set of constraints on the symbolic program state. Listing 4.1 shows a simple example of a function-call sequence for the code (in Listing 1.4) that uses the ring data structure. The seven calls in this function-call sequence result from the execution of the corresponding lines in Listing 1.4. `loop_invariant_produce` and `loop_invariant_consume` are markers indicating the beginning and the end of a loop iteration. In the `ring_pop_front` call, `packet` is an output parameter pointing to the popped packet; the function-call sequence records its initial and final value.

The *constraints* section shows the relationship between the different symbols. In this simple example, there is only one constraint: $y \neq 9$ is the result of the application of `packet_constraints` (Listing 1.3) in the ring model. The initial value of x is unconstrained.

```
 1  loop_invariant_produce(ring=[..]) ==> []
 2  ring_full(ring=[..]) ==> true
 3  ring_empty(ring=[..]) ==> false
 4  can_send() ==> true
 5  ring_pop_front(ring=[..],
 6                 packet={.port=:x:} --> {.port=:y:}) ==> []
 7  send(packet={.port=:y:}) ==> []
 8  loop_invariant_consume(ring=[..]) ==> []
 9  --- constraints ---
10  :y: != 9
```

Listing 4.1: Function-call sequence for a path through the example code in Listing 1.4. Colons (`:val:`) designate a symbol, `-->` separates the input and output value of a pointer argument, `==>` marks the return value of a function call, `[..]` indicates omitted details.

The Validator now takes each function-call sequence, weaves into it the properties to be proven, and turns it into a verification task. The verification task is a C program that contains the sequence of calls from the function-call sequence, enriched with metadata on the symbolic variables used as arguments and return values, as well as the constraints that describe the relationships between these symbolic variables at each point in the function-call sequence. To help the proof checker, the Validator also inserts lemmas into the function-call sequence. In essence, the Validator translates each function-call sequence into a proof that the call sequence satisfies the desired properties. It then passes the proof to the proof checker to verify it.

Listing 4.2 shows the Validator-transformed version of Listing 4.1. The seven calls are now on lines 6,8,10,12,15,18,22. The uninitialized *arg_1* and *arg_2* variables are unconstrained symbols initially. The constraints on return values recorded in Listing 4.1 turn into the `@assume` statements on lines 9,11,13,17. The symbolic constraint from line 10 in Listing 4.1 turns into the `@assume` on line 17 of the proof. These four `@assume` statements constitute the pre-condition of this function-call sequence. In order to set up the post-condition that needs to be verified, the Validator initializes special handle variables `packet_is_sent` and `sent_packet` on lines 19 and 20 to capture externally visible effects immediately after the `send()` call. Then it inserts the NF specification (semantic property) into the function-call sequence on lines 24–26:

```
if (packet_is_sent) {
  assert(sent_packet->port != 9);
}
```

Vigor verifies that the NF specification holds after every loop iteration.

```
 1 struct ring* arg1;
 2 struct packet arg2;
 3 bool packet_is_sent = false;
 4 struct packet* sent_packet = NULL;
 5
 6 loop_invariant_produce(&arg1);
 7 //@ open loop_invariant(&arg1);
 8 bool ret1 = ring_full(arg1);
 9 //@ assume(ret1 == true);
10 bool ret2 = ring_empty(arg1);
11 //@ assume(ret2 == false);
12 bool ret3 = can_send();
13 //@ assume(ret3 == true);
14 //@ close packetp(&arg2, packet(arg2.port));
15 ring_pop_front(arg1, &arg2);
16 //@ openpacketp(&arg2, _);
17 //@ assume(arg2.port != 9);
18 send(&arg2);
19 packet_is_sent = true;
20 sent_packet = &arg2;
21 //@ close loop_invariant(&arg1);
22 loop_invariant_consume(&arg1);
23
24 /*@ if (packet_is_sent) {
25   assert(sent_packet->port != 9);
26 } @*/
```

Listing 4.2: The function-call sequence of Listing 4.1, translated into a proof.

Once the proof checker completes all verification tasks received from the Validator, we have a proof that the function-call-sequence properties weaved in by the Validator hold for all possible executions of the SE-friendly code. Vigor proves that VigNAT satisfies the NAT specification by weaving the properties translated from the specifications in our Python dialect [254] into the function-call sequences, similar to lines 24–26 in Listing 4.2. Function-call sequence verification is highly parallelizable: to verify all 431 function-call sequences resulting from the 108 execution paths of SE-friendly part of VigNAT takes 38 minutes on a single core and 11 minutes on a 4-core machine. As we note later, this verification time includes not only proving P_1 but also P_4 and P_5.

4.2 Validating the libVig Symbolic Model

We say a symbolic model is *valid* if the behavior it exhibits is indistinguishable from the behavior of the libVig implementation captured by the formal interface contracts[1]. Any behavior of the model that is not observed during ESE is irrelevant to its validity *for this particular proof.* This is the insight behind our lazy proof technique: It does not matter whether a model is universally valid, but it does matter whether *the parts of the model used during symbolic execution* are valid. This is a much weaker property than universal validity but is sufficient for our purposes. The function-call sequences capture all necessary information on how the model is used.

The technique for proving that the libVig model is consistent with the libVig interface contracts is similar to the one we use for proving NAT semantics. Only this time, instead of weaving the NAT pre- and post-conditions into the function-call sequences, the Validator weaves in the assertions for the given function-call sequence's path constraint. It then asks the proof checker to verify whether the assertions hold based solely on the post-conditions of the libVig functions. If the verification succeeds, then it means that, after each invocation of the libVig model, the outcome covers all possible outcomes prescribed by the libVig interface contracts.

A libVig model can be over-approximating, under-approximating, or both. The question that the Validator aims to answer is whether, for the particular NF and properties, the model is sufficiently accurate. If a model is too under-approximating for the desired proof, it causes the validation phase to fail because its narrow behavior does not cover the spectrum of behaviors allowed by the contracts. If it is too over-approximating for the target proof, it either causes exhaustive symbolic execution (ESE) or validation to fail: It will be the former if the model exhibits behavior that is too general and makes it impossible to verify the low-level properties, and it will be the latter if low-level properties verify but a loop invariant or a high-level semantic property is violated. When ESE completes, we have proof that the low-level properties hold, as long as the model is valid. ESE failure means that either there is a violation of a low-level property or the model offered by libVig is not suitable. Vigor does its best to help the developer distinguish between the two, but there is still room for improvement. In either case, it is back to the drawing board: either the NF developer needs to fix her bug, or the Vigor developer needs to alter the model.

Vigor also uses models we wrote of the DPDK packet processing framework's `send`, `receive`, and `free` calls. We do not formally validate these models, though there

[1]It is sufficient to show that the model over-approximates the contracts because any proof that holds for the over-approximation holds for an exact model as well.

is no fundamental reason it cannot be done. We make it part of Vigor's trusted computing base (Section 4.4).

4.3 Proving That VigNAT Correctly Uses libVig

There is one caveat to the proof in the previous section: if a libVig method implementation is invoked and the corresponding pre-condition does not hold, then the behavior of that method is undefined. For example, passing a null argument when the contract says it must be non-null could cause the implementation to crash, behave incorrectly, or behave correctly. It is therefore imperative that, in conjunction with validating the model's behavior, Vigor also validates the caller's behavior with respect to the interface contracts.

The method for proving that the VigNAT SE-friendly code uses the libVig data structures consistently with the libVig interface contracts is the same as above, except that the Validator weaves in the pre-conditions contained in the libVig interface contracts. In fact, the Validator weaves in the NAT pre- and post-conditions and the libVig pre- and post-conditions in the same run, and it generates a single verification task per function-call sequence, which simultaneously verifies properties P_1, P_4, and P_5.

The trickiest part in verifying the libVig pre-conditions is tracking memory ownership across the interface. A pointer returned by a libVig method (either as a return value or via an output parameter) references memory owned at first by libVig; upon return, ownership transfers to the caller. After using/modifying the pointer, the NF code calls another function to return ownership to libVig.

A pointer passed as an argument to a libVig method can be the address of a libVig data structure (equivalent to the `this` pointer in C++/Java or the `self` reference in Python). This type of pointer remains opaque to the SE-friendly code: it can be copied, assigned, and compared for equality, but cannot be dereferenced. The Validator and proof checker need not look at the memory pointed to by such pointers but only keep track of aliasing information. The pointed-to memory is owned by libVig at all times. During symbolic execution, Vigor verifies that the SE-friendly code obeys this pointer discipline. To verify this Vigor uses our addition to the SEE for enabling/disabling the dereferenceability of a pointer between libVig calls.

A pointer used as an output parameter points to where the caller expects libVig's return result to be written. In this case, the pointed-to memory is owned by the calling code. The Validator and the proof checker trace the evolution of the pointed-to memory by including it, before the call, in the function's input set, and, after the call, in the function's output set. A special case of output pointer is a double pointer to a library data structure (e.g., `X** p`) as appears in the data-structure allocation functions. The VigNAT code owns the pointee `*p`, hence the Validator tracks it, but the pointee of the pointee `**p` is a library data structure, therefore the memory is owned by the library, and there is no need to track it. Vigor currently does not support other cases of double or deeper pointers.

Vigor also checks for memory leaks. Even though SE-friendly code cannot dynamically allocate memory, leaks are possible if it uses libVig incorrectly, such as forgetting to call a release method. Unlike simple low-level properties (e.g., an integer overflow) that can be stated as a simple `assert`, absence of memory leaks is a global property. Vigor, therefore, must keep track of memory ownership and validate that ownership is properly returned to libVig before the end of the execution. This facility, for example, caught an accidental memory leak in VigNAT where we failed to release DPDK memory corresponding to a packet returned by DPDK, thus violating the DPDK interface contracts.

4.4 Summary of the Verification Workflow

The Vigor workflow described above can be summarized as follows (see Figure A4.1): We split the NAT NF into a SE-friendly and a non-SE-friendly part, the latter contained in the libVig library. Then we use theorem proving to verify P_3, the correctness of the data structures implemented in libVig. We use exhaustive symbolic execution (ESE) with a modified version of KLEE [36] to explore all paths in the SE-friendly part (using symbolic models of the data structures) and verify P_2 (low-level properties, such as crash freedom, memory safety, and no overflows), as well as VigNAT's disciplined use of pointers. This step proceeds under the assumptions that the SE-friendly code uses the libVig data types according to their interface contracts (P_4) and that the libVig model is faithful to the libVig interface contracts (P_5) for the particular execution paths explored during ESE. Both of these assumptions we prove a posteriori by using a combination of our Vigor Validator and the VeriFast proof checker [110]. Finally, we use this same combination of tools to prove VigNAT's semantic properties *(P_1)*, i.e., that it conforms to our formalization of RFC 3022 [207].

$P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5$ together formally prove VigNAT's correctness, under the assumptions described below.

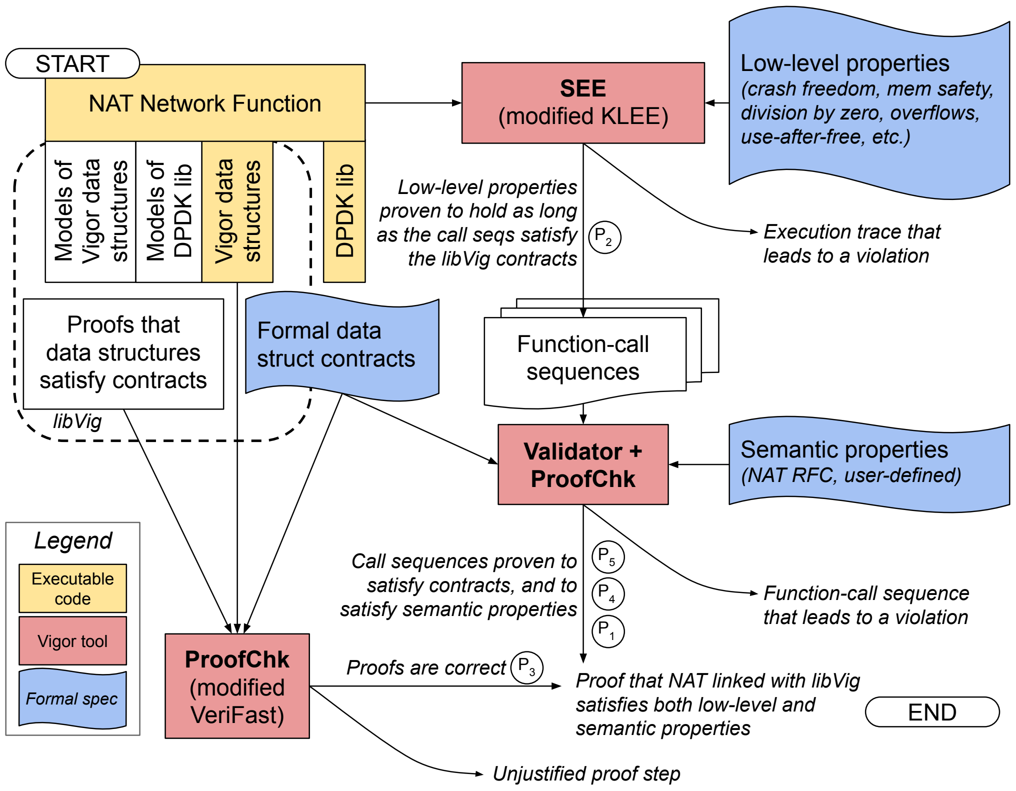

START
NAT Network Function
Models of Vigor data structures
Models of DPDK lib
Vigor data structures
DPDK lib
Proofs that data structures satisfy contracts
libVig
Formal data struct contracts
SEE (modified KLEE)
Low-level properties (crash freedom, mem safety, division by zero, overflows, use-after-free, etc.)
Low-level properties proven to hold as long as the call seqs satisfy the libVig contracts
P2
Execution trace that leads to a violation
Function-call sequences
Validator + ProofChk
Semantic properties (NAT RFC, user-defined)
Call sequences proven to satisfy contracts, and to satisfy semantic properties
P5
P4
P1
Function-call sequence that leads to a violation
Legend
Executable code
Vigor tool
Formal spec
ProofChk (modified VeriFast)
Proofs are correct
P3
Proof that NAT linked with libVig satisfies both low-level and semantic properties
END
Unjustified proof step

Figure A4.1: The Vigor workflow: how it verifies. In circles are the sub-proofs established on each step assuming their dependencies hold (see Figure A1.1)

Assumptions

The trusted computing base for a Vigor-verified NF consists of the Vigor toolchain (the Clang LLVM compiler, VeriFast, KLEE, and our own Validator) and the environment in which the NF runs (DPDK, device drivers, operating system kernel, BIOS, and hardware). We assume that the compiler implements the same language semantics employed by Vigor (e.g., same byte length for C primitive types). We wrote symbolic models for three DPDK functions and for system time, which as of this writing we have not verified. They are small (about 400 LOC), hence we manually convinced ourselves that they are correct over-approximations.

Note that in Part B we show how to remove DPDK, device drivers, operating system kernel, and DPDK SE models from the trusted computing base (TCB).

The resulting TCB:

- Compile time
  - Vigor toolchain:
    Clang compiler, VeriFast, patched KLEE, Vigor Validator;
  - GCC C compiler
  - hardware SE model
- Run time
  - hardware
  - BIOS
  - GRUB bootloader

Chapter 5

Preliminary Evaluation: VigNAT Verification Does Not Come at the Cost of Performance

By building VigNAT, we demonstrated that it is possible to verify an NF, hence there is a promise for our approach. Now the question is whether the resulting NF can deliver performance on-par with unverified NFs. We, therefore, do a preliminary empirical evaluation of VigNAT's performance and find that the answer is yes, we can have both verification and performance at the same time. This result motivates us to continue with the full-blown approach, as described in the next part of the thesis.

We focus our evaluation on comparing VigNAT (labeled **Verified NAT** in the graphs) to three other NFs:

(a) **No-op forwarding** is implemented on top of DPDK; it receives traffic on one port and forwards it out to another port, without any other processing. It serves as a baseline that shows the best throughput and latency that a DPDK NF can achieve in our experimental environment.

(b) **Unverified NAT** is also implemented on top of DPDK; it implements the same RFC as VigNAT and supports the same number of flows (65,535) but uses the hash table that comes with the DPDK distribution. It was written by an experienced software developer who had little verification expertise, which is different from the developer who wrote and verified VigNAT. It serves to compare VigNAT to a NAT that was not written with verification in mind.

(c) **Linux NAT** is NetFilter [26], set up with straightforward masquerade rules and tuned for performance [115]. We expect it to be significantly slower than the other two because it does not benefit from DPDK's optimized packet reception and transmission. We use it to make the point that VigNAT performs significantly better than the typical NAT currently used in Linux-based home and small-enterprise routers, as would be expected from a DPDK NAT.

We use the testbed shown in Figure A5.1 as suggested by RFC 2544 [28]. The Tester and the Middlebox machines are identical, with an Intel Xeon E5-2667 v2 processor at 3.30 GHz, 32 GB of DRAM, and 82599ES 10 Gbps DPDK-compatible NICs. The Middlebox machine runs one of the four NFs mentioned above (we use one core). The Tester machine runs MoonGen [74] to generate traffic and measure packet loss, throughput, and latency; for the latency measurements, we rely on hardware timestamps for better accuracy [184]. We use DPDK v.16.07 on Ubuntu Linux with kernel 3.13.0-119-generic.

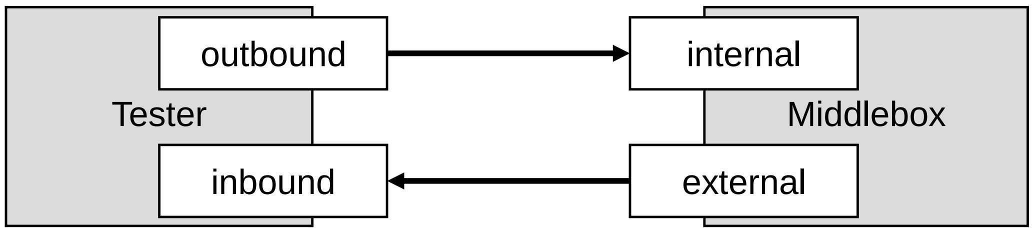


Figure A5.1: Testbed topology for performance evaluation.

First, we measure the latency experienced by packets between the Tester's outbound and inbound interfaces. We run a set of experiments in which all the NATs are configured to expire flows after 2 seconds of inactivity. In each experiment, the Tester generates 10–64,000 "background flows" that produce in total 100,000 pps and never expire throughout the experiment, and it generates 1,000 "probe flows" that produce 0.47 pps and expire after every packet. We use the background flows to control the occupancy of the flow table, while we measure the latency of the packets that belong to probe flows. We focus on the probe flows because, from a performance point of view, they are the worst-case scenario for a NAT NF: each of their packets causes the NAT to search its flow table for a matching flow ID, not find any match, and create a new entry.

Figure A5.2 shows the average latency experienced by the probe flows as a function of the number of background flows, for the three DPDK NFs: the Verified NAT (5.13μsec) has 2% higher latency than the Unverified NAT (5.03μsec), and 8% higher than No-op forwarding (4.75μsec). Therefore, on top of the latency due to packet reception and transmission, the Unverified and Verified NAT add, respectively, 0.28μsec and 0.38μsec of NAT-specific packet processing. For all three NFs, latency remains stable as flow-table occupancy grows, which shows that the two NATs use good hash functions to spread the load uniformly across their tables. The only case where latency increases (to 5.3μsec) is for the Verified NAT, when the flow table becomes almost full (the green line curves upward at the last data point). The Linux NAT has significantly higher latency (20μsec).

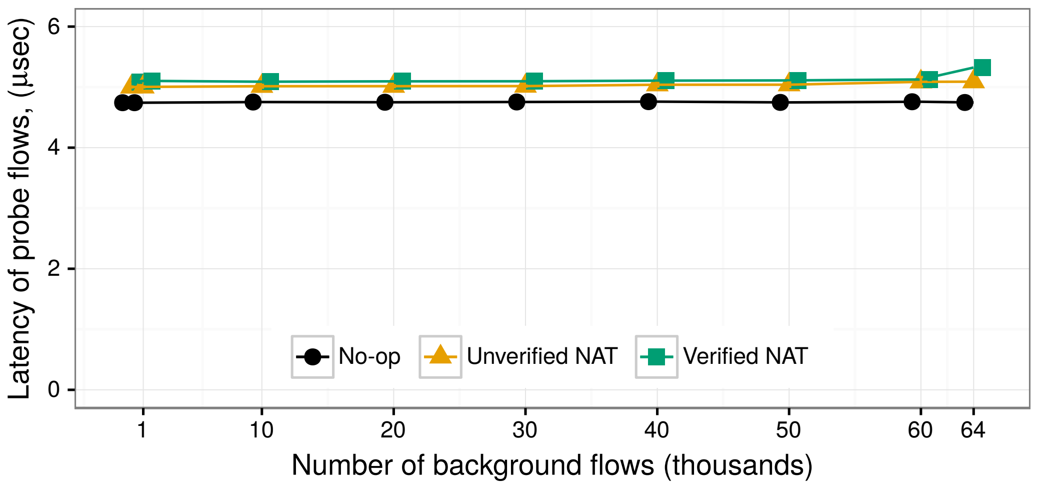


Figure A5.2: Average latency for probe flows. Confidence intervals are approximately 20 nanosec, not visible at this scale.

To give a sense of latency variability, Figure A5.3 shows the complementary cumulative distribution function (CCDF) of the latency experienced by the probe flows when there are 60,000 background flows (i.e., 92% occupancy): The Verified NAT has a slightly heavier tail than the Unverified NAT, and all three NFs have outliers that are two orders of magnitude above the average, but these are due to DPDK packet processing, not NAT-specific processing (the three curves coincide for latency exceeding 6.5μsec). The CCDFs computed for different numbers of background flows look similar.

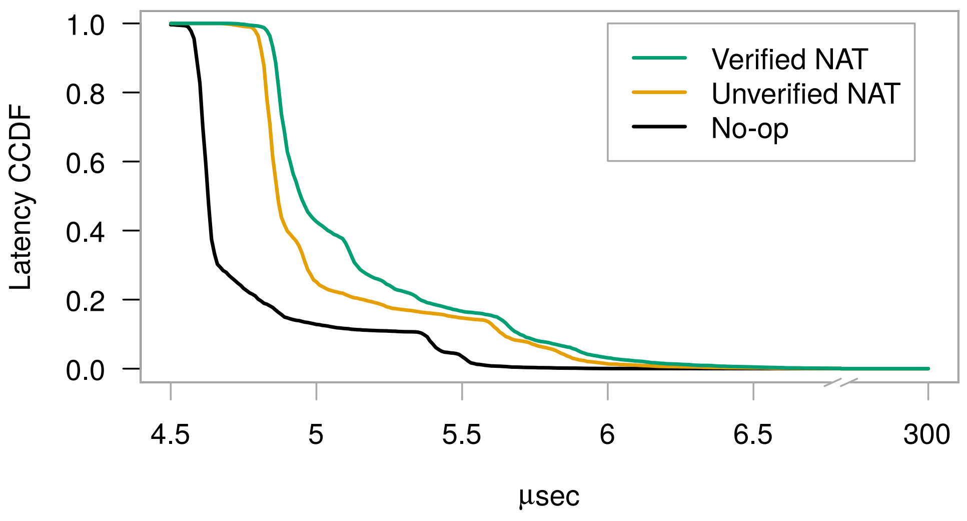


Figure A5.3: Latency CCDF for probe flows.

We obtain similar results in the second set of experiments, where the Tester produces the same flow mix as before, but the NATs are configured to expire flows after 60 seconds of inactivity (hence neither the probe flows nor the background flows ever expire). In this case, the average latency of the Verified NAT is slightly lower (5.07μsec), whereas that of the Unverified NAT the same as before (5.03μsec).

Finally, we measure the highest throughput achieved by each NF. In each experiment, the Tester generates a fixed number of flows that never expire, each producing 64-byte packets at a fixed rate, and we measure throughput and packet loss. During all experiments, the Middlebox is CPU bound. Figure A5.4 shows the maximum throughput achieved by each NF with less than 0.1% packet loss, as a function of the number of generated flows. The Verified NAT (1.8 Mpps) has a 10% lower throughput than the Unverified NAT (2 Mpps). This difference in throughput comes from the difference in NAT-specific processing latency (0.38μsec vs. 0.28μsec) imposed by the two NATs: in our experimental setup, this latency difference cannot be masked, as each NF runs on a single core and processes one packet at a time. The Linux NAT achieves significantly lower throughput (0.6 Mpps).

These results indicate that the performance of the libVig flow table (which has a formal specification and proof) is close to that of the DPDK hash table (which has neither), though not the same. The implementations of the two data structures are quite different. We did not try to reuse or adapt the implementation of the DPDK hash table because it resolves hash conflicts through separate chaining (items that hash to the same array position are added to the same linked list) and might fail to allocate an entry even before it fills up to the capacity. a behavior that is hard to specify in a formal contract. Instead, the libVig flow table resolves conflicts through open addressing: If an item hashes to an occupied array position, it is stored in the next array position that is free, together with auxiliary metadata that accelerates lookup. We have not yet optimized our implementation at the instruction level, hence it has an overall slower access time because there are, on average, more candidate memory locations for each item[1]; the difference is the most substantial for lookups that find no match because they result in searching for all candidate memory locations. Nevertheless, the next part presents the benchmarks of the optimized flow table implementation that performs on par with the DPDK baseline.

In summary, our experimental evaluation shows that it is possible to have a NAT network function that both offers competitive performance and is formally verified.

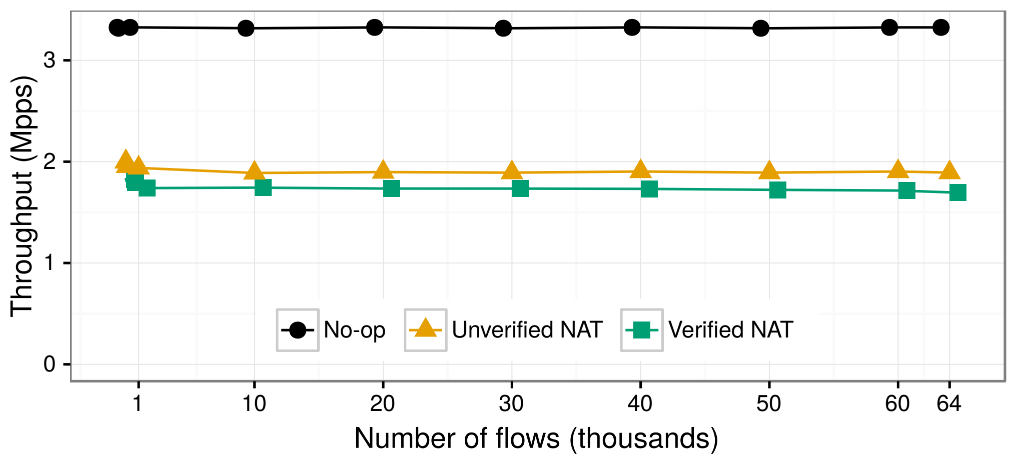


Figure A5.4: Maximum throughput with a maximum loss rate of 0.1%.

[1]This is the case for the particular implementations used, respectively, by the Unverified and Verified NAT, not for separate chaining and open addressing in general.

Chapter 6

Summary

This part has demonstrated some of the key ideas behind the Vigor toolchain:

- Separation to SE-friendly and non-SE-friendly code that enables the application of verification approaches with different tradeoffs to the parts of the code with different needs: high-overhead and high-power theorem proving on the compact, complex and reusable components, and low-overhead and low-power exhaustive symbolic execution on the custom and relatively straight forward NF logic.
- Lazy proofs that enable Vigor to automatically combine the proofs of the two different verification approaches.

Our preliminary performance evaluation shows that the proposed development model and the verification framework preserve the competitive latency and throughput of the NAT. In particular, theorem proving is powerful enough to handle a standard open-addressing hash table and the Vigor framework is compatible with DPDK (a high-performance packet processing framework).

Inspired by the success of the ideas on VigNAT, we continue to expand to more NFs, smaller trusted computing base, and complete automation in the next part.

Part B

Vigor: Making Network-Function Verification Practical

Chapter 1

Overview

In this part, we define Vigor's application domain (NFs) structure and objective (practical verification) and show how it uses the former to achieve the latter. Vigor works for software data-plane network functions because of their particular common structure. Vigor strives to achieve practical verification, which means it should be generic, full-stack, push-button, and pay-as-you-go.

1.1 Software NF as a Composition of SE-Friendly and Non-SE-Friendly code

The control flow observed for a discard-protocol example in the previous part turns out to fit a NAT as well as most of the network functions. Figure B1.1 depicts a control-flow structure for a typical NF. It represents an event processing loop, where the event can be either a packet received or a timer tick. The three dashed boxes 2, 3, 4 are "ghost code" that our toolchain inserts to handle the infinite loop, using an automatically generated loop invariant. After allocating all the necessary memory for packet pools and data structures, the NF enters an infinite event-processing loop. Each iteration of the loop polls the network cards for received packets and the schedule for a ready timer. Once an event arrives, the iteration processes it, and finishes by either sending a packet or not. There is no exit condition, hence the NF needs no deinitialization code.

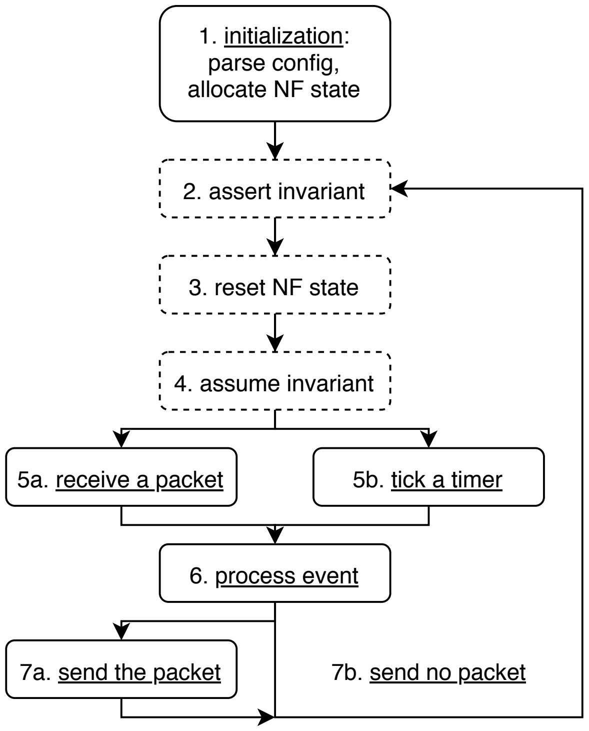


Figure B1.1: Typical NF control flow. The dashed boxes are a verification aid and are compiled out.

When verifying the NF code, our framework also inserts the support ghost code to manage the infinite loop by using a loop invariant. To keep verification sound, Vigor checks that the loop invariant holds (in node #2) after initialization, and after each iteration. To verify the generic case, Vigor resets all the persistent NF state (in node #3), i.e., havocs all the changed values and resets the state of the libVig models, right before entering the loop and, to prepare verification of the generalized iteration, it assumes the loop invariant combined from individual libVig data structure invariants (in the node #4).

Both the initialization code and the loop iteration consist of SE-friendly code—the NF logic—that invokes API methods from well-defined data structures.

We call SE-friendly code the code amenable to symbolic execution (SE). To enable SEE exhaustively explore its all possible behaviors, the SE-friendly code must comply with the three properties below:

- SE-friendly code must be isolated: the only external (non-symbolically-executed) code it can interact with is the library provided by the Vigor framework. This means it cannot interact with the user, use files, or use devices other than NICs.

- SE-friendly code must be bounded: all the looping (cycles or recursion) must have bounds on the number of repetitions. Otherwise, SEE might loop infinitely, exploring paths with more and more repetitions. A limitation of our prototype is that Vigor requires these bounds to be inferable statically.
- SE-friendly code must access only specific pointers:
  - static pointers within the SE-friendly code scope and
  - pointers returned by libVig API calls that reference part of the internal data of the data structure or a packet and that are designated explicitly to communicate with SE-friendly code (usually to avoid copying buffers and records across the libVig API).

If a part of the NF logic does not comply with the restriction above, it is likely to implement a data structure or a part of the data structure. For example, initially, the libVig expiring-map data structure supported the erasing of only a single expired entry at a time. NF logic implemented a loop that would expire all the expired flows that had a statically unbounded termination condition and blew the number of paths to infinity. Refactoring the loop within another API function "expire all" transformed this piece of NF logic into a single call. Such code has to go into the Vigor library of data structures.

Once the state of the NF is isolated in well-defined uniform data structures, we can deconstruct the loop invariant (which includes the data-structure invariants) into three simple parts.

- The first part is the constants in the NF logic; for the most part, they represent the
  - allocated resources, such as a buffer pool, call stack pointer, function pointers, or the data-structure pointers; and
  - configuration, such as the number of NICs, data-structure capacity, NIC proper MAC addresses, and NAT external IP address.

  The rest of the data in the NF logic is transient and is reset at the beginning of each iteration.
- The second part is the data structure's own invariant that guarantees that their implementation state is well-formed.

- The third part is the condition on the elements of the data structures. As the data structures are uniform, the complete data-structure invariant is the same for any NF, except for the condition on each entry. Our experience shows that the condition on the entry is quite simple for data plane NFs: the bounds on the value, e.g., the NIC-ID must be in the range [0,#NICs) and forbidden value (e.g., the internal NIC-ID has to be different from the external NIC for a NAT).

The loop invariant structure enables our SEE to automatically infer it, using user input, the extended entry type, only for the last part.

Unfortunately, some popular NFs fit imperfectly into the described structure. Any NF can be encoded in the event-processing loop form, with spawning or forwarding a packet as the only observable behavior. The SE-friendly/non-SE-friendly code boundary also exists in every NF because, according to our definition, any code can be either SE-friendly or not. However, in some NFs we have to classify most of the code as non-SE-friendly, and it is quite specific to the NF for the amortization argument to take effect.

Examples of NFs that fit imperfectly are the NFs implementing protocols with complex packet headers, such as DNS, LDAP, or XMPP. NFs related to these protocols require specialized parsers that are unlikely to satisfy the SE-friendly code requirements. A parser for each protocol has to be manually verified with theorem proving, included in libVig, and reused rarely if at all. In such a case verification is not push-button. The remaining added value of Vigor is the full-stack, as the SE-friendly code consists only of the framework, driver, and operating system. See Chapter 1 of the next part for a longer discussion.

1.2 Practical Verification Is Full-Stack, Push-Button, and Pay-As-You-Go

The previous part left two questions open:

1. What do we do about the software layers underneath the NF? All the NF verification work we are aware of (see Chapter 2 of the introduction) assumes that the I/O framework, the operating system, and the network device drivers are correct. This means that the resulting proofs are just as correct

as the thousands or millions [250] of lines of code running underneath the NF. Furthermore, a single misunderstanding about how a particular system call works could render the entire NF proof false.

2. How to lower the cost of verifying NF correctness in the real world? Practitioners at major Internet companies tell us that the upfront cost of verifying NFs is prohibitive. Learning a new language for writing specifications, investing multiple days in writing a full specification before being able to verify even the first line of code, writing lemmas that a theorem prover can understand, and figuring out whether a proof is failing because of an error in the specification or an error in the code are all reasons for verification being perceived by modern-day developers as an undue burden.

Vigor takes as input an NF implementation and a specification that the implementation must satisfy, and it automatically produces either a proof that the implementation satisfies the specification or debug information, in the form of a counterexample. We refine the approach described in the previous part in three ways:

1. We verify the entire software stack, reducing the trusted computing base (TCB) to the hardware, the Clang and GCC compilers, a small piece of operating system startup code, and our verification toolchain. A key element that enables this is an NF-specific operating system we built and a mechanism that verifies only those parts of the stack that are required by the NF in question. As we expect NFs to run on dedicated machines, either physical or virtual, using a custom operating system is reasonable.
2. We provide "pay-as-you-go" verification, a mode that reduces the cost of using verification in practice, both for developing and for deploying NFs.

The key Vigor elements that enable push-button verification are the level of abstraction of the NF specification and a sophisticated algorithm that bridges between specification and implementation. Like many modern tools that perform automated verification of low-level code, we use symbolic execution (SE) to analyze the behavior of each NF. The challenge is that there are always "difficult" pieces of code that cannot be SE-ed. A typical solution is to replace them with simplified models, but then the difficult pieces become part of the TCB. In NFs, the difficult pieces are the functions that access state, e.g., a function that performs a lookup in an IP forwarding table, or a function that updates a NAT flow table. To complete SE, we replace each such function with a simplified model, but then we retroactively validate each model: We prove that, for the particular NF specification and the particular NF implementation, the model did not under-approximate the original function. This model validation is the most challenging part of the verification process because it must bridge the state abstractions used by the NF specification and those used by the NF implementation. We argue that the NF domain is such that it makes sense to write implementations and specifications by using the same state abstractions; this is what makes automatic model validation—and push-button verification—possible.

The key Vigor elements that enable full-stack verification are an NF-specific operating system and a simple mechanism that verifies only those parts of the stack that are required by the NF in question. The straightforward approach to full-stack verification would be to analyze each layer of the stack separately, then to reason about putting them all together; we rejected this, due to the size of modern I/O frameworks and the size and complexity of modern operating systems. Our insight is that each NF uses only a small subset of the functionality provided by the layers below. Hence, when we SE an NF, instead of abstracting the layers below with models, we SE the full stack (we abstract away only the hardware); this way, we explore only the parts of the stack that are exercised by the NF in question. Still, our approach would not work with a standard operating system. Therefore, we write an NF-specific operating system (NFOS) that provides the minimum functionality needed to run NFs: it sets up the hardware and I/O framework, and it relays timing information from the hardware to the NF.

Vigor also provides "pay-as-you-go verification": it minimizes the burden of writing specifications. Practitioners tell us that the upfront cost of writing a full specification is typically prohibitive [165]: a developer cannot afford to invest days writing a full specification before being able to verify the first line of code. An added feature of how we express and verify NF specifications is that developers can write and verify short, partial specifications (e.g., "the firewall never forwards inbound traffic for an unknown flow", or "the bridge always learns the source MAC-to-NIC association of a frame"), without specifying the rest of the NF behavior.

1.2.1 Benefits for Different Audiences

We refine the four categories of the audience from the previous part:

1. **NF operators** are network operators who deploy NFs in their networks.
2. **NF developers** write NF code in C on top of standard packet-processing frameworks such as DPDK [249]. They are competent in popular programming languages, such as C and Python, but are not (and do not seek to become) verification experts. For instance, they would not have the time or experience to productively use an interactive theorem prover such as Coq or Isabelle or to write specifications in first-order logic.
3. **Standardization bodies** are the entities that define what NFs ought to do. For example, they write the RFCs that define IP forwarding or Network Address Translation. They are not verification experts either, but they might be willing to specify formally the correct behavior of the NF they standardize, as long as it can be done in an accessible language.
4. **Vigor contributors** are the maintainers of the Vigor stack and toolchain. They do have plenty of verification expertise and enjoy using it.

The Vigor contributors develop and maintain a library of formally verified data structures (libVig in Figure B1.2), a small NF-specific operating system, the Vigor toolchain, and models of newly supported NICs (i.e., specifications of NIC behavior to be used for verification). They also provide a standard packet I/O framework (in our prototype, DPDK [249]) and NIC drivers, all of which are off-the-shelf but slightly patched. These patches are minor ($\approx$ 100 LOC added/removed) and include fixes for bugs that prevent correctness proofs from succeeding (see Section 6.2) plus adjustments to make DPDK verifiable. DPDK maintainers have fixed the bugs in the newer releases. We do not upstream the adjustments that enable verification of DPDK because they reduce the generality of DPDK, but we publish the patches in our repository [254]. They do not change the public API.

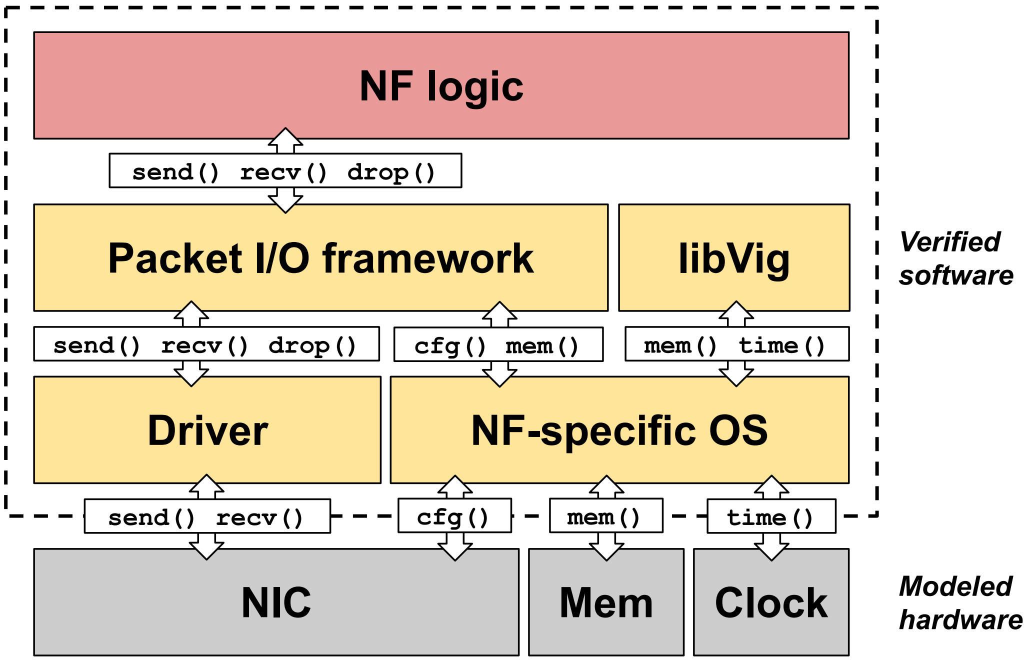


Figure B1.2: The Vigor stack for running NFs.

Standardization bodies write full specifications of NF functionality and publish them with the corresponding RFCs, IEEE standards, etc. Their goal is to reduce the confusion inherent to natural language and enable testing and mechanical verification of the standard's implementations.

1.2.2 Kinds of Specifications

Vigor verifies low-level properties (e.g., memory safety, crash freedom, hang freedom, absence of undefined behavior) and semantic properties, and it supports two types of semantic specifications:

1. **Full NF specifications** define the entire semantics of an NF. A standardization body might publish such a specification along with a new RFC, in order to
    - reduce the confusion inherent to natural language.
    - enable testing and mechanical verification of the RFC implementations.
2. **One-off properties** specify particular aspects of an NF implementation's behavior that perhaps are not even part of the RFC, such as "Could this load balancer ever send connections to those high-per-connection-cost nodes when

any of the inexpensive ones over here are under-utilized?" NF developers and operators have little interest in formulating an entire specification only to verify that such a one-off property holds in all possible circumstances. This ad-hoc way of verifying properties is particularly suitable when we are trying to learn things about the NF implementation in the context of its deployment rather than trying to prove that it is correct in an absolute sense.

NF developers write NF implementations in C, using the libVig API to store all state that persists across packets, and the I/O framework to send/receive packets. For each release of an NF, developers use Vigor in a push-button manner (Chapter 3) to verify the implementation against an RFC-derived specification. While developing the code, they use Vigor in a pay-as-you-go manner (Chapter 5) to check one-off properties and thereby to better understand their code, to debug it, and to converge onto a correct implementation more quickly. Neither mode requires verification expertise, but it does require NF developers to keep Vigor's limitations (Chapter 1 of the next part) in mind. Figure B1.3 illustrates this workflow.

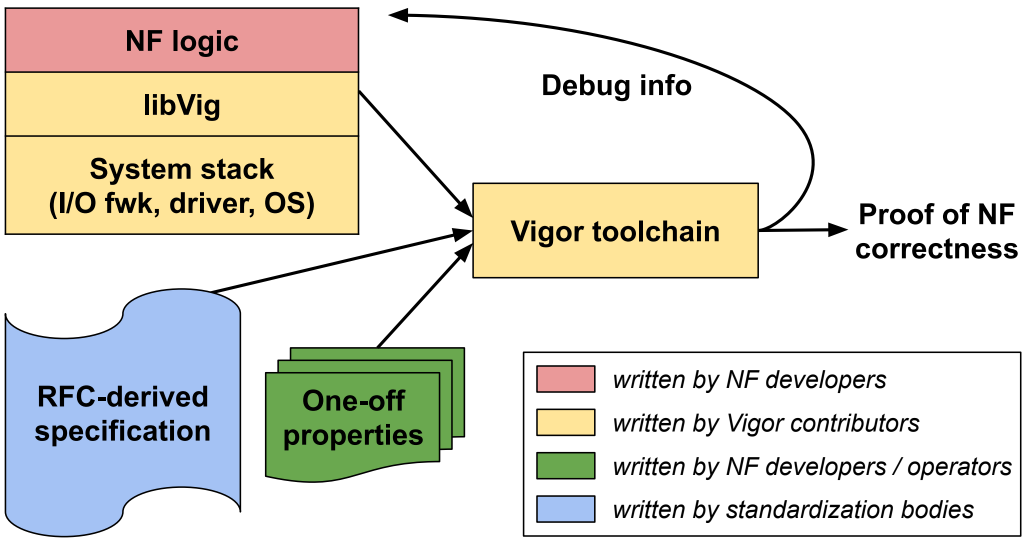

NF logic
libVig
System stack (I/O fwk, driver, OS)
Debug info
Vigor toolchain
Proof of NF correctness
RFC-derived specification
One-off properties
written by NF developers
written by Vigor contributors
written by NF developers / operators
written by standardization bodies

Figure B1.3: Vigor workflow: who writes what.

Similarly, NF operators verify NF implementations against both full specifications provided by standardization bodies and one-off properties specific to their setup (see Figure B1.3). Operators are not required to possess verification expertise, and they can use Vigor either in a push-button manner (e.g., against RFC-derived specifications) or in an ad-hoc pay-as-you-go manner (e.g., check whether the NF fulfills custom, “in-house” expectations, such as the load-balancer question above). They can also use the specifications and proofs to confidently employ network verification tools that reason about their network as a whole.

1.2.3 Verification Stages

The Vigor verification process has two stages.

1. **libVig verification**: For each libVig function, the Vigor contributors write a contract (i.e., pre- and post-conditions) and prove using a theorem prover that the function implementation satisfies this contract, i.e., given the preconditions, an invocation of the function leads to the post-conditions. Further details on how this verification is done appear in Section 2.3 of Part A.
2. **NF stack verification**: Vigor takes as input the code of the entire software stack, the contracts of the libVig functions called by the NF, and a specification to verify. It then uses a combination of symbolic execution and theorem proving (see Chapter 3) to prove that the NF running on the given stack does or does not satisfy the specification for all possible execution scenarios. If the proof fails, Vigor provides debug information to help pinpoint the cause. More details appear in Chapter 4.

Specifications for Vigor are written in a dialect of Python. Listing 1.1 shows an example. We believe that using Python makes writing specifications easier and friendlier for all three categories of Vigor users. Our dialect uses a small subset of Python, and we define its semantics in Appendix B. The most thorough formalization of Python subset we know of is [202]. It describes a subset of the language much larger than we need. We also slightly augmented the semantics to enable a more concise expression of NF specifications, see Appendix B for the details.

```
from state import macTable

macTable.expireOlder(now - EXP_TIME)
macTable[pkt.src_mac] = (port, now)

if pkt.dst_mac in macTable:
  out_port = macTable[pkt.dst_mac]
  if out_port == port:
    return DROP
  else:
    return out_port, pkt
else:
  return BROADCAST, pkt
```

Listing 1.1: Specification for a MAC learning bridge.

libVig provides a small number of data structures that we consider sufficient for writing most NFs: a single-key hashtable, a consistent hashtable, a longest-prefix-match table, a number allocator (a.k.a. ID manager), a packet queue, and a vector. With some further engineering, to complete the library, we can add verified crypto primitives [8, 70, 139, 236], verified regexp parsing primitives [4], and (pseudo) random generators (e.g. with the x86 rdrand instruction). libVig also provides primitives for reading system time, for parsing packet headers, and for expiring elements of the various data structures. As we show in Chapter 1 of the next part, we can reasonably expect this expanded version of libVig to be sufficient to build most NFs. Today, we consider libVig to be complete as far as research is considered, even if not in implementation. Compared to the data structure library designed for VigNAT only (described in Chapter 2 of Part A), libVig required some reorganization and reimplementation, as well as the addition of new primitives.

A key challenge in automating verification is to establish the correspondence between abstract state (used by the specification) and concrete state (used by the implementation). Vigor specifications are written in terms of abstract state that is at the same level of abstraction as that exposed by the libVig API; and Listing 1.1 illustrates this level of abstraction: `macTable` is a hash table with automatic expiration that in this case associates MAC addresses with the NIC IDs and an Ethernet header. The specification first describes the `macTable` updates: expiration and potentially learning of a new address. Then, it describes a decision tree for how the bridge responds to packets: forward, broadcast, or drop. The developer implements the NF by using similar abstractions: a set of packet headers and a hashtable with expiration. In Chapter 3, we discuss the reason we consider this to be the right level of abstraction for writing NF specifications: these are the common abstractions used in the networking domain.

Chapter 2

libVig: A Verified Library of Data Structures for Network Functions

LibVig trades off sophistication for reusability. By abandoning specialized and highly sophisticated data structures, we accommodate a wide range of NFs with a small number of generic components.

In order to increase code reuse, we replace the monolithic double keyed map data structure with two separate ones: a vector responsible for storing keys and values and a single-keyed map.

We implement the single-keyed map similar to the double-keyed hash map described in Section 2.1 of Part A. The difference is in the ownership strategy and the key-value memory layout. This map also features an interesting (single-character) optimization: for power-of-two capacities, we replace modulo operation with a bitwise mask when wrapping around indices. This brings in the order of 3x improvement in execution time for the lookup, insert, and delete methods compared to the version that used the slow modulo operation. This optimization demonstrates how there can be different implementations for the same set of API contracts. The map does not manage the keys but requires their addresses to remain valid and preserve the content unchanged, as long as they are in use. It accepts `void` (untyped) pointers to keys, as long as there are two operations defined for these pointers:

- `uint32_t hash(void* p)` : compute a 32-bit integer hash value, based on the data pointed to.
- `bool eq(void* p1, void* p2)` : compare two keys for equality.

Note that these functions themselves must know the structure of the key objects their arguments point to.

The operations need to be formally proven and to satisfy the following constraints:

- both functions are pure (i.e., have no side effects and compute the same value for the same pointers and the data pointed to)
- `eq(x, x) == true` (a key is always equal to itself)
- `eq(x, y)` ⇒ `hash(x) == hash(y)` The hash function applied to equal keys must produce equal hash values.

The logic of such functions is trivial and defined by the C-struct layout, which enables their automatic generation.

Vigor framework includes a code generator [254] that automatically creates the necessary functions based on the C-struct definitions provided by the NF developer. We also generate the logical definitions (an inductive datatype and a predicate) of the user structs. We generate VeriFast contracts, the implementation, and the proof annotations for the following operations independently for each C-struct `X` used in the NF:

- `uint32_t X_hash(void* p)`: A 32-bit hash function aggregating the `X` C-struct POD (plain old data, built-in integer types) fields by repeatedly applying the x86 crc32 built-in transformation function.
- `bool X_eq(void* p1, void* p2)`: A field-by-field comparison with recursive traversal for pointer fields.
- `void* X_allocate()`: A “do-nothing” function that takes a chunk of memory of the size of `X` and logically represents it as a memory chunk that holds the uninitialized logical value of `X`.
- `void X_log(void* p)` (not part of the verified base, so only implementation): A function that dumps all the POD fields with their names and recurses on the pointers.

We use the same mechanism to handle arbitrary protocol headers (with the statically fixed layout). `ether_hdr`, `ipv4_hdr`, `udp_hdr` are all described as plain C structures that our code generator uses to produce tracing layouts, manipulation functions, and logical definitions.

libVig also includes a few trivial data structures such as a static *array*, dynamic *vector*, and a *ring* buffer for the example in the previous part. libVig includes two more relatively sophisticated data structures at the moment: longest prefix match (LPM) table and a consistent hash table (CH).

LPM table is inevitable, as we support the IPv4 router, one of the fundamental algorithms of the Internet. We take the DPDK dir-24-8 LPM table. The table is optimized for lookup performance, as the router performs only lookups at runtime, and as initialization performance is much less important. We cleared out the cache and prefetcher hints, as they are not supported by VeriFast and, in our experience, do not affect performance.

The dir-24-8 LPM table implementation consists of a large array addressable by 24-bit index and a collection of small arrays addressable by the remaining 8 bits. The table performs a lookup in two stages. First, it indexes the large array with the first 24 bits of the key. Then, if the cell has the corresponding flag, it uses the value as an index of the short array that contains the mapping for the remaining 8 bits of the key.

Consistent hashing (CH) improves consistency in our Maglev [73]-inspired dynamic load balancer (LB). CH allows the LB to assign the same backend to the same client with high probability, even if the initial flow has expired and the set of active backends has changed.

The key element that makes the hashing mostly consistent is the permutation table. For each hash value, the table contains a unique permutation of the backend IDs. Whenever a new flow needs a backend ID, the algorithm searches in the permutation list corresponding to the hash of the flow-ID for the first backend that is online. This ensures that the same flow-ID is assigned to the same backend in most of the situations and that it always has an assignment, as long as there is at least one backend that is online. Note that this algorithm does not guarantee consistency: if a new flow appears when the first backend B in the permutation list for the corresponding hash is off-line, the flow will be switched to B once it comes online: thus it will automatically move from the backend that handled its creation to a different one.

Packet parsing is a non-trivial operation hence is a part of libVig. The library enables sequential reading chunks of the packet with known length. This mechanism successfully parses Ethernet, IPv4, TCP, and UDP headers. Given an unconstrained IHL (internet header length, the size of the IPv4 header) value, we can skip and ignore the IP options array, without bursting the number of symbolic paths. However, parsing the options requires reading them one by one, as a result still involves enumeration for all possible IP options counts.

Our parsing abstraction is a good fit for simple headers with a predefined number of fields of predefined length. It fails, however, as soon as the header contains fields with non-trivial length or number. For example, DNS packet parsing has to be done in the NF logic code and leads to path explosion.

In summary, libVig contains the following data structures proven correct:

- static *array* is a zero-cost abstraction of a plain statically allocated array.
- dynamic *vector* is a minimal abstraction over an array sized at startup time.
- *ID_manager* enables the user to allocate and to free integers (IDs) from a range and keeps track of access to them.
- *hash_table* is the single-keyed hash table described above that maintains no ownership over keys or values. Normally, it is used together with a *vector* to store the keys and *ID_manager* to manage the entries.
- *expirator* is layered on top of the *hash_table*, the key-holding *vector* and the *ID_manager* and performs the expiration and deallocation of expired flows from the table and the vector.
- *ring* is a classic ring buffer, introduced in Part A.
- *CH* table augments the *hash_table*, as described in [73], to implement consistent hashing.
- *LPM* (longest prefix match) table is the classic data structure used in IP routing. We implement an optimized version that limits keys length to 32 bits, and that also supports only a limited number of prefixes longer than 24. We copy the design described in the DPDK documentation for lpm_lib.

The implementations and proofs are available in the project repository [254] in the libvig/verified subdirectory. VeriFast, the proof checker, runs as part of the continuous integration routine to make sure the proofs hold upon any change of libVig.

Chapter 3

Push-Button Verification: Specifications in libVig-Level Terms

Push-button verification is a form of verification that does not require developer assistance: simply write the code and optionally provide a specification, then push a button, and out comes the proof. This stands in contrast to standard verification that requires users to write proofs or to guide the theorem prover with lemmas, usually as code annotations in a special proof language.

The Vigor approach has four key steps, illustrated in Figure B3.1:

1. NF developer runs exhaustive SE of the SE-friendly NF code to obtain all live paths through the code.
2. Validator tool converts the resulting function-call sequences to C programs, each one representing one path through the SE-friendly code.
3. Validator tool annotates each function-call sequence with lemmas corresponding to the NF specification.
4. Validator validates each annotated function-call sequence by using the pre- and post-conditions from the libVig API contracts, i.e., verify (using a theorem prover) that the SE-friendly code uses the libVig API correctly and that the resulting post-conditions imply the desired specification.

Once this process is completed, the theorem prover has formally verified that the NF is a correct implementation of the specification. As mentioned earlier, libVig is already verified and provides formal contracts for its API.

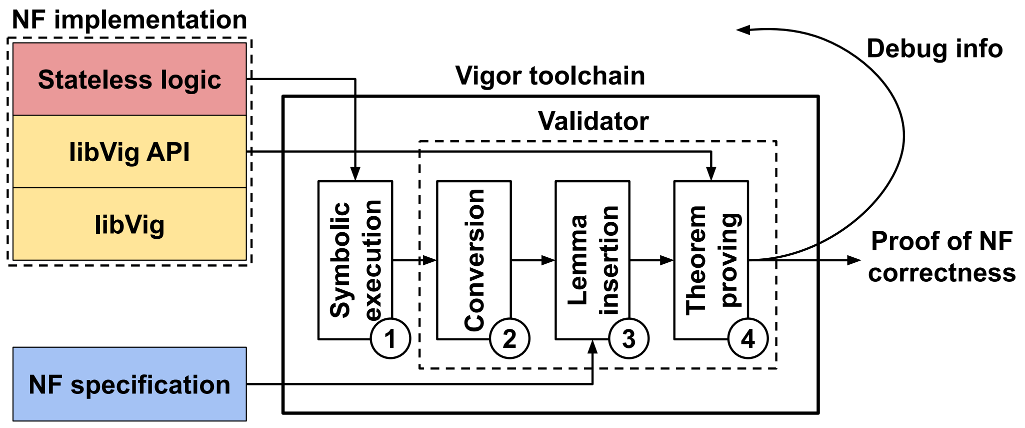


Figure B3.1: The Vigor verification process. Dashed lines group parts together to delineate "NF implementation" and "Validator".

The challenge in making the Validator (steps **2**, **3**, and **4**) generic across NFs, even when libVig is shared by all NFs, is that each NF uses the libVig data structures differently. Our first approach to generalizing the Validator was to design heuristics that infer how each NF uses libVig then to generate automatically the corresponding support lemmas in order to tie the API calls together. But, for each attempt, we always found a scenario in which our heuristics did not work. This pushed us to think differently about the problem, i.e., to find a way to side-step automatic lemma generation.

The key is to identify a "language" that enables efficient communication between specification writers/operators, developers, and verification tools. On the one hand, standardization bodies and NF operators use a certain vocabulary when they discuss NF functionality, and we should expect them to use the same vocabulary to write NF specifications. On the other hand, NF developers use the primitives provided by libVig to write NF implementations. So, to verify that an NF implementation honors an NF specification, the Vigor toolchain must correctly translate between (a) libVig primitives and (b) specification vocabulary. This can be hard, and we found that current verification tools cannot close non-trivial conceptual gaps between (a) and (b) without assistance from a human verification expert. Hence, if we want to achieve push-button verification today, the primitives used by NF specifications ought to be at the same level of abstraction as those provided by libVig.

To illustrate the importance of the level of abstraction, consider describing the behavior of a MAC-learning bridge. It could be described as an NF that lets all the directly connected endpoints to communicate with each other with the minimum amount of duplicate traffic. Or it could be described as in Listing 3.1. Or it could be phrased in terms of a hash table using open-addressing hash collision resolution and XOR of the two halves of a 6 byte MAC address as a hash function that stores source MAC addresses of all arriving packets in a contiguous array. Could we choose an abstraction level that makes it possible for specification writers, operators, developers, and tools to talk to each other, without the need for translation?

After reading RFCs, IEEE standards, and papers that specify NF functionality, we found that such a common language does exist and that it consists of a handful of popular data structures. For example, RFC 3022 [207] expresses NAT behavior in terms of a "translation table," and describes "mapping of tuples of the type (local IP address, local port number) to tuples of the type (registered IP address, assigned port number)." The IEEE standard 802.1Q [56] defines the "flow classification rules" of a MAC-learning bridge in terms of "destination MAC address, source MAC address, VID, or Priority." According to [73], the Maglev load-balancer's "connection tracking table uses a fixed-size hash table mapping 5-tuple hash values of packets to backends." NF specifications in general talk about network addresses, ports, flows, etc. and data structures that map these to relevant state; essentially the same primitives that developers use to write NF implementations.

We, therefore, designed libVig to provide primitives at a level of abstraction that enables not only NF developers to write NF implementations but that also enables specification writers to naturally express NF functionality. In theory, writing specifications with such primitives might seem inappropriate because they seemingly would end up dictating implementation details. In the specific case of NFs, however, we find that specifications *anyway* imply such implementation details, e.g., RFC 3022 essentially defines the NAT translation table as a map that translates between local and external IP addresses/ports. In fact, trying to define a NAT without using a map might even be confusing.

Using the same primitives in both the specification and the implementation reduces the conceptual gap between abstract state and implementation state. This makes the proofs simpler because the theorem prover needs fewer lemmas to match libVig call sequences in the function-call sequences with abstract-state manipulations in the specifications. In other words, instead of automating the human task of proving refinement from high-level abstract state to low-level implementation state, in Vigor, we remove the need for doing it in the first place.

Vigor requires NF developers to explicitly link the NF's persistent state with the specification's abstract state. The NF specification or one-off property manipulates abstract state (e.g., the specification in Listing 1.1 references on line 1 a MAC-learning table `macTable`). The NF implementation, written in C, manipulates a concrete state (e.g., the code in Listing 3.1 references `mac_table`). The correspondence between abstract and implementation state is made explicit by the developer via an `NF_EXPORT_STATE` macro, as on line 2 in Listing 3.1. This statement informs Vigor that the concrete instance `mac_table` corresponds to `macTable` in the specification.

```
 1  NF_STATE(mac_table, EMap, CAPACITY, ether_addr, uint32_t, 0, DEV_COUNT);
 2  NF_EXPORT_STATE(mac_table, macTable);
 3
 4  int main(int argc, char *argv[]) {
 5    while (1) {
 6      uint8_t in_port;
 7      packet_t pkt;
 8
 9      if (receive_packet(&in_port, &pkt)) {
10        expire_entries(mac_table, now - EXP_T);
11
12        if (map_has_key(mac_table, pkt.src_mac))
13          map_refresh(mac_table, pkt.src_mac, now);
14
15        map_put(mac_table, pkt.src_mac, in_port);
16
17        if (map_has(mac_table, pkt.dst_mac)) {
18          uint8_t out_port = map_get(mac_table, pkt.dst_mac);
19          if (out_port == in_port)
20            drop_packet(pkt);
21          else
22            forward_packet(out_port, pkt);
23        } else {
24          broadcast_packet(pkt);
25        }
26      }
27    }
28  }
```

Listing 3.1: Simple implementation of a MAC learning bridge.

To use libVig data structures, developers instantiate generic type templates from libVig. As part of this instantiation, they select parameters such as key and value types, and convey information on bounds (e.g., max number of elements in a vector) and on forbidden values (e.g., disallowed values for an external device ID). For example, the first two lines in Listing 3.1 declare `mac_table` as an `EMap` of total capacity CAPACITY with keys of type ether_addr and values of type 32-bit integer in the range 0..DEV_COUNT. This is akin to type instantiation in languages that are typed more precisely than C—this does not require verification expertise and poses little burden on modern developers. These declarations provide hints to Vigor that help it reason about the code. For example, it uses the bounds information to verify both that the bounds are observed and that values can be safely used as an index into the array of NICs. Listing 3.2 shows the API of the EMap data structure in libVig.

```
void expire_entries(struct EMap* map, time_t t);
void map_refresh(struct EMap* map, void* key, time_t t);
void map_put(struct EMap* map, void* key, value_t value);
bool map_has(struct EMap* map, void* key);
value_t map_get(struct EMap* map, void* key);
```

Listing 3.2: libVig's map with automatic expiration.

Vigor, in essence, automatically generates a custom Validator for each NF, which was at first done by hand for VigNAT (see Chapter 4 of Part A). Using the information in the template instantiations above, Vigor automatically produces lemmas to annotate the function-call sequences used in the verification process. The Vigor Validator uses these declarations to choose which lemmas to instantiate and how to combine them to verify the NF's data structure usage patterns.

There is an interesting design question about what the "right" level of abstraction for libVig is. An alternative to the "low level" primitives provided by libVig would be the interface provided by the Click modular router [128]: construct an NF by connecting various packet processing modules, called elements. Although this makes it easy to quickly compose NFs, [171] finds that "in practice, when implementing new NF functionality, developers commonly need to implement new modules, and optimizing the performance of such a module is difficult and time-consuming." We reached a similar conclusion (see Figure B6.3): With libVig's small number of data structures, we can write a large number of NFs, whereas a higher-level interface requires periodic implementations of new components underneath that interface. So far, experience suggests that the choice of abstraction presented here works well for both specifying and implementing the NF state.

It might seem that writing NFs requires verification knowledge (to formulate the loop invariants), but with Vigor, this is not the case in practice. In the case of a general program, writing invariants that the proof toolchain understands and can take advantage of requires understanding how the toolchain performs verification. However, in the case of NFs, with each processed packet the NF transitions between clearly defined states. Such valid states can be described by generic invariants, such as the data structure being consistent, or the state automaton being in one of the few allowed states. In our experience, the state declaration (`NF_STATE`) is sufficient for filling in the NF specific parameters of such invariants. We claim that NFs are mostly expressed in terms of a small set of common abstractions (see Figure B6.2 and Figure B6.3 in Section 6.4), and we argue that it makes sense to write NF specifications in terms of these abstractions. It then becomes natural that the invariants on the specification's abstract state are the same as on the implementation state.

For example, Vigor uses the declarations at the top of Listing 3.1 to automatically compose a loop invariant that describes the state that persists across the event loop iterations, including the user-provided constraints on the table values. The generated loop invariant is:

- `mac_table` is a valid `EMap[ether_addr -> uint32]`.
- `mac_table` has up to `CAPACITY` entries.
- `mac_table` owns all the values and keys (none is borrowed by the NF code).
- Each value in `mac_table` lies between `0` (inclusive) and `dev_count` (exclusive).
- `argc` equals to `argc0` (its value before the loop).
- `argv` equals to `argv0` (its value before the loop).

In summary, the turning point for enabling push-button verification in Vigor was choosing a common level of abstraction for the state used by NF specifications and NF implementations. This choice makes it easier for Vigor to reason about the correspondence between abstract state and implementation state, which enables it to automatically generate the supporting lemmas needed for verification. This automation, in turn, means that NF developers no longer need to modify the toolchain for each new NF, as a result, Vigor generalizes across NFs.

Our experience in developing five NFs suggests that this level of abstraction works well for both specifying and implementing NFs. In Section 6.4, we attempt to partly quantify why.

Chapter 4

Full-Stack Verification: Harnessing Exhaustive Symbolic Execution

To make the vision of NF virtualization (NFV) [47, 95] come true, we argue for formal guarantees of correctness that cover middlebox software in its entirety. Vigor focuses on the data-plane software stack—the NF and its libraries, the packet-processing framework, the operating system, and the device drivers. But our argument also applies to the control plane, as well as to any compiler used to generate the middlebox code. When deploying hardware middleboxes, operators have to trust only hardware, not software. For them to make the switch to NFs with peace of mind, they need strong guarantees that hardware (and humans) are the only source of failures.

Merely verifying the NF code is not sufficient, especially when using a kernel-bypass framework such as DPDK, because a bug or security vulnerability in the framework can compromise the entire machine that runs the middlebox. We have direct evidence that blindly trusting the layers underneath the NF is unsafe: during verification, we found bugs in both DPDK and an Intel NIC driver (see Section 6.2).

The following observation enables Vigor to verify the full NF stack: even when an NF stack is too big to be verified efficiently, the fraction of the stack used to accomplish a specific purpose is likely small [178]. In the case of the NFs we wrote, only $\approx$ 3.5% of DPDK's roughly 62 KLOC end up ever running, respectively

$\approx$ 18% of the Intel 82599ES NIC driver's 24 KLOC. The rest of the code is dead as far as the execution of that NF is concerned. Therefore, it makes sense to use exhaustive symbolic execution of the NF and to not model the library level (as described in the previous part and done in most NF verification work), but rather to model the *hardware layer.* The NF calls the NIC driver directly and uses only simple DPDK utility functions that we can symbolically execute; most of DPDK's functionality is only used during initialization that is executed concretely (i.e., not with symbolic input). The NIC driver code is a sequence of reads and writes to hardware, with little branching.

Unfortunately, even informed by this observation we cannot scale the straightforward symbolic execution approach when running on a commodity operating system. Even though packet processing frameworks such as DPDK bypass the kernel, and the operating system gets out of the way shortly after setup, it is still the case that the operating system could affect the NF any time after setup (e.g., by preempting the NF, unmapping its memory pages, reconfiguring the way the NF talks to hardware). This is because a large part of the commodity operating system remains live, even if it is not directly invoked by the NF or the packet processing framework. Consequently, we would need to prove isolation between the operating system and the packet processing framework, and doing this is hard. Using symbolic execution would result in path explosion, and interactively verifying Linux is not practical. For reference, tools such as VeriFast [110] require $\approx$ 10 lines of annotations for each line of code, and the Linux kernel contains over 25 MLOC [250].

Our solution to this challenge is a small custom operating system that can be automatically verified. We think of this as an NF-specific operating system or NFOS. The NFOS performs the necessary setup for the packet I/O framework to take over: sets up the CPU and memory, scans the PCI bus to find all available NICs, configures the NICs, then runs the NF. Then it gets out of the way; and this we can prove formally. Beyond regular setup, the NFOS also provides timing information from the hardware to the NF (see Figure B1.2). We use this information, for example, to expire TCP flows, MAC address entries or to police IP traffic. As a side benefit, as is discussed in Chapter 2 of the next part, an NFOS also offers the opportunity to build NF domain-specific abstractions.

Being specifically intended for running NFs built with Vigor, there are many features the NFOS simply does not need to do. Given that Vigor formally verifies the NF, DPDK, and the NIC driver to be crash-free and memory-safe, the NFOS need not provide any inter-process isolation. Kernel-bypass frameworks such as DPDK often come with their own NIC drivers that use polling rather than interrupts. This means that the NFOS need not include device drivers or provide support for handling interrupts. Finally, given that NFs in Vigor are currently single-threaded (see Chapter 2 of the next part for a discussion), the NFOS does not need a scheduler. This lean design ensures that the NFOS can be symbolically executed exhaustively.

This makes it possible to achieve push-button verification of not only the NF but also DPDK, the NIC driver, and the operating system. In the case of the NFs we wrote, this results in symbolically executing on average $\approx$ 6.5 KLOC of DPDK, operating system, and driver code for each NF, with absolute certainty that the remainder of $\approx$ 82.5 KLOC is dead, hence cannot influence the correctness of this NF. In contrast, verifying, just the NF (e.g., VigNAT) code itself would cover about 1 KLOC of NF code. Our use of exhaustive symbolic execution formally proves the complete separation between the part of the stack that is invoked by the specific NF being verified and the rest of the code. In other words, the observation above enables Vigor to prove that most of the code in the stack running underneath the NF is dead as far as the NF is concerned, i.e., it would never have the opportunity to run. Verification then focuses automatically, for each NF, on the live code only. Of course, the part of the code that is dead might differ from NF to NF.

The only part of the NFOS that we do not symbolically execute is a piece of code that consists of 140 `x86` assembly instructions run during boot, 170 lines of C code to scan the PCI bus, and 40 lines of C code for a trivial memory allocator. Early startup code is typically excluded from operating system verification due to a simple trade-off: we would need to formally define and/or model large amounts of hardware details in order to prove correct tiny, straightforward code fragments. In our case, we argue the code's correctness with detailed code comments that explain why each instruction and LOC has the intended effects [254]. Our correctness argument was vetted by several developers other than the NFOS author. However, to be conservative, we include this code in Vigor's TCB.

An alternative to our approach is to verify the packet-processing framework, NIC drivers, and operating system in isolation. For the operating system and drivers, we could have used an approach similar to seL4 [126] or Hyperkernel [161]; this would require translating the operating system interface contracts to stitch with our proofs, and would likely impose a performance penalty on the resulting NF. For

DPDK, verifying the entire framework through exhaustive symbolic execution is impractical due to path explosion. Interactively verifying it with a theorem prover is not practical either, due to the human effort it entails, and to the fact that it would have to be repeated for every new release. Furthermore, as DPDK contains bugs, a verification effort would have to either fix them or explicitly specify the conditions under which DPDK is correct. Overall, we believe that enabling the NF code to implicitly indicate which parts of the stack it needs, and then letting Vigor automatically verify only those parts, is a more practical approach.

In conclusion, Vigor's full-stack verification reduces the TCB to a level we expect NF operators to be comfortable with: the early startup code mentioned above, the hardware, the clang and GCC compilers, and the Vigor toolchain. This toolchain is composed of the KLEE symbolic executor [36], the VeriFast theorem prover [110], and the Vigor Validator. The Vigor stack uses the unverified GRUB bootloader, but it could be replaced with a formally verified one [50].

Just as in the case of push-button verification being possible for other systems domains beyond NFs, we conjecture that writing NF-specific operating systems is also feasible in other systems domains (Chapter 2 of the next part) and could make full-stack verification possible in those domains.

Chapter 5

Pay-As-You-Go Verification by Using Open-World Specifications

Any verification task has a fixed cost and a variable cost, and pay-as-you-go verification is about lowering the fixed cost as much as possible. The fixed cost is dominated by writing a complete specification that ends up including many properties that are not needed for verifying the first line of code. In Vigor, developers need only to write a sufficient specification to cover the property they seek to verify. Thus, it becomes possible to specify a single property, verify the NF against that property, fix any bugs, write another property, and so on. This feature of Vigor is inspired by [166], and our use of the term "pay-as-you-go" was suggested to us by the author.

Vigor enables pay-as-you-go verification by changing the closed-world assumption present in VigNAT to an open-world assumption [166]. In the logic under a closed-world assumption, a true statement is always known to be true; as a result, a specification must include or imply all true properties of the NF code. Any properties that are not part of, or implied by, the specification, are deemed to not hold. This imposes the need to write the entire specification before verifying any property of the system; and complete specifications are often long and tedious. As a result, practitioners are dissuaded from writing them. Under an open-world assumption, however, what is not known to be true is simply unknown rather than false. This removes the need for all knowledge about the NF's properties to be captured at once in a specification.

First, Vigor enables writing specifications in a compositional way, by using references. The open-world assumption is amenable to an incremental build-up of knowledge about the NF's properties, and this matches well the modern software development process. For instance, a developer can write a specification saying "The bridge always learns the association between the source MAC address and the port" (Listing 5.1). Then, they can write another specification saying "The bridge forwards the frame according to the association in its MAC address table" and reference the previous specification (Listing 5.2). This specifies that on top of manipulating the MAC address table, a bridge forwards the frames according to the corresponding entry. Both Listing 5.1 and Listing 5.2 are sound specifications, in the sense that they specify a bridge behavior that must hold for *all* possible executions. This compositional approach dovetails with the monotonicity of first-order separation logic used in our toolchain: adding new information cannot falsify a previous conclusion.

```
from state import macTable
macTable.expireOlder(now - EXP_TIME)
macTable[pkt.src_mac] = (port, now)
```

Listing 5.1: Example property specification: after expiring state entries, the bridge learns the source MAC address of a frame.

```
import bridge_learn
if pkt.dst_mac in macTable:
  out_port = macTable[pkt.dst_mac]
  if out_port == port:
    return DROP
  else:
    return out_port, pkt
else:
  return BROADCAST, pkt
```

Listing 5.2: Composed specification: the bridge forwards a frame according to the entry in the MAC address table.

Second, in Vigor, one-off properties can be described with small specifications that exclude, via the Python `pass` keyword, knowledge about an arbitrarily large number of NF properties that remain unspecified. For instance, if we wanted to focus only on the broadcast case shown in Listing 5.2, we would use `pass` for other behaviors, as shown in Listing 5.3. Here, Vigor only checks that, for unknown MAC addresses, the frame is indeed broadcast, ignoring the I/O and state changes for known MAC addresses. This property enables the developer to focus on one aspect of NF behavior at a time, while still obtaining formal guarantees.

```
import bridge_learn
if pkt.dst_mac in macTable:
  pass
else:
  return BROADCAST, pkt
```

Listing 5.3: Broadcast case in the MAC-learning bridge specification.

One-off properties in Vigor can be independent of the implementation, or specific to it. Independent properties are those implied by the NF standard (such as the properties in Listing 5.1 and Listing 5.2), or by a combination of the standard and specific configuration or deployment details (such as "a correct Maglev load balancer never sends connections to my high-per-connection-cost nodes over here when any of the cheap nodes over there are under-utilized"). These properties do not have to be literal subspecifications of the complete specification, they can generalize the behavior of the NF from a different perspective. For example, Listing 5.4 demands that the bridge never echoes a frame back to where it arrived from. This is an important property on its own and is independent of the learning functionality. It is necessary to prove the absence of trivial loops.

```
assert not member_of(received_on_port, sent_on_ports)
```

Listing 5.4: One-off property: Bridge does not echo frames.

Implementation-specific properties capture the behaviors of the NF implementation that are outside the standardized specification. These might or might not appear in the NF's own specification, but they are absent from the official specification. For example, NAT middleboxes that correctly implement RFC 3022 are free to choose what to do with a non-SYN TCP packet that arrives from the internal network and does not correspond to an established flow: forwarding or dropping it are both correct behaviors according to the RFC. An operator might want to know whether the NAT she is about to deploy would forward such spurious TCP packets to the external network. Or perhaps she uses network verification tools, and no NAT model captures the detailed non-SYN behavior of this particular NAT implementation. She would formulate this property, then ask Vigor to verify it against the NAT implementation. If it verifies, then she can add it to the model.

In Vigor, it is possible not only to choose relevant parts of the specification but also to select which layers of the stack to verify. The NF developer can choose whether to verify only their NF code or to include additional parts of the stack, such as DPDK. Verifying more code provides more guarantees but also takes more time. To enable the developer to choose which parts of the stack they want to verify, we wrote models for each layer in the stack. As we show in Table B6.3, depending on how much of the stack one wants to verify, the developer can speed up verification significantly. It makes sense for NF developers to use partial-stack verification during their daily development workflow and to use full-stack verification before each release of the software.

Pay-as-you-go verification makes it less daunting to write standards as precise specifications. First, with pay-as-you-go, specifications can be written incrementally, during the development of a reference implementation for an NF by a standardization body. Second, even in the absence of an initial formalization of an RFC by a standardization body, development teams building NF implementations could collaboratively formalize the RFC as they write one-off properties to verify their code; and these properties can then be curated into a unified specification of the RFC. Third, as specifying properties requires only Python knowledge, network operators could write specifications that correspond to their own data center or network setup, which NF developers and vendors would likely find useful.

To complement pay-as-you-go verification, Vigor also helps NF developers pinpoint whether a proof failure is the result of an erroneous specification or erroneous implementation (see “Debug info” edge in Figure B1.3). As it often happens, semantic specifications can be formulated after the implementation is done. When verifying against an erroneous specification, our tool reports the function-call sequence that corresponds to the failed validation. This function-call sequence is essentially a piece of linear C code representing the execution that led to the alleged property violation, and it makes it easy for the developer to understand the mismatch between the formulated property and the discovered behavior. If the property specification turns out to be erroneous, the developer can use this single function-call sequence as a test case for the fixed specification, without having to re-run the verification workflow. For instance, if Listing 5.2 was missing the `DROP` case on lines 4 and 5, Vigor would produce the function-call sequence leading to line 20 in Listing 3.1 and flag a mismatch. The developer can then use this function-call sequence to correctly formulate the `DROP` case and skip running the symbolic execution and validation of the other function-call sequences.

The three aspects described in this section—one-off properties and composable specifications, verification by layers, and debugging by counter-example—enable an incremental form of push-button, full-stack verification. They enable developers and operators to benefit from the formal verification quickly, with little upfront cost.

5.1 Lower-Level Abstractions to Reduce Specification Length

We described, in Chapter 3, how lowering the level of abstraction for writing specifications facilitates push-button verification. This lower level of abstraction enables describing the state in terms of basic, generic data structures (maps, vectors, queues, etc.) that have well-understood semantics. As a result, a specification expressed in terms of such data structures needs not to define the data structures but merely to reference them (via `import` in Vigor's case). This makes the specification shorter, as well as more comfortable to write. In contrast, the initial specification for VigNAT [234] consisted of ≈ 150 lines of Verifast (a subset of C extended with inductive data types and some generics), and ≈ 150 lines that defined the abstract state independently of libVig definitions. Now, with generalized Vigor, the VigNAT specification has 47 lines of Python and 19 lines that, in terms of libVig data structures, define the abstract state.

Chapter 6

Evaluation

We experimentally show that Vigor does not trade performance for guaranteed behavior. We present five typical NFs implemented with Vigor—a NAT, a Maglev-like load-balancer, a learning bridge, a traffic policer, and a stateful firewall. And show that each performs on par with a standard, non-verified NF that implements the same functionality and runs on the same stack.

In this section, we evaluate the main aspects of Vigor by answering the following four questions:

1. Does Vigor generalize? In Section 6.1, we show that the answer is yes: we developed five NFs that cover a variety of representative NF features, and we push-button verified them with Vigor.
2. Does verification have tangible benefits? In Section 6.2, we discuss how Vigor helped prevent four bugs in NF code and discovered eight bugs in DPDK and an Intel NIC driver.
3. Does verification come at the price of performance? In Section 6.3, we show that it does not: NFs verified with Vigor perform on-par with third-party alternatives.
4. Does verification come at the price of reduced productivity? In Section 6.4, we show that using Vigor is easy and can improve productivity by helping the developer write correct code faster. Our code, specifications, proofs, and results are available [254].

6.1 Does Vigor Generalize?

To evaluate the extent to which verification with Vigor generalizes, we used it to develop and verify the five NFs shown in Table B6.1. These NFs cover a wide range of features found in NFs. Some notable NF types are missing, such as intrusion-detection systems, because our current prototype of libVig does not yet provide primitives for regular-expression matching or cryptography (see Chapter 1 of the next part). For each NF, we wrote a full specification of its behavior and used Vigor to prove that the NF correctly implements that specification. Section 6.4 provides details on the size of the specifications.

Table B6.1: The NFs we developed and verified with Vigor.

Name	Description	Class of NFs
VigNAT	Network address translator	Per-flow state; Header rewriting
VigBridge	Eth bridge with MAC learning	Packet duplication
VigLB	Load balancer	Per-flow state; Consistent hashing
VigPol	Traffic policer	Per-flow state; Fine-grained timing
VigFW	Firewall	Per-flow state

VigFW Standard stateful firewall that interconnects an internal and external network and permits traffic only from connections that originated in the internal network. This is representative of NFs that keep per-flow state.

VigNAT Standard Network Address Translator that rewrites the source (destination) IP address and TCP port number of the outbound (inbound) traffic. This is representative of NFs that keep per-flow state and additionally perform header rewriting.

VigLB Implementation of Google's Maglev load-balancer algorithm [73] that sends traffic from the same TCP connection to the same back end. This is representative of NFs that keep per-flow state and implement consistent hashing [117].

VigBridge Standard Layer-2 bridge that learns MAC address/output port mappings by observing traffic and broadcasts packets destined to MAC addresses for which it has not learned a mapping yet. This is representative of NFs that generate multiple copies of an incoming packet.

VigPol Standard policer that rate-limits traffic per source IP address. This is representative of NFs that employ fine-grained timing information in their packet processing.

For every row in Table B6.1, all software is verified except for the Vigor toolchain, the GRUB bootloader, and compilers. Table B6.2 shows the size in LOC of each layer of the Vigor stack that is verified. As explained in Chapter 4, the entire stack is mechanically verified, except for $\approx$ 350 LOC of assembly and C, whose correctness argued by hand. We reiterate that we could replace GRUB with a formally verified bootloader [50] thus eliminate it from the TCB.

Table B6.2: Size of each layer in the Vigor stack.

NF	LOC	Stack Layer	LOC
VigNAT	969	libVig	1,674
VigBridge	815	KLEE-uClibc (libc)	60,556
VigLB	850	DPDK	62,380
VigPol	725	Ixgbe Driver	24,211
VigFW	754	NFOS	1,958

Table B6.3 shows the amount of time it takes to complete the verification. We measured three scenarios:

1. verifying only the NF code against the full specification,
2. verifying the NF with DPDK, driver, and libC, and finally
3. adding the NFOS as well, to verify the entire software stack.

The difference between verifying with or without NFOS is negligible ($\pm$ 20 sec), hence we report the first and third scenarios only. Total verification time is the sum of the time for exhaustive symbolic execution to obtain the function-call sequences (columns $2+3$) plus the time to validate all the function-call sequences (columns $4+5$ multiplied by column 6). We report validation time as *# of seqs* $\times$ *per-seq validation time* because validation is an "embarrassingly" parallel task, as a result, total completion time depends linearly on the number of thread contexts available. The reported number of function-call sequences corresponds to code paths analyzed using symbolic models for the hardware and the libVig data structures and *after* loop havoc-ing is applied (see Chapter 3); otherwise, the number of function-call sequences would be infinite.

Table B6.3: Verification statistics.

NF	**Symbolic execution time**		**# of seqs**			**Per-seq validation time (avg)**
	NF only	*with rest of stack*	*NF only*	*with rest of stack*		
VigNAT	7 sec	+8 min	54	+434	$\times$	88 sec
VigBridge	7 sec	+10 min	69	+542	$\times$	80 sec
VigLB	23 sec	+26 min	146	$+1,190$	$\times$	219 sec
VigPol	14 sec	+6 min	37	+272	$\times$	82 sec
VigFW	6 sec	+7 min	43	+326	$\times$	88 sec

For the verification measurements, we used a setup consisting of Intel DPDK v.17.11 for the packet I/O, with the ixgbe driver for the Intel 82599ES NIC. We ran the verification on a dual-socket Intel Xeon Gold 6132 machine @ 2.6 GHz, which provided a total of 28 cores (56 thread contexts). Full-stack verification consumed $<$ 700 GB of DRAM, and verifying only the NF took $<$ 2 GB; the machine had 1.48 TB available. Each NF was configured with table sizes of 65,536 entries.

The data shows that verifying the NF code alone is fast enough to be done on every commit to the code repository, or as part of continuous integration, and that verifying the full software stack is fast enough to be done at least once per release cycle. It is, however, possible to significantly speed up the validation phase through parallelization because each function-call sequence can be validated independently from all others. In the extreme, validating each function-call sequence on a separate CPU core would make the overall process complete in minutes. On our one machine, it took as long as several hours of wall-clock time; if a large cluster was devoted to verification, then it would become feasible for the full stack to be verified on every commit.

The growth of the number of function-call sequences is typically superlinear over the code size. This is due to the nature of the function-call sequences. A function-call sequence is homomorphic to a prefix of an execution path. If the number of paths, as well as their average length, grows linearly, their prefixes' growth rate is between linear and quadratic, depending on the distribution of branches in the execution tree.

In summary, we developed and verified five varied and representative NFs with Vigor, thus showing that the Vigor approach generalizes to multiple kinds of NFs. The verification time matches well the patterns of modern software development. Therefore, we conclude that Vigor can provide practical push-button, full-stack verification for NFs.

6.2 Does Verification Have Tangible Benefits?

Formal verification traditionally offers at least two benefits: it prevents *all* implementation bugs from making it into released code, and it increases users' confidence in the software. The latter is hard to assess without the experience of wide, real-world deployment, but we focus on the former and report our own experience. As Vigor verifies both semantic properties and low-level properties such as memory safety, it can identify both high-level bugs (e.g., packet-injection vulnerabilities) and low-level ones (e.g., incorrect writes to reserved bits). Table B6.4 shows some of the bugs Vigor found in our own, as well as in third-party code.

Table B6.4: Bugs uncovered by Vigor, and the step in the Vigor workflow that discovered each bug: libVig verification (LV), symbolic execution of the NF logic (SE), or validation (VA).

Stack Layer	**Bug Description**	**Vigor Step**
NF	Incorrect use of by-ref parameter	VA
	Missing checks for packet spoofing	VA
	Incorrect use of the IP header	VA
	Infinite loop in consistent hashing	LV
DPDK	Out-of-bounds array access	SE
	Incorrect use of *mmap()*	SE
	Incorrect use of *libnuma*	SE
ixgbe NIC driver	Out-of-order register write	SE
	Use of potentially invalid NIC register	SE
	Incorrect timing of register write	SE
	Incorrect writes to reserved bits	SE
	Write to unknown NIC registers	SE

First, using Vigor during the development of our NFs helped us to identify numerous bugs in the code, ranging from crashes and infinite loops to packet injection vulnerabilities and packet corruption. Most of the bugs that are not low-level (such as crashes or hangs) were caught during the last validation stage. This is because they are of two types:

(1) violations of the use of contracts for the libVig API, and
(2) violations of the NF's semantics.

As neither the symbolic-execution phase nor theorem proving can witness both sides of the libVig API on their own, neither of them can individually catch bugs of type (1). As the symbolic execution phase runs on the SE-friendly code, it is unaware of the higher level NF semantics (which get checked during the validation phase), as a result, it misses all bugs of type (2).

Second, by doing full-stack verification, Vigor uncovered existing bugs in DPDK and the Intel ixgbe driver we used. There were nine bugs in DPDK and the driver [240–248]; five of them have been confirmed by the DPDK maintainers of which three have been fixed at the time of this writing. These bugs primarily result from nuanced corner cases in the code. As Vigor uses exhaustive symbolic execution, it discovers corner cases that typically elude a developer. For instance, we found that DPDK can crash during initialization if the first 128 CPU cores are disabled. Although this scenario is unlikely today, it would be hard to debug when and if it occurred.

In summary, our experience demonstrates the benefit of using verification during development, as well as the use of full-stack verification. By preventing many bugs from making it into our final code, Vigor saved us a substantial amount of debugging time. By identifying bugs in the off-the-shelf software we were relying on, it further saved us from debugging unknown code. If real operators were to use the NFs we wrote, they would probably feel more confident.

6.3 Does Verification Come at the Expense of Performance?

Whenever verification is discussed for real-world use, the elephant in the room is performance: in many cases, verified code tends to perform more poorly than its unverified counterpart. In this section, we show that writing verified NFs with Vigor does not have to compromise performance.

We compare each Vigor NF to a baseline that provides similar functionality but was not developed with verification in mind. All baselines are popular third-party NFs that offer competitive performance. Ideally, each baseline would provide functionality identical to the corresponding Vigor NF, but we were unable to find popular third-party NFs that satisfied this; we could have implemented our own baselines, but then we could not have argued that they are representative unverified NFs. Instead, for each Vigor NF, we selected the baseline that, in the context of our experiments, provided as much as possible (but always a subset of) the functionality provided by the Vigor NF. Comparing to these baselines is a good indicator of whether verification hurts performance. Although their functionality is not identical to the respective Vigor NF, the baselines serve as an indicator of the expected performance of a deployable NF.

For all NFs except VigPol, the baseline is a chain of standard Click [128] elements. Click is arguably one of the most popular ways to implement high-performance NFs today. For VigPol, we could not find appropriate Click elements, so the baseline is the Moonpol open-source policer [252] that relies on a DPDK-based framework [74]. All the baselines run on a standard stack: DPDK v.17.11 and Ubuntu Linux with kernel 4.15.0-55-generic. The Click baselines run on the latest stable version of Click (v2.1).

We run the Click baselines and Moonpol, with and without batching. Receiving and sending packets in batches can amortize the cost of certain bookkeeping operations (e.g., updating memory-mapped NIC registers), thus increasing throughput at the cost of higher latency. Vigor does not yet support batching (see Chapter 1 of the next part), but the Click baselines do, therefore we run them both with batching (throughput-optimized) and without (latency-optimized). For batching, we use the default Click batching parameters.

To assess whether performance differences are due to the NF code or the layers underneath the NF, we also measure a no-op NF that receives packets at a fixed input port and forwards them to a fixed output port without any processing.

In summary, for each NF type, we compare the Vigor NF running on top of the Vigor stack to a baseline NF running on top of the standard Linux stack, with and without batching. We also ran Vigor NFs on the standard Linux stack, and their performance is the same as on top of the Vigor stack.

Similarly to the experiment in the previous part, we use the testbed, as per RFC 2544 [28], identical to the one used in the previous part, shown again on Figure B6.1. The Tester and Middlebox machines are identical, with an Intel Xeon E5-2667 v2 processor at 3.30 GHz, 32 GB of DRAM, and 82599ES 10 Gbps DPDK-compatible NICs. The Tester machine runs MoonGen [74] to generate traffic and measure packet loss, throughput, and latency; for the latency measurements, we rely on hardware timestamps for better accuracy [184]. In each experiment, the Middlebox machine runs either our NF or a baseline, always on a single core.

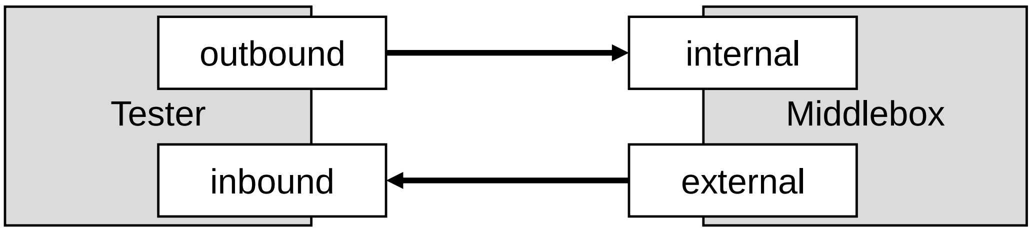


Figure B6.1: Testbed topology for performance evaluation (identical to Figure A5.1).

For each NF, we measure latency and throughput. In particular, for each NF, we generate

(a) "background traffic" that causes the NF's main data structure to reach 90% occupancy; and

(b) "probe packets" that always correspond to a new flow ID, and thus exercise the longest path in the NF.

For example, in the case of a NAT, the background traffic consists of established, long-running TCP flows, whereas the probe packets consist of packets from new, short-running TCP flows. Vigor NFs receive the same traffic as their baselines. We report the overall throughput achieved by each NF when it starts dropping 0.1% of the packets it receives. We also report the latency experienced *by the probe packets* because they experience the worst latency (each of them has to wait until the entire data structure, which is 90% full, is searched). This methodology measures performance under stressful but realistic conditions—an operator would not provision a middlebox to operate at more than 90% occupancy. Under less stressful conditions, the performance differences are less evident.

Table B6.5 shows the results. Each reported latency number is the average latency experienced by all the 100K probe packets in a single experiment. For Vigor NFs and their no-batching baselines, the standard deviation of the latency across different packets in the same experiment is $100 - 200$ nanoseconds. Both average latency and standard deviation are stable across different runs of the same experiment.

Table B6.5: Throughput and latency of Vigor NFs and the corresponding baselines.

NF type	**Vigor**		**Baseline (no batch)**		**Baseline (batch)**	
	Latency (μsec)	Thruput ($Mpps$)	Latency (μsec)	Thruput ($Mpps$)	Latency (μsec)	Thruput ($Mpps$)
NOP	3.90	8.27	4.62	4.07	15.51	14.7*
NAT	4.07	4.86	5.59	1.63	16.30	2.80
Bridge	4.07	4.94	4.76	2.88	15.84	11.2
Load Balancer	4.12	4.02	7.24	1.63	16.26	2.79
Policer	4.03	5.21	5.28†	2.91†	5.20†	11.5†
Firewall	4.02	5.36	5.59	1.63	16.19	2.79

*10 Gbps saturated † Moonpol

First, we see that our NFs have similar latency to their latency-optimized (no-batching) baselines. This is not surprising: The only fundamental difference between baseline and our NF is that the latter uses libVig, and there is no reason a libVig data structure should be slower or faster than any other data structure.

Second, our NFs have a throughput that is on par with their baselines, including the throughput-optimized (batching) ones, with the exception of VigBridge and VigPol. In some cases, our NFs have higher throughput (e.g., VigNOP achieves 74% better throughput than its no-batching baseline), but this has little to do with the Vigor approach. It is mainly due to the fact that Click loads modules dynamically, which disables a number of the link-time optimizations (LTO) that apply to the Vigor stack. We confirmed this explanation by disabling the optimizations for our stack and by finding that the differences vanish. Not surprisingly, in the case of the no-op NF, batching has ample room to compensate for this, so the batching baseline saturates the network interface. In the real NFs, however, there is less opportunity to exploit batching (i.e., to pipeline the processing of consecutive packets), as a result, our NFs achieve higher throughput than even some batching baselines.

Naturally, when batching is enabled in the baselines (Vigor currently does not support it), we witness a significant improvement in throughput due to

1) the use of SIMD instructions for processing multiple packets simultaneously and

2) the concomitant improvement in instruction cache locality, prefetching effectiveness, and branch prediction accuracy.

However, this is also accompanied by a corresponding deterioration in latency.

Simultaneously processing multiple packets somewhat breaks our assumption of a single-packet processing function that takes just one packet as input. Incorporating this into Vigor would involve modeling possible interactions between these packets within the logic, as opposed to simply via the state abstraction. Hence, we leave this to future work.

We conclude that developing and verifying NFs with Vigor does not come at the cost of NF performance. Any performance differences between our NFs and their baselines have nothing to do with them being verified or not.

6.4 Does Verification Come at the Expense of Developer Productivity?

Whenever verification is discussed in a practical context, the other elephant in the room (besides performance) is developer productivity. Although it is generally true that introducing formal verification can increase the burden on developers, we designed Vigor specifically to lighten this burden: Desired properties can be specified in Python, there is no need to write lemmas or understand formal methods, the high upfront cost of verification can be avoided with pay-as-you-go verification, and Vigor automatically provides debug information that helps resolve proof failures. In our subjective assessment, using verification improved our productivity when developing our NFs. In this section, we evaluate more objectively the extent of the burden introduced by Vigor.

The first question is whether developers can indeed use Vigor in a fully push-button mode? The answer is yes, as long as the code they write manages all its persistent state by using libVig primitives. Then, is libVig sufficient to write all (or at least most) NFs? Figure B6.2 summarizes our argument by demonstrating the number of new and reused primitive API calls required to implement each NF. The first NF we wrote using libVig was VigNAT, and we added to libVig all the VigNAT data structures and primitives we thought would be useful. The second one was VigLB, and this required adding a consistent hashtable to libVig. For the remaining three NFs, we did not need to add anything new to libVig.

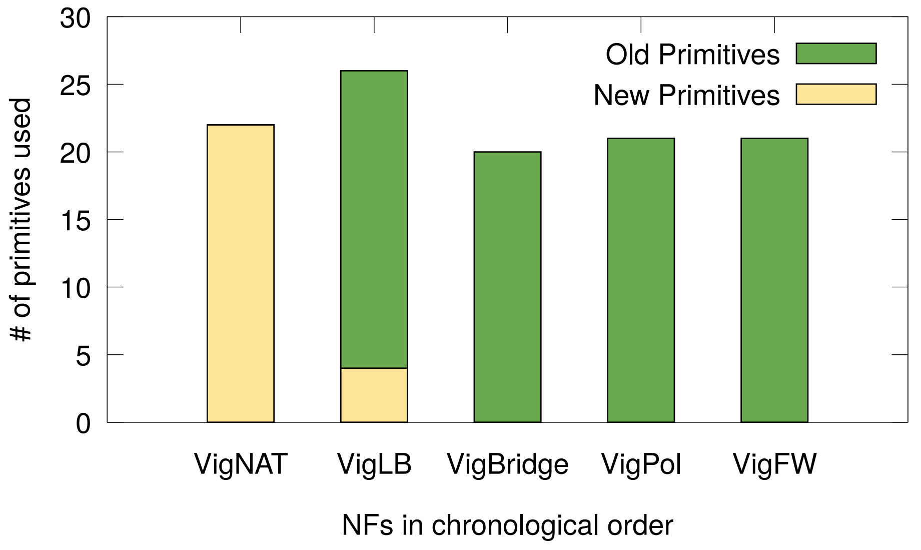


Figure B6.2: The number of new primitives (for NF state-management) added to libVig reduces over time.

As to whether the libVig API is sufficient, we also asked ourselves whether the choice of abstraction level influences the rate at which libVig's API converges. In other words, had we chosen a different level of abstraction, would libVig have to be larger or smaller to be sufficient, and how rapidly would the API stabilize? We performed the following thought experiment: We designed "on paper" a number of other NFs (18 in total) using libVig's API, and compared it to building them in Click. Then we looked at how many new data structures we needed to add to libVig with each new NF vs. how many new elements had to be added to Click according to [144]. Figure B6.3 shows the result. The NFs are listed along the x-axis, sorted in decreasing order of the number of Click elements they use. The continuous lines show how the number of data structures and elements evolve in libVig and Click, respectively, serving as a metric for convergence of the API. We assume we start with an empty libVig and Click library.

We conclude (with no claim of statistical significance) that, had we chosen a higher level of abstraction for libVig such as the Click API, we would have had to add many more primitives to libVig. This is because the less "primitive" a primitive is, the more likely it is to require changes to be useful to a new NF. With our current choice, once we add verified crypto primitives from HACL* [236] and regexp primitives from LMS [4], libVig will have all the data structures needed to write all the 18 NFs. At this point, NF developers should be able to truly write any NF they want in Vigor, push a button, and verify it all the way down to the hardware. Note that this comparison has no bearing on Click itself because the purpose of the Click framework is quite different from that of libVig's.

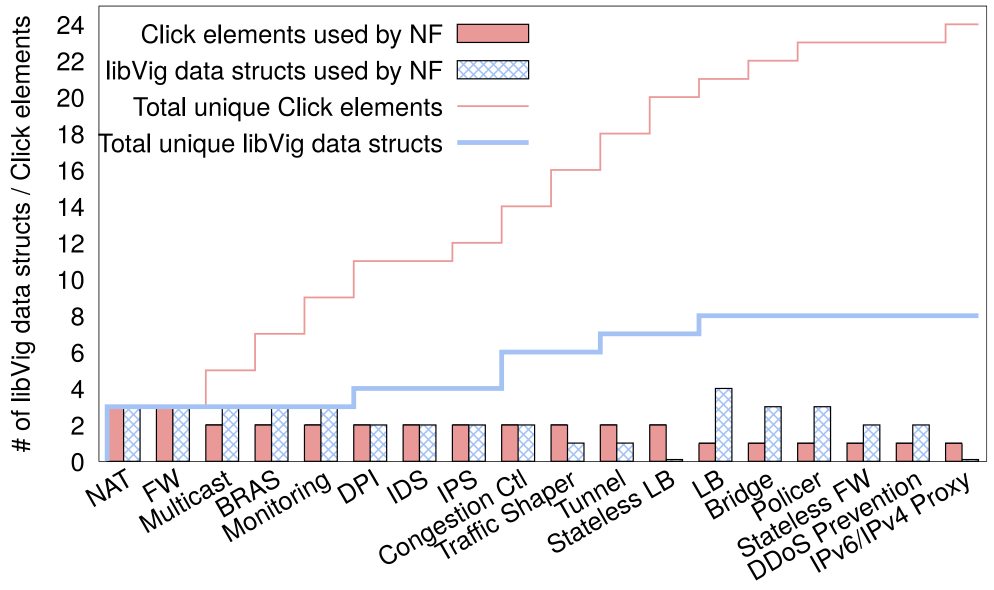


Figure B6.3: Hypothetical evolution of libVig and Click, as developers use them to write new NFs. Click data based on [144].

The second question is, How hard is it for developers and standardization bodies to specify the properties they're interested in? For each NF, Table B6.6 shows the lines of code in the corresponding specification, the amount of time it took to generate the specification from the corresponding RFC, and the number of bounds supplied by the developer to instantiate the necessary libVig types (as described in Chapter 3). The full RFC-derived specifications are relatively short because Python is expressive. The number of user-supplied bounds is quite low. The average full specification size across our NFs is 40 LOC, which is $10 - 20$ times fewer LOC than the implementation. Generating a specification from an RFC takes a non-trivial amount of time, but this is done once per RFC. Plus, we envision that generating a specification will become part of the process of writing an RFC; this process is already a scientifically and socially complex process that can take from weeks to years. We hope that generating a specification will not be the bottleneck but help the process converge faster.

Table B6.6: Statistics on writing NF specifications in Vigor.

NF	LOC in spec	Time to write spec	# of bounds
VigNAT	47	3 days	2
VigBridge	29	2 days	2
VigLB	56	3 days	4
VigPol	41	3 hours	2
VigFW	32	1 hour	1

The third question is whether pay-as-you-go verification really helps. Table B6.7 shows the number of modular properties in each NF specification, and the fraction of the specification they account for. Each NF possesses a handful of meaningful, well-modularized properties. To verify the one with the shortest specification in isolation requires writing from 1 to 13 LOC of specification; this represents $3 - 40\%$ of the full specification. To verify subsequent properties takes, on average across each NF, 4.1 – 9.1 LOC of specification. This suggests that a developer can express meaningful properties in a few minutes, as opposed to taking multiple days to formalize an RFC with all of its (possibly not of interest) corner cases.

Table B6.7: Cost of writing one-off semantic properties.

NF	# modular properties	Cost of 1st prop. (LOC, % of spec)	Avg. cost of each further prop (LOC,% of spec)
VigNAT	9	13 (27%)	4.1 (9%)
VigBridge	7	1 (3%)	4.5 (15%)
VigLB	7	3 (5%)	8.9 (15%)
VigPol	4	13 (31%)	9.1 (22%)
VigFW	5	13 (40%)	5.3 (16%)

Combining this evidence with that on the verification time (Section 6.1), we conclude that the effect on developer productivity would at worst be negligible. We believe that, based on our experience described in Section 6.2, productivity would be enhanced. This is especially true when factoring in the use of function-call sequences for diagnosing the root cause of a proof failure.

Chapter 7

Summary

We have presented Vigor, a software stack and toolchain for building and running formally verified software network functions (NFs). Our goal was to build a verification toolchain that can be used without any verification expertise. Vigor employs symbolic execution—a good candidate because it requires little human effort. We designed Vigor knowing that there will always exist NF code that cannot be symbolically executed without running into path explosion,

We based our work on two conjectures about the NF domain. First, the NF code that cannot be, in practice, symbolically executed tends to be common across NFs, hence it can be stored in a specialized library and verified by experts using theorem proving, without any input from NF developers. Second, the same primitives provided by this specialized library to NF developers can be productively used to write useful functional specifications. Our two conjectures make it possible to outsource the complex verification tasks to experts and to integrate the outcome of their work automatically into proofs of the entire NF. We tested Vigor on five representative NFs, and it was able to verify them within minutes or hours, depending on whether we verified only the NF or the entire software stack.

Our experimental evaluation shows that the confidence that formal verification offers need not come at the cost of performance. When performing non-batched processing, our formally verified NFs performed at least as well as their non-verified counterparts. Once we extend Vigor to support batching, we hope to show that formal verification does not come at any cost in performance.

Discussion and Future Work

Chapter 1

Limitations

In this chapter, we discuss the limitations of the Vigor approach and the Vigor prototype. First, we enumerate the fundamental limitations, such as restrictions on the NFs and the properties that Vigor can verify. Second, we enumerate the incidental limitations caused by the limited scope of our prototype; then, we suggest possible solutions that could lift these limitations.

1.1 Limitations of the Approach

In this section, we discuss the fundamental limitations of the Vigor approach. We consider practical issues that would appear in any verification framework following the Vigor approach. Such issues include the limitations on the development practices, properties that can be verified, and the software-NF stack and types of NFs that can be implemented.

To facilitate verification, Vigor assumes the separation for SE-friendly/non-SE-friendly code, which limits its ability to verify arbitrary systems. Vigor expects the system to be partitioned, with SE-friendly and non-SE-friendly logic being compartmentalized. Though libVig was designed to minimize the impact and the difficulty of developing NFs within this framework, there is a more fundamental argument to be had about the inherent ability to frame all NFs in such a way. From recent work that extracts the semantics of state from NF implementations [120], to new languages for programmable data-planes [25], a large body of ongoing work either assumes or reinforces a certain degree of stateful/stateless separation, which is close to SE-friendly/non-SE-friendly separation. By making this separation explicit, and by formalizing the APIs at the boundary, Vigor makes NFs more amenable to analysis, thus enabling push-button verification.

In particular, SE-friendly code must satisfy the following constraints:

- Event loop: Vigor assumes that the SE-friendly NF code has one top-level infinite loop that receives/sends packets, and that all the other loops in the code are finite. In particular, SE can handle any loop that can be completely unrolled. Loops inside NFs are almost always bounded by maximum packet size, thus this imposes negligible restrictions in practice. The event loop must handle at most one event (packet or a timer tick) per iteration. Vigor assumes that the specification describes the result of exactly one iteration. It might be nontrivial to deconstruct some of the more complicated algorithms into such event handling logic. However, most protocols are defined in such terms, hence we expect this not to be an issue for NFs.
- Handling persistent state: Any state that persists across packet receives must reside in libVig data structures. This eliminates the need in the NF code for complex pointer structures that could render exhaustive symbolic execution impractical. However, this is a catch-all rule that is broader than necessary: not all persistent state must go into libVig, rather only the state that cannot be effectively havoc-ed. If a developer accidentally keeps the persistent state in the NF code itself, then verification time is likely to be long, due to path explosion. But it could also happen that Vigor succeeds in havoc-ing that state and the developer never notices the “mistake.” In fact, there does exist a persistent state outside libVig, namely in DPDK, the NIC driver, and NFOS, which is successfully havoc-ed. To ensure that NF developers are presented with a simple and verification-agnostic model of state handling, we decided to make this rule broad, even if it is over-constraining.

Apart from defining the heuristics above, we define no clear rule that discerns SE-friendly from non-SE-friendly code, hence the developers must rely on their judgment and trial and error. We give a few guidelines for the SE-friendly code, such as prohibiting unbounded loops and input-dependent pointer dereferencing, but some bounded loops or pointers that happen to be havoc-ed might still hinder SE. Moreover, some input-dependent pointers might not pose a challenge to the SEE. Furthermore, even sufficiently complex arithmetic, such as hash computation, might render SE untractable. In some cases, to produce code amenable to SE, the developer has to try multiple solutions and adapt them according to the SEE output.

The range of NFs that any framework implementing the Vigor approach supports depends on the finite set of data structures offered by libVig. Vigor enables NF developers to verify their particular NF only if libVig provides suitable data structures. As shown in Section 6.4 of Part B, a small set of data structures covers a broad range of NFs. Even though, as libVig grows, it enables more and more NFs yet it will unlikely ever become complete for all NFs. Some NFs, such as an LPM-based IP router, use unique data structures that are rarely used elsewhere. For such NFs, Vigor offers no push-button verification because, in addition to their NF, the developers must also verify the data structure they use manually.

A Vigor framework might fail to support NFs with a specific performance profile, even if libVig contains the data structures with APIs required by these NFs. There exist multiple implementations of a data structure with an identical semantic specification but with different performance characteristics. If libVig provides a data structure required by an NF, it might fail to provide an implementation that meets the required performance characteristics, thus altering the performance profile of the NF.

Vigor NFs are limited in the choice of the lower levels of the stack, such as the OS, NIC driver, and the packet processing framework, because of the SE-friendly-code constraints. Lower levels of the stack must satisfy the SE-friendly-code constraints, as Vigor runs exhaustive SE to explore them. Although DPDK and its ixgbe driver happen to fit these constraints, with minimal tweaking, this might not hold for alternative frameworks and drivers. As a result, for the full-stack verification, the Vigor approach limits the implementation freedom of the OS, of the packet processing framework, and of the NIC driver.

The Vigor approach does not support multithreading in the NFs, such as embarrassing parallelism, batch packet processing, or concurrent packet processing. It might be possible (see Section 2.2) to extend the approach for some form of parallelism in the future. One could imagine an automatic generation of a refinement proof for multithreaded NF implementations with respect to the same sequential specification Vigor NFs have today. Alternatively, the Vigor specification language could be extended with multithreaded clauses that relax semantics of the NF if it processes multiple packets at a time.

Vigor supports only reasoning about a reaction to a single event and not sequences of events. Vigor specifications describe the NF behavior in response to a single event, such as a reception of a packet or a timer tick. Although a specification can introduce an abstract state that enables properties that span multiple packets (in the extreme the abstract state can keep all the previous packets), our toolchain cannot automatically prove them. This limitation includes existential qualifiers, e.g., Vigor fails to directly prove the fact that, “For each flow, there is an outbound packet initiating it,” even though it proves the one-event-at-a-time specification that the NF either reuses an existing flow from the flow table or allocates a new one in reaction to an outbound packet. A developer has to resort to other reasoning tools to prove these higher-level properties based on the one-event-at-a-time specifications proven by Vigor.

Finally, the Vigor specification language, in its simplicity, limits the expressiveness of the NF specifications.

The Vigor approach is designed for precise and deterministic reasoning and does not support probabilistic statements. For example, the essential property of a consistent hashing used in our load balancer is the flow affinity that is expressed in terms of probability of events. Although the Vigor load-balancer specification is precise, all it can say about the backend for a particular packet is that it is chosen by using the CHT data structure. The specification makes no assertions about the probability of routing a packet to the backend that processes the corresponding flow. The user has to trust or inspect manually that the CHT respects the flow affinity.

The Vigor specification might grow complex because Vigor fixes the abstraction of the specification. Vigor specification has to be expressed in terms of the libVig data structures. For NFs that use multiple data structures and involve complex manipulations of these data structures, such specification might become too detailed and conceal bugs in itself. The developer has to use higher-level verification tools to formulate a higher-level specification and to prove that the Vigor-verified specification refines it.

Beyond the fundamental limitations, the Vigor prototype framework carries incidental ones, reflecting our implementation choices and available development resources.

1.2 Limitations of the Prototype

We now discuss the main limitations of our current prototype: lack of support for certain NF types, parallel NFs, and batch packet processing. We also discuss if and how we think we can address them.

1.2.1 Restrictions on NF Code

Writing NFs with Vigor does not require verification expertise, but it does impose constraints on the NF code:

- SE-friendly-code requirements, such as having a single top-level infinite event-processing loop and keeping all persistent state in libVig data structures, mentioned in Section 1.1.
- Memory ownership: NF developers must respect the memory ownership model of libVig data structures, as documented in the libVig interface. For instance, after inserting a value into a map, the caller is not permitted to change that value by reference; instead, the caller must remove the entry from the map, modify it, then re-insert it. Vigor does not currently enforce the memory ownership model at compile time rather only at verification time, which could make debugging harder. To make this requirement more transparent for the developer, it is possible to perform a custom compile-time analysis or to piggyback on a language, such as Rust [146], which offers compiler support for this ownership model by default.
- Inherited constraints: Vigor also inherits restrictions imposed by the underlying tools. First, KLEE does not handle well memory access with symbolic pointers, even with tight constraints. Fortunately, pointer arithmetic and input-dependent pointers are infrequent in the NF code. Second, VeriFast does not support the whole of the C standard library. We have not found this to be a problem as most of the unsupported functions would anyway impose undue performance overheads on the NF, hence a developer is unlikely to use them.

The set of data structures in libVig essentially defines the applicability of Vigor to NFs. Vigor supports the NFs that use only data structures represented in libVig.

1.2.2 Completeness of Data-Structure Library

The current version of libVig does not provide primitives for regular expression matching, cryptography, or variable-length headers. As a result, we cannot verify NFs that need such primitives, e.g., deep packet inspection and intrusion-detection systems or a VPN endpoint. We can fix this by incorporating into libVig verified open-source libraries, such as LMS [4] for regular expression matching and HACL* [236] for cryptographic primitives.

Packet parsing might prove to be an issue for protocols with complex grammars. Currently, Vigor supports fixed-sized fields with a fixed layout except for arbitrary gaps. That is just enough to unmarshal TCP/IP stack headers. However, even DNS protocol specifies an arbitrary number of arbitrary length entries that require complex parsing logic to handle. Future work can address this issue by implementing packet parsing as the non-SE-friendly code, verifying it, and shipping it with libVig.

1.2.3 Network Function Parallelism

Other limitations of Vigor relate to the assumptions about the underlying hardware model. Vigor currently assumes the NF is implemented by using a single execution context, thus reasoning about concurrency, memory consistency, and other forms of cross-core interference is avoided. Other advanced features, such as the use of SIMD instructions and inline assembly optimizations are also currently not supported by our prototype. All of these features play an important role in achieving high-performance in NFs, especially at higher line rates, such as 40Gbps, 100Gbps, or more. Handling these constructs poses a mix of fundamental and engineering challenges. The ability to handle special instructions could easily be added to our prototype, but SIMD instructions, in particular, are often used to simultaneously handle batches of packets, which somewhat breaks the assumption of a single packet-processing function that takes just one packet as input. Modeling how each of these packets could interact with each other within the logic poses a deeper challenge that we leave for future work.

Batch Processing

Our current Vigor prototype assumes no batching, i.e., that the CPU processes one packet at a time. As discussed in Section 6.3 of Part B, batching is an important element of modern I/O frameworks, which enables trading off latency for throughput. To support it, we have to remove from our prototype the assumption of a single packet-processing function that takes a single packet as input. One challenge this poses is the modeling of potential interactions between packets that are processed within the same batch.

One could imagine supporting batching transparently, i.e., an NF with an identical one-packet-at-a-time specification might batch packets under the covers. In the case of transparent batching, the specification language remains as is and the specification hides the fact that the NF might implement batching. The framework in such a case has to ensure proper synchronization and to analyze the possible reorderings within a batch in order to make sure the batched implementation still conforms to the same sequential specification.

Chapter 2

Future Work

In this chapter, we explore the possible future extensions of the Vigor approach and the prototype framework. We demonstrate how verification expertise (which can be incorporated into the framework) still makes a difference in the verification efficiency. We speculate on the possible extension of the Vigor verification approach to the parallel NFs. We propose a possible fusion of our NF verification approach and the network verification to achieve end-to-end correctness guarantees. We explore the possibility to generate the C implementation from the specification rather than proving the refinement between the two. We then consider possible alternatives to the Vigor specification language. Finally, we propose to use modularity to improve the scalability of the approach and conjecture its applicability to domains other than NFs.

2.1 Impact of Verification Expertise

We ran an experiment to check if a basic level of verification expertise on the part of NF developers could significantly impact verification time. We made two minor modifications to the NFs, in order to reduce the number of code paths while preserving the exact same semantics. First, we replaced short-circuiting Boolean operators `&&` and `||` (which produce one code path per term) with their non-short-circuiting counterparts `&` and `|` whenever there were no side effects. Second, we replaced `if` blocks that only assign Boolean values with equivalent Boolean expressions (e.g., `if(c) {b = true;}` turned into `b = b|c;`). As can be computed based on Table B2.1, these minor changes netted an average reduction of 51% in overall verification time on our 56-contexts machine: The number of paths, thus the symbolic execution time, is roughly halved; but per-sequence validation time increases due to the higher complexity of path constraints. This effect is well understood [133] by verification experts, but verification-agnostic developers can also try these heuristics out with potentially big rewards.

Table B2.1: Verification statistics for NFs written with some verification expertise. We show decreases (−) and increases (+) as a percentage of the numbers in Table B6.3.

NF	**Symbolic execution time**		**# of seqs**			**Per-seq validation time (avg)**
	NF only	*with rest of stack*	*NF only*	*with rest of stack*		
VigNAT	−29%	−38%	−52%	−58%	×	−16%
VigBridge	−14%	−40%	−44%	−50%	×	0%
VigPol	−52%	−58%	−60%	−67%	×	+66%
VigLB	−43%	−17%	−32%	−40%	×	−4%
VigFW	−17%	−43%	−42%	−50%	×	−15%

2.2 Parallel NF Implementations

Our current Vigor prototype can verify only single-threaded NFs. This does not mean that Vigor is not useful in parallel environments. A typical approach to scaling up NF performance is to run multiple independent instances of the same NF on different cores of the same server and to use shared-nothing receiver-side scaling (RSS) [251] in order to ensure that packets from the same flow always go to the same core. In this case, the NIC itself uses a hash of packet headers to select which core to dispatch the packet to, and each core operates exclusively on the local state. In essence, this execution model runs multiple copies of the NF across the cores and each operates independently. Verifying NFs under these circumstances implies proving the correctness of a single core and proving that neither core can access the state of another. As long as different NF instances do not share state, we can use Vigor to verify that each instance conforms to an NF specification.

Currently, Vigor does not support this mode. It would require two steps to enable RSS support:

1. Augment NFOS to run multiple instances of the NF code with different configurations and configure RSS on the interfaces.
2. Add non-interference static analysis, that validates that given header fields can be used for sharding (e.g., VLAN tags for Ethernet) and does not dete-

riorate the observed semantics.

One can imagine extending Vigor so that it can verify that a set of NF instances running on different cores conform to an NF specification, i.e., behave as a single NF. For example, consider a standard NAT that maintains two pieces of state: a flow table and a port allocator (the latter keeps track of free/allocated port numbers). And consider a server running multiple NAT instances on different cores. Concerning the RSS, all packets from the same flow go to the same instance, hence each instance can have its own flow table. If each instance also has its own port allocator, then each instance is an independent NAT, and we can use Vigor to prove that it conforms to a NAT specification. However, statically partitioning the port-number space across the NAT instances can lead to inefficiency—new flows being dropped despite the availability of port numbers. Ideally, we would want some form of lightweight coordination that prevents this situation; and we would want to prove that all the NAT instances together behave as one, i.e., they drop no new flow as long as a port number is available at any instance. We think we could extend Vigor to automatically compute such proofs.

A key element in this next step is the abstraction of a *logically single NF* exposed by our NF-specific operating system. Our vision is that an operator will use this abstraction to deploy a single NF; under the covers, NFOS will run parallel instances of this NF, while providing a formal guarantee that all the instances together behave as a single NF.

2.3 Fusion With Network Verification

As mentioned in Section 2.5 of the introduction, a large body of work on the verification of the entire data plane of a network (network verification) relies on trusted models for the NFs. Vigor can fill in this gap, and extend the network verification results down to the source code of the NFs actually running in the network. In particular, we consider using the SEFL models used in [211] as a specification. Vigor could also handle the "controller" part of the P4 program that is currently modeled in [140, 210].

Another interesting collaboration might be in skipping the NF specification altogether, and integrating Vigor with the network verification framework to check the NF properties necessary for a particular all-network property. This approach can save the unnecessary formulation of the full functional specification and proof effort for irrelevant semantic cases.

2.4 Translation Between NF Specifications and C Code

In Vigor, a developer writes the NF implementation, which is then verified against the NF specification written in Python. An alternative approach is to automatically translate the specification into an implementation. This is non-trivial, and we question whether, given a specification, there exists a single efficient C implementation to translate to. Or is the decoupling between specification and implementation indeed essential to practicality? Future work on this would need to find ways to translate one-off properties ("partial specifications") to code, to translate negative properties (such as "packet is never sent back on its source port") to code and, finally, to check that libVig contracts are obeyed by the specification.

The opposite direction, automatic abstraction of the given NF implementation into a short specification, might prove to be more feasible. Trusting the translation tool, the developer has to check only the resulting specification for the protocol conformance, which would generally be much easier. The challenge lies in deciding what details can be abstracted away without compromising the important semantic properties. One could imagine a feedback-guided abstraction, with a user listing the details pertinent only to the implementation, and refining this list as the generated specification gets leaner.

2.5 Specification Language Alternatives

Another question concerns the expressivity of our specification language. To increase familiarity and expressivity, we designed our specification language around a subset of the Python programming language. Although the Python language, in general, is not fully specified with formal semantics, we selected a narrow subset of it with clear semantics that we define more precisely (see Appendix B). The resulting language, as can be seen in our examples (Listing 1.1 in Part B), retains a practical resemblance to a typical Python program but does not fully support the baggage that comes with supporting the entirety of the Python language [202].

Yet another line of work is automatic formal specification extraction out of natural language standards. Our case-study NFs are relatively easy to specify (see Section 6.4 of Part B). However, this might not always be the case and, for some standards or for an array of similar protocols, an automatic specification distiller using natural language processing might prove effective.

2.6 Modular Verification

In some cases, opportunities for more efficient verification of the stack do exist. For instance, the result is a symbolic execution of the `send` function over and over again. We could imagine therefore verifying in isolation perhaps parts of the stack by using partly constrained symbolic inputs—preconditions—and then reusing the SE results when NF code acts within those constraints. We have not explored these opportunities yet.

2.7 Beyond NFs

We conjecture (i.e., we believe this to be true, but have no proof) that the Vigor approach applies to other systems code. Our framework requires three essential features of the target code:

(1) The system is event-handling: every IO or state change necessarily follows one of a few clearly defined events;

(2) The system is built out of simple (amenable to symbolic execution) SE-friendly code, and commonly used (for verification cost amortization) data structures, which are cleanly compartmentalized.

(3) The system's specification can be written naturally in terms of these commonly used data structures.

Such a system can be essentially represented as F in $(Output, NewState) = F(Input, OldState)$. The software data plane is not the only domain that satisfies these requirements. Embedded systems, such as alarm systems, or industrial controllers are also good application domains for Vigor. Moreover, for low complexity systems, there is no need to introduce higher abstraction, as long as the number of abstractions (data structures) fits into human working memory.

Chapter 3

Conclusion

With the advent of software NFs, comes a software verification challenge. Accelerated innovation has brought more uncertainty and failures into the critical domain of computer networks. The traditional method of quality assurance, testing, is either too superfluous or too expensive. The alternative, verification, is difficult for a generic domain.

However, practical software verification is possible in the domain of software NFs. The common structure of software NFs enables Vigor to fuse two separate verification approaches: exhaustive symbolic execution and theorem proving, and combines their advantages: automation, scalability, and expressiveness.

Vigor exploits the structure of software NFs to enable push-button, full-stack, pay-as-you-go software verification.

The verified NFs do not suffer from a performance overhead. Our evaluation has demonstrated that all five of our NFs perform on par with the unverified alternatives.

The Vigor approach is extensible and generalizable. We believe that the Vigor approach can be extended to parallel NFs, at least by adding concurrent data structures into libVig or by replicating single-threaded NF images and offloading flow steering to hardware. Although it performs well in the domain of software NFs, future work can apply the same approach to other domains with a similar structure, such as embedded software.

Appendix A : Person-Hours to Implement and Verify libVig

During the initial development of libVig, we kept track of the interactive verification effort went into the libVig version as it was initially designed for VigNAT (before generalization to other NFs). We report it here, as a few data points witnessing formal verification experience for medium-sized targets without any pretension for objectivity.

Table B3.1 breaks down the development and verification effort. The * marks an extrapolated value. Unsurprisingly, the verification effort dominates other activities (the Validator plug-ing effort comprises the transition lemmas and the Validator heuristics for inserting them). This disparity (17:1) in proof:development effort can be explained partly by the lack of practical experience with the formal verification tool, and partly by the differences in proof:code size in lines (20:1).

Table B3.1: Developer person-hours spent on each component of libVig.

Component	Interface design	Implementation	Formal specification	Symbolic model	Proof	Validator plug-in
Hash Map	2.5*	2*	4.5*	5.5*	73.5	11.5
Port Allocator	1*	1.5*	3.5*	1.5*	70	7.5
Static Array	1.5	0.5	2.5	2	6	2
Batcher	1	0.2	0.5	1	0.5	0.2

Appendix B : Vigor Specification Language

The following BNF summarizes the syntax of the Vigor specification language:

```
<spec>                    ::= <imports> <constant-decls> <block>
<imports>                 ::= <import> <imports> | ""
<import>                  ::= "import" <python-symbol> "\n"
                            | "from" <python-symbol> "import" <symbols>"\n"
<symbols>                 ::= <python-symbol>
                            | <python-symbol> "," <symbols>
<constant-decls>          ::= <constant-decl>
                            | <constant-decl> <constant-decls>
<constant-decl>           ::= <python-symbol> "=" <expression> "\n"
<block>                   ::= <statements>
<statements>              ::= <statement> | <statement> <statements>
<statement>               ::= <if-statement>
                            | <if-else-statement>
                            | <assignment-statement>
                            | <pop-header-statement>
                            | <assert-statement>
                            | <return-statement>
                            | <pass-statement>
<if-statement>            ::= "if" <expression> ":" "\n" "   " <block>
<if-else-statement>       ::= "if" <expression> ":" "\n" "   " <block> "\n"
                              "else: \n" "   " <block>
<assignment-statement>    ::= <python-symbol> "=" <expression> "\n"
<pop-header-statement>    ::= <python-symbol>"= pop_header("<python-symbol>
                                  "," "on_mismatch=" <expression> ")" "\n"
<assert-statement>        ::= "assert" <expression> "\n"
<return-statement>        ::= "return (" <list-expression> ","
                                         <list-expression> ")" "\n"
<pass-statement>          ::= "pass" "\n"
<expressions>             ::= <expression> | <expression> <expressions>
<expression>              ::= "(" <expression> ")"
                            | <aexpression> | <bexpression> | <term>
                            | "..." | <list-expression>
<list-expression>         ::= "[" <expressions> "]"
<aexpression>             ::= <aexpression> "/" <aexpression>
                            | <aexpression> "*" <aexpression>
                            | <aexpression> "-" <aexpression>
                            | <aexpression> "+" <aexpression>
<bexpression>             ::= <aexpression> "<" <aexpression>
```

```
                         | <aexpression> "<=" <aexpression>
                         | <aexpression> ">" <aexpression>
                         | <aexpression> ">=" <aexpression>
                         | <aexpression> "==" <aexpression>
                         | <aexpression> "!=" <aexpression>
                         | <aexpression> "<>" <aexpression>
                         | <bexpression> "and" <bexpression>
                         | <bexpression> "or" <bexpression>
                         | <bexpression> "not" <bexpression>
<term>                 ::= <python-symbol> | <literal>
                         | <call> | <attribute> | <constructor>
<literal>              ::= NUMBER | "True" | "False"
<call>                 ::= <python-symbol> "(" <expressions> ")"
                         | <attribute> "(" <expressions> ")"
<attribute>            ::= <python-symbol> "." <python-symbol>
<constructor>          ::= <python-symbol> "(" <initializers> ")"
                         | <python-symbol> "(" <python-symbol> ","
                                               <initializers> ")"
<initializers>         ::= <initializer> <initializers> | ""
<initializer>          ::= <python-symbol> "=" <expression>
```

- Import subspecification (the first form of `<import>`): Include the file (add the `.py` extension to the module name) as if it were a part of the current specification (like C preprocessor `#include`)
- Import state variable (the second form of `<import>`): Declare that the present specification uses a state variable, that is declared in the NF state declaration. The variable that enters the context has the name and type specified in the NF declaration. The variable is initialized at the beginning of the packet processing iteration. The variable is mutable, and the final mutated value is compared against the computed value at the end of the iteration.
- Constant declaration (`<constant-decl>`): Declare the constant with the given value (like C preprocessor `#define`).
- If statement (`<if-statement`, `<if-else-statement>`): A regular Python `if` statement with optional `else` branch. If the condition holds, the block that immediately follows is executed, otherwise the block that follows the `else` keyword, if present, is executed. The block is identified by its indentation.
- Assignment (`<assignment-statement>`): Creates or updates a variable on the left-hand side and assigns it the value from the right-hand side of `=`.
- Pop header (`<pop-header-statement>`): Parses a header of the protocol that is specified as the first element, and stores it in the variable on the left-hand side of `=`.
- Assert (`<assert-statement>`): Evaluates an expression and checks if it is equivalent to `True`
- Return (`<return-statement>`): Accepts a tuple of two lists—`interfaces`,

`headers`. The first list `interfaces` enumerates all the interfaces where the NF has to send a packet. The second list `headers` is a stack of headers (from the outermost to the innermost protocol) in the outcast packet. A header is set using the constructor syntax as described below.

- Pass (`<pass-statement>`): Immediately succeeds the verification. Used to discard the behaviors and circumstances that are not interesting for the property. Forces the verification engine to finish verification and report success.
- Expressions (`<expression>`): The standard set of arithmetic and Boolean expressions, comparisons, lists, and a special ellipsis symbol ... used to designate ignored parts in assertions. For example, when used as a value of a field in a constructor, it means that the outcast packet may have any value at that place.
- Call (`<call>`): A function or a method call. Only the predefined functions can be called. The function name can not be indirect.
- Attribute (`<attribute>`): Some objects, such as packet headers and state have a predefined set of attributes, accessible through the `.` syntax.
- Constructor (`<constructor>`): Currently defined only for packet headers, a constructor of an object, initializing all its fields. Constructors assign fields as listed in the parentheses using the `=` signs (`<initializers>`). The constructor accepts as an optional argument an existing object of the same type (the second form of `<constructor>`). It copies the values of all the fields from the given object and overrides them with the values provided by explicit assignment arguments.

References

[1] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H. and Ulbrich, M. 2016. *Deductive software verification—the KeY book—from theory to practice.* Springer.

[2] Al-Shaer, E. and Al-Haj, S. 2010. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. *Proceedings of the 3rd acm workshop on assurable and usable security configuration* (2010), 37–44.

[3] Amani, S. et al. 2016. Cogent: Verifying high-assurance file system implementations. *International conference on architectural support for programming languages and operating systems* (Apr. 2016).

[4] Amin, N. and Rompf, T. 2017. LMS-Verify: abstraction without regret for verified systems programming. *Symposium on principles of programming languages* (2017).

[5] Anand, S., Păsăreanu, C.S. and Visser, W. 2007. JPF–se: A symbolic execution extension to java pathfinder. *International conference on tools and algorithms for the construction and analysis of systems* (2007), 134–138.

[6] Anderson, C.J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C. and Walker, D. 2014. NetKAT: Semantic foundations for networks. *Symposium on principles of programming languages* (San Diego, California, USA, 2014).

[7] Anwer, M.B., Motiwala, M., Tariq, M. bin and Feamster, N. 2010. SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM Conference* (2010), 183–194.

[8] Appel, A.W. 2015. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems.* 37, 2 (2015).

[9] Arisholm, E. and Briand, L.C. 2006. Predicting fault-prone components in a java legacy system. *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering* (2006), 8–17.

[10] Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C. and Finocchi, I. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2018).

[11] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K. and Ustuner, A. 2006. Thorough static analysis of device drivers. *ACM*

SIGOPS Operating Systems Review. 40, 4 (Apr. 2006), 73–85. DOI:https://doi.org/10.1145/1218063.1217943.

[12] Ball, T., Bounimova, E., Kumar, R. and Levin, V. 2010. SLAM2: Static driver verification with under 4. *International conference on formal methods in computer-aided design* (2010).

[13] Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pak, K. and Urban, J. 2015. Mizar: State-of-the-art and beyond. *Conferences on intelligent computer mathematics* (2015), 261–279.

[14] Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B. and Leino, K.R.M. 2005. Boogie: A modular reusable verifier for object-oriented programs. *Formal methods for components and objects* (2005).

[15] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C. and others 1997. The coq proof assistant reference manual: Version 6.1. (1997).

[16] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A. and Tinelli, C. 2011. CVC4. *International conference on computer aided verification* (Berlin, Heidelberg, 2011), 171–177.

[17] Beckett, R., Gupta, A., Mahajan, R. and Walker, D. 2017. A general approach to network configuration verification. *ACM SIGCOMM Conference* (2017).

[18] Beckett, R., Gupta, A., Mahajan, R. and Walker, D. 2018. Control plane compression. *ACM SIGCOMM Conference* (2018).

[19] Beizer, B. 2003. *Software testing techniques.* Dreamtech Press.

[20] Beringer, L., Petcher, A., Katherine, Q.Y. and Appel, A.W. 2015. Verified correctness and security of OpenSSL HMAC. *USENIX security symposium* (2015).

[21] Beyer, D., Henzinger, T.A., Jhala, R. and Majumdar, R. 2007. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer.* 9, 5-6 (2007), 505–525.

[22] Bird, D.L. and Munoz, C.U. 1983. Automatic generation of random self-checking test cases. *IBM systems journal.* 22, 3 (1983), 229–245.

[23] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M. and Wansbrough, K. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. *SIGCOMM Computer Communication Review.* 35, 4 (2005).

[24] Boonstoppel, P., Cadar, C. and Engler, D. 2008. RWset: Attacking path explosion in constraint-based test generation. *International conference on tools and algorithms for the construction and analysis of systems* (2008).

[25] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. and Walker, D. 2014. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review.* 44, 3 (2014).

[26] Boye, M. 2013. Netfilter connection tracking and NAT implementation. *Seminar on network*

protocols in operating systems (2013).

[27] Boyer, R.S., Elspas, B. and Levitt, K.N. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*. 10, 6 (1975).

[28] Bradner, S. and McQuaid, J. 1999. *Benchmarking methodology for network interconnect devices*. Technical Report #2544. Internet Engineering Task Force; Internet Requests for Comments.

[29] Brand, D. and Joyner Jr, W.H. 1978. Verification of protocols using symbolic execution. *Computer Networks*. 2, 4-5 (1978), 351–360.

[30] Brouwer, L.E.J. 1925. Intuitionistische zerlegung mathematischer grundbegriffe. *Jahresbericht der Deutschen Mathematiker-Vereinigung*. 33, (1925), 251–256.

[31] Brumley, D., Caballero, J., Liang, Z., Newsome, J. and Song, D. 2007. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. *USENIX security symposium* (2007).

[32] Brumley, D., Jager, I., Avgerinos, T. and Schwartz, E.J. 2011. BAP: A binary analysis platform. *International conference on computer aided verification* (2011), 463–469.

[33] Bucur, S., Ureche, V., Zamfir, C. and Candea, G. 2011. Parallel symbolic execution for automated real-world software testing. *Proceedings of the sixth conference on computer systems* (2011), 183–198.

[34] Buechler, C.M. and Pingle, J. 2009. *PfSense: The definitive guide*. Reed Media Services.

[35] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L. and Hwang, L.-J. 1992. Symbolic model checking: 10^{20} States and beyond. *Information and computation*. 98, 2 (1992), 142–170.

[36] Cadar, C., Dunbar, D. and Engler, D.R. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Symposium on operating systems design and implementation* (2008).

[37] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L. and Engler, D.R. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*. 12, 2 (2008), 10.

[38] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N. and Visser, W. 2011. Symbolic execution for software testing in practice: Preliminary assessment. *International conference on software engineering* (2011), 1066–1071.

[39] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J. and Rodriguez, D. 2015. Moving fast with software verification. *NASA formal methods symposium* (2015), 3–11.

[40] Canini, M., Venzano, D., Perešíni, P., Kostić, D. and Rexford, J. 2012. A NICE way to test OpenFlow applications. *Symposium on networked systems design and implementation* (2012).

[41] Carpenter, B. 2002. *Middleboxes: Taxonomy and issues*. Technical Report #3234. Internet Engineering Task Force; Internet Requests for Comments.

[42] Cérin, C. et al. 2014. Downtime statistics of current cloud solutions. (2014).

[43] Cha, S.K., Avgerinos, T., Rebert, A. and Brumley, D. 2012. Unleashing mayhem on binary code. *IEEE symposium on security and privacy* (2012), 380–394.

[44] Chappel, C. 2013. Unlocking network value: Service innovation in the era of SDN. *HeavyReading, white paper.* (2013).

[45] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F. and Zeldovich, N. 2015. Using crash hoare logic for certifying the FSCQ file system. *Symposium on operating systems principles* (2015).

[46] Chiosi, M. et al. 2012. Network functions virtualisation. *SDN and OpenFlow World Congress.* (2012).

[47] Chiosi, M., Clarke, D., Willis, P., Reid, A., Feger, J., Bugenhagen, M., Khan, W., Fargano, M., Cui, C., Deng, H. and others 2012. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. *SDN and OpenFlow world congress* (2012), 202.

[48] Chipounov, V., Georgescu, V., Zamfir, C. and Candea, G. 2009. Selective symbolic execution. *Workshop on Hot Topics in Dependable Systems* (2009).

[49] Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D. 2001. An empirical study of operating systems errors. *Symposium on operating systems principles* (New York, NY, USA, 2001), 73–88.

[50] Chou, E. 2016. A secure bootloader for demonstrating formal verification of hardware-firmware interactions on SoCs. Princeton University.

[51] Clapeyron, É. 1834. Mémoire sur la puissance motrice de la chaleur. *Journal de l'École polytechnique.* 14, (1834), 153–190.

[52] Clarke, D.G., Potter, J.M. and Noble, J. 1998. Ownership types for flexible alias protection. *Proceedings of the 13th acm sigplan conference on object-oriented programming, systems, languages, and applications* (1998), 48–64.

[53] Clarke, E., Kroening, D. and Lerda, F. 2004. A tool for checking ansi-c programs. *International conference on tools and algorithms for the construction and analysis of systems* (2004), 168–176.

[54] Clarke, E.M. and Emerson, E.A. 1982. Design and synthesis of synchronization skeletons using branching-time temporal logic. *Logic of programs, workshop* (1982).

[55] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W. and Tobies, S. 2009. VCC: A practical system for verifying concurrent C. *Theorem proving in higher order logics* (Berlin, Heidelberg, 2009), 23–42.

[56] Committee, L.S. 2014. *IEEE standard for local and metropolitan area networks—bridges and bridged networks.* IEEE Standards Association.

[57] Cook, S.A. 1971. The complexity of theorem-proving procedures. *Proceedings of the third annual acm symposium on theory of computing* (1971), 151–158.

[58] Cooper, D.C. 1972. Theorem proving in arithmetic without multiplication. *Machine intelligence.* 7, 91-99 (1972), 300.

[59] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. 2009. *Introduction to algorithms.*

[60] Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. *Symposium on principles of programming languages* (1977).

[61] Curry, H.B. 1934. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America.* 20, 11 (1934), 584.

[62] Curry, H.B., Feys, R., Craig, W., Hindley, J.R. and Seldin, J.P. 1958. *Combinatory logic.* North-Holland Amsterdam.

[63] De Moura, L. and Bjørner, N. 2008. Z3: An efficient SMT solver. *International conference on tools and algorithms for the construction and analysis of systems* (2008).

[64] Detlefs, D., Nelson, G. and Saxe, J.B. 2005. Simplify: A theorem prover for program checking. *J. ACM.* 52, 3 (2005), 365–473.

[65] Dijkstra, E.W. 1976. *A discipline of programming.* Prentice-Hall, Englewood Cliffs.

[66] Dijkstra, E.W. 1972. *Chapter I: Notes on structured programming.* Academic Press Ltd.

[67] Dillon, L.K. 1990. Using symbolic execution for verification of ada tasking programs. *ACM Transactions on Programming Languages and Systems.* 12, 4 (Oct. 1990), 643–669. DOI:https://doi.org/10.1145/88616.96551.

[68] Dobrescu, M. and Argyraki, K. 2014. Software dataplane verification. *Symposium on networked systems design and implementation* (2014).

[69] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M. and Ratnasamy, S. 2009. RouteBricks: Exploiting parallelism to scale software routers. *Symposium on operating systems principles* (2009).

[70] Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D. and Tomb, A. 2016. Constructing semantic models of programs with the software analysis workbench. *Verified software: Theories, tools, experiments* (2016).

[71] Duran, J.W. and Ntafos, S.C. 1984. An evaluation of random testing. *IEEE Transactions on Software Engineering.* 4 (1984), 438–444.

[72] Eén, N. and Sörensson, N. 2004. An extensible SAT-solver. *International conference on theory and applications of satisfiability testing* (Berlin, Heidelberg, 2004), 502–518.

[73] Eisenbud, D.E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W. and Hosein, J.D. 2016. Maglev: A fast and reliable software network load balancer. *Symposium on networked systems design and implementation* (2016).

[74] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F. and Carle, G. 2015. MoonGen: A scriptable high-speed packet generator. *Internet measurement conference* (Tokyo, Japan, 2015).

[75] Ernst, M.D., Cockrell, J., Griswold, W.G. and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering.* 27, 2 (2001).

[76] Eyolfson, J., Tan, L. and Lam, P. 2011. Do time of day and developer experience affect commit bugginess? *Proceedings of the 8th working conference on mining software repositories* (2011), 153–162.

[77] Fayaz, S.K., Yu, T., Tobioka, Y., Chaki, S. and Sekar, V. 2016. BUZZ: Testing context-dependent policies in stateful networks. *Symposium on networked systems design and implementation* (2016).

[78] Ferraiuolo, A., Baumann, A., Hawblitzel, C. and Parno, B. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. *Symposium on operating systems principles* (2017).

[79] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R. and Millstein, T. 2015. A general approach to network configuration analysis. *Symposium on networked systems design and implementation* (2015).

[80] Foster, N., Kozen, D., Milano, M., Silva, A. and Thompson, L. 2015. A coalgebraic decision procedure for NetKAT. *SIGPLAN notices* (2015).

[81] Fuchs, C. 2012. Implications of deep packet inspection (DPI) internet surveillance for society. (2012).

[82] Ganesh, V. and Dill, D.L. 2007. A decision procedure for bit-vectors and arrays. *International conference on computer aided verification* (2007).

[83] García Villalba, L., Valdivieso, L., Lopez, D. and Barona, L. 2015. Trends on virtualisation with software defined networking and network function virtualisation. *IET Networks.* 4, (Aug. 2015). DOI:https://doi.org/10.1049/iet-net.2014.0117.

[84] Gember-Jacobson, A., Viswanathan, R., Akella, A. and Mahajan, R. 2016. Fast control plane analysis using an abstract representation. *ACM SIGCOMM Conference* (2016).

[85] Geuvers, H. 2009. Proof assistants: History, ideas and future. *Sadhana.* 34, 1 (2009), 3–25.

[86] Godefroid, P. 2007. Compositional dynamic test generation. *SIGPLAN notices* (2007), 47–54.

[87] Godefroid, P., Klarlund, N. and Sen, K. 2005. DART: Directed automated random testing. *SIGPLAN Notices.* 40, 6 (Jun. 2005), 213–223. DOI:https://doi.org/10.1145/1064978.1065036.

[88] Godefroid, P., Levin, M.Y., Molnar, D.A. and others 2008. Automated whitebox fuzz testing. *Network and distributed system security symposium* (2008).

[89] Godefroid, P., Nori, A.V., Rajamani, S.K. and Tetali, S.D. 2010. Compositional may-must program analysis: Unleashing the power of alternation. *ACM Sigplan Notices.* 45, 1 (2010), 43–56.

[90] Gordon, M. 2000. From LCF to HOL: A short history. *Proof, language, and interaction*

(2000), 169–186.

[91] Gordon, M.J.C. and Melham, T.F. 1993. *Introduction to hol a theorem proving environment for higher order logic.*

[92] Graves, T.L., Karr, A.F., Marron, J.S. and Siy, H. 2000. Predicting fault incidence using software change history. *IEEE Transactions on software engineering.* 26, 7 (2000), 653–661.

[93] Greenhalgh, A., Huici, F., Hoerdt, M., Papadimitriou, P., Handley, M. and Mathy, L. 2009. Flow processing and the rise of commodity network hardware. *SIGCOMM Computer Communication Review.* 39, 2 (2009), 20–26.

[94] Gunawi, H.S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J.M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Sen, K. and Borthakur, D. 2011. FATE and DESTINI: A framework for cloud recovery testing. *Symposium on networked systems design and implementation* (2011), 239.

[95] Han, B., Gopalakrishnan, V., Ji, L. and Lee, S. 2015. Network function virtualization: Challenges and opportunities for innovations. (2015).

[96] Harrison, L.J. and Kemmerer, R.A. 1988. An interleaving symbolic execution approach for the formal verification of ada programs with tasking. *[Proceedings 1988] the third international ieee conference on ada applications and environments* (1988), 15–26.

[97] Havelund, K. and Pressburger, T. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer.* 2, 4 (2000), 366–381.

[98] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S. and Zill, B. 2015. IronFleet: Proving practical distributed systems correct. *Symposium on operating systems principles* (2015).

[99] Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D. and Zill, B. 2014. Ironclad apps: End-to-end security via automated full-system verification. *Symposium on operating systems design and implementation* (2014).

[100] Heule, M.J., Järvisalo, M. and Suda, M. SAT RACE 2019.

[101] Heyting, A. 2013. *Mathematische grundlagenforschung intuitionismus beweistheorie.* Springer-Verlag.

[102] Hoare, C.A.R. 1969. An axiomatic basis for computer programming. *Communications of the ACM.* 12, 10 (1969), 576–580.

[103] Hodes, L. 1971. Solving problems by formula manipulation in logic and linear inequalities. *Proceedings of the 2nd international joint conference on artificial intelligence* (1971), 553–559.

[104] Holler, C., Herzig, K. and Zeller, A. 2012. Fuzzing with code fragments. *Presented as part of the 21st USENIX security symposium (USENIX security 12)* (2012), 445–458.

[105] Holzmann, G.J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering.* 23, 5 (1997), 279–295.

[106] Howard, W.A. 1980. The formulae-as-types notion of construction. *To HB Curry: essays*

on combinatory logic, lambda calculus and formalism. 44, (1980), 479–490.

[107] Howden, W.E. 1977. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering.* 3, 4 (1977).

[108] Institute, P. 2016. *Cost of data center outages.* https://www.vertiv.com/globalassets/documents/reports/2016-cost-of-data-center-outages-11-11_51190_1.pdf.

[109] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K. and Candea, G. 2019. Performance contracts for software network functions. *Symposium on networked systems design and implementation* (2019).

[110] Jacobs, B. and Piessens, F. 2008. The VeriFast program verifier. *CW Reports.* Department of Computer Science, K.U.Leuven; Leuven, Belgium.

[111] Jacobs, B., Smans, J. and Piessens, F. 2017. The VeriFast program verifier: A tutorial. Zenodo.

[112] Janicic, P., Green, I. and Bundy, A. 1997. A comparison of decision procedures in presburger arithmetic. *DAI RESEARCH PAPER.* (1997).

[113] Jenn, E., Arlat, J., Rimen, M., Ohlsson, J. and Karlsson, J. 1995. Fault injection into VHDL models: The MEFISTO tool. *Predictably dependable computing systems.* Springer. 329–346.

[114] Ju, X., Soares, L., Shin, K.G., Ryu, K.D. and Da Silva, D. 2013. On fault resilience of openstack. *Symposium on cloud computing* (2013), 1–16.

[115] Kadlecsik, J. and Pásztor, G. 2004. *Netfilter performance testing.* Netfilter Project, Berlin, Germany; http://people.netfilter.org/kadlec/nftest.pdf.

[116] Kanawati, G.A., Kanawati, N.A. and Abraham, J.A. 1995. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers.* 44, 2 (1995), 248–260.

[117] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the twenty-ninth annual ACM symposium on theory of computing* (1997).

[118] Kazemian, P., Chan, M., Zeng, H., Varghese, G., McKeown, N. and Whyte, S. 2013. Real time network policy checking using header space analysis. *Symposium on networked systems design and implementation* (2013).

[119] Kazemian, P., Varghese, G. and McKeown, N. 2012. Header space analysis: Static checking for networks. *Symposium on networked systems design and implementation* (2012).

[120] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A. and Akella, A. 2016. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. *Symposium on networked systems design and implementation* (2016).

[121] Khoshgoftaar, T.M., Allen, E.B., Jones, W.D. and Hudepohl, J.P. 1999. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering.* 9, 05 (1999), 547–563.

[122] Khurshid, A., Zou, X., Zhou, W., Caesar, M. and Godfrey, P.B. 2013. VeriFlow: Verifying network-wide invariants in real time. *Symposium on networked systems design and implementation* (2013).

[123] Kim, S., Whitehead, E.J. and Zhang, Y. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering.* 34, 2 (2008), 181–196.

[124] King, J.C. 1976. Symbolic execution and program testing. *J. ACM.* 19, 7 (1976).

[125] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J. and Yakobowski, B. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing.* 27, 3 (2015).

[126] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M. and others 2009. seL4: Formal verification of an OS kernel. *Symposium on operating systems principles* (2009).

[127] Kochhar, P.S., Bissyandé, T.F., Lo, D. and Jiang, L. 2013. An empirical study of adoption of software testing in open source projects. *International conference on quality software* (2013), 103–112.

[128] Kohler, E., Morris, R., Chen, B., Jannotti, J. and Kaashoek, M.F. 2000. The Click modular router. *ACM Transactions on Computer Systems.* 18, 3 (2000).

[129] Kolmogoroff, A. 1932. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift.* 35, 1 (1932), 58–65.

[130] Kothari, N., Mahajan, R., Millstein, T., Govindan, R. and Musuvathi, M. 2011. Finding protocol manipulation attacks. *ACM SIGCOMM Conference* (2011).

[131] Kreibich, C., Weaver, N., Nechaev, B. and Paxson, V. 2010. Netalyzr: Illuminating the edge network. *Internet measurement conference* (2010), 246–259.

[132] Kuznetsov, V., Chipounov, V. and Candea, G. 2010. Testing closed-source binary device drivers with DDT. *USENIX annual technical conference* (2010).

[133] Kuznetsov, V., Kinder, J., Bucur, S. and Candea, G. 2012. Efficient state merging in symbolic execution. *International conference on programming language design and implementation* (2012).

[134] Kuzniar, M., Peresini, P., Canini, M., Venzano, D. and Kostic, D. 2012. A SOFT way for OpenFlow switch interoperability testing. *Proceedings of the 8th international conference on emerging networking experiments and technologies* (2012).

[135] Lee, C.-Y. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal.* 38, 4 (1959), 985–999.

[136] Leino, K.R.M. 2010. Dafny: An automatic program verifier for functional correctness. *International conference on logic for programming artificial intelligence and reasoning* (2010).

[137] Leino, K.R.M. and Logozzo, F. 2005. Loop invariants on demand. *Asian symposium on programming languages and systems* (2005), 119–134.

[138] Leszak, M., Perry, D.E. and Stoll, D. 2002. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*. 61, 3 (2002), 173–187.

[139] Lewis, J. 2007. Cryptol: Specification, implementation and verification of high-grade cryptographic applications. *Fmse* (2007).

[140] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N. and Foster, N. 2018. P4v: Practical verification for programmable data planes. *ACM SIGCOMM Conference* (2018), 490–503.

[141] Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K. and Varghese, G. 2015. Checking beliefs in dynamic networks. *Symposium on networked systems design and implementation* (2015).

[142] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B. and King, S.T. 2011. Debugging the data plane with Anteater. *ACM SIGCOMM Conference* (2011).

[143] Majumdar, R. and Sen, K. 2007. Hybrid concolic testing. *International conference on software engineering* (2007), 416–426.

[144] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. and Huici, F. 2014. ClickOS and the art of network function virtualization. *Symposium on networked systems design and implementation* (2014).

[145] Matichuk, D., Murray, T., Andronick, J., Jeffery, R., Klein, G. and Staples, M. 2015. Empirical study towards a leading indicator for cost of formal software verification. *Proceedings of the 37th international conference on software engineering-volume 1* (2015).

[146] Matsakis, N.D. and Klock, F.S. 2014. The rust language. *ACM SIGAda Ada Letters*. 34, 3 (2014), 103–104.

[147] McCarthy, J. 1959. A basis for a mathematical theory of computation. *Studies in logic and the foundations of mathematics.*

[148] McKeeman, W.M. 1998. Differential testing for software. *Digital Technical Journal*. 10, 1 (1998), 100–107.

[149] McMillan, K.L. 1993. Symbolic model checking. *Symbolic model checking*. Springer. 25–60.

[150] Miller, B.P., Fredriksen, L. and So, B. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*. 33, 12 (1990), 32–44.

[151] Mitchell, M. 1998. *An introduction to genetic algorithms*. MIT press.

[152] Moon, S., Helt, J., Yuan, Y., Bieri, Y., Banerjee, S., Sekar, V., Wu, W., Yannakakis, M. and Zhang, Y. 2019. Alembic: Automated model inference for stateful network functions. *Symposium on networked systems design and implementation* (2019).

[153] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. *Design automation conference* (New York, NY, USA, 2001), 530–535.

[154] Musuvathi, M., Engler, D.R. and others 2004. Model checking large network protocol

implementations. *Symposium on networked systems design and implementation* (2004).

[155] Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R. and Dill, D.L. 2002. CMC: A pragmatic approach to model checking real code. *Symposium on operating systems design and implementation* (2002).

[156] Myers, A.C. and Myers, A.C. 1999. JFlow: Practical mostly-static information flow control. *Symposium on principles of programming languages* (1999), 228–241.

[157] Nagappan, N. and Ball, T. 2005. Use of relative code churn measures to predict system defect density. *Proceedings of the 27th international conference on software engineering* (2005), 284–292.

[158] Nagarakatte, S., Zhao, J., Martin, M.M. and Zdancewic, S. 2010. CETS: Compiler enforced temporal safety for C. *SIGPLAN Notices*. 45, 8 (2010).

[159] Nagarakatte, S., Zhao, J., Martin, M.M. and Zdancewic, S. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *SIGPLAN Notices*. 44, 6 (2009).

[160] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E. and Wang, X. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. *Symposium on operating systems principles* (Huntsville, Ontario, Canada, 2019).

[161] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E. and Wang, X. 2017. Hyperkernel: Push-button verification of an OS kernel. *Symposium on operating systems principles* (2017).

[162] Nipkow, T., Paulson, L.C. and Wenzel, M. 2002. *Isabelle/HOL: A proof assistant for higher-order logic*. Springer Science & Business Media.

[163] Norell, U. 2007. *Towards a practical programming language based on dependent type theory*. Citeseer.

[164] Ognawala, S., Hutzelmann, T., Psallida, E. and Pretschner, A. 2018. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. *Symposium on applied computing* (2018), 1475–1482.

[165] O'Hearn, P. 2019. Personal Communication.

[166] O'Hearn, P.W. 2018. Continuous reasoning: Scaling the impact of formal methods. *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science* (2018).

[167] O'Hearn, P.W. and Pym, D.J. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic*. 5, 2 (1999), 215–244.

[168] O'Hearn, P.W., Reynolds, J.C. and Yang, H. 2001. Local reasoning about programs that alter data structures. *International workshop on computer science logic* (2001).

[169] Owre, S., Rushby, J.M. and Shankar, N. 1992. PVS: A prototype verification system. *International conference on automated deduction* (1992), 748–752.

[170] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G. 2011. Faults in

linux: Ten years later. *SIGPLAN Not.* 46, 3 (Mar. 2011), 305–318. DOI:https://doi.org/10.1145/1961296.1950401.

[171] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S. and Shenker, S. 2016. NetBricks: Taking the V out of NFV. *Symposium on operating systems design and implementation* (2016).

[172] Panda, A., Lahav, O., Argyraki, K., Sagiv, M. and Shenker, S. 2017. Verifying reachability in networks with mutable datapaths. *Symposium on networked systems design and implementation* (2017).

[173] Papastergiou, G., Fairhurst, G., Ros, D., Brunstrom, A., Grinnemo, K.-J., Hurtig, P., Khademi, N., Tüxen, M., Welzl, M., Damjanovic, D. and others 2016. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*. 19, 1 (2016), 619–639.

[174] Pedrosa, L., Fogel, A., Kothari, N., Govindan, R., Mahajan, R. and Millstein, T. 2015. Analyzing protocol implementations for interoperability. *Symposium on networked systems design and implementation* (2015).

[175] Pedrosa, L., Iyer, R., Zaostrovnykh, A., Fietz, J. and Argyraki, K. 2018. Automated synthesis of adversarial workloads for network functions. *ACM SIGCOMM Conference* (2018).

[176] Penninckx, W., Mühlberg, J.T., Smans, J., Jacobs, B. and Piessens, F. 2012. Sound formal verification of Linux's USB BP keyboard driver. *NASA formal methods symposium* (2012).

[177] Perkins, J.H. and Ernst, M.D. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. *ACM SIGSOFT Software Engineering Notes*. 29, 6 (2004).

[178] Pirelli, S., Zaostrovnykh, A. and Candea, G. 2018. A formally verified NAT stack. *ACM sigcomm workshop on kernel-bypass networks* (2018).

[179] Plotkin, G.D., Bjørner, N., Lopes, N.P., Rybalchenko, A. and Varghese, G. 2016. Scaling network verification using symmetry and surgery. *Symposium on principles of programming languages* (St. Petersburg, FL, USA, 2016).

[180] Post, H. and Küchlin, W. 2007. Integrated static analysis for Linux device driver verification. *International conference on integrated formal methods* (2007).

[181] Postel, J. 1983. *Discard protocol*. Technical Report #863. Internet Engineering Task Force; Internet Requests for Comments.

[182] Potharaju, R. and Jain, N. 2013. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. *Internet measurement conference* (2013), 9–22.

[183] Presburger, M. 1929. Uber die vollstandigkeiteines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt. *Comptes-rendus du ler congres des mathematiciens des pays slavs* (1929).

[184] Primorac, M., Argyraki, K. and Bugnion, E. 2017. How to measure the killer microsecond. *ACM sigcomm workshop on kernel-bypass networks* (2017).

[185] Queille, J.-P. and Sifakis, J. 1982. Specification and verification of concurrent systems in

CESAR. *International symposium on programming* (1982).

[186] Rajagopalan, S., Williams, D., Jamjoom, H. and Warfield, A. 2013. Split/Merge: System support for elastic execution in virtual middleboxes. *Symposium on networked systems design and implementation* (Lombard, IL, 2013), 227–240.

[187] Rakamarić, Z. and Emmi, M. 2014. SMACK: Decoupling source language details from verifier implementations. *Computer aided verification* (2014), 106–113.

[188] Ramamoorthy, C.V., Ho, S.-B. and Chen, W.T. 1976. On the automated generation of program test data. *IEEE Transactions on Software Engineering.* 2, 4 (1976).

[189] Renzelmann, M.J., Kadav, A. and Swift, M.M. 2012. SymDrive: Testing drivers without devices. *Symposium on operating systems design and implementation* (2012).

[190] Reynolds, J.C. 2002. Separation logic: A logic for shared mutable data structures. *Proceedings 17th annual IEEE symposium on logic in computer science* (2002).

[191] Rice, H.G. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society.* 74, 2 (1953), 358–366.

[192] Rojas, J.M., Fraser, G. and Arcuri, A. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability.* 26, 5 (2016), 366–401.

[193] Ryzhyk, L., Bjørner, N., Canini, M., Jeannin, J.-B., Schlesinger, C., Terry, D.B. and Varghese, G. 2017. Correct by construction networks using stepwise refinement. *Symposium on networked systems design and implementation* (2017).

[194] Saltzer, J.H., Reed, D.P. and Clark, D.D. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems.* 2, (1984), 277–288.

[195] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S. and Song, D. 2010. A symbolic execution framework for JavaScript. *2010 IEEE symposium on security and privacy* (May 2010), 513–528.

[196] Sekar, V., Ratnasamy, S., Reiter, M.K., Egi, N. and Shi, G. 2011. The middlebox manifesto: Enabling innovation in middlebox deployment. *Proceedings of the 10th acm workshop on hot topics in networks* (2011), 1–6.

[197] Sen, K., Marinov, D. and Agha, G. 2005. CUTE: A concolic unit testing engine for C. *ACM sigsoft software engineering notes* (2005).

[198] Serebryany, K. 2015. libFuzzer—a library for coverage-guided fuzz testing. *LLVM project.* https://llvm.org/docs/LibFuzzer.html.

[199] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S. and Sekar, V. 2012. Making middleboxes someone else's problem: Network processing as a cloud service. *SIGCOMM Computer Communication Review.* 42, 4 (2012), 13–24.

[200] Sigurbjarnarson, H., Bornholt, J., Torlak, E. and Wang, X. 2016. Push-button verification of file systems via crash refinement. *Symposium on operating systems design and implementation* (2016).

[201] Sigurbjarnarson, H., Nelson, L., Castro-Karney, B., Bornholt, J., Torlak, E. and Wang, X. 2018. Nickel: A framework for design and verification of information flow control systems. *Symposium on operating systems design and implementation* (2018).

[202] Smeding, G.J. 2009. An executable operational semantics for python. Universiteit Utrecht.

[203] Smith, J.M. and Nettles, S.M. 2004. Active networking: One view of the past, present, and future. (2004).

[204] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P. and Saxena, P. 2008. BitBlaze: A new approach to computer security via binary analysis. *International conference on information systems security* (2008), 1–25.

[205] Srisuresh, P. and Egevang, K. 1996. *Architectural principles of the Internet.* Technical Report #1958. Internet Engineering Task Force; Internet Requests for Comments.

[206] Srisuresh, P. and Egevang, K. 2001. *Protocol complications with the IP Network Address Translator.* Technical Report #1958. Internet Engineering Task Force; Internet Requests for Comments.

[207] Srisuresh, P. and Egevang, K. 2001. *Traditional IP network address translator (traditional NAT).* Technical Report #3022. Internet Engineering Task Force; Internet Requests for Comments.

[208] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. and Vigna, G. 2016. Driller: Augmenting fuzzing through selective symbolic execution. *Network and distributed system security symposium* (2016), 1–16.

[209] Stewart, G., Beringer, L., Cuellar, S. and Appel, A.W. 2015. Compositional CompCert. *SIGPLAN Notices.* 50, 1 (2015).

[210] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L. and Raiciu, C. 2018. Debugging P4 programs with Vera. *ACM SIGCOMM Conference* (2018).

[211] Stoenescu, R., Popovici, M., Negreanu, L. and Raiciu, C. 2016. SymNet: Scalable symbolic execution for modern networks. *ACM SIGCOMM Conference* (2016).

[212] Stoenescu, R., Popovici, M., Negreanu, L. and Raiciu, C. 2016. SymNet: Scalable symbolic execution for modern networks. *ACM SIGCOMM Conference* (2016).

[213] Tahat, L.H., Vaysburg, B., Korel, B. and Bader, A.J. 2001. Requirement-based automated black-box test generation. *25th annual international computer software and applications conference. COMPSAC 2001* (2001), 489–495.

[214] Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T. and Reps, T. 2010. Directed proof generation for machine code. *International conference on computer aided verification* (2010), 288–305.

[215] Tsai, T.K. and Iyer, R.K. 1995. Measuring fault tolerance with the FTAPE fault injection tool. *International conference on modelling techniques and tools for computer performance evaluation* (1995), 26–40.

[216] Volpano, D., Irvine, C. and Smith, G. 1996. A sound type system for secure flow analysis. *Journal of computer security.* 4, 2-3 (1996), 167–187.

[217] Walfish, M., Stribling, J., Krohn, M.N., Balakrishnan, H., Morris, R.T. and Shenker, S. 2004. Middleboxes no longer considered harmful. *Symposium on operating systems design and implementation* (2004), 15–15.

[218] Wang, C., Pastore, F., Goknil, A., Briand, L. and Iqbal, Z. 2015. Automatic generation of system test cases from use case specifications. *Proceedings of the 2015 international symposium on software testing and analysis* (2015), 385–396.

[219] Wang, P., Wang, D. and Chlipala, A. 2017. TiML: A functional language for practical complexity analysis with invariants. (2017).

[220] Wang, T., Wei, T., Gu, G. and Zou, W. 2011. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC).* 14, 2 (Sep. 2011). DOI:https://doi.org/10.1145/2019599.2019600.

[221] Wang, Z., Qian, Z., Xu, Q., Mao, Z. and Zhang, M. 2011. An untold story of middleboxes in cellular networks. *SIGCOMM computer communication review* (2011), 374–385.

[222] Weitz, K., Woos, D., Torlak, E., Ernst, M.D., Krishnamurthy, A. and Tatlock, Z. 2016. Scalable verification of border gateway protocol configurations with an smt solver. *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications* (Amsterdam, Netherlands, 2016).

[223] Xi, H. 2002. Dependent types for program termination verification. *Higher-Order and Symbolic Computation.* 15, 1 (2002), 91–131.

[224] Xie, G.G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G. and Rexford, J. 2005. On static reachability analysis of IP networks. *International conference on computer communications (infocom)* (2005).

[225] Xu, X., Jiang, Y., Flach, T., Katz-Bassett, E., Choffnes, D. and Govindan, R. 2015. Investigating transparent web proxies in cellular networks. *International conference on passive and active network measurement* (2015), 262–276.

[226] Yang, H. and Lam, S.S. 2015. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking.* 24, 2 (2015).

[227] Yang, X., Chen, Y., Eide, E. and Regehr, J. 2011. Finding and understanding bugs in c compilers. *International conference on programming language design and implementation* (2011), 283–294.

[228] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S. and Bairavasundaram, L. 2011. How do fixes become bugs? *Proceedings of the 19th ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering* (New York, NY, USA, 2011), 26–36.

[229] Yu, Y., Manolios, P. and Lamport, L. 1999. Model checking TLA+ specifications. *Advanced research working conference on correct hardware design and verification methods* (1999), 54–66.

[230] Yurichev, D. 2019. *SAT/SMT by example*. https://yurichev.com/writings/SAT_SMT_by_example.pdf.

[231] Zalewski, M. 2014. Technical “whitepaper” for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt.

[232] Zaostrovnykh, A., Argyraki, K. and Candea, G. 2016. Nework software verification survey. https://vignat.github.io/survey.

[233] Zaostrovnykh, A., Pirelli, S., Iyer, R., Rizzo, M., Pedrosa, L., Argyraki, K. and Candea, G. 2019. Verifying software network functions with no verification expertise. *Symposium on operating systems principles* (2019).

[234] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K. and Candea, G. 2017. A formally verified NAT. *ACM SIGCOMM Conference* (2017).

[235] Zhang, K., Zhuo, D., Akella, A., Krishnamurthy, A. and Xi, W. 2020. Automated verification of customizable middlebox properties with Gravel. *Symposium on networked systems design and implementation* (Santa Clara, CA, Feb. 2020).

[236] Zinzindohoué, J.-K., Bhargavan, K., Protzenko, J. and Beurdouche, B. 2017. HACL*: a verified modern cryptographic library. *Conference on computer and communication security* (2017).

[237] Clang 5 documentation—UndefinedBehaviorSanitizer—available checks. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#available-checks.

[238] 2020. Customized KLEE used as part of vigor framework. https://github.com/vigor-nf/klee.

[239] 2019. Debian: Release-critical bugs status. https://bugs.debian.org/release-critical/.

[240] 2018. DPDK bug: Crash on initialization if first RTE_MAX_LCORE cores are disabled. https://bugs.dpdk.org/show_bug.cgi?id=19.

[241] 2018. DPDK bug: Ixgbe driver changes FCTRL without first disabling RXCTRL.RXEN. https://bugs.dpdk.org/show_bug.cgi?id=21.

[242] 2018. DPDK bug: Ixgbe driver does not ensure FWSM firmware mode is valid before using it. https://bugs.dpdk.org/show_bug.cgi?id=26.

[243] 2018. DPDK bug: Ixgbe driver sets RDRXCTL with the wrong RSCACKC and FCOE_WRFIX values. https://bugs.dpdk.org/show_bug.cgi?id=22.

[244] 2018. DPDK bug: Ixgbe driver sets TDH register after TXDCTL.ENABLE is set. https://bugs.dpdk.org/show_bug.cgi?id=25.

[245] 2018. DPDK bug: Ixgbe driver sets unknown bit of the 82599’s SW_FW_SYNC register. https://bugs.dpdk.org/show_bug.cgi?id=24.

[246] 2018. DPDK bug: Ixgbe driver writes to reserved bit in the EIMC register. https://bugs.dpdk.org/show_bug.cgi?id=23.

[247] 2018. DPDK bug: mmap with MAP_ANONYMOUS should have fd == -1. https://bugs.dpdk.org/show_bug.cgi?id=18.

[248] 2018. DPDK bug: Undefined behavior caused by NUMA function in eal_memory. https://bugs.dpdk.org/show_bug.cgi?id=20.

[249] 2020. DPDK: Data plane development kit. https://dpdk.org.

[250] 2018. GitStats—Linux—lines. https://phoronix.com/misc/linux-20180915/lines.html.

[251] 2017. Introduction to receive side scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[252] 2018. moonpol. https://github.com/erkinkirdan/moonpol.

[253] 2019. OpenWRT. https://openwrt.org/.

[254] 2020. Vigor project repository. https://vigor.epfl.ch.