Thèse n° 9541

Automated Formal Verification of Software Network Functions

Présentée le 21 mars 2024

Faculté informatique et communications
Laboratoire des systemes fiables
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Solal Vincenzo PIRELLI

Acceptée sur proposition du jury

Prof. C. Pit-Claudel, président du jury
Prof. G. Candea, directeur de thèse
Dr R. Beckett, rapporteur
Prof. M. Kapritsos, rapporteur
Prof. S. Kashyap, rapporteur

École
polytechnique
fédérale
de Lausanne

2024

I have very little regard for consensus if it blinds you to the truth.
— Harry Belafonte

Acknowledgements

Where to even begin? After not only 6 years of PhD studies but my Bachelor and Master studies as well, there are too many people I should thank given the space available in both this section and my own memory. If you are reading this, thank you for reading it, and forgive me if I should have thanked you by name below but did not.

First, let me thank George, my advisor, for not just advising me but bearing with my unusual approach to research and frequent decisions to try something unrelated to what we had discussed I should be doing. George also encouraged me to give lectures in his Software Engineering course, which made me discover I enjoy the teaching side of academia just as much as the research. Once again, this is not something many advisors would have trusted an early-career PhD student with, so thank you, George.

I would also like to thank my thesis committee: Clément Pit-Claudel, Sanidhya Kashyap, Manos Kapritsos, and Ryan Beckett. They gave insightful feedback and asked pointed questions about the work, which not only improved the thesis but my own way of thinking about my work. Two other professors I want to thank are Katerina Argyraki, who co-authored some of the work in this thesis and provided excellent advice on other work as well, and James Larus, who helped start my research career by giving me a major stake in an interesting project when I was still a Master's student.

For another side of my work, I want to thank Anjo Vahldiek-Oberwagner, Natacha Crooks, and Salvatore Signorello for taking my overly blunt feedback seriously when they were EuroSys Artifact Evaluation co-chairs and making me fourth co-chair, which strengthened my interest in reproducibility and repeatability. Similarly, I want to thank Vaastav Anand and Roberta De Viti, my co-organizers of a conference panel about reproducibility which was quite successful and certainly an order of magnitude better than I could have done on my own.

I must also thank the IC Doctoral School admins for always being there to help, and especially Marta Bellone for helping me help everyone by improving the admissions process and for her patience dealing with my never-ending suggestions in the process.

At DSLAB, I want to thank everyone and especially Arseniy, Rishabh, and Can who did part of their PhD along side me and were always there to help, listen to my latest idea, or listen to my latest rant. Céline, our admin, also deserves an extra-special mention for being all-around awesome and making all administrative processes easy. I also want to thank the people in labs around DSLAB, in particular for the weekly joint labs meeting, a weekday that I always looked forward to. Catalina, Mia, Muhammad, Zeinab, Alexandre, Basil, Shardul, Yawen, Marios, Samuel, Yugesh, Diyu... I'm probably missing some, and there are many more reasons to thank you besides this paragraph.

Besides work, I was also involved in a number of associations that made my time here much more enjoyable. In PolyProg, a now defunct association for programming competitions, I met many cool people: Amer, Christian, Jakub, Joey, Joseph, Jonas, Michalina, Robert, Ruofan, Tatiana, Wajeb... we had lots of fun organizing HC2 and other competitions together, and I hope you all remember these times as fondly as I do. In LauzHack, the hackathon association that is still going strong, I met even more cool people: Alina, Antoine, Audrey, Boubacar, Camille, Eric, Fabien, Florian, Güneş, Jacopo, Kopiga, Lionel, Luca, Mariia, Natalia, Nathalie, Qinyue, Simona, Sourabh, Tristan, Victor... I'm sad I will no longer be as active in organizing these events, but I'm very glad they're in capable hands.

And then there's even more people inside and outside of EPFL. Chiara and Tatiana, thank you for all of the conversations and the support when I needed it. Thanks to the awesome people around EDIC and its EPIC association like Vinitra, Jade, and Akhil. Thanks to the members of the Spanish InQuizition, our pub quiz team: Jess, Anna, Uday, ... the team is growing so fast any list would be obsolete by the time you read this.

I also had the pleasure of meeting many awesome interns in my internship at Microsoft Research in Redmond. Behnaz, Siva, Ryan, Santi, Pooria, thank you for being great colleagues and mentors. All of the Hyatt House gang, Ammar, Arasu, Foteini, Geetanjali, Joanne, João, Lauren, Siddharth, thank you for all of the memories.

Finally, I would like to thank my family, who supported me in the somewhat odd choice to go for a PhD and never doubted I could do it.

Lausanne, February 28, 2024

Abstract

Formally verifying the correctness of software is necessary to merit the trust people put in software systems. Currently, formal verification requires human effort to prove that a piece of code matches its specification and code changes to improve verifiability at the expense of other goals such as run-time performance. Exhaustive symbolic execution (ESE), a technique that automatically explores all paths in a piece of code, is a promising direction for automating formal verification. However, ESE faces the *path explosion* problem: a large number of different program paths with equivalent behavior prevent ESE from completing in reasonable time, since ESE enumerates each path individually.

This thesis addresses some of the challenges ESE currently faces, in the context of software network functions (NFs). These are real-world programs that the Internet relies on and are amenable to verification, because they are typically self-contained and have a clear specification.

In the first part of this thesis, we propose abstractions that enable ESE to analyze more code in reasonable time. We propose *ghost maps* to abstract over complex data structure code by executing equivalent simpler code written in terms of maps. We also propose *imperative loop summaries* to abstract over loops by executing equivalent loop-free code also written in terms of maps. Instead of exploring a large number of equivalent paths, an ESE engine can use our techniques to explore only paths that have different high-level behavior. As a result, our techniques can automatically verify the correctness of a set of NFs using only hand-written contracts for data structures, as opposed to the step-by-step proof annotations necessary in previous work.

In the second part of this thesis, we propose abstractions that make code amenable to ESE while also improving performance and safety. We present a new network card driver template specifically designed for NFs. Drivers following this template are not only easier to automatically verify, as they don't use complex data structures, but also provide better performance for the NFs that can use them compared to more general drivers. We additionally show that drivers written in safe languages can reach the same performance as their unsafe counterparts, as long as safe languages expose to developers a handful of specific type invariants that enable compilers to automatically prove the safety of potentially unsafe operations and thus avoid emitting run-time checks.

Résumé

Vérifier formellement qu'un logiciel est correct est nécessaire pour mériter la confiance investie par les utilisateurs dans les systèmes logiciels. Actuellement, la vérification formelle demande de l'effort humain pour prouver qu'un bout de code correspond à sa spécification, ainsi que des changements dans le code pour faciliter le raisonnement automatisé, ce qui peut avoir des effets négatifs sur d'autres buts comme la performance. L'exécution symbolique exhaustive (ESE), une technique qui explore automatiquement tous les chemins dans un bout de code, est une direction prometteuse pour automatiser la vérification formelle. Cependant, l'ESE rencontre le problème de *l'explosion de chemins* : une quantité trop large de chemins différents mais dont le comportement est équivalent empêche l'ESE de finir en un temps raisonnable, vu que l'ESE doit énumérer chaque chemin individuellement.

Cette thèse s'attaque à certains des problèmes qui font face à l'ESE, dans le contexte des fonctions réseau logicielles. Ces fonctions sont à la fois des vrais programmes dont l'Internet dépend et suffisament simples pour être vérifiées, car elles sont typiquement autonomes et ont une spécification claire.

Dans la première partie de cette thèse, nous proposons des abstractions qui permettent à l'ESE d'analyser plus de code dans un temps raisonnable. Nous proposons des *tables associatives fantômes* pour abstraire les structures de données complexes en exécutant du code plus simple écrit en terme de tables associatives à la place. Nous proposons aussi des *sommaires impératifs de boucles* pour abstraire les boucles en exécutant du code équivalent sans boucles utilisant également des tables associatives. Au lieu d'explorer un grand nombre de chemins équivalents, un moteur d'ESE peut utiliser nos techniques pour explorer seulement les chemins qui ont un comportement différent. Il en résulte que nos techniques permettent de vérifier automatiquement que des fonctions réseaux obéissent leur spécification en utilisant uniquement des contrats pour les structures de données, sans autres annotations de preuves contrairement à ce qui était nécessaire avec l'état de l'art précédent.

Dans la deuxième partie de cette thèse, nous proposons des abstractions pour rendre le code plus facile à analyser avec l'ESE, tout en améliorant aussi les performances et la sécurité. Nous présentons un nouveau modèle de pilote de carte réseau spécifiquement pour les fonctions réseaux. Les pilotes suivant ce modèle ne sont pas seulement plus faciles à vérifier automatiquement, car ils n'utilisent pas de structures de données complexes, mais fournissent également de meilleures performances aux fonctions réseaux par rapport aux pilotes plus généraux. Nous montrons également que des pilotes écrits dans des langages de programmation sûrs peuvent avoir les mêmes performances que les équivalents non sûrs, si ces langages fournissent quelques invariants de types de données qui permettent au compilateur de prouver automatiquement que les opérations sont sûres et éviter des tests à l'exécution.

Contents

1 Introduction

People trust software to perform the tasks it claims to perform, and in particular to not lie, crash, or hang. Our sensitive communications should arrive in the right inbox, our radiation therapy machines should deliver the right doses, and our plane autopilots should fly safely. Software failures can lead to security vulnerabilities, bodily injury, or even death. Because such failures breach the trust put into software, they can also lead to a regression to less efficient and potentially more dangerous manual practices such as in driving [45]. Yet, software trust is mainly backed up by experiments such as specific test cases: we run the code in a set of specific conditions and we check its behavior compared to what we expect. There have been impressive software verification efforts to merit the trust put in software components such as the deepest layers of an operating system [60], upon which the rest of the software stack depends, as well as cryptographic code [121], required to provide privacy, but these efforts required person-years worth of effort from experts. Even for software whose goal we fully understand, such as where Internet packets should be sent based on the packets' addresses, the time it currently takes to obtain hard guarantees can be a constraint on how quickly we would like software to evolve.

In this thesis, we focus on networking software, such as firewalls and routers. These are employed both in commercial networks from Internet Service Providers (ISPs) and in home devices. The latter are sometimes called "routers", despite doing much more than routing, and provide Internet connectivity in many homes. While a network failure is unlikely to cause death as a radiation therapy failure would, network failures can lead to tangible breaches of trust: the network might become unavailable due to a crash in its core infrastructure [54], or an email server might let unauthorized users read private emails [39].

Historical examples of networking-related problems abound. Some security vulnerabilities were severe enough to be given names to educate end users about them. Heartbleed [104] enabled "anyone on the Internet to read the memory of the systems protected by the vulnerable versions" of the OpenSSL cryptographic library. Spring4Shell [94] enabled attackers to execute their own code on a remote server running the Spring web framework, and was named after Log4Shell [114], itself an infamous vulnerability in the widely-used software logging component Log4j. SMBGhost [40], a vulnerability in a Microsoft Windows networking component, was severe enough to merit a public warning from the U.S. Department of Homeland Security.

To avoid security vulnerabilities and software bugs in general, we would like to formally and automatically prove the correctness of software components as defined by a specification. Ideally, there should be no constraints on the software implementation besides those that already exist for goals other than verification, such as performance and maintainability. Developers should be able to write code, write a specification of what properties the code should have, run a verification tool with the code and specification as inputs, and automatically obtain either a proof of correctness or a counter-example highlighting a bug in the code. Such a tool could even be used by operators deploying software if they have access to the source code, or if the tool accepts compiled programs as input.

This thesis brings the state of the art in automated formal verification closer to its overall goal in two ways: (1) we enable tools to verify more code automatically, resulting in the automated verification of network functions such as a firewall and a NAT with orders of magnitude fewer human annotations than previous work; and (2) we show that, at least in the case of network card drivers, changing code to make it more verifiable also improves performance and thus is not a verification-specific burden.

The specific goal of this thesis is to automate the formal verification of real-world software network functions by trading completeness for automation: it is acceptable to fail to verify contrived examples if doing so improves the technique in other ways, such as verifying more real-world exampls or making verification faster. The run-time performance and design effort for verified NFs should not suffer from verification.

The key challenge comes from the conflict between automated verification, which is traditionally done on high-level code, and real-world NF code, which extensively uses low-level constructs.

This thesis does not address the problem of *validation*, which is checking whether the specification captures the intent of users, such as whether a firewall should treat some types of packets in a special way or whether that is a bad idea. Validation is an interesting and challenging problem but it is a social one that is outside the scope of this technical thesis.

1.1 Motivation

Why this thesis? In this section, we justify our choices of target, network functions, of technique, formal verification, and of approach, systems research.

1.1.1 Network Functions

For the purpose of this thesis, we define *network functions*, or NFs for short, as packet-processing appliances that implement network policies such as “individual computers should not use too much bandwidth” using mechanisms such as “count the number of bytes each computer transmits, and drop packets if there are too many over a short period of time”. They are also known as *middleboxes* [15] and their use is on the rise [41]. NFs can be simple, such as a traffic policer counting packets, but they can also be more complex and encompass entire protocols, such as 4G and 5G mobile networks.

Hardware network functions are the traditional way to implement NFs for high-traffic networks. They are physical boxes with custom hardware that are part of the network, distinct from standard computers. These are known as *ASICs*, application-specific integrated circuits. Because modifying hardware after its deployment to fix bugs is only possible if it has been designed for the specific modifications required, such as using microcode, hardware appliances have a higher quality bar for shipping compared to software ones. Thus, there have been considerable efforts in hardware verification [57].

Software network functions run on general-purpose hardware such as x86 and mainstream operating systems such as Linux, using a software stack that includes network cards drivers and implementations of protocols such as IP and TCP [27].

The networking world is moving from hardware to software to increase flexibility. Software NFs are flexible since they have low deployment costs. This means that correctness guarantees, while important, are not a hard requirement for deployment. Developers and operators currently test network functions hoping to catch bugs, but this is not enough especially in light of frequent software updates. For instance, network address translators from Microsoft [76], Linux [77], and Cisco [78] have had headline-causing bugs and vulnerabilities.

Software NFs are a good application domain for research in systems and formal methods because they are a “Goldilocks” sweet spot in three domains:

1. They are small enough to be analyzable, but still large enough to be interesting real-world software rather than toy examples;
2. They are self-contained enough for their environment interactions to be analyzable, but still interact with their environment in meaningful ways; and
3. Their purpose is clear enough to be formally definable, but still flexible enough that there is no single implementation to fit all use cases.

1.1.2 Formal Verification

Formal verification is the process of proving, in mathematical terms, that a given piece of code has a given property. The code can in theory be in any programming language and do any task, though some programming languages are easier to reason about than others because they impose more constraints, such as only manipulating memory in well-defined ways. The property can be as short as "don't divide by zero" or as complex as a full specification defined over multiple program executions. However, all proofs make assumptions, even if they are implicit such as "if someone unplugs your desktop computer's power cable, the software will not be able to finish".

Formal verification can prove properties in software network functions. For instance, going back to the traffic policer, one could prove the property "given two different computers A and B, the count of bytes for A has no influence on what happens to B's packets within the policer". As another example, a firewall may have a specification stating "a packet P from the global Internet is only let through if there exists a previous packet P2 from the internal network that P is replying to".

The reason why formal verification is not yet widely used in software development is that it requires effort and expertise. Writing a proof, even assisted by good tools, takes time, and writing code that is easy to reason about requires deep knowledge of formal verification. This workflow hurts developer productivity, making it unlikely to be adopted outside of extreme cases in which safety has life-or-death consequences or in which a safety bug could cost extremely large amounts of money.

The approach we take in this thesis is to start from existing techniques in *automated formal verification* and extend them to support more kinds of code. Such techniques do not require any human interaction after being given the program and the specification. For instance, Vigor [119] is a state-of-the-art tool that, given the code of a software network function and a specification, automatically proves or disproves that the code meets the specification, as long as both are constrained in specific ways such as using only specific data structures in the code rather than arbitrary ones. This is a promising direction to reduce the need for effort in particular, since the process is "push-button". It also reduces the need for expertise since there is no need to learn any separate language for proofs or lemmas. However, "automated" does not imply "automatic". Developers must still have a sense of what is verifiable and what is not, and write code in consequence. It is not possible to automatically verify arbitrary properties for arbitrary pieces of code, in practice because of how long some properties can take to prove even on well-behaved pieces of code, and in theory due to impossibility results such as Rice's theorem [64] and the Halting Problem [109].

There is another approach that we did not take: making manual tools easier to use and more automated. Manual verification tools can check a proof written by a human, but cannot generate the proof themselves. For instance, seL4 [60] is an operating system kernel whose proof was written by humans and checked by a machine, meaning one no longer needs to trust the humans involved in the proof but only the tool that checks the proof. The seL4 proof is an order of magnitude larger than its source code, though such comparisons are hard to generalize because some of the proof is reusable rather than being seL4-specific, and not all systems prove the same kinds of properties. Such tools can be improved so they can take larger steps in the proof instead of requiring an explanation for every small step the proof makes. There is also interesting work to be done in improving the usability of these tools, such as by enabling developers to write more graphical proofs for concepts that are easier to explain with graphics than with text. Such work is out of scope for this thesis.

1.1.3 A Systems Approach

We approach the problem of formally verifying network functions from the point of view of computer systems, i.e., with a whole-system perspective rather than isolating small portions. In this case, we observe that software network functions are currently being deployed as software stacks without formal guarantees, and we design formal methods that apply to what developers currently use to design the entire system. This includes not only the network functions themselves, but also the frameworks used by developers, such as DPDK [27], and the programming languages used by developers, specifically C.

We thus build our techniques “bottom-up”, starting from the concrete use cases we know of and designing abstractions that enable their verification as well as the verification of related software. We chose this direction because we had concrete problems in mind already.

The choice of C as a programming language is because it is the de facto standard for writing systems software, even though it is not as easy to reason about as other languages. For instance, it would be easier to reason about software NFs written in Python since they could not manipulate memory in arbitrary ways, or in OCaml since they would involve fewer side-effects and global state, but such work would be a harder sell for developers. This does not mean we support every possible C program. We can still impose restrictions on the kind of code we support, as long as they match good software engineering practices and thus do not impose a verification-specific burden on developers. The reason why we want the burden to be as low as possible, ideally zero, is that from experience most developers will not use any tooling that requires additional effort on their part.

Overall, we trade complexity and power for applicability. Because we handle real-world software written in an unsafe imperative programming language, our techniques must handle more constructs and especially lower-level constructs, but this makes them more applicable to existing code.

1.2 Problem Statement

1.2.1 Network Functions

We want to verify real-world software NFs, i.e., those that can be deployed in real networks such as home or corporate networks, or even in the backbones of Internet service providers.

These NFs have strict performance constraints, which limits how much time they can spend per packet since taking too much time may cause subsequent packets to be buffered or even dropped if the buffer is full. To illustrate how crucial performance is, consider the time budget for serially processing 64-byte packets and their 20-byte Ethernet header using a 10 Gb/s link: $(64+20) \times 8/(10 * 10^9) = 67.2$ ns per packet. This is the same order of magnitude as a memory read: a network function will exceed its time budget if it needs data outside the CPU cache.

These NFs are written in imperative low-level languages, typically C, with frameworks oriented towards performance and programmer control, such as DPDK, rather than high-level abstractions. The resulting code uses pointers, transfers and shares ownership of data structures in non-obvious ways to minimize copying, and generally requires a human or a machine to reason about memory at a low level of abstraction to understand the code. We cannot expect developers to use high-level operations that impose even small run-time overheads for low-level operations, nor can we expect them to learn entirely new programming languages or proof languages.

1.2.2 Formal Verification

Formal verification has two key objectives, which are sometimes at odds: *soundness*, meaning no incorrect program can pass verification, and *completeness*, meaning every correct program can be verified. An ideal tool is of course both sound and complete: it can verify all correct programs and no incorrect ones. However, tools can be useful even if they are unsound, such as a static analysis tool that flags potential errors but cannot find all errors, or incomplete, such as a verification tool that gives up on some code that it cannot reason about. Even a tool that is both unsound and incomplete can be useful if its "signal to noise" ratio is high enough and it applies to enough code.

We require soundness because it is necessary for trustworthiness in the context of formal verification. Developers should be able to trust that if our tools say a program obeys its specification, there are no bugs hiding in dark corners. While unsoundness can be useful during development to quickly catch some mistakes, it is not enough to build reliable software for key infrastructure such as the Internet. However, we accept trading more completeness than strictly necessary if it makes the tool more automated. Since we have concrete examples of code to verify in mind, our techniques do not need to be able to verify edge cases we do not use.

1.2.3 Automation

Manual verification requires multiple times more lines of proof than lines of code [60; 44], making it too expensive for all but the most safety-critical systems. Even for such systems, having to update the proofs every time the code changes limits the pace of evolution. In practice, this implies that proofs may perpetually lag behind the associated code.

We thus want automated verification with as little human effort as possible. This effort is measured in terms of how many code annotations are necessary and how complex these annotations are. Type systems are a good example of such low-effort high-reward annotations: by giving each variable a type, even with a simple type system, developers can catch common errors such as calling a function with the wrong argument type.

We also want to minimize verification-related constraints on how the code must be written. Code changes are acceptable if they can be justified in other ways such as improving maintainability. For instance, instead of convoluted `while` loops with `goto` statements jumping in and out of the loop, a structured `for` loop not only helps automated verification tools, but also helps humans maintain the code, and may even help compilers produce faster machine code using vectorization and other loop-related optimizations.

1.2.4 Summary

In summary, the goal of this thesis is to automate the formal verification of real-world software network functions. We want high completeness so that common code patterns can be verified, but we are more interested in automation and usability than completeness. The run-time performance of the verified NFs should be as close as possible, or better, than that of unverified code. The manual effort to design, implement, and verify the code should be as close as possible, or lower, to the effort of designing and implementing unverified code.

The key challenge comes from the conflict between automated verification, which is traditionally done on high-level code, and real-world NF code, which extensively uses low-level constructs.

1.3 Approach

This thesis is formed of two distinct parts that advance the automation of software verification.

1.3.1 Improving automated tooling

In the first part of this thesis, we propose techniques to improve the reach of automated verification. Because automation is an all-or-nothing property, “90% automatic” is equivalent to “not automatic” for developers who do not have verification expertise, and thus finding automation bottlenecks that prevent tools from handling entire categories of code can have high returns.

From experience with manual proofs, we believe such bottlenecks exist, especially around the handling of loops. Even straightforward loops often cannot be handled by state-of-the-art automated tools. This appears to be partly due to a chicken-and-egg problem: since tools cannot automatically handle loops, practical work on loops focuses on bug-finding rather than verification to be maximally applicable, thus there is not enough work on loops for automated verification, and so on.

To achieve this, we propose abstractions that fit common coding patterns, specifically data structures and loops, and enable a tool to automatically reason about these patterns. This is by opposition to general-purpose reasoning, which must be able to handle all sorts of theoretical edge cases that developers mostly do not use. In particular, whereas previous work required tools to consider each kind of data structure separately, as well as all of the possible combinations and their interactions, our abstractions require reasoning only about one kind of data structure, the map.

1.3.2 Simplifying programs

In the second part of this thesis, we provide evidence that making low-level systems code such as network card drivers more verifiable also improves other goals. That is, properties that make code verifiable are desirable independently of verification. For instance, structuring code in a simple and predictable way should help both automated verification tools reason better and compilers produce faster code.

To achieve this, we strip general-purpose abstractions down to the minimal components necessary for a specific use, software NFs in this case. As with verification improvements, these abstractions enable compilers to produce smaller and simpler assembly code, which executes faster.

This evidence goes against stereotypes of automated verification as a process that requires sacrifices in functionality or performance for low-level code. We do not claim that all systems can be made better by making them verifiable, only that we have evidence some systems fall in this category. There is also existing evidence that high-level systems can be made faster using higher-level abstractions, such as Halide [95].

1.4 Thesis statement

Appropriate abstractions enable the automated formal verification of software network functions. Abstracting data structures and loops at verification time enables automated verification of the network functions themselves, and simplifying network card drivers not only enables their automated verification but improves their performance and reliability.

1.5 Thesis contributions

This thesis makes the following contributions:

1. A technique to automatically reason about data structures and infer related invariants, based on the idea of *ghost maps*, a new verification-time data structure;
2. A technique to represent common loops using ghost maps, and to summarize these loops automatically such that an automated verification tool can handle them without user assistance;
3. A novel driver model for network cards specialized for common network functions that enables automated formal verification.

We implement our contributions in KLINT, a verification tool for software network functions implementing ghost maps, which we use to verify 7 network functions using only their binaries without debugging symbols, as well as BPF-based network functions; LOOPSY, a symbolic execution engine using loop summaries; and TINYNF, a formally verified driver for the Intel 82599 network card written in C.

We also provide evidence, using TINYNF and translating it in other programming languages, that verifiability and performance can go hand-in-hand.

1.6 Thesis roadmap

In the first part of this thesis, we show how to extend the reach of automated verification. In Chapter 3, we propose the use of maps to abstract data structures in symbolic execution, and present KLINT, a tool that uses such maps to automatically verify network functions. In Chapter 4, we extend maps to abstract common loops as well, and propose a technique to summarize these loops into loop-free code, making it more amenable to symbolic execution.

In the second part of this thesis, we build network card drivers to show that verifiability has benefits beyond verification. In Chapter 5, we present a new network card driver model for network functions that process packets one-by-one, which enables automated verification and also improves their performance even compared to the unverifiable state-of-the-art driver. In Chapter 6, we translate our driver implementation in C into safe programming languages, and show that improvements in safety and maintainability do not require sacrificing performance.

In Chapter 7, we discuss the limitations of this thesis, in Chapter 8 we present related work, and in Chapter 9 we conclude and propose future work.

1.7 Bibliographic notes

This thesis primarily builds upon the ideas presented in the following publications:

- A Simpler and Faster NIC Driver Model for Network Functions, S. Pirelli, G. Candea, OSDI 2020 [90].

- Automated Verification of Network Function Binaries, S. Pirelli, A. Valentukonytė, K. Argyraki, G. Candea, NSDI 2022 [92].
- Safe Low-Level Code Without Overhead is Practical, S. Pirelli, G. Candea, ICSE 2023 [91].

2 Background

In this section, we introduce the background necessary for the rest of the thesis. We recap the basics of network functions as well as the network cards and drivers they use, key techniques to formally verify systems code such as network functions, and the automation of verification.

As explained in the introduction, network functions are a good application domain because they are interesting real-world pieces of software that are also constrained enough to be in scope for work on formal verification, especially in terms of how they interact with their environment. An example of code that would not fall in the same category is the networking stack of a mainstream operating system, such as Linux or Windows. While it is in the same domain, such a stack is much more complex as it must support many different protocols, possibly deviate from the specification for compatibility with existing devices, and integrate with the rest of the operating system. Modeling the precise behavior of an operating system to verify a component that depends on this behavior would require tremendous effort and most likely be just as complex as the real code.

Network functions are not the only target we could have chosen. One other such target, which we considered but did not have time to evaluate further, is storage. Some storage applications also have a small surface for environment interactions, deal with hardware directly, and have stringent performance requirements. Automatically verifying storage applications could be interesting future work, especially in combination with the networking work we present here. Such systems already exist [61] but have not been automatically verified.

Because we focus on automating the verification of systems code such as network middleboxes and drivers, we focus on techniques that we believe have the most potential as the foundations for our work. We thus do not cover functional programming languages for the code, nor manual verification with annotations.

2.1 Networking code

2.1.1 Network functions

Network operators are moving from hardware *network functions* to software ones for flexibility, but still deploying them as black boxes. Historically, network functions such as firewalls, NATs, and load balancers used special-purpose hardware for performance at the cost of flexibility, but this tradeoff does not always make sense any more thanks to the performance of modern general-purpose hardware. Developers write software network functions using programming languages such as C or Rust and frameworks such as DPDK [27] or BPF [73; 21]. Marketplaces for distribution of software network functions are emerging [32], and most network functions on these marketplaces are proprietary and distributed in binary form.

Developers and operators currently test network functions hoping to catch bugs, but this is not enough especially in light of frequent software updates. For instance, network address translators from Microsoft [76], Linux [77], and Cisco [78] have had headline-causing bugs and vulnerabilities.

In this thesis, we restrict ourselves to network functions that use mutable state to process packets and contain two phases: initialization and packet processing. When processing a packet, network functions decide whether to transmit packets in response and what data to transmit based on state and on the packet's contents and metadata, such as its length. We assume that network functions are single-threaded, and that they only execute code when initializing and receiving packets. We do not support timers as triggers to run code; they must instead be checked during packet processing.

To create, read, and update state, network functions acquire and use data structure *capabilities* through an environment, such as the DPDK [27] framework. These capabilities correspond to opaque pointers in C, preventing network functions from directly accessing data structure internals.

Acquiring capabilities is only allowed during initialization, and may fail due to external factors, for instance running out of memory. Using capabilities is allowed at any time, and may only fail through incorrect use, for instance indexing arrays out of bounds. Packet processing may also use other environment calls that cannot fail from external factors, such as obtaining the current time.

Our assumptions about data structures correspond to good programming practices: developers should use data structures in a modular fashion for maintainability, and network functions should not allocate memory while processing packets, to avoid performance issues and out-of-memory errors that could allow for denial-of-service attacks. These assumptions are enforced during verification.

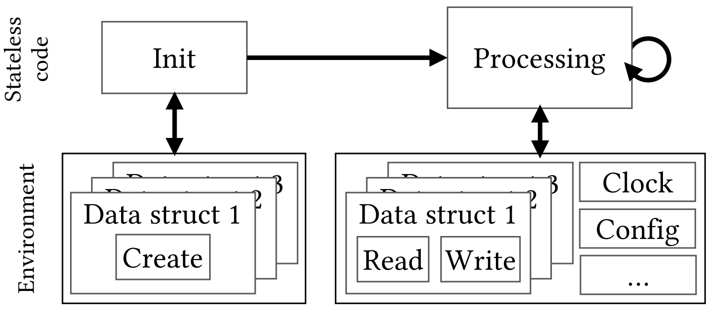


Figure 2.1: Network function components: initialization allocates state, packet processing reads and writes state and may also use other utilities. Arrows represent control flow.

We represent our model of a network function in Figure 2.1. This is only a formalization of what developers already do, not a proposal of a new model.

2.1.2 Network cards

We summarize the architecture of modern Network Interface Controllers, or “NICs” for short, which is necessary to understand network driver design. While NICs are diverse, the core concepts are similar. We estimate that our description applies to 90% of the network cards supported by DPDK. The remaining ones are proprietary cards and some based on reconfigurable hardware.

Communication between CPU and NIC uses three channels: PCI registers, NIC registers, and RAM. PCI registers are stored on the NIC and accessed by the CPU using port-mapped I/O. The CPU only uses them for the first stage of NIC initialization. NIC registers are stored on the NIC and accessed by the CPU using memory-mapped I/O. Their latency is an order of magnitude higher than RAM [84], making them a performance bottleneck. RAM is the main shared storage. The CPU accesses it as usual, and the NIC uses Direct Memory Access, or “DMA” for short, to transfer data into it. RAM holds packet buffers and metadata. The CPU and the NIC are not notified when the other has modified RAM; if they want to be aware of changes, they must poll RAM or use a side channel.

The packet descriptor is the main NIC data structure, containing a pointer to a data buffer and some metadata. The metadata typically contains required fields such as the packet length, and optional fields such as whether to use advanced hardware offloading features. Software chooses the total number of descriptors when initializing hardware. Descriptors are given from the CPU to the NIC to issue commands, such as packet transmission, and given by the NIC back to the CPU when the associated command has finished. Different NICs have different ways to manage descriptor ownership, such as flags in metadata. Software can change the buffer pointer before giving a descriptor to the NIC. This lets developers implement buffer pools, to reuse descriptors without losing received data. This is useful for cases such as TCP,

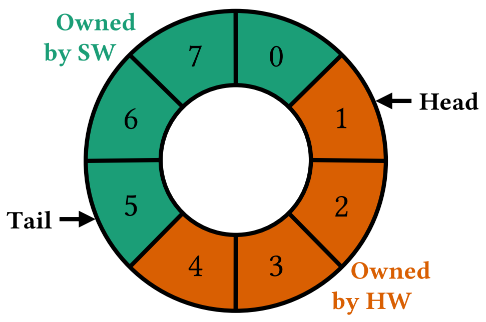


Figure 2.2: A descriptor ring with 8 elements; the head is 1 and the tail is 5, thus hardware owns elements 1, 2, 3 and 4.

where packets must be kept until an entire message has arrived.

Reception and transmission are the core operations of network cards, and work in symmetric ways. To receive packets, the CPU gives descriptors to the NIC indicating where to deposit packets in memory. The NIC gives descriptors back when it has received packets. The NIC sets descriptor metadata to indicate the packet length and other such information. To transmit packets, the CPU gives descriptors to the NIC indicating where packets are in memory, and the NIC gives descriptors back once it has transmitted packets. The metadata is set by the CPU, to inform the NIC of the packet length and other such information.

Descriptor rings are the main mechanism for descriptor ownership in modern cards. We present here their inner workings in Intel's 82599 NIC as a concrete example. A descriptor ring is composed of a region of memory, a head pointer, and a tail pointer. The memory is in RAM, while the pointers are NIC registers. Descriptors between the head, inclusive, and the tail, exclusive, belong to the NIC. Other descriptors belong to the CPU. If the head and tail are equal, the CPU owns all descriptors. We present an example ring in Figure 2.2. Since descriptors start in an unknown state, descriptor metadata has a "Descriptor Done" flag to let the CPU know whether a descriptor has been processed by the NIC or was just never initialized. The CPU gives descriptors to the NIC by clearing their "Descriptor Done" flag and incrementing the tail pointer. Since the tail pointer is a NIC register, the NIC immediately notices the change. The NIC gives descriptors back to the CPU by setting the "Descriptor Done" flag in metadata and incrementing the head pointer. To know when a descriptor has been given back, the CPU polls the metadata. The head and tail can only be incremented, though they can be incremented by more than 1 to give descriptors in batches. Decrementing is forbidden since it would logically be an attempt to steal descriptors.

NIC queues are a hardware mechanism to allow for parallel packet processing. A queue consists of a descriptor ring and some configuration. The NIC places all received packets in the first reception queue by default; developers can configure the NIC to route packets to a queue based on packet headers, such that packets belonging to the same logical flow are

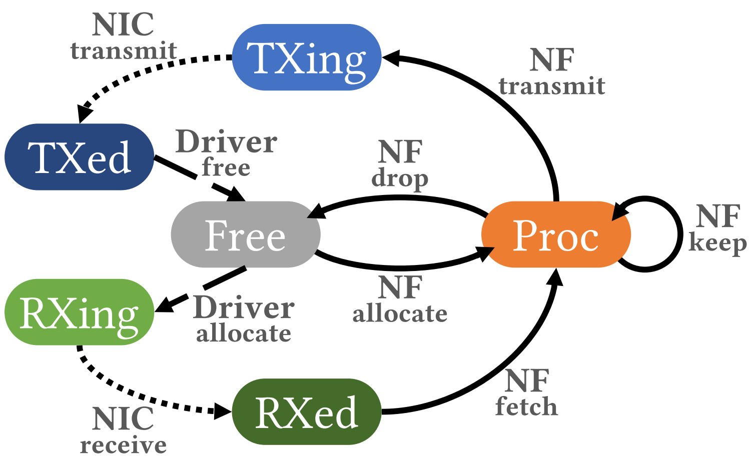


Figure 2.3: Diagram of the kernel-bypass driver model. Each box is a queue, each arrow is a step moving one packet from one queue to another. Steps are annotated with their actor and their name. “RX” is reception, “TX” is transmission, “Proc” is processing, and “NF” is network function

routed to the same queue. For transmission, all queues behave in the same way, without flow tracking: packets added to any transmission queue are sent to the wire regardless of which queue it is. Queues allow multiple CPU cores to handle packets without having to synchronize NIC accesses, increasing software scalability.

2.1.3 Network card drivers

Drivers are low-level pieces of software that interact with hardware directly. We focus on *user-mode* drivers, which perform the same task as *kernel-mode* drivers but run outside of the kernel, because they are much easier to isolate and replace. Such drivers are not toy examples. Networking code can use libraries such as DPDK to avoid the overhead of system calls at the cost of exclusive access to a network card. In practice, that network card can be virtualized [89], multiplexing it into virtual cards that applications can have exclusive access to.

We focus in particular on high-performance drivers intended for network functions, which must process many packets and thus minimize overheads even at the expense of some flexibility. The smallest Ethernet packet, including framing, is 84 bytes, thus handling 10 Gb/s of such packets means each packet must be handled in 67 ns. Handling packets in two directions, which is common for network software such as firewalls, halves the time budget. Spreading load across multiple cores can help, but Ethernet speeds are growing as well, currently up to 400 Gb/s.

The driver model in modern frameworks such as DPDK [27] is based around reusing a fixed set of packet buffers, in order to avoid the overheads of memory allocation. We formalize this model as a diagram in Figure 2.3, where the overall system is a set of first-in-first-out buffer queues. Each conceptual “step” of the system is performed by one of three actors: the NIC, the driver, or the network function. In the initial state, all buffers are in the “Free” state, which represents unused buffers in the buffer pool. The driver typically takes the first steps, by “allocating” buffers from the pool and giving them to the NIC for reception. This “allocation” refers to taking buffers from the pool, not creating new ones. If there are buffers in the “receiving” state, the NIC can transition them to the “received” state once it gets data from the network. The network function typically runs a polling loop to move buffers into the “processing” step. From there, the network function can choose to transmit the buffer, possibly after modifying it. The NIC will then send out the buffer contents to the network and move the buffer to the “transmitted” state. The driver moves transmitted buffers back to the ‘free” state at well-defined points, for instance when there are too few free buffers left. The NF can also choose to keep the buffer for later, or to “drop” it and return it to the pool. The network function can also allocate buffers from the pool and process them like received buffers.

Unlike classical driver models found in mainstream operating systems and exposed to programmers in libraries such as BSD sockets, the system is closed: none of the actors can insert buffers into the system from the outside, such as by asking the operating system for memory. Actors cannot remove buffers either, though the network function is allowed to keep buffers indefinitely by using its “keep” transition to reorder buffers in the “processing” state. The reason for a closed system is performance: buffer allocation and deallocation are expensive. This is not only due to general software issues such as the overheads of keeping a “free list” of memory blocks, or the cost of asking the operating system for more memory, but also to an issue specific to drivers: memory pinning. The driver gives physical memory addresses to the network card when specifying buffer addresses. If the operating system were to change which physical page backs a virtual page used by the driver, the network card would not see the change and write to the wrong page. Thus, the operating system has to be informed of which memory is used for buffers and give it special treatment. While modern hardware can use I/O memory management units to allow devices to address virtual memory, there is a cost to changing I/O memory mappings.

2.2 Techniques for verifying systems code

How can one verify systems code in practice? Recall that systems code, unlike general-purpose application code, involves low-level operations such as manipulating hardware registers, and has strict performance requirements due to its key position in the software-hardware stack. Because of this, systems code may not always be written in a way that’s easiest to verify, or using abstractions at as high a level as one would like. In this section, we discuss the steps beyond testing.

Software engineers often resort to *testing*, i.e., running their code with specific inputs and checking that the outputs match expectations. However, this is not enough for verification, since bugs may lurk in the untested portions of the code. Even with metrics such as code coverage, measuring how much of the code has been tested, there may be bugs in the interactions between components, which could only be found by tests if every single path in the program has a corresponding test. This requires too many tests for a human, or even a team of humans, to write in reasonable time. As we will see later, this problem with the number of paths also exists for automated tools.

This does not mean testing is useless: it can be a good way to quickly check that basic operations work as expected, and that the interface of the code is easy to work with since tests are merely another kind of code that uses the code under test. However, humans often forget corner cases such as integer overflows, and are thus likely to forget them both in the implementation and in the tests.

2.2.1 Key enabler: SMT solvers

Satisfiability Modulo Theories, or "SMT" for short, is the key technique that underlies the verification work in this thesis. At a high level, it consists of asking the question "are there values for all the variables in a given Boolean formula such that the formula is true?". If so, a tool should answer "satisfiable" and provide a *model*, i.e., assignments of values for each variable such that the formula holds. Otherwise, a tool should answer "unsatisfiable", and ideally provide some form of proof that there exists no assignment that could make the formula hold, known as an "unsat core", though this is not always practical. The tool may also answer "unknown" after some user-configured timeout, i.e., it could neither prove nor disprove the satisfiability of the formula. Such a tool is called a *solver*.

SMT can be used to check whether an invalid program state is reachable by asking a solver whether the conditions necessary to reach the state are satisfiable. For instance, if a program contains `assert(x > 0)` within the body of an `if(x > 100)` statement, one can ask the solver whether $x > 100 \wedge x \leq 0$ is satisfiable, i.e., whether the asserted condition can be false given the branch condition. Since the query is trivially false, the assertion must always hold. This use generalizes to specifications: given a set of program states, one can check whether they comply with a specification by asking the solver, for each state, if there is an assignment of variables such that the program state's conditions hold and the specification does not hold. Conveniently, because failing verification corresponds to the "satisfiable" answer by the solver, such a failure always comes with a model that represents a counter-example. For instance, if an NF forwards packets without modifying them, but its specification states the MAC address of outgoing packets should never be all-zeroes, the counter-example given by the solver will be an incoming packet with an all-zero MAC address.

While SMT is not new, there has been considerable progress in recent years, such as the Z3 [24] and CVC5 [4] solvers which have easy-to-use APIs from common programming languages and can answer many queries in practice, even if they need heuristics which in theory cannot handle everything. Thus, it has become practical to throw large queries at a solver without worrying too much about the number of variables or size of the query, and expect an answer in seconds. Only complex models need careful thought to formulate a query that can be solved in reasonable time.

The T in SMT, which stands for Theories, refers to the kinds of variables and expressions within formulas. The most basic kind is Booleans, in which case SMT reduces to SAT, the satisfiability problem. While SAT is NP-complete [17], solvers can handle large problem instances in practice. A slightly higher-level theory is *bitvectors*, i.e., strings of Booleans typically used to represent machine integers such as `int` in C-like languages. Bitvectors are a *decidable* theory, i.e., a correct solver will always answer “satisfiable” or “unsatisfiable” in finite time no matter the formula it is given. Intuitively, because bitvectors are of finite size, a solver could try every possible value for every possible variable. This process of translating an SMT problem in the theory of bitvectors to a SAT problem which can only use Booleans is known as “bit blasting”. Some theories are *undecidable*, such as non-linear integer arithmetic, meaning that no solver can solve all formulas in finite time. This is one reason why “unknown” is an answer, the other being that “finite time” for decidable theories does not mean “reasonable time”. *Quantifiers*, such as “for all”, typically lead to undecidable theories, though subsets of quantified theories have been proven decidable [10; 42].

2.2.2 Theorem proving

Complex operations such as manipulating data structures are too hard for SMT solvers to solve alone, since they often involve abstract state that cannot be directly expressed in an existing theory and reasoning steps that are too large for a solver to find on its own. This is where *theorem provers* come in: they let users add annotations to guide the verification process, such as lemmas that prove the correspondence between concrete data structures and their abstract counterparts step-by-step. These annotations can be added automatically for specific cases a tool understands, such as automatically trying lemmas related to an abstract data structure when an instance of that structure is encountered. In fact, this mirrors what SMT solvers do internally: when they encounter universally quantified formulas, they will attempt to instantiate these formulas over concrete instances found elsewhere in the input formula. Theorem proving can thus be seen as hand-holding a solver to take bigger steps than it could take on its own in a given amount of time. The search space for complex problems is so large that solvers need some directions.

Some theorem provers use their own language, such as Dafny [69]. Designing a new language specifically for verification makes verification easier, but also less applicable since existing code must be rewritten. In contrast, some provers such as VeriFast [49] use existing programming languages such as C, with a superset of the language for annotations.

Consider the following example, taken from the VeriFast tutorial [50]:

```c
void account_set_balance(struct account* myAccount, int newBalance)
//@ requires account_balance(myAccount, _);
//@ ensures account_balance(myAccount, newBalance);
{
  myAccount->balance = newBalance;
}
```

This trivial function has explicit pre- and post-conditions, ensuring VeriFast only needs to verify its body once and can then summarize its effects through the conditions. While a simple function like this could also be inlined without increasing verification time, inlining a larger function would likely cause extra work for the solver. Conveniently, this function is also an example of a good software engineering practice: instead of exposing the `balance` field directly, the programmer provides a setter function, so that logic such as checking the validity of the balance can be centralized. This example only used C structures and integers, but VeriFast also supports more advanced abstract state that cannot be defined in C, such as inductive data types. One may for instance represent a C data structure as a VeriFast list, which is a classical “either nil, or a value and the rest of the list” data structure. One can write *lemmas* to prove facts about data structures:

```
// Universal quantification is preserved on a subset
// Proved by induction
lemma void forall_filter<t>(fixpoint (t,bool) q, fixpoint (t,bool) f, list<t> l)
requires true == forall(l, q);
ensures true == forall(filter(f, l), q);
{
  switch(l) {
    case nil:
    case cons(h, t):
      forall_filter(q, f, t);
  }
}
```

The most important recent milestone in systems verification is seL4 [60], an operating system kernel formally verified to be correct through theorem proving. seL4 is a microkernel, meaning it has fewer features than traditional kernels, but this again conveniently corresponds to a good engineering practice of avoiding to bloat the kernel with features it should ideally not be responsible for.

Human assistance in theorem proving is not an all-or-nothing requirement: the prover will attempt to infer basic facts on its own, and can be used alongside other techniques for code that does not require human assistance. Vigor [119], which directly inspired some of the work in this thesis, uses theorem proving for complex data structures and automated reasoning for the simpler code that uses these data structures. As another example of a tradeoff, Amazon uses what they call “lightweight” formal methods to prove some facts about their code, but not total correctness [8].

2.2.3 Symbolic execution

A *symbolic execution engine* executes code with *symbols* as inputs instead of concrete values. Whenever it encounters a branch on a symbolic condition, it explores both alternatives, remembering the choices it made in a *path constraint*. This leads to a set of *paths*, which are sequences of choices that each represent one possible program execution. For instance, instead of executing an "absolute value" operation on -5 and obtaining 5, a symbolic execution engine will execute it on α and obtain two paths: one with path constraint $\alpha \geq 0$ and result α, and one with path constraint $\alpha < 0$ and result $-\alpha$.

Some code has too many paths to exhaustively explore in reasonable time. For instance, consider looking for a value in an array. The value could be in the first position, or the second, or the third, and so on, until the array length. The number of paths is limited by the array length, which could be, e.g., 2^{32}. If the code which looked up the value then looks up another value, each path resulting from the first lookup will lead to new paths for the second lookup, squaring the total number of paths. This problem is known as *path explosion*. While the number of paths can sometimes be reduced by merging related paths at the expense of producing more complex constraints [65], the paths to be merged must still be partially explored, which does not solve path explosion.

If the number of paths in a program is "reasonably" small, a symbolic execution engine can exhaustively enumerate them and verify the program by verifying that each path satisfies the program's specification. One way to achieve this is writing code with few paths, such as the Hyperkernel [83], a simplified operating systems kernel written without loops. The Hyperkernel's system call interface instead offloads loops to applications, ensuring the kernel itself can be symbolically executed quickly.

Symbolic execution is often used for bug finding instead of verification because of path explosion. For instance, SAGE [9] found a large number of bugs in Windows by combining symbolic execution with fuzzing. This is useful, but fundamentally cannot lead to a proof of correctness. Because this thesis focuses on verification, we need to extend symbolic execution to bypass path explosion.

Widely used symbolic execution engines such as KLEE [14], S2E [16], and `angr` [102] were designed primarily to find bugs. That is, their goal is to find one or more paths that trigger a bug as quickly as possible, typically an illegal operation such as an out-of-bounds memory access. Bug-finding does not require exhaustiveness, since finding paths that trigger bugs is enough, and in fact enumerating paths that do *not* trigger bugs can be seen as a waste of time. An important part of bug-finding engines, in terms of complexity, is the algorithm choosing which path to explore next among a set of candidates. However, existing engines can be repurposed to be exhaustive as part of network function verification such as Vigor [119] and Dobrescu and Argyraki [26]. In this case, exhaustiveness is necessary, and the order in which paths are explored does not matter for correctness. It may matter for performance since exploring similar paths may provide opportunities for caching. These are part of a recent line of work in using exhaustive symbolic execution to verify systems code, which also includes Gravel [120] and Serval [81].

One source of path explosion we do not consider in this thesis is concurrency: in its most general form, given N threads, each program step causes N new paths since it could be taken by any thread, and each interleaving might have different effects. The techniques required to deal with such interleavings, such as dynamic partial order reduction [35], are different from those used and proposed in this thesis, and we leave the verification of concurrent systems to future work.

2.3 Automation

2.3.1 Challenges

The two key challenges this thesis makes progress on with regards to automation are data structures and loops. While automatically handling all possible data structures and loops is impossible, we focus on common kinds that developers use in real-world code.

Data Structures

There is a gap between the variety of data structures that exist in the code and the restricted set of abstractions that any given tool can reason about, as we illustrate in Figure 2.4. For instance, our firewall may use a "least recently used" data structure to keep track of which packet flows should be expired due to a lack of activity, with operations such as "add a new item" and "remove and return the oldest item". The firewall may also combine the use of this data structure with other structures, such as a map tracking per-flow statistics or a configurable set of ports that are open to the external world at all times. However, it is not feasible for an automated tool to understand the semantics of every possible data structure operation purely based on the code, let alone map these semantics to those of abstract structures used in specifications.

Previous work proposed two ways to bridge the gap between implementation and specification: Vigor [119] requires the use of specific data structures in both implementation and specification, while Gravel [120] requires the use of specific data structures that it knows how

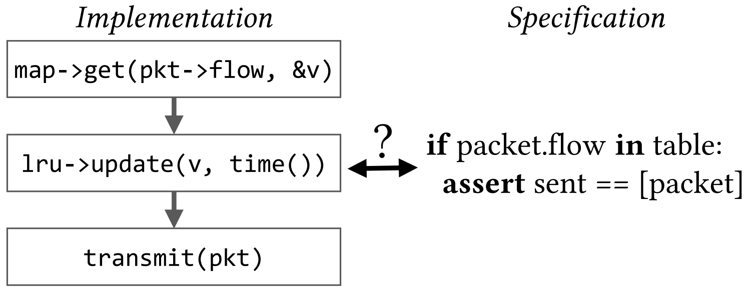


Figure 2.4: There is a gap between the abstractions used in the implementation and the specification.

to model in terms of high-level operations that can be used in specifications.

Vigor imposes two constraints to verify our firewall. First, the firewall must be written using data structures that Vigor knows about, either by modifying the firewall's code to only use Vigor data structures or by modifying Vigor itself to handle new data structures, including proof annotations for invariants that can hold across these data structures such as "all flows in the LRU are also in the map". Second, the firewall specification must be written in terms of the same data structures used in the implementation. These two constraints limit both developers and operators. Developers must restrict themselves to specific data structures or learn verification techniques to add new ones, and operators must learn the semantics of the data structures used in an implementation to understand its specification.

Gravel removes the second limitation: it translates data structure operations to a small set of high-level operations for use in specifications, thus operators do not need to learn all data structure semantics. However, the restriction on developers remains. Developers must either modify the firewall's code to only use existing Gravel data structures or modify Gravel itself to handle new data structures.

Loops

The reason traditional symbolic execution does not enable the automated verification of real-world code is that even simple-looking code fragments such as array-based data structures cause trouble. Consider the following C function:

```c
bool contains(int value, int* arr, size_t len) {
  for (size_t n = 0; n < len; n++) {
    if (arr[n] == value) return true;
  }
  return false;
}
```

Intuitively, this function has three paths: either `value` is found in `arr`, or it is not, or there is an out-of-bounds array access. But the loop it contains causes path explosion, i.e., too many paths to explore in reasonable time.

Loops are a hurdle because symbolic execution unrolls them to a sequence of branches, each of which creates new paths. In the example above, there is one path that finds `value` at `arr[0]`, one that does not find it there and ends because `len` is 0, one that finds it at `arr[1]`, and so on. If `len` is not constrained, there are twice as many paths as there are `size_t` values, i.e., 2^{64} on a 64-bit machine.

This loop could be summarized with a contract describing what it does using quantifiers, i.e., a loop invariant, but just as handling one data structure at a time does not scale, handling one loop at a time does not scale either. We need an approach that can infer contracts for entire classes of loops automatically.

Concurrency

While this is not a focus of this thesis, concurrency is a common challenge for symbolic execution. Just as a loop could run any number of times, a thread could execute first, or second, or third, and so on for each instruction.

One reason why we are able to ignore concurrency in this thesis is that shared-nothing parallelism, in which separate instances of a network function run without even being aware of each other, is good enough for many cases [116]. This is thanks to hardware features: network cards can deliver traffic from different parts of a network to different CPU cores, thus any network function that can be cleanly partitioned does not need explicit reasoning about concurrency.

2.3.2 Real-world uptake

Formal verification is not merely an academic exercise: it has been successfully used to verify the full or partial correctness of useful systems. seL4 [60] has been used as a basis for verified systems, and is growing both in terms of kinds of proven properties and number of features. Amazon uses “lightweight” formal methods to prove partial correctness of core infrastructure [8], and Microsoft incorporated symbolic execution in their bug-finding tools [9]. Model-checking, such as TLA+ [66], is also in use in industry [85] to verify the correctness of protocols, such as the property that a distributed system will make progress and will not get stuck in an incorrect state. However, TLA+ requires manual effort in addition to writing code since developers must write models in the TLA+ language.

There are two current obstacles to real-world uptake: first, automated verification is often inapplicable to real-world code due to the limitations of current techniques and tools; and second, verification is not always fast enough to be run as part of a developer’s everyday workflow, or even daily continuous integration.

2.4 Summary

In summary, verification techniques currently require too much effort from developers to be widely applicable to software network functions. Developers must provide models for every data structure they use and avoid using even simple loops in their code. This is because symbolic execution, the main automated technique in use for verifying network functions, cannot effectively deal with the large amount of different paths created by data structures and the loops within them, even though most of these paths lead to similar behavior.

In this thesis, we address this problem in two ways: by increasing the reach of automated verification, so that developers need less effort and expertise to use it, and by showing that some of the effort developers spend on adapting their code has benefits outside of verifiability.

Automating systems-code verification Part I

3 Ghost maps to model data structures

This chapter is based on "Automated Verification of Network Function Binaries, S. Pirelli, A. Valentukonytė, K. Argyraki, G. Candea, NSDI 2022". The vast majority of the work was done by the author of this thesis.

To automatically verify network functions, which are a form of systems code since they deal with the lowest levels of the stack and have strict performance requirements, one must deal with the parts of network functions that are not automatically verifiable: data structures. For instance, a hash map is too complex for symbolic execution alone, especially if it uses internal optimizations that add even more complexity. However, this hash map can be written once and reused many times, thus a verification tool is primarily concerned with network functions that reuse the hash map and not with the hash map itself.

We propose the use of maps as a "universal" data structure to specify network functions and data structures. We define the abstract state of both network function specifications and data structure contracts in terms of maps from keys to values using a programming language such as Python. We call these maps *ghost maps* by analogy to "ghost variables" that exist only in proofs, not in implementations. Contracts can be written even for data structures that cannot be implemented in terms of maps, by using "for all" quantifiers to describe an operation's effects on the data structure in a declarative manner without describing its implementation. Verification is thus proving that the network function manipulates concrete data structures in a way that conforms to the manipulation of abstract maps in the specification, using contracts to match concrete operations with abstract ones.

Ghost maps enable efficient and automated invariant inference. Inferring invariants ensures automated analysis does not explore impossible program states, which would cause spurious verification failures. The sweeping simplification of maps enables a tool to reason about invariants across any data structures instead of limiting itself to specific ones. It also leads to simpler tools, as there is only one kind of data structure to reason about, the map. Such a tool uses contracts to translate data structure operations into map operations and infers invariants on the resulting maps. If the map operations are familiar to developers, writing contracts is similar to writing documentation or unit tests.

In this section, we present the design of ghost maps and their implementation in KLINT, a new tool we built to prototype this technique.

3.1 Insight: Common NF data structures can be modeled with maps

Fundamentally, tools that handle a specific set of data structures do not scale. Adding support for a new data structure requires not only encoding the data structure's operations, but also its interactions with other data structures, such as invariants that can exist across structures. Even if a tool is limited to invariants across two data structures, adding the Nth structure requires adding N kinds of invariants, one for each data structure including the new one.

We introduce a level of conceptual indirection: we express the semantics of both data structures and specifications in terms of one data structure, the *map*. Operations on data structures such as arrays, hash tables, longest-prefix-match tables, and port allocators can be defined in terms of map operations, regardless of how they are implemented. Verification tools can use contracts to translate any data structure operation into map operations, which become the only kind of operation the tool needs to reason about for invariant inference and verification.

We refer to such maps as *ghost maps*, by analogy to "ghost variables" which are variables only used in proofs. We present an example of contracts for a "least recently used" data structure in Listing 1. Using this contract, a verification tool can translate the LRU semantics into maps, and thus does not need special knowledge of what an LRU data structure is, only knowledge of maps. Importantly, contracts can be declarative, not just imperative. Ghost maps can define even data structures that cannot be implemented using maps thanks to the "for all" quantifier. `LRU_expire`'s contract only describes *what* it does, not *how*: the returned value is the oldest, but the contract does not need to explain how this value is found.

This sweeping simplification also makes invariant inference easier since there is only one kind of data structure, as we show later.

```
# struct LRU;
  LRU = namedtuple('LRU', ['items'])
# void LRU_add(struct LRU* lru,
#              void* item, int age);
  assert item not in lru.items
  lru.items[item] = age
# void* LRU_expire(struct LRU* lru);
  age = lru.items[result]
  assume(lru.items.forall(lambda k, v: v <= age))
  lru.items.remove(result)
```

Listing 1: Contracts written by a developer using maps, in Python, for a "least recently used" data structure, in C. `result` is the return value from `LRU_expire`.

3.2 Design: Ghost maps

We introduce ghost maps as a "vocabulary" in contracts to describe the semantics of data structures and network functions to both verification tools and human readers. Ghost maps only exist within contracts, not implementations.

$$\begin{aligned}
\mathit{length}(M) &\rightarrow \mathit{Int} \\
\mathit{get}(M, K) &\rightarrow V \mid \mathit{None} \\
\mathit{set}(M, K, V) &\rightarrow M' \\
\mathit{remove}(M, K) &\rightarrow M' \\
\mathit{forall}\big(M, \lambda(k, v) \rightarrow \mathit{Bool}\big) &\rightarrow \mathit{Bool}
\end{aligned}$$

Listing 2: Ghost map operations. M and M' are maps; K, V are keys and values. *Int* denotes bitvector-based integers, and *Bool* Booleans. *None* is the lack of value.

3.2.1 Expressivity, decidability, and completeness

We propose the ghost map abstraction in Listing 2 which is expressive enough to define data structures and network functions, while still enabling a tool to reason in a decidable, sound, and "complete enough" manner. We describe each of these properties next.

Ghost maps are *expressive* enough to abstract the data structures we care about. Simpler abstractions require too much detail in contracts to be practical for either humans or tools. For instance, representing a hash table as a sequence of 0s and 1s is possible but impractical. We are concerned with data structures used in network functions, such as hash tables and port allocators, thus we limit our vocabulary to what they need, not to all possible code.

However, expressiveness is at odds with *decidability*. Symbolic execution engines use a solver to tell whether a logical formula is *satisfiable*, i.e., whether there exists an assignment of variables such that the formula holds. For instance, if "the firewall's variables, given the firewall's constraints, violate its specification" is satisfiable, then the assignment of variables is a counter-example to the firewall's correctness. Logical formulas are written using *theories*, which are the "vocabulary" of solvers. Some theories are *decidable*, meaning that a correct solver will always return a correct answer. Some are not, meaning that the answer may be "unknown" instead of yes or no. Even with decidable theories, "unknown" may be returned if the solver is given a timeout and cannot find an answer in time.

Verification tools must be *sound* and as *complete* as possible. A tool is sound if it verifies *only* correct programs. A tool is complete if it verifies *all* correct programs. Verification of general-purpose programming languages is incomplete [64], thus the goal is to verify "interesting" correct programs, i.e., those that humans actually write, even if some contrived theoretical examples cannot be verified.

Ghost maps are an intermediate step between quantifier-free bit vectors, which are decidable due to their finite size but not expressive enough, and arrays with universal quantifiers, which are more than expressive enough but undecidable.

3.2.2 Representing ghost maps

To remain as decidable as the theory of bit vectors while offering more expressivity, we present a translation of ghost maps to bit vectors in the context of symbolic execution. This translation includes a decidable encoding of ghost maps' "for all" quantifier *without* using universal quantifiers, enabling non-imperative contracts for data structures that cannot be implemented with maps. Ghost maps can be more expressive despite being translated to bit vectors because the symbolic execution engine internally uses data structures, such as lists, to build the logical formulas it sends to the solver.

We expect the code to manipulate maps of large size, but to only interact with a small number of items in any given map. This is true of network functions, which by nature only perform a small number of operations for each packet to remain within their performance budget.

Tracking *known* and *unknown* items separately is our key insight to handle maps of arbitrary size. That is, none of the map operations require "forking" the current path, and the engine handles map operations in a time linear in the number of known items, not total items.

Thus, instead of keeping track of every item in every map, the engine only needs to track the *known* items that are explicitly used in one iteration of the network function's packet-processing loop. Other items are "summarized" into a single pseudo-item that tracks their constraints, such as "all unknown values are non-zero".

This scheme naturally enables ghost map lengths to be symbolic, since their actual size does not matter to exhaustive symbolic execution performance as much as the number of known items. A verification tool can thus represent maps whose length is determined by configuration parameters using a symbolic configuration, instead of verifying only one specific configuration as is for instance done in Vigor [119]. The tool can thus catch all bugs that only occur in specific configurations, such as the maximum capacity of the firewall's flow table being zero, without requiring developers to think of which configurations to try.

Counter-intuitively, the engine must mutate its internal representation of maps during read-only operations on maps. Known items must include those that have been retrieved from the map, even if they have not explicitly been set before. For instance, consider the following:

$$get(M, K_1) \rightarrow V_1$$

$$get(M, K_2) \rightarrow V_2$$

If $K_1 = K_2$, then $V_1 = V_2$ by definition. But the engine must remember the first *get* in some way in order to guarantee the implication, and it cannot store high-level map operations in the path constraint, thus it must modify its internal representation of M. From an outside perspective, M has not changed, but internally the engine must remember this *get*.

We informally describe each operation first, then add additional details to handle subtleties, then provide a formal algorithm for the core *get* operation.

The engine tracks each map's length explicitly.

Known items are triples: (key, value, presence bit). If the presence bit is *false*, the value is ignored and the key is considered absent from the map. *None* only exists conceptually; the theory of bit vectors cannot represent it.

Known items may be redundant, but their values and presence bits must match if their keys match. This is because their keys may be symbolic, thus the engine cannot know for sure whether two items have equal keys.

Unknown items are represented by an *invariant* that they all satisfy. The unknown items invariant only concerns unknown items. Known items need not satisfy it. For instance, a map may have as invariant "all unknown values are non-zero" and two known items, $(K_1, 0, \mathit{true}), (K_2, 1, \alpha)$. The first known item is definitely present, whereas the second may or may not be present depending on the value of α.

The unknown items invariant is represented as a formula on a special *unknown item*, unique to each map. For instance, the non-zero invariant example is represented as $UP_M \Rightarrow UV_M \neq 0$, where UP_M and UV_M are the presence bit and value of M's unknown item. Including the presence bit in the invariant allows it to be constrained for cases such as arrays. For instance, a zero-based array of length L described using a ghost map A would have $UP_A = (0 \leq UK_A < L)$ as part of its unknown items invariant, indicating that keys are in A if and only if they are between 0 and L, matching the semantics of array indexing in languages such as C.

3.2.3 Translating ghost map operations

We use $ITE(c, t, f)$ to denote "if c then t else f", and *fresh* to mean a symbol that was not used before, i.e., that is not constrained in any way. "Applying" the unknown invariant of a map to an item means substituting the map's unknown item for the item.

$\boldsymbol{length}(M)$: Return the map's length, which the engine tracks explicitly.

$\boldsymbol{get}(M, K)$: Create a fresh tuple (V, P). Add (K, V, P) to the map's known items. Add constraints to the current path to encode that, within the map, (1) if K matches a known item then so do V and P, (2) if K does not match any known item then the unknown items invariant applies on (K, V, P), and (3) the number of unique known items cannot exceed its length. Return (V, P), which encodes "if P then V else *None*".

$\boldsymbol{set}(M, K, V)$: Let $(\_, P) = get(M, K)$. Return a new map whose length is $length(M) + ITE(P, 0, 1)$, whose known items are $\big(K', ITE(K = K', V, V'), P' \vee (K = K')\big)$ for each known item (K', V', P') in M, plus $(K, V, true)$, and whose unknown items invariant is the same. That is, (1) the length only grows if K was not in M, and (2) known items must match if they are redundant.

$\boldsymbol{remove}(M, K)$: The opposite of *set*, i.e., the length change is $ITE(P, -1, 0)$, the new known item is $(K, V, false)$, where V is any arbitrary value, and the known items are changed to $\big(K', ITE(K = K', V, V'), P' \wedge (K \neq K')\big)$.

$\boldsymbol{forall}(M, F)$: The result is *true* iff $P \Rightarrow F(K, V)$ for each known item (K, V, P) in the map and $UP_M \Rightarrow F(UK_M, UV_M)$ for the unknown item of the map. That is, the result indicates whether the predicate holds on known items and on unknown items. Add to the map's invariant that if the result of this operation is *true*, then the predicate holds on unknown items if their presence bit is *true*, ensuring that even if the result is not yet known, it will be consistently applied in the future. If the result is false, future items are not constrained, which is sound but not complete.

***Layers* are necessary to handle dependencies** that arise when code uses multiple versions of a map at once. Consider:

$$set(M, K, V) \rightarrow M'$$

$$get(M, K') \rightarrow V'$$

$$get(M', K') \rightarrow V''$$

If $K \neq K'$, then V' and V'' must be the same, since only the value associated with K is affected by the *set*. However, the representation described earlier cannot guarantee this, because the known items of M and M' are independent.

To handle this issue, *set* and *remove* return layers instead of entirely new maps, as the example in Figure 3.1 illustrates. That is, they return a map which includes the newly added or removed item among its known items, but which links to the previous map for the other known items, transforming them when necessary, instead of evaluating the new known items at creation time. Items and invariants created by *get* and *forall* are always added to the bottom-most layer. Map layers also share the unknown items invariant, and the associated "unknown item", of their base map.

Thus, in our example above, the known items "seen" by the second *get* operation include the result of the first one, and thus the second *get* will return a result that lets a tool prove $K \neq K' \Rightarrow V' = V''$.

Invariant recursion must be explicitly handled to avoid infinite recursion when two maps' invariants refer to each other. For instance, a bi-directional map may be represented as two maps whose contents are inverses: for each key-value pair (K, V) in map M_1, there is a pair (V, K) in map M_2, and vice versa. The invariants are:

$$forall\big(M_1, \lambda(k, v).\ get(M_2, v) = k\big)$$

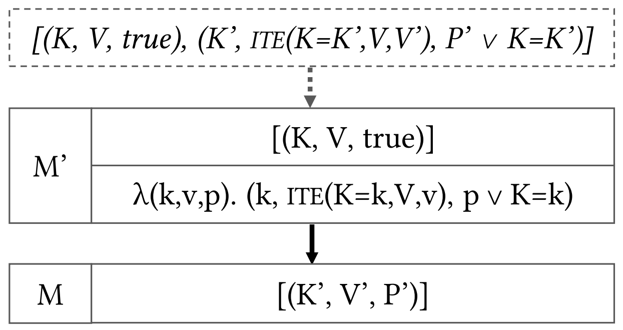

[(K, V, true), (K', ITE(K=K',V,V'), P' ∨ K=K')]
M'
[(K, V, true)]
λ(k,v,p). (k, ITE(K=k,V,v), p ∨ K=k)
M
[(K', V', P')]

Figure 3.1: Example of a *set* layer M' on top of a map M, and the resulting known items of M'.

$$\mathit{forall}\big(M_2, \lambda(k,v).\ \mathit{get}(M_1, v) = k\big)$$

Consider what would happen using the representation described earlier when the engine handles *get*(M_1, K) for some K. As part of adding the invariant on the newly-known item of M_1 to the path constraint, the engine will call *get*(M_2, V) with the fresh V from the original *get*. As part of adding the invariant on the newly-known item of M_2 to the path constraint, the engine will call *get*(M_1, V') with the fresh V' from the second *get*. The engine then calls *get* on M_2, and so on, leading to infinite recursion.

Solving this issue requires the engine to recognize that in the third *get* call, the V' argument is equal to K, which it knows from the first *get* call. The result should thus be the existing V, not some fresh V''.

The engine thus tracks a *condition* and a *value hint* during ghost map operations, which are set when handling invariants and used to stop recursion when handling *get*.

When handling a map invariant of the form *get*(...) = ..., the engine adds to the condition the presence bit given as argument to the invariant and sets the value hint to the value expected by the invariant. These changes are reverted when the engine is finished handling the invariant.

get(M, K) needs two changes at the start: First, if K cannot be different from an existing item's key assuming the condition holds, return that item's value and presence bit. Second, after creating the fresh V, if there is a condition, add "the condition implies $V = value\_hint$" to the path constraint.

Applying this logic to our example solves the issue. When handling *get*(M_2, V), the value hint is K and the condition is P, both from the newly-known item of M_1. When M_2's invariant calls *get*$\big(M_1, V'\big)$, the path constraint contains $P \Rightarrow V' = K$, thus the *get* on M_1 will start by checking whether its known item (K, V, P)'s key is equal to V' assuming P, which it is, and return (V, P), ending the recursion.

This strategy avoids recursion in common cases such as our example of maps with reciprocal keys and values, but it is not complete, as the engine may recurse infinitely. For instance, "M_1 has $min(K-1,0)$ for all K in M_2, and M_2 has $min(K-1,0)$ for all K in M_1" could hold, but will lead to infinite recursion in the engine given our implementation.

We present the final *get* algorithm, which is the core of our ghost map technique, in Algorithm 1. Our technique is decidable and expressive but not complete, unlike prior work that focused on completeness at the expense of expressiveness [10; 25]. Our technique enables a symbolic execution engine to translate ghost map operations into formulas on quantifier-free bit vectors. This enables the engine to bypass the path explosion caused by data structure code by executing the code's contract instead.

```
function KnownSize(M)
    $result = 0$, $known = \emptyset$
    for $(k, v, p)$ in $M$'s known items do
        $result \mathrel{+}= ITE(k \notin known \wedge p, 1, 0)$
        $known \mathrel{+}= k$
    return $result$
function Get(M, K)
    for $(k, v, p)$ in $M$'s known items do
        if $unsatisfiable(condition \wedge K \neq k)$ then
            return $(v, p)$
    if $unsatisfiable(condition \wedge K \neq UK_M)$ then
        return $(UV_M, UP_M)$
    Let $(V, P)$ be a fresh value and presence bit
    if $condition$ is set then
        Add $condition \Rightarrow V = value\_hint$ to the PC
    Let $U = \bigwedge(K \neq K')$ for each known key $K'$ in $M$
    Add $(K, V, P)$ to $M$'s known items
    for $(k, v, p)$ in $M$'s items do
        Add $K = k \Rightarrow \big((V = v) \wedge (P = p)\big)$ to the PC
    Add $U \Rightarrow invariant_M(K, V, P)$ to the PC
    Add $KnownSize(M) \leq length(M)$ to the PC
    return $(V, P)$
```

Algorithm 1: The *get* operation on ghost maps. *PC* is the Path Constraint. UK_M, UV_M, and UP_M are the triple forming map M's unknown item. The *condition* and *value_hint* are those required to handle invariant recursion.

3.2.4 Invariant Inference

Handling data structures by translating them to ghost maps is not sufficient for automated verification. Developers use data structures in well-defined patterns, named *invariants*, but these patterns are not always explicit in the code.

Verification tools must *infer* such invariants to avoid failures. When executing a contract instead of an implementation, the tool must have enough information to prove the contract's precondition. Previous tools bypassed this problem by special-casing data structures and their invariants.

The ghost maps representation we propose enables tools to infer invariants without special-casing templates.

Consider these motivating examples in pseudo-code:

```
if packet.flow in table:
  statistics.increment(packet.flow)
```

If `increment`'s contract requires the item to be in `statistics`, symbolic execution will fail if it cannot prove this fact.

```
device = destinations.get(packet.flow)
transmit_packet(packet, device)
```

If `transmit_packet`'s contract requires the device to exist, symbolic execution will fail if it cannot prove this fact.

```
if not items.full:
  items.add(x)
  metadata.add(x, y)
```

If `add`'s contract requires free space, symbolic execution will fail if it cannot prove `not metadata.full` in the second `add` call.

All three examples could be bugs depending on how the data structures they deal with are updated. If the first example is in code that always adds `packet.flow` to both `statistics` and `table`, the code is valid. If the second example always puts a device known to be valid in `devices`, such as the incoming device of a packet, the code is valid. If the third example is the only occurrence of `add` in the code and both structures have the same length, the code is valid.

To infer invariants, our algorithm starts from the strongest possible ones then iterates by relaxing them as needed until it finds a fixed point, as we illustrate in Algorithm 2. The starting point is the program states resulting from symbolically executing the network function's initialization code. At this point, the invariants are the strongest possible ones: "all maps will always be exactly as they are after initialization".

The initial invariants are unlikely to hold on the packet processing code, unless the code does nothing. After symbolically executing the packet processing code for one iteration of the network function's infinite packet-processing loop, the engine *relaxes* the invariants to match the program states that result from the iteration. For instance, the initial invariant "the firewall's `flow_table` is always empty" could be relaxed into "the `flow_table` may have items, and its length is always the same as the `statistics`". The engine then symbolically executes a packet-processing iteration again using these new invariants, which may lead the engine to explore new paths in the code such as a path in which the packet's flow is found in

```
function FindFixedPoint(code)
    $invs = InitialInvariants(code)$
    $states = InitialStates(code)$
    do
        $invs = Relax(invs, states)$
        $states = SymbolicallyExecute(states, invs)$
    while $invs$ do not hold on $states$
    return $invs$
```

Algorithm 2: Core of the invariant inference algorithm.

the `flow_table`, previously infeasible as the table was assumed to be empty. The engine then relaxes the invariants again, and executes an iteration with these new invariants, until the invariants no longer need relaxing after an iteration. By definition, the final relaxation yields invariants that hold on the initial state as well, thus the result is a correct set of invariants. The algorithm is guaranteed to converge since the set of invariants can only shrink. It may converge towards the empty set of invariants because the code has no invariants or because the engine could not infer any. The goal is not to find some "ideal" set of perfect invariants, only enough invariants to be able to symbolically execute and verify the code. If some properties happen to hold but are not necessary, inference need not find them.

The key to inferring useful invariants is to use ghost maps' *known* items to form invariant candidates, and then check whether these candidates hold using the *unknown* items. Thus, instead of using low-level invariant templates such as "the value is nonzero" as in Houdini [36], the tool can find invariants based on the constraints that hold on known items, using high-level templates such as "one map's values may be another map's keys" and "one map's length may be the same as another map's length".

Besides finding invariants among map items and lengths, tools can also find invariants *across* maps. Such invariants are of the form "if M_1 associates key K with value V, then M_2 has some key and possibly value related to K, V". For instance, in the first motivating example, finding the invariant "all keys of `table` are also keys in `statistics`" enables the tool to prove that the code uses `increment` correctly. Inferring such invariants requires finding functions F_K, F_V, F_P for two maps M_1, M_2 such that $((get(M_1, K) = V) \wedge F_P(K, V)) \Rightarrow (get\big(M_2, F_K(K, V)\big) = F_V(K, V))$, or alternatively find only F_K, F_P and merely infer that M_2 contains $F_K(K, V)$ without inferring the associated value. Candidates for the functions are found using the known items and confirmed using the unknown items invariant.

In summary, the representation we propose for ghost maps also enables an elegant invariant inference algorithm based on finding candidate invariants and relaxing them as necessary. Representing known items explicitly lets a verification tool find useful candidates for invariants, instead of limiting itself to special-cased low-level templates.

3.3 Implementation

We implement our technique in KLINT, a tool which uses the `angr` [102] symbolic execution engine and the Z3 [24] solver.

KLINT takes a network function binary and a Python specification as inputs and proves that the binary satisfies the specification or produces a counter-example. KLINT can also be used without a specification to check that no path in the code crashes or accesses memory out of bounds.

KLINT first identifies all environment interactions in the binary, which correspond to data structure operations or networking operations, using the symbols that the binary must export since it dynamically links the environment code. Then, KLINT maps these operations to their contracts, provided by the developers of the data structures and of the networking environment. Data structure contracts are written in terms of ghost maps, while the handful of fundamental networking operations such as transmitting a packet have manually written contracts for KLINT. When the binary calls its environment, KLINT symbolically executes the corresponding contract instead of the implementation, inferring types and control flow information: KLINT knows the types of parameters to environment interactions, and extracts control flow in the form of path constraints. This enables KLINT to understand the semantics of the binary under verification in terms of its environment.

KLINT symbolically executes the network function's initialization code, then infers invariants that hold in all packet-processing paths by symbolically executing the packet-processing code until it finds a fixed point. KLINT then checks whether all packet-processing paths satisfy the specification, which indicates whether the network function binary as a whole does.

Specifications are Python programs that use ghost maps, which KLINT interprets using peer symbolic execution [13] on each state resulting from symbolic execution of the binary. The binary may abstractly manipulate more than one ghost map, since it may use multiple data structures and individual data structures may be modeled by contracts as multiple maps. Thus, when the specification refers to a map, KLINT must infer which of the maps it is, using heuristics to try likely candidates first, and only fail verification if the specification is violated for all possibilities. For instance, a firewall can track outgoing flows and maintain statistics per IP address. KLINT must infer that the flow table mentioned in a specification is the abstract form of the former data structure, not the latter.

Developers must comply with good programming practices such as state separation if they want verification to succeed, and KLINT enforces this during verification. If developers do not separate data structure code from the rest of the network function, KLINT will encounter too many paths and fail once it has executed a configurable instruction threshold. Developers already follow even stricter practices to write BPF programs due to the strict Linux verifier [20].

KLINT models heap memory with ghost maps, treating memory as just another kind of data structure and including it in invariant inference. Developers use a standard `calloc`-like interface to allocate memory. During symbolic execution, memory allocations return *symbolic pointers*, ensuring that KLINT will explore all paths even if some paths depend on the value of pointers. This is not the case in a tool such as Vigor [119], which uses concrete pointers.

To ensure that developers cannot "hide" memory from KLINT, all memory outside of the stack and the maps-modeled heap is read-only after initialization. This is not a limitation on sensible developers, who allocate on the heap through the environment and cleanly separate mutable state.

Our memory model is similar to Memsight [18] and KLEE's segmented memory [55], but it requires almost no effort to implement thanks to the flexibility of ghost maps.

KLINT can do *full-stack* verification, verifying the entire software stack, including the network driver and most of a minimal operating system.

To verify the network driver, KLINT matches hardware accesses to the *actions* they correspond to, using a hardware description of the network card written in a domain-specific language and based on publicly available data. KLINT intercepts reads and writes to network card registers, which usually go through port- or memory-mapped I/O. When the network driver writes a value to a register, KLINT reverse-engineers what the driver might be doing by finding all actions that could match the write, and checking which ones are feasible in the current hardware state. For instance, if the driver sets the "enable promiscuous mode" flag in the network card, KLINT looks up the corresponding action and checks its precondition, which states packet reception must be disabled. If the precondition does not hold, or if no action matches the write performed by hardware, KLINT aborts verification. Actions can also have postconditions describing what happens to the hardware as the result of an action, such as a self-clearing bit in a register.

The only environment operations for which KLINT uses contracts instead of implementations during full-stack verification are the data structures and the memory allocator due to their complexity. We verified that they obey their contracts using machine-checked proofs.

Full-stack binary verification does impose one constraint: while the network function and its network driver can be compiled together, the environment has to be compiled separately and then linked together without link-time optimizations, ensuring the symbols corresponding to environment operations exist and can be given as an input to KLINT.

Our trusted computing base is made up of KLINT itself, including `angr` and Z3; the bootloader; the hardware; and the VeriFast [49] theorem prover we use to verify the memory allocator and data structures. Since VeriFast verifies source code, we trust the C compiler for the data structures and memory allocator, but this is not a fundamental limitation as one could use a theorem prover for assembly code instead, and thus not need to trust the compiler.

3.4 Evaluation: Completeness

We evaluate KLINT using the binaries, without debug symbols, of 6 network functions: an Ethernet bridge with a spanning tree protocol, a firewall, an implementation of Google's Maglev [28], a network address translator, a traffic policer, and an IPv4 router with longest prefix matching. The first five are based on publicly available code from the Vigor [119] project, which verified source code.

Kinds of network functions

KLINT is applicable to real-world network functions: it does not require additional inputs from developers, it enables operators to verify the entire stack of network functions, its network function model is a superset of the widely used BPF, and it enables developers to use any programming language including those not considered mature enough to be trusted.

As we show in Table 3.1, KLINT operates on binaries, can handle any data structure for which map-based contracts are provided, does not require intermediate inputs, does not limit developers' use of pointer arithmetic beyond memory safety, and precisely tracks the contents of data structures and packets enabling functional correctness proofs.

Previous work falls into two categories. Vigor and Gravel prove functional correctness but require a typed intermediate language, extra inputs, and specific data structures. Prevail [37] handles BPF bytecode and maps, which BPF developers must use anyway, and can even handle unbounded loops, but can only prove memory safety and crash freedom. Prevail can be viewed as a superset of the Linux BPF verifier. KLINT provides the best of both worlds: it requires neither a typed intermediate representation nor extra inputs, and it does not limit data structures, while also enabling proofs of functional correctness. As an example of unnecessary inputs, we were able to remove around 3000 lines of proofs for invariants when porting Vigor network functions to KLINT.

Operators can verify the entire software stack, and thus do not need to trust software such as network drivers, using KLINT's full-stack verification. Developers can thus tune the drivers for performance by removing features they do not need, or even rewrite their own driver to suit

	Lang.	**Data structs**	**Extra inputs**	**Pntr. arith.**	**Loops**	**Precise**
Gravel [120]	LLVM	C++ STL	Interm. specs	Limited	Bounded	Yes
Prevail [37]	BPF	BPF maps	No	If safe	Unbounded	No
Vigor [119]	LLVM	Custom	Per-NF models	No	Bounded	Yes
KLINT	x86_64	Any w/ contracts	No	If safe	Bounded	Yes

Table 3.1: Comparison of KLINT with previous network function verification efforts.

it to a specific usage pattern. PacketMill [33] showed that such transformations can be done with developer hints to increase network function performance.

KLINT can be used to verify network functions in containers, i.e., statically linked with a Linux implementation of the environment abstraction and running within Docker [74]. Containers are a convenient way to deploy and manage programs and have been proposed as a deployment model for network functions [112]. KLINT can verify such network functions in the same way it verifies full-stack ones, although the trusted code base is larger since operators need to trust the container runtime as well as the Linux environment implementation running inside the container.

Our network function model is a superset of BPF. The Linux kernel verifies that BPF programs are memory safe and have no unbounded loops. KLINT verifies functional correctness, though it also requires a lack of unbounded loops. BPF requires developers to use a fixed set of data structures, mostly maps. KLINT enables developers to use any data structure that has contracts based on ghost maps.

To show that KLINT's model is at least as expressive as BPF's, we use five existing BPF programs: Facebook's Katran [101] load balancer, the CRAB [62] load balancer, a filter from Suricata [87], a firewall from hXDP [12], and a bridge from Polycube [75]. We extend KLINT to analyze the assembly code compiled by the kernel after dumping it through a Linux debugging facility, using contracts we wrote for the BPF maps. Since we have no specifications for these BPF programs, KLINT only verifies memory safety and crash freedom. KLINT verifies the bridge and firewall in seconds, and CRAB and the Suricata filter in less than 2 minutes, but Katran requires almost 4 hours. This is due to our choice to model packet contents with a ghost map for simplicity, even though packet contents do not factor into invariant inference. Katran reads and writes to dozens of fields in the packet, which means the ghost map representing the packet has too many items to be efficient. This could be fixed by using a different way to model packet memory, such as Z3 arrays.

Developers can use any programming language, even if that language is considered too "exotic" or "immature" for operators to blindly trust it, since KLINT verifies binaries, and thus can catch errors resulting from compiler bugs.

For instance, the Rust [79] programming language is a promising direction for writing low-level systems code, including networking, but a developer or an operator might be concerned about its maturity level. Indeed, as of this writing there are currently 69 issues in the Rust bug tracker [99] marked with the "unsound" tag, meaning the compiler allows code that violates Rust's safety guarantees. Furthermore, some Rust features such as removing unused parts of the standard library are currently experimental.

However, we are able to write a traffic policer in Rust, compile it using these experimental features, and verify it using KLINT with the same specification as our C implementation. We can thus be confident that no matter what bugs lie in the Rust compiler, our binary is correct.

Verifying network functions

We summarize the time it takes KLINT to verify our network functions in Table 3.2, split into the time to symbolically execute the code, infer invariants, and verify the resulting paths. KLINT runs multiple iterations of symbolic execution and invariant inference to find a fixed point, thus we report the total time spent in each category.

We measure all times on an Intel i7-7700HQ CPU running at 3.60GHz. KLINT is single-threaded, though invariant inference is embarrassingly parallelizable. We prototyped parallelization but ran into a complex bug between `angr` and Z3 due to garbage collection [2], and since KLINT is fast enough we did not investigate further.

Overall, even the most complex of our network functions, the bridge, takes about 2 minutes to verify, which we believe is reasonable. We could further reduce verification time by reducing the redundancy in some invariants, and by using less flexible tools than `angr` and Z3, since we do not use their full power. The router uses a single data structure, the longest-prefix-match table, thus it only has one invariant.

We caution against over-interpreting the exact results, as most of the time is spent by the Z3 solver, and we noticed that the time Z3 takes to solve queries can vary significantly with small changes in queries, due to the heuristic-based nature of solving. Thus, total verification time can vary by dozens of seconds with minor changes in KLINT or Z3.

We show the KLINT equivalent of a firewall, our running example of a specification, in Listing 3. The full specification is too long to show here, but KLINT can also verify this partial specification.

The specification abstracts away implementation details such as the type of values in the firewall's flow table, as seen in line 15. This uses Python's "ellipsis" literal, meant for domain-specific languages.

	Time (seconds)				**#invariants**
	Symbolic execution	**Invariant inference**	**Verification**	**Total**	
Bridge	82	18	19	**119**	32
Firewall	26	7	10	**43**	20
Maglev	36	11	10	**57**	25
NAT	43	7	8	**58**	20
Policer	53	10	6	**69**	25
Router	0	0	1	**1**	1

Table 3.2: Our network functions, the time KLINT takes to verify them, and the number of invariants it finds.

```
 1 def spec(pkt, config, sent_pkt):
 2   if pkt.ipv4 is None or pkt.tcpudp is None:
 3     assert sent_pkt is None
 4     return
 5   if pkt.device == config["external device"]:
 6     flow = {
 7       'src_ip': pkt.ipv4.dst,
 8       'dst_ip': pkt.ipv4.src,
 9       'src_port': pkt.tcpudp.dst,
10       'dst_port': pkt.tcpudp.src,
11       'protocol': pkt.ipv4.protocol
12     }
13     # there must exist a map that tracks flows,
14     # regardless of what it maps them to
15     table = Map(typeof(flow), ...)
16     if sent_pkt is not None:
17       assert flow in table.old
18       assert sent_pkt.data == pkt.data
19       assert sent_pkt.device == 1 - pkt.device
```

Listing 3: Partial specification of a firewall with 2 devices. This specification can be given to KLINT unmodified.

This specification is actual Python code run in the standard Python interpreter by KLINT, thus developers can write specifications using their existing programming knowledge. KLINT currently requires the order of fields in specification structures such as the flow declared in line 6 to match the order in the corresponding implementation structure. Having KLINT try all possible orders would not finish in reasonable time, but there may be better strategies.

We found and fixed two implementation bugs using specifications we wrote based on standards. First, section 7.8 of the IEEE 802.1D standard [67] forbids adding Ethernet group addresses to the filtering table, which we originally forgot to implement. Second, our NAT originally misused its port allocator if configured to use only a sub-range of ports. This was already present in the Vigor NAT, because Vigor requires a concrete size for data structures during verification, and its authors only verified it for the full port range.

Verifying the entire stack requires an additional 15 minutes per network function, most of which is spent inferring the network driver's actions because the driver performs thousands of writes to registers and each of these writes requires work from KLINT. This could be improved by recognizing simple loops, since most initialization operations are performed in loops over categories of registers.

3.5 Evaluation: Runtime performance

KLINT enables developers to write faster verified network functions in two ways: developers can quickly prototype changes to data structures without having to verify the data structures first nor writing any additional invariants or proofs, and they can use the abstractions they want instead of verification-specific ones. In practice, we expect developers to either use existing verified data structures or write contracts for existing "trusted" data structures such as BPF maps. KLINT enables this workflow by only requiring contracts and not proofs for these trusted data structures, whereas previous tools required an all-or-nothing approach.

First, ghost maps and invariant inference enable quick prototyping of new data structures. For instance, we rewrote our port allocator to have different performance characteristics with slightly different semantics, including stricter preconditions as we believed our network functions did not need the generality of the existing ones. Using an approach such as Vigor or Gravel, even when prototyping, we would have needed to add a model of the new port allocator to the verification tool, including annotations for invariants, to check whether our network functions would still be correct when using it. With KLINT, we wrote new contracts, automatically verified that our network functions already satisfied the new stricter preconditions, then manually verified the implementation once we finished prototyping.

Second, verifying binaries enables simpler and faster code by removing abstractions over low-level hardware features that existed for the sake of verification. For instance, using a tool such as Vigor, obtaining the time from the environment requires calling a C function that the tool replaces with a model at verification time. Unless the compiler can inline this function, the developer will pay the performance cost at run time. With KLINT, the code can use inline assembly to read the CPU's time stamp counter, which KLINT handles in the same way as other assembly instructions. Furthermore, in the Vigor model, the C function must be verified separately using a different tool, a problem KLINT does not have.

We benchmark our network functions using the Vigor ones as baselines, with the TINYNF [90] driver instead of the original DPDK subset since it is faster and is the base for our own network driver. We use the same setup as Vigor to make the comparison useful: two machines as in RFC 2544 [11], one running a network function and one running the MoonGen packet generator [29]. Both machines have Intel Xeon E5-2667 v2 CPUs at 3.30GHz, Intel 82599ES NICs, and run Ubuntu 18.04. These network cards are the ones we modeled for KLINT's

	Vigor			**KLINT**		
	Throughput	**Latency** (*us*)		**Throughput**	**Latency** (*us*)	
	(*Gb*/*s*)	50%	99%	(*Gb*/*s*)	50%	99%
Bridge	5.54	3.98	4.25	10*	3.84	4.26
Firewall	7.77	3.92	4.26	10*	3.84	4.25
Maglev	6.34	3.96	4.28	10*	3.90	4.27
NAT	3.47	3.97	4.32	10*	3.87	4.27
Policer	9.12	3.87	4.25	10*	3.83	4.24

Table 3.3: Maximal single-link throughput without loss, and latency with 1 Gb/s background load of the original Vigor network functions and our versions. * = link saturated

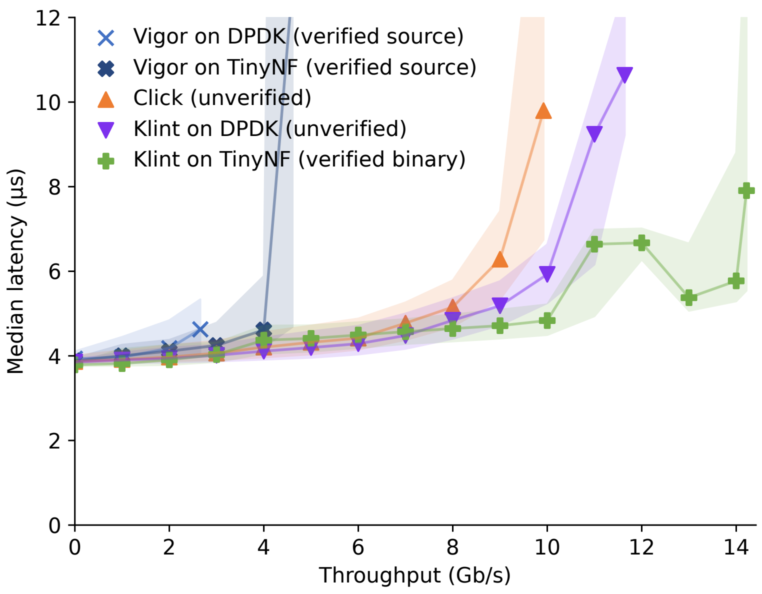


Figure 3.2: Throughput without loss vs. median latency of different bridges. Shaded areas delimit 5th and 95th percentiles.

full-stack verification. We measure throughput with minimally-sized packets, filling network functions' flow tables to 90% of their capacity.

We first run the same benchmark used in the Vigor paper: find the maximum throughput the network functions can sustain without dropping packets using one 10 Gb/s link and measure their latency when fed 1 Gb/s of background load. All our network functions can handle 10 Gb/s whereas the Vigor ones cannot, as we show in Table 3.3.

Since the Ethernet link is a bottleneck in the Vigor benchmarks, we use two links, for a theoretical maximal throughput of 20 Gb/s. To obtain more details about performance, we measure latency at increments of 1 Gb/s until the network function drops packets. We focus on the bridge for lack of space. We included the original Vigor bridge running on its verified DPDK subset, the Vigor bridge running on the verified TinyNF driver, the Click [63] bridge originally used as a baseline by Vigor, our bridge running on our verified driver, and our bridge running on DPDK, which is not verified. We do not know of any "standard" network functions we could use as a baseline beyond Click.

Despite our bridge having extra features compared to the Vigor one, such as support for a spanning tree protocol, it can reach more throughput before dropping packets than any of the other bridges, including the bridge from the widely used Click toolchain, as we show in Figure 3.2.

3.6 Evaluation: Manual effort

KLINT requires one kind of manual effort and encourages another. Developers must write contracts for their data structures, and should verify the implementations of these data structures for their entire network function to be verified. KLINT can automatically verify the network function code developers write everyday, which is code that uses common data structures. These common data structures may be the developers' own custom ones, but they are typically written once and reused many times.

Writing contracts does not require expertise in formal methods since they are defined with common map operations that developers are familiar with. The universal "forall" quantifier on ghost maps allows contracts even for operations that cannot be implemented using maps by describing *what* they do instead of *how*. However, we believe some data structures are not a good fit for ghost maps, specifically ordered ones. While queues and stacks can be viewed as maps from indexes to elements, the resulting contracts are unlikely to be conducive to invariant inference.

For data structure implementations, manual verification tools currently remain necessary. We used VeriFast [49], which uses annotation for C, but other approaches exist such as Dafny [69], a programming language designed for verification.

There is one more unlikely need for manual effort: if developers want to use a different networking stack. KLINT includes models for specific networking stacks, which must be completely precise for verification to succeed. Thus, swapping the network stack requires modifying KLINT.

3.7 Summary

In summary, we showed in this chapter that it is possible to automatically verify the network function code that developers write and evolve, even though the data structures they use remain too complex for automated verification.

Using maps to define data structure contracts enables even non-experts in verification to write such contracts for existing data structures, and provides a sweeping simplification that a verification tool such as KLINT can use internally.

4 Loop summarization for imperative code

In the previous chapter, we proposed a way to abstract data structures using maps so that a symbolic execution engine can still handle the rest of the code. However, this is unsatisfactory for two reasons: first, some data structures are intuitively so simple they should be symbolically executable, and second, defining anything that cannot be symbolically executed as a data structure means even basic operations such as searching over an array must be considered special.

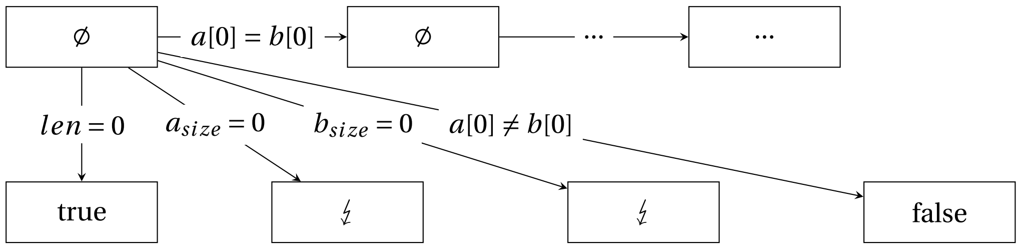
In this chapter, we propose to push the use of maps to abstract over loops as well. Instead of unrolling loops into a sequence of branches as in traditional symbolic execution, we propose a map-based intermediate representation for loops. This enables a symbolic execution engine to handle loops explicitly, such as by summarizing them into loop-free code, rather than implicitly unrolling them into redundant paths.

4.1 Insight: Common loops match well defined templates

Consider the following function comparing two C arrays:

```c
bool equal(int* a, int* b, size_t len) {
    for (size_t n = 0; n < len; n++) {
        if (a[n] != b[n]) return false;
    }
    return true;
}
```

Traditional symbolic execution will yield a symbolic graph of the following shape given unconstrained symbolic inputs:



Because the loop is unrolled, every iteration creates one state where the loop has finished and the result is `true`, one where the loop continues but the body returns `false`, two where the memory accesses are out-of-bounds, and one where the loop continues and the body does not return. That final state then leads to more states for the next loop iteration. The number of states is thus linear in `len`. This is despite the function logically having three paths: the blocks of memory are equal, or they are not, or the operation is ill-defined because `len` is too large. Even if `len` was concrete, which may happen if this function is called with a constant argument as part of a larger piece of code, there would still be a number of paths linear in that concrete value.

Loops must thus exist in a symbolic execution engine's intermediate representation, so that the engine can deal with loops as a high-level construct and avoid unrolling them, and maps provide a convenient answer to "looping over what?". Maps can represent common data structures, including ranges to handle the common C-style `for` loop, and heap objects to precisely model memory. While we obviously cannot deal with arbitrarily convoluted theoretical examples due to the Halting Problem [109], our representation can handle common kinds of code that developers write everyday in compliance with good software engineering practices.

Instead of many different kinds of loops such as `while`, C-style `for`, C#-style `foreach`, and so forth, we propose one loop statement: for κ in map by $order$ do st. The order is a function that provides a total order on keys. It is necessary for both known ordering, such as C-style `for` loops on an integer range, and unknown but fixed ordering, such as C#-style `foreach` loops on sets or maps which can be modeled with an uninterpreted function.

While recovering some loops from low-level intermediate representations is possible [105], it is by nature imprecise, and symbolic execution should ideally not depend on heuristics.

4.2 Design: Imperative loop summaries

Symbolically executing loop statements requires summarizing loops to avoid the redundant states created by traditional symbolic execution. There is an inherent tradeoff between precision, applicability, and automation when it comes to summarizing loops, since this is an undecidable problem in the general case. We focus on verification and thus require both precision and automation, trading applicability. Previous work instead focused on the bug-finding aspect and explored automatically handling many loops but proving only their termination or non-termination [110], automatically and precisely handling loops that use specific operations such as string manipulation in C [56], and handling many loops but requiring the user to provide a grammar [100] or to specify variables of interest [115].

We show that using maps enables straightforward loop summarization for common loop shapes, i.e., those that can be represented with our intermediate representation, by translating the result of symbolically executing the loop body to a loop-free statement. This generalizes the compact symbolic execution technique proposed by Slaby et al. [103], which does not support collections.

The kind of summarization we propose does not cause redundant symbolic states because it symbolically executes the loop body exactly once. Thus, as long as the body itself does not have redundant states, neither the summarization process nor symbolic execution of the summary introduce redundancy. This is unlike partial loop summaries [38], which are more widely applicable but require executing the loop body multiple times to confirm candidate invariants.

4.2.1 Preliminaries

We extend the map representation introduced in the previous chapter to cover all symbolic execution operations. Maps can be polymorphic, as in Boogie2, meaning the kind of a value depends on the exact key it is associated with. This also enables the modeling of data structures such as tuples and records.

We reuse the same four map query expressions: `get(map, key)`, `has(map, key)` (which is now explicit compared to `get` returning an optionally present value), `forall(predicate)`, and `length(map)`; and add two new map update statements for combining maps, bringing the total to four: `set(map, key, value)`, `unset(map, key)`, `add(map1, map2)`, and `subtract (map1, map2)`. The statements are imperative rather than functional, in the spirit of symbolic store updates. There are no map literals.

Instead of a traditional symbolic store, we use a *symbolic map store*, which contains map updates. There are no variables and thus no assignment statement. Instead, variables can be represented using an “environment” map from variable names, which are constant, to variable values. The environment map is not special cased. This scheme is inspired by interpreters for languages such as Python, and should enable modeling dynamic constructs such as Python’s `getattr` though we have not tried this.

To handle loops, we must handle not only halting, as in traditional representations, but also control flow such as `continue` and `break` in loops, and early `return`. While these can be simulated by adding indicator variables and branches to the rest of the code, the result is verbose and loses high-level information. For loop bodies in particular, adding more variables is undesirable because handling loops is complex enough already. On the other hand, adding one construct to the intermediate language for each kind of early return in each source language does not scale. We generalize the “halt” instruction using `throw "tag"` and `catch` `"tag" in <statement>` statements, where the tag must be a constant. Execution of a throw statement transfers control flow to the successor of the nearest parent catch statement with the same tag. Such a scheme has been proposed for specific cases such as `break` already [88]. As with halt, any additional data such as the returned value or the asserted expression must be stored in a specific variable and retrieved by the engine.

4.2.2 Examples

Consider these Python loops:

```python
for n in range(len(lst)):
    lst[n] = 0

for x in lst:
    if x == y:
        return True

for k, v in a.items():
    if v == 0:
        break
    b[k] = v
```

These loops all intuitively have one or two paths. However, traditional symbolic execution does not terminate since it unrolls the loops and there is no bound on the length of `lst` and `a`. Even with a bound, loop unrolling yields a set of paths with a redundancy factor close to 1 since all but one or two paths are redundant.

The first loop can be summarized as "all values in `lst` are now 0", which we model as an add operation on maps adding to `lst` a map associating all of `lst`'s keys to 0. The second loop can be summarized as "return `True` if any `x` in `lst` is equal to `y`", which we model using a negated $forall$ expression whose predicate is $m = x \wedge p \implies v \neq y$. The third loop can be summarized as "copy the part of `a` before a value of 0 into `b`", which we model as an add operation adding to `b` a map that contains some of `a`'s items. The last loop exemplifies why our loop statement requires an order: if the code later iterates on `a` again, the result must be consistent with `b`'s contents.

These summaries require the engine to support assumptions in order to define additional maps. Assumptions can be desugared into `throw` statements. Since the summary is a statement, there are no other requirements for the engine, i.e., the summarization technique does not need to control symbolic execution in any way, and can be a module separate from the symbolic execution engine.

4.2.3 Algorithm sketch

The key idea is to start from the states created by symbolically executing the loop body and emit a summary in two parts: (1) summarizing the map operations that happen when the loop body does not `throw`, and (2) copying the map operations and the throwing that happen when the loop body does `throw`. For loop bodies that cannot throw, the second part is empty. Thanks to the throw/catch abstraction, loop summarization does not need to worry about specific constructs such as breaks, continues, exceptions, or early returns.

A loop summary thus consists of constructing the following statements:

1. Assume that the loop has not finished before its last iteration, i.e., according to the loop order no key smaller than the last iterated one satisfies any early throw condition.
2. For each throw in the loop body:
   (a) Branch if the map is nonempty and the throw condition holds
   (b) Summarize map operations performed in iterations that did not throw
   (c) Perform the map operations in the iteration that threw
   (d) Throw.
3. Assume no early exit happened
4. Summarize map operations performed in all iterations of the loop body

4.2.4 Summarizing map operations

There is a large body of work on summarizing variable assignments in loops, such as finding precise closed forms of mathematical formulas [111], approximate ones for loop invariants [58], and induction variables for use in optimizing compilers [31]. Many of these methods could be adapted for use in a symbolic execution engine.

We focus here on describing how using maps helps. Because our intermediate representation includes add, $subtract$, and $forall$, a tool can transform a single map set or $unset$ operation into statements that create a map summarizing the operation over all keys and add or subtract this map from the operation's target. Handling `throw` statements works the same as when summarizing the loop structure: by restricting the keys in the added or subtracted map to those that did not throw.

While the simplification of having only map operations and no explicit variables helps, it is still necessary to distinguish set and $unset$ operations that use the loop iteration key as a key and those who do not. The latter are equivalent to variables in source languages, e.g., $set(E, "a", \kappa_0)$ should be summarized as a single set using the last iterated key. Unfortunately, since many mathematical formulas do not have closed form, it is not always possible to summarize the non-iteration-key-dependent operations, in which case the summarization algorithm fails.

Summarizing map operations in loops consists of the following steps:

1. If the operation reads data modified in previous operations, find a recurrence relation or fail
2. If the operation uses the loop iteration key as a set or $unset$ key, create a map representing the corresponding items using a $forall$ assumption, and return an add or subtract operation respectively
3. Otherwise, return a branch that, if the map is nonempty, contains the operation with the loop iteration key replaced by the last iterated key

4.3 Implementation

We prototype our techniques in LOOPSY, a new symbolic execution engine for C code that uses Z3 [24] as the underlying solver. Our engine takes source code as input so that it can detect loops, instead of an intermediate representation such as LLVM IR or even assembly that requires heuristics to reconstruct loops.

We use this engine to symbolically execute motivating examples from previous work, to show that map-based symbolic execution appears to generalize. Our engine is only a prototype intended to show that our techniques can work, not a general tool, and as such needs more engineering work to handle syntax beyond the basics.

LOOPSY uses the following intermediate representation (κ represents loop iteration variables), and thus acts as a “back end” which can be combined with “front end” translators from a programming language to the intermediate representation:

$$
\begin{array}{lll}
\mathrm{e} & ::= & \mathrm{b} \mid \mathrm{n} \qquad \textit{Expressions} \\
 & | & \mathtt{get}(\mathrm{e}, \mathrm{e}) \\
 & | & \mathtt{has}(\mathrm{e}, \mathrm{e}) \\
 & | & \mathtt{forall}(\lambda m, k, v, p.\ \mathrm{e}) \\
 & | & \mathtt{length}(\mathrm{e}) \\
 & | & \kappa_n \\
 & | & \mathrm{unop}\ \mathrm{e} \\
 & | & \mathrm{e}\ \mathrm{binop}\ \mathrm{e}
\end{array}
\qquad
\begin{array}{lll}
\mathrm{s} & ::= & \mathtt{set}(\mathrm{e}, \mathrm{e}, \mathrm{e}) \qquad \textit{Statements} \\
 & | & \mathtt{unset}(\mathrm{e}, \mathrm{e}) \\
 & | & \mathtt{add}(\mathrm{e}, \mathrm{e}) \\
 & | & \mathtt{subtract}(\mathrm{e}, \mathrm{e}) \\
 & | & \mathtt{throw}\ \mathrm{tag} \\
 & | & \mathtt{catch}\ \mathrm{tag}\ \mathtt{in}\ \mathrm{s} \\
 & | & \mathtt{if}\ \mathrm{e}\ \mathtt{then}\ \mathrm{s} \\
 & | & \mathtt{for}\ \kappa_n\ \mathtt{in}\ \mathrm{e}\ \mathtt{by}\ \mathrm{order}\ \mathtt{do}\ \mathrm{s} \\
 & | & \mathrm{s}\ ;\ \mathrm{s}
\end{array}
\qquad
\begin{array}{ll}
\mathrm{b} & \in \textit{Bools} \\
\mathrm{n} & \in \textit{Ints}
\end{array}
$$

with the following big-step semantics for statements, going from triples of statement, map store, and constraints to triples of tag, map store, and constraint:

$$\frac{st \in \{\text{set}, \text{unset}, \text{add}, \text{subtract}\}}{\langle st, s, c\rangle \Downarrow \{\langle \text{nil}, s + st, c\rangle\}}$$

$$\frac{}{\langle \textbf{throw}\ tag, s, c\rangle \Downarrow \{\langle tag, s, c\rangle\}}$$

$$\frac{\langle st, s, c\rangle \Downarrow \{\langle r_i, s_i, c_i\rangle^*\} \qquad \langle \text{nil}, s_i, c_i\rangle = x_i \text{ if } r_i = tag \qquad \langle r_i, s_i, c_i\rangle = x_i \text{ otherwise}}{\langle \textbf{catch } tag \textbf{ in } st, s, c\rangle \Downarrow \cup x_i}$$

$$\frac{\langle st, s, c \wedge s\{e\}\rangle \Downarrow x}{\langle \textbf{if } e \textbf{ then } st, s, c\rangle \Downarrow x \cup \{\langle \text{nil}, s, c \wedge s\{\neg e\}\rangle\}}$$

$$\frac{\langle st, \emptyset, true\rangle \Downarrow x \qquad \text{summarize}(x, \kappa_n, s\{m\}, order) = st' \qquad \langle st', s, c\rangle \Downarrow y}{\langle \textbf{for } \kappa_n \textbf{ in } m \textbf{ by } order \textbf{ do } st, s, c\rangle \Downarrow y}$$

$$\frac{\langle st_1, s, c\rangle \Downarrow \{\langle r_i, s_i, c_i\rangle^*\} \qquad \{\langle r_i, s_i, c_i\rangle\} = x_i \text{ if } r_i \neq \text{nil} \qquad \langle st_2, s_i, c_i\rangle \Downarrow x_i \text{ otherwise}}{\langle st_1 \,;\, st_2, s, c\rangle \Downarrow \bigcup x_i}$$

The standard `continue` and `break` constructs can be translated into catch statements inside and outside a loop respectively:

```
for (...) {
  if (...) continue;
  if (...) break;
}
```

$\Rightarrow$

```
catch "break" in
  for ...
    catch "continue" in
      if ... then throw "continue";
      if ... then throw "break"
```

Loop handling algorithms thus do not need to handle these cases explicitly.

4.4 Evaluation sketch

LOOPSY is currently still a prototype and thus we do not have a fully-fledged evaluation. It can handle all loops that satisfy the "for key in map" shape and only contain assignments that it can summarize, without any human effort. Importantly, LOOPSY does *not* require any proof annotation, loop invariant, or other such manual step.

In terms of network functions, consider the following example of a loop written by undergraduate students tasked with writing a DHCP server and verifying it with KLINT:

```
for (uint32_t next_avail_ip = ip_subnet + 1; next_avail_ip < ip_subnet +
  num_ipv4_addr_avail; next_avail_ip++) {
  if (next_avail_ip != dhcp_server_ip && next_avail_ip != gateway_ip &&
  next_avail_ip != dns_server_ip) {
    ip_addresses[index] = next_avail_ip;
    index++;
  }
  if (index == capacity) {
    break;
  }
}
```

While KLINT cannot handle this loop because the bounds are symbolic as they come from the

NF's configuration, LOOPSY can handle it by summarizing the loop's effect on the IP addresses array and on the index variable.

In terms of previous work, consider the motivating example for MemSight [18]:

```
void bomb(char* a, char i, char j) {
  char boom;
  a[i] = 23;
  if (a[j] == 23) boom = 0;
  else boom = 1;
  assert(!boom);
}
```

Because we model `char*` as a map from pointers to characters, this example is trivial: the assert is triggered iff $get(a, j) = 23$ given the symbolic store $set(a, i, 23)$. If we add the assumption that a is originally zeroed, our engine simplifies this to $i = j$, without having to call the solver. Interestingly, MemSight's memory model is similar to the maps translation we described earlier, though they do not explicitly mention detecting out-of-bounds memory accesses, but we generalize maps to the entire engine instead of only handling heap memory with it.

As another example, consider the running example of Trtík and Strejček [108]:

```
int* A;
int foo(int n, int i) {
  A = (int*)malloc(n * sizeof(int));
  A[3] = 777;
  A[4] = 888;
  A[3*i+1] = 999;
  return A[3] + A[4];
}
```

Once again, modeling arrays as maps makes this example trivial since the get operations for the return value will naturally include if-then-else expressions based on the value of i. The modeling also includes the implicit bounds checks that must happen since n may not be large enough compared to 3, 4, and $3 * i + 1$.

Consider the following excerpt from a Windows NT driver used as an example for Tracer [51]:

```
lock = false;
new = old + 1;
while (new != old) {
  lock = true;
  old = new;
  if (*) {
    lock = false;
    new++;
  }
}
assert(lock);
```

Tracer requires multiple passes through the loop body to compute candidate pre- and post-conditions, learn that some of the candidates do not work, and refine the guesses.

We represent the `while` loop as iterating over an infinite map M, with a test to break out of the loop if $new = old$, and the nondeterminism as an uninterpreted function f over the loop iteration key κ. Symbolically executing the loop body leads to three states, where E is the environment map:

- $\langle "break", \emptyset, new = old\rangle$
- $\langle nil, \{set(E, "old", get(E, "new")), set(E, "lock", false), set(E, "new", get(E, "new") + 1)\}, new \neq old \wedge f(\kappa)\rangle$
- $\langle nil, \{set(E, "lock", true), set(E, "old", get(E, "new"))\}, new \neq old \wedge \neg f(\kappa)\rangle$

In the loop summary, given ω the original value of old and N the map containing all keys in M that satisfy f before the loop breaks:

- $lock$ is $\neg f(\kappa)$
- old is $\omega + 1 + length(N)$
- new is $ITE(f(\kappa), 1, 0) + \omega + 1 + length(N)$

Thus, the assertion is equivalent to checking the satisfiability of $f(\kappa) \wedge (ITE(f(\kappa), 1, 0) + \omega + 1 + length(N) = \omega + 1 + length(N))$ which is trivially unsatisfiable without needing to call the solver.

4.5 Summary

The work presented in this chapter is only the beginning of a possible future for exhaustive symbolic execution, which we believe could preserve the fundamental simplicity of the core algorithm while handling more code, at the cost of side complexity in the algorithm for summarizing loops.

Part II

Making systems code verification-friendly

5 Verification-friendly NIC driver architecture

This chapter is based on "A Simpler and Faster NIC Driver Model for Network Functions, S. Pirelli, G. Candea, OSDI 2020".

The current network driver model is too flexible for the needs of common network functions, which must pay the complexity costs of modern drivers without reaping their benefits. This is mainly because the current driver model allows network functions to process packets out of order, a powerful feature that is not needed in many of the core functions making up the Internet's backbone.

We propose a simpler model that (1) makes network functions easier to formally verify, (2) is faster than the current most complex driver model that can be formally verified, (3) provides competitive performance against the fastest state-of-the-art drivers regardless of complexity, and (4) is applicable to most network functions that are deployed today.

5.1 Insight: Common network functions do not need a complex driver model

The traditional network card driver model provides flexibility to network functions: they can keep buffers aside to reassemble messages from high-level protocols such as TCP, and can allocate buffers from the pool in response to non-network events such as timers indicating a request needs to be retried. The model also lends itself well to concurrency: the "free" queue is the central element shared by any number of reception, transmission or processing queues. A network function can receive and transmit packets from multiple NICs, and it can use multiple processing queues that each communicate with different reception and transmission queues on the same NIC to process packets concurrently and increase overall throughput.

But this flexibility comes at a cost: the steps that the network function can perform besides transmission introduce forks in the path of packet buffers. This requires buffer management within the "free" queue, including support for concurrent accesses. It also requires the driver to

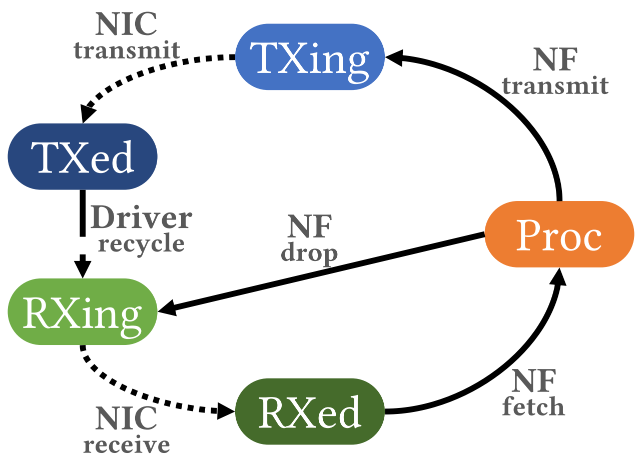


Figure 5.1: Diagram of our proposed driver model, with the same semantics as Figure 2.3.

implement a policy for buffer freeing and allocation, adding complexity to the overall system. The model additionally introduces a failure case that is not fundamental to the concept of a network function. If there is a state within the processing logic in which any buffer is kept, and the only way to get out of that state is to receive new data, the system will only make progress if there are buffers outside of the processing queue, which is not guaranteed. Reasoning about the existence of such a state requires reasoning about the invariants that hold in the network function code across packets.

This flexibility is not always needed: some of the network functions that power the backbone of the Internet, such as IP routers or Ethernet bridges, process packets one by one, never keep buffers aside, and never allocate buffers. Overall, they are conceptually simpler than the general case of a network function, yet they must currently pay the price of driver flexibility they do not use.

5.2 Design: New driver model

We propose a new driver model designed for common network functions that do not need the flexibility provided by existing models. It is based on two key insights: we can remove the buffer pool altogether, and we can implement buffer drops on modern NICs without the theoretical branch they introduce, minimizing the amount of state that the driver must keep track of.

Our model is designed to be as simple as possible, thus improving correctness and performance. Its simplicity makes it easier to formally or informally reason about and requires less code and simpler code to implement.

Our model is a subset of the existing model: as shown in Figure 5.1: the core differences are that it has no pool of free buffers, and does not allow network functions to keep buffers. The driver moves transmitted buffers directly to the reception queue, and the network function must choose to either transmit or drop received packets. This simplifies the driver by giving it only one choice when transmitting a packet: recycling transmitted buffers to the receiving queue now or later. Removing the buffer pool also makes progress easier to reason about: the software can only halt if the driver does not recycle buffers when the receiving queue is empty, or if the network function halts. While termination is impossible to prove in the general case due to the halting problem, network functions have strict performance requirements, thus their code is unlikely to have loops whose termination is not obvious because such loops could be performance bugs.

Our model minimizes state by combining reception, processing, and transmission into a single logical descriptor ring containing all buffers, without the need for any other data structure. While it is implemented using one reception ring and one transmission ring, the driver mirrors the head of the transmission ring to the tail of the reception ring, thus ensuring that buffers that have finished transmitting are reused for reception without any intermediate steps. The key hardware feature that allows this is called “null transmit descriptors”: as its name implies, it allows some descriptors in a transmission ring to have no effect. Packet drop is thus a special case of packet transmission, which removes the fork in buffers’ paths and allows for a regular buffer flow. For instance, a network card can implement this by dropping packets whose length in metadata is zero.

The driver’s job consists of three tasks: move buffers from the “received” queue to the “processing” queue when the network function asks for a packet, move buffers from the “processing” queue to the “transmitting” queue when the network function asks to transmit or drop its current packet, and recycle buffers from the “transmitted” queue to the “receiving” queue to ensure the “receiving” queue is never empty. Since this last operation is not a response to a specific input, the driver must choose when to perform it, for instance once every few transmitted packets.

Our model supports multiple outputs by using multiple transmission rings and making the driver synchronize their state. That is, the driver must set the tails of all transmission rings at the same time and use the earliest head in all rings as the head to mirror to the reception tail. Transmitting a packet when the driver has multiple outputs conceptually maps to transmitting it on some outputs and dropping it on all others; all rings still have a descriptor pointing to the buffer, but that descriptor is null in some of the rings. This may cause packet drops if an output link is too slow, in which case the entire ring will be used for transmission with no space left for reception. The same could happen in a traditional model if all buffers in the pool were used for transmission due to a slow output.

Multiple inputs can be handled concurrently: while the same processing queue cannot have multiple inputs, since it is not possible to synchronize the state of reception rings, the entire system can be duplicated so that there is one reception queue per input, one associated

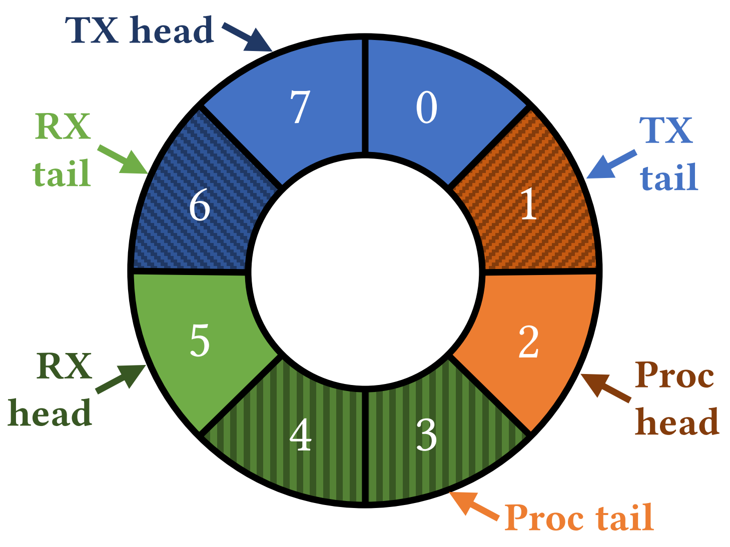


Figure 5.2: Logical ring composed of reception, processing and transmission queues. "In progress" queues are light, "done" ones are dark and shaded. Heads and tails refer to "in progress" queues, but implicitly delimit the others.

processing queue, and any number of synchronized transmission queues. Modern NICs have hundreds of queues, thus it is not a problem to use one transmission queue per input. This does not mean our model requires parallelism: a single thread of execution can implement many instances, which are thus concurrent but not parallel.

Our model is amenable to parallelism: multiple threads of execution can run in parallel, each implementing any number of instances, without having to synchronize any state. Only the state of the rings within an instance needs to be kept in sync. This is similar to existing models.

The key limitation of our model is the flip side of its strength: since network functions must process buffers one by one without keeping any aside, they cannot reconstruct multi-packet messages without copying buffers that arrive out of order. Thus, while core functions such as routing and network address translation can be implemented with our model, one cannot terminate TCP connections or otherwise reassemble fragments without copying buffers, which is an expensive operation given modern network speeds.

5.3 Implementation

In this section, we describe an implementation of our driver model for the Intel 82599 NIC which we call "TinyNF", short for "Tiny Network Function".

TinyNF's goals are to be easy to reason about and fast. The former is different from "correct" because it is hard to tell whether a driver operates as expected without hardware schematics, since the data sheet may be incorrect. However, we want to make it simple enough that it is not a bottleneck in network function verification efforts. For simplicity, TINYNF processes buffers one at a time: there is always at most one buffer in the processing queue. One key hypothesis in this project was that TINYNF could be fast without explicitly processing packets in batches.

The keys to TINYNF's performance are the avoidance of any operation that is not absolutely required and the use of a few small but surprisingly effective scheduling algorithms for synchronizing queue state.

TINYNF avoids unnecessary work, even metadata copy. Because each buffer always belongs to exactly one queue, and because queues are ordered, it is enough to set the buffer pointers at initialization time and never change them afterwards. Moving a buffer from one queue to another only requires writing to the source head and destination tail.

There are fewer delimiters in practice than in theory since some of them are implicit, as shown in Figure 5.2. The "transmitted" head and tail are the "receiving" tail and "transmitting" head, respectively. Similarly, the "received" head and tail are the "processing" tail and "receiving" head. While there is technically a "processed" queue that does not exist in the conceptual model, its head and tail are the "transmitting" tail and "processing" head respectively. The "processing" tail does not need explicit tracking, because it is always either one buffer ahead of the head or equal to it, due to the one-packet-at-a-time constraint.

TINYNF avoids reading from NIC registers entirely after initialization. To check for received buffers, the "descriptor done" metadata flag of the descriptor at the processing tail is enough. To check for transmitted buffers, the 82599 NIC provides a "transmit head write-back" feature: software can request hardware to write the transmit head to RAM after hardware has finished transmitting a buffer.

TINYNF cannot avoid updating the receive and transmit tails, which are NIC registers and thus slower than RAM, but it can avoid doing so after every packet. Updating the receive tail, which moves buffers to the "receiving" queue, is only necessary once every few transmitted buffers since reception continues working as long as there are buffers in the queue, even if there are less than there theoretically could be. Updating the transmit tail is necessary for buffers to be transmitted to the network, but this can be done once every few transmissions, or when there are no packets to receive and thus no other work to do.

TINYNF carefully schedules operations to minimize the amount of communication between software and hardware. This improves overall latency and reduces the fraction of PCIe throughput used for metadata.

Two operations can be scheduled together: asking the NIC to update the transmission tail and checking for such updates to recycle buffers. The request is made with a bit in transmission metadata, and the check is made by reading the value that the NIC wrote to RAM via DMA.

TinyNF schedules both operations once every 64 packets. The check will thus see the update that was requested 64 packets ago.

The most important scheduling decision is updating the transmission tail: frequent updates decrease latency by making the NIC aware of packets sooner, but they increase throughput by performing less book-keeping. Networking stacks such as DPDK solve this with adaptive batching: they check for multiple received buffers at a time up to a limit, let the network function process them all, then update the transmission tail. This theoretically allows drivers to make better scheduling decisions because they have more data: they know how many packets have arrived, rather than whether there is at least one packet.

TinyNF's one-packet-at-a-time model is incompatible with batching, thus we chose an algorithm based on past data instead. TinyNF updates the transmission tail either once every few transmitted packets, or as soon as there are no packets to receive since this likely indicates there is time to perform this expensive operation. This keeps the period short under low load, avoiding latency spikes, but allows for longer periods under high load, avoiding throughput drops, without looking at packets beyond the current one.

Overall, TinyNF is around 550 lines of C code, and its only dependency is a 300-line environment abstraction. It runs entirely in user mode, without kernel dependencies.

5.4 Evaluation: Completeness

In this section, we evaluate the applicability of our model to real-world network function deployment. Did we strike a good trade-off choosing not to support some functions to simplify the model? And is our model useful in the context of network function virtualization? As previously explained, the core limitation of our driver model is that network functions cannot keep buffers aside for later use. For instance, they cannot reconstruct messages in TCP or other higher-level protocols. Our model targets network functions that do not need to do so because they logically handle packets one at a time.

Our model supports many well-known functions, though there is no standard list of network functions. Despite their increased importance in modern networking, there is no consensus on what is a "network function" and what is not. There have been attempts such as RFC 3234 [96] to classify "middleboxes", which are functions that are not crucial to the network, but to the best of our knowledge there is no commonly accepted list of network functions. We chose to use the list of functions from the ClickOS [71] paper, which were also used by the authors of Vigor [119] to estimate the applicability of their verification technique. We complement this list with our own knowledge, for lack of a more standard source.

Our driver model supports 13 of the 14 types of network functions listed in ClickOS: load balancing, DPI, NAT, firewalls, tunnel, multicast, BRAS, monitoring, DDoS prevention, IP proxies, congestion control, IDS, and IPS. The only one that our model cannot support without compromises is a traffic shaper, because shaping requires keeping packets to send them later in the desired traffic shape. Among the network functions not mentioned by ClickOS, our model can be used for Ethernet bridges, ARP clients and servers, DNS proxies, statistics collectors, traffic policers, and Google's Maglev [28] load balancer.

However, our driver model cannot efficiently support functions based on entire TCP messages, since this requires keeping IP packets around to reorder and merge them into logical messages. Such functions include proxies and HTTP servers. While one could implement reordering by copying buffers before giving descriptors back to the hardware, this would hinder performance. We believe our model is a good fit for network functions that form the backbone of networks, such as routing, load-balancing, NAT and DNS, access control and statistics. However, it is not suited to high-level functions that deal with entire connections or protocols that fragment packets.

Some requirements are orthogonal to our model. For instance, offloading checksums to hardware would remove the main bottleneck in the NAT we benchmarked. Any such feature that can be used by providing metadata to the NIC can be implemented in a driver using our model.

TINYNF can be used for virtualization, which is a key tool for the practical deployment of network functions [86]. Virtualization allows operators to deploy multiple network functions on the same physical machine, instead of having to dedicate an entire machine to a single function. They also provide an easier way to manage network functions, in the same way virtual machines ease software management.

We experimented with virtualization using Single-Root I/O Virtualization, or "SR-IOV" for short, a PCIe standard with which network cards can expose virtual network cards with the same packet-processing features as the physical card. The virtual machine monitor can let virtual machines access virtual devices directly, without surrendering control over the physical card. The physical card includes hardware to route packets to virtual cards based on packet headers, for instance by Ethernet address. The physical card can limit the rate at which each virtual card transmits packets and can prevent virtual cards from transmitting packets with a different source address than their own. Virtual machines thus gain the benefits of direct access without the ability to monopolize the link or lie about their network identity.

The Intel 82599's virtual cards do not support some of the physical features. Notably, using transmit head write-back causes virtual cards to hang, a problem not mentioned in the card's data sheet but already reported by the authors of Arrakis [89]. Another missing feature is legacy packet descriptors, which are simpler to use, though the data sheet calls this out. We wrote a version of TINYNF that does not use these features, making it slightly slower. The Arrakis authors estimated that the lack of transmit head write-back causes a 5% performance penalty.

We used 16 virtual functions on each of the two network cards, for a total of 32 virtual cards, and a setup we describe in the next section. Each virtual card has an Ethernet address, and physical cards route packets to virtual cards based on these addresses. The only code changes are due to the missing features mentioned above, as well as a few dozen lines of configuration. The functions forward each packet using a virtual card on the physical card opposite the one whose virtual card received the packet. The Vigor policer handles 12.2 Gb/s of minimally-sized packets without loss when using TinyNF in this setup. A no-op function reaches 14 Gb/s. Both are bottlenecked by reading packet descriptors for packet fetches, as the data from packets and descriptors no longer fits in the L2 cache. This experiment is only intended to show that our driver model is applicable to virtualized environments. With this number of devices, other concerns arise such as load skew across devices and non-uniform memory accesses, which we do not capture here. We believe TinyNF is as sensitive to these concerns as other stacks. In particular, the order in which the function checks virtual cards for packets matters. For instance, if packets mostly arrive on one card, checking the other cards for packets will limit performance.

5.5 Evaluation: Run-time driver performance

In this section, we compare the performance of TinyNF and DPDK for general purpose network functions, regardless of verifiability. To measure performance, we used two machines in a setup based on RFC 2544 [11], with a "device under test" running a network function and a "tester" running the MoonGen packet generator [29], which can measure latency using NIC timestamps. Both machines run Ubuntu 18.04 on two Intel Xeon E5-2667 v2 CPUs at 3.60GHz with power-saving features disabled and have two Intel 82599ES NICs, using only one port per card to ensure PCIe bandwidth is not a bottleneck. We measure throughput using minimally sized packets. Our workload fills the internal flow table of the network functions to 90% of their capacity. Measuring latency with MoonGen instead of on the device under test allows us to capture the latency of NIC register writes as well as the effects of drivers' NIC configuration. This setup is similar to the one used to originally evaluate Vigor, and could replicate Intel's DPDK performance numbers at the time. We use both directions for throughput, for a maximum of 20 Gb/s. We keep throughput symmetric during the benchmarks, i.e., if a function cannot handle a given load, we reduce the load of both directions by the same amount and retry. We then measure the latency at load increments of 1 Gb/s to paint a clear picture of the function's overall performance profile.

TinyNF can outperform a fully optimized DPDK setup, as we show in Figures 5.3 and 5.4 using a traffic policer as an example. We compare the Vigor policer using TinyNF as its driver to the same code using either "unbatched" DPDK, which is the simpler version used by Vigor, or "batched" DPDK, which is the standard way to use DPDK that enables optimizations such as adaptive batching and vectorization. We also implemented a 2-core parallelization of the policer for all three variants. We chose the policer because, by design, traffic in one direction is independent of traffic in the other, which means it admits a trivial 2-core parallelization for our experiments. We are not proposing a new way to parallelize network functions, but merely showing that TinyNF can be parallelized in a similar way to existing drivers. This also shows how much improvement parallelization can bring compared to batching.

Using TinyNF, the policer achieves better throughput than using batched DPDK, with an even starker difference when using two cores. The bottleneck that prevents the dual-core TinyNF version of the policer from reaching line rate is the frequent reads from the CPU time, which it needs for flow expiration.

TinyNF leads to better latency at low and high loads but worse latency in the middle, especially the 99th percentile latency. Looking at individual data points, which we show in Figure 5.5, the TinyNF-based policer has lower latency in some cases, but this advantage is lost in the tail latency. We believe this is a case where DPDK's batching shines: it can detect "gaps" between packets, in which updates to the transmission tail do not compete with packet processing, by looking at how many packets there are in the queue.

A no-op function can handle more throughput with DPDK than with TinyNF, even though the opposite holds with real functions. We reached this surprising conclusion by benchmarking DPDK's "testpmd" built-in application, which DPDK developers use in performance reports [106] to benchmark driver speed. We configured testpmd to update packets' MAC address to provide some realism. Using our setup, both TinyNF and DPDK in its batched mode could saturate two 10 Gb/s links, as we show in Figure 5.6. We also included the Ixy driver [30], which performed admirably given its educational purpose but could not sustain line rate even with batching.

Since our setup was bottlenecked by link capacity, we chose to lower the CPU frequency to 2 GHz and re-run the benchmark. In this setup, DPDK can reach 97.5% of line rate while TinyNF peaks at around 92.5% of line rate, as we show in Figure 5.7, though its latency is lower. We believe the bump around 11 Gb/s is due to hardware issues, since it appears in three independently written drivers and in both a no-op and a nontrivial function.

This result is interesting, since the no-op benchmark is the one used by DPDK developers to measure their progress when optimizing DPDK's performance. If this benchmark does not accurately represent driver performance on real network functions, the DPDK developers may believe they are improving DPDK's performance but do the opposite.

To explain this finding, we started by plotting the no-op function's latency in more detail. We did this because of an observation we made while running the other benchmarks: TinyNF's performance appeared more stable than DPDK's, yielding more consistent results across runs, such as never dropping packets under high loads whereas DPDK would sometimes drop a few packets per million.

As expected, TinyNF has a more stable latency profile than DPDK: without background load, TinyNF's latency remains low up until the 99.9th percentile, whereas DPDK's latency starts jittering before this, as show in Figure 5.8. We stop at the 99.99th percentile because Primorac et al. showed that NIC timestamping is not accurate after that point [93].

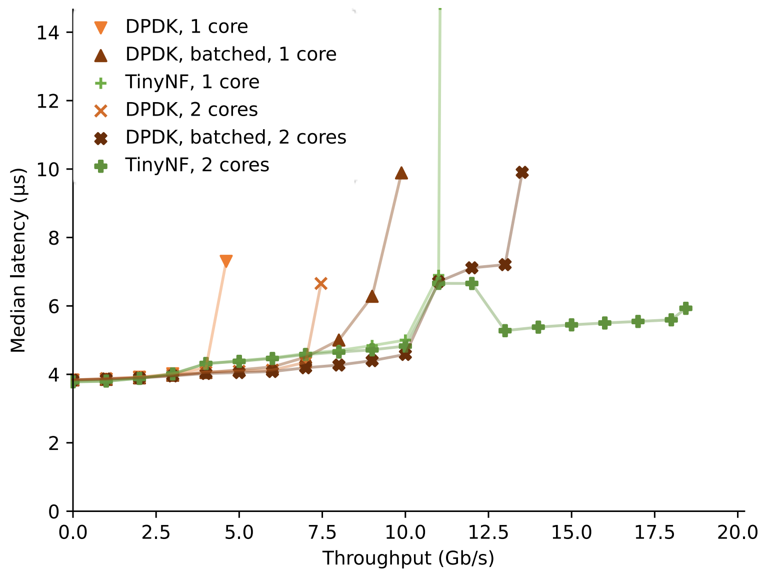


Figure 5.3: Throughput and median latency of a traffic policer using DPDK with and without batching, TinyNF, and 2-core versions of all three.

This measurement highlights a key issue with DPDK's driver model: the driver has to manage buffers explicitly instead of merely moving them from one queue to the next, which leads to a distinct bump in latency before the 99th percentile. The same holds for Ixy, since it uses the same driver model as DPDK.

We used the toplev microarchitectural measurement tool [107] to investigate bottlenecks in DPDK's driver when running the Vigor policer. While the tool indicates that the policer is bottlenecked on memory writes, there is no single write that dominates. Some of the memory writes that take the most time are fundamental to DPDK's design, such as moving buffer pointers to and from the buffer pool, while others could be removed at the cost of some functionality, such as writes to packet buffer metadata.

5.6 Evaluation: Manual effort

In terms of verifiability, we succeeded in our goal to automatically verify the TinyNF codebase. Developers do not need any manual effort to move from Vigor's modified DPDK subset to TinyNF, keep the benefits of automated verification, and reap increased performance. We evaluate TinyNF by using it with Vigor's formally verified network functions, using the Vigor tooling to verify network functions and the drivers they use.

TinyNF is 1/11th the code of the DPDK driver and has exponentially fewer paths, which leads to drastic improvements in verification time. Instead of a geometric mean of 26822 seconds of verification time for Vigor functions, we obtain 3108 seconds with TinyNF. This is

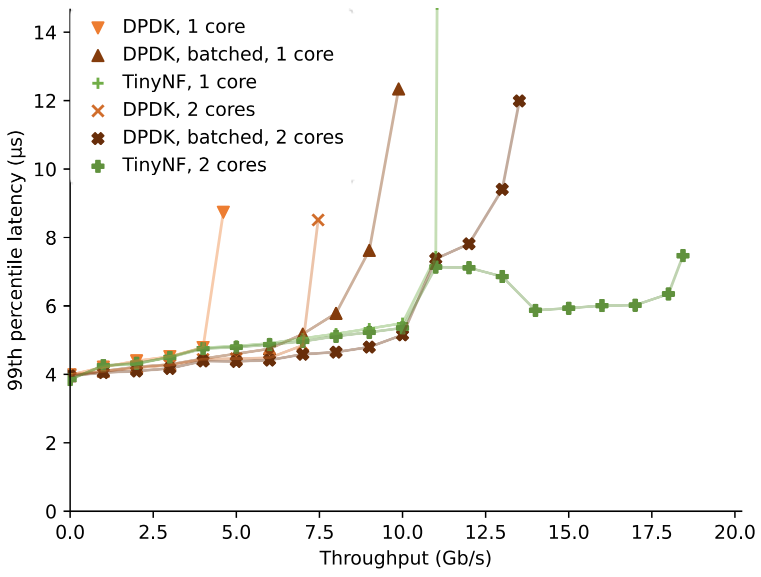


Figure 5.4: 99th percentile latency version of Figure 5.3.

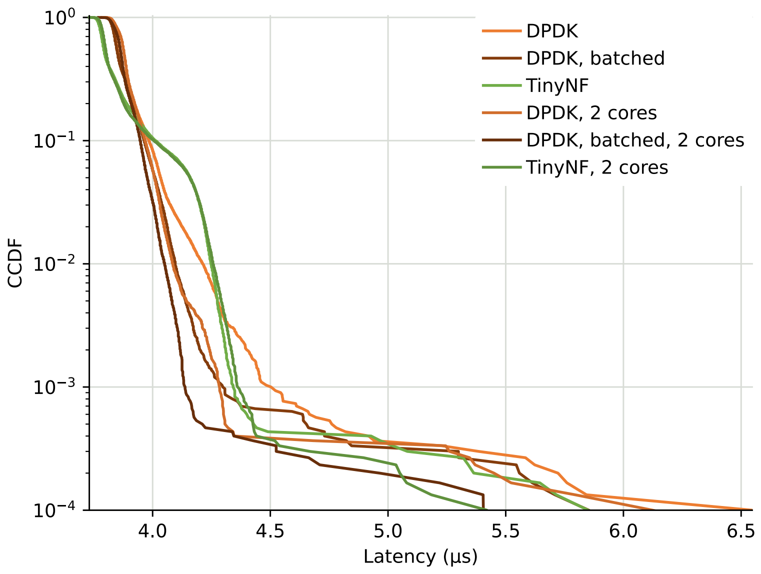


Figure 5.5: Complementary cumulative latency distributions of a traffic policer using the same alternatives as Figure 5.3, with 1 Gb/s background load.

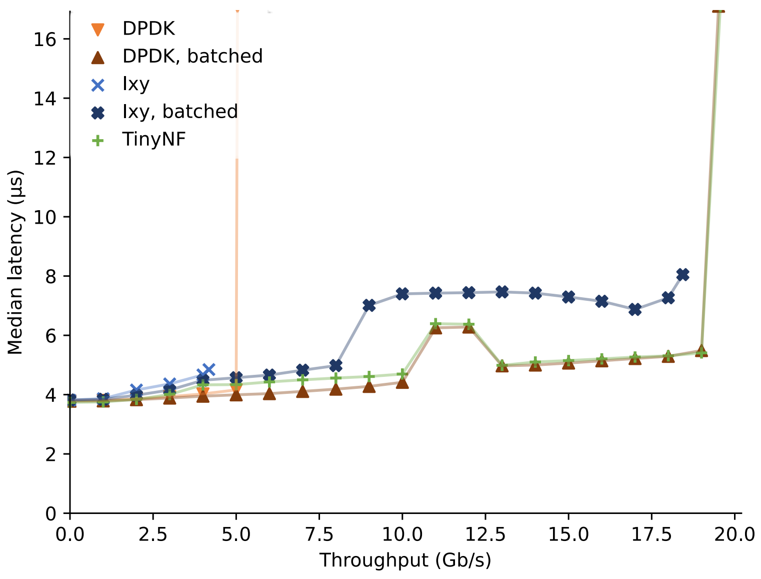


Figure 5.6: Throughput and median latency of DPDK's no-op function with and without batching, a port of it on TinyNF, and a port of it on Ixy with and without batching.

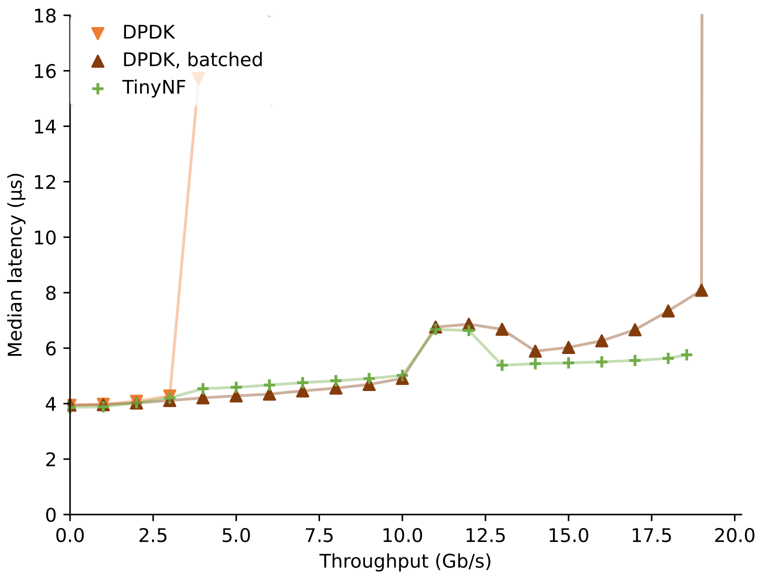


Figure 5.7: Same benchmark as Figure 5.6 but with the CPU capped to 2 GHz. We do not show Ixy since it could not sustain line rate even at full CPU speed.

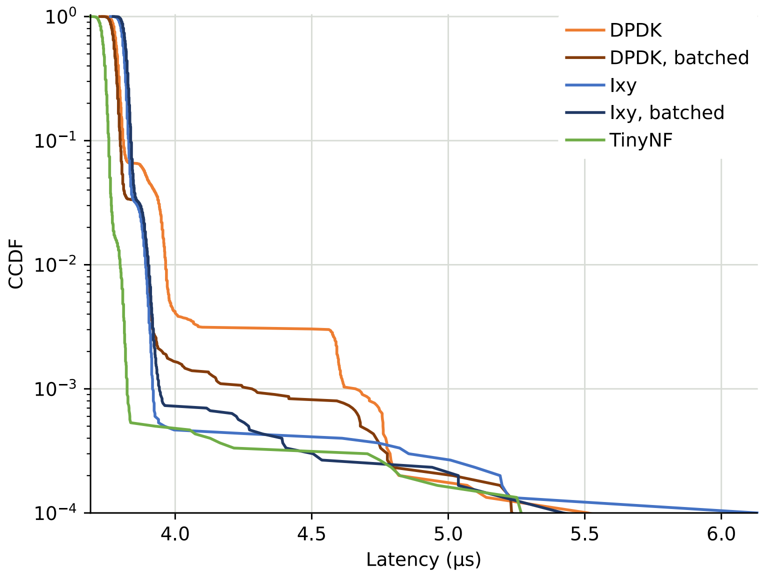


Figure 5.8: Complementary cumulative latency distributions of the no-ops from Figure 5.6 without background load

too long for developers' inner loop, but verification with the driver is not part of the inner loop anyway.

We measured the code complexity of TinyNF and of the verified subset of DPDK's driver. We manually counted paths, so that we could define them in terms of the public parameters: the arguments passed in the code, and the choices made at DPDK build time when picking a data structure implementation. Automating this using symbolic execution would have only found the number of paths given a concrete configuration. When counting paths, we assume that NIC hardware behaves as per its data sheet. To show the effect of a change in driver model and not only in implementation, we also included the "Ixy" driver by Emmerich et al. [30], a simplified implementation of DPDK's design for educational purposes that does not aim for comparable performance. As expected, TinyNF and Ixy use similar amounts of code to initialize, since they both use a limited set of NIC hardware features. However, TinyNF has less code and exponentially fewer paths than Ixy in the reception and transmission functions that form the core of the driver, providing more evidence in favor of our model.

We find that TinyNF has 3 paths for reception and $2+2^L$ paths for transmission, where L is the number of output links. For comparison, Ixy has $1+A_{ixy}$ paths for reception and 14^L for transmission, where A_{ixy} is the number of paths in Ixy's memory allocation function. DPDK has $1+AF_{dpdk}+288AS_{dpdk}$ paths for reception and $(8+14(FF^T_{dpdk}+P((FS_{dpdk}+FF_{dpdk})^T - FF^T_{dpdk}))^L$, where AS_{dpdk}, AF_{dpdk}, FS_{dpdk}, FF_{dpdk}, represent the number of success and failure paths in DPDK's memory allocation and failure, and T is a DPDK parameter that must be positive.

We note that the number of paths can change based on programmer decisions: using Boolean expressions rather than conditionally executed code can lower the number of paths, such as writing `x = c ? y : x;` instead of `if (c) { x = y; }` in C. We could have used this to bring down the number of paths in TinyNF's transmission function to 4, without any exponent regardless of the number of output links, but chose not to as such code is compiled to conditional move instructions which have poor tail latency on our machines.

In terms of effort to get run-time performance, since we had to modify the policer code to use TinyNF, we wanted to see whether the same performance benefits could be obtained without code changes. We wrote a compatibility layer that implements some of the DPDK API on top of TinyNF. The layer cannot implement all of the DPDK API, by design, but can replace DPDK for functions that fit the TinyNF model by changing an environment variable at compile time. The compatibility layer allows for 1% more maximum throughput than batched DPDK, at the cost of increased latency.

5.7 Summary

In summary, we showed in this chapter that simplifying network card drivers to only what is necessary for a specific set of network functions, namely those that process packets one-by-one, has benefits not only in terms of enabling automated verification but also performance. These benefits are linked: automated verification requires the code to have fewer paths and ideally simpler ones, and this means there are fewer branches to take and fewer instructions to execute. The fastest code is the code that does not need to run. Interestingly, we beat the performance of DPDK on real-world workloads despite DPDK using advanced CPU features such as vector instructions, which require manual work to write using C language extensions, whereas TinyNF is written in pure C with no such extras.

While TinyNF is only applicable to a subset of network functions, the same principle of specialization may apply to other network functions, since those are unlikely to need the full flexibility of a driver with a buffer pool. We leave this to future work.

6 Safe NIC drivers by construction with type invariants

This chapter is based on "Safe Low-Level Code Without Overhead is Practical, S. Pirelli, G. Candea, ICSE 2023".

In the previous chapter, we studied how verifiability helps performance. In this chapter, we study how it can help maintainability: verifiably safe high-level code does not need to sacrifice performance, yet is easier to read and maintain. In particular, there is no need to worry about accidentally introducing low-level vulnerabilities such as out-of-bounds buffer accesses.

Programming languages that provide type and memory safety eliminate entire classes of bugs and security vulnerabilities, yet unsafe languages remain in use even for new projects due to performance concerns. Safe language compilers try to avoid run-time checks if they can prove safety at compile-time but rely on heuristics. Compilers often do not have the information necessary to formally prove the safety of operations at compile-time. We study the necessity of "unsafety". Is there a practical way for languages to provide safety without run-time overhead?

By "safety" we mean memory safety and crash freedom: safe code never accesses memory it does not own and never performs invalid operations such as divisions by zero. Such safety prevents vulnerabilities such as Heartbleed [104].

6.1 Insight: Common unsafe operations match well-defined templates

Can developers give enough information to a compiler to prove at compile-time that all possibly unsafe operations are safe? We propose the use of *type invariants* as a solution to this problem, i.e., predicates that hold on all instances of a type. Type invariants are already used in practice, but we propose that languages should systematically provide them for each possibly unsafe operation. Consider the following C code:

```c
int* value;
bool set_value(int* v) {
  if (v == NULL) { return false; }
  value = v;
  return true;
}
```

The `set_value` function guarantees that value is non-null. Thus, a developer can use `value` once `set_value` has been called without having to write a safety check.

Consider the following C# code, which looks equivalent:

```
class Example {
  int[] value;
  Example(int[] v) {
    if (v == null) { throw new ...; }
    value = v;
  }
}
```

The same invariant holds: `value` is not null once the object is initialized. However, this information is not available to the compiler outside of the `Example` constructor, as it is not a type invariant of `int[]`. The compiler can only analyze signatures rather than implementations, since there are too many paths within implementations. Thus, it must insert a null check when value is used, as the signature lacks information to prove value is not null. C# developers can optionally annotate their code with nullability information, but the compiler cannot trust that the annotations are correct because they can describe arbitrarily complex conditions and thus must insert checks anyway.

Rust, however, does have an invariant for "non-null", which is in fact the default:

```
// Lifetime annotations omitted for brevity
struct Example {
  pub value: &[u32]
}
```

The member `value` in this struct is never null, thus the compiler does not have to insert null checks when it is used. Instead, the compiler enforces that any instance of Example is created with a non-null value. If a developer wants a possibly absent array, they can use Rust's `Option` type to wrap the array type, at which point the compiler requires a check before every use to ensure the Option indeed contains an array.

Rust enables safe code without null checks, but null checks are not the only kind of checks that are typically necessary in safe languages. Going back to the Rust struct above, indexing the array forces the compiler to insert a bounds check, unless the compiler can prove that the index is in bounds, for instance in a loop.

Type invariants can help avoid bounds checks just as they do with null checks. Consider the following Ada declarations:

```
-- Ranges are inclusive in Ada,
-- i.e., 0 <= n <= 9
type Small is range 0 .. 9;
type SmallArray is array(Small) of Integer;
```

The first declaration is for a ranged integer, and the second is for an array indexed by that range. Because this information is contained within the type's invariant, an Ada compiler can omit bounds checks when a `SmallArray` is indexed by a `Small`, as any value of type `Small` must be in bounds. The developer or the compiler must instead insert checks when converting from an integer with a different range, since the integer may not be in the target range's bounds.

Safety checks when converting a value of a general-purpose type to a type with an invariant are not overhead compared to C, because when writing C, developers must write these checks manually. For instance, checking that some user input is smaller than the size of an array is necessary if the input is used to index that array later. But in languages without type invariants, this check is only recorded in the implementation of some function and in the developer's memory. With type invariants, compilers can use signatures to know that a check was performed and avoid generating pointless run-time checks.

Programming languages can thus enable safe code without overhead as long as they only permit unsafe operations on types with invariants that trivially prove safety. Converting a piece of untrusted data to a type with the right invariant is only needed once, and it is not overhead since unsafe code must also convert and potentially reject the input. Once a value is converted to a type with an invariant, some operations on that value return a value with that same invariant, while others need a conversion. For instance, dividing a value of type "integer modulo N" by 2 returns a value of the same type, since the resulting value must mathematically be below N. However, incrementing that integer returns a value with a less strict type invariant, since the value can exceed N. Converting the result back to its original type code can be implicit given hardware instructions that match. For instance, incrementing an "integer modulo 256" can be done with a "byte increment" instruction.

Type invariants also enable compilers of safe languages to avoid memory overheads in the same way a programmer would in an unsafe language. Consider again Rust's `Option` type, which represents "a value, or nothing". Since Rust references cannot be null, per their type invariant, the compiler can use the memory representation of "null", e.g., all-zeroes, to represent an empty `Option` of a reference type, instead of using a separate Boolean value to track whether the option contains a value. Rust similarly has "non-zero" integer types with an invariant enabling the compiler to optimize the representation of `Option`.

6.2 Design: Type invariants for safety

We present below a systematic overview of type invariants that modern languages need to provide safety without run-time overhead. This list matches the potentially unsafe operations exposed by modern programming languages for low-level code: using references, dividing integers, and indexing arrays. It may not be exhaustive for future languages that could add more potentially unsafe operations.

Valid references. To avoid overhead, memory safety must be a type invariant, i.e., references that always point to valid memory without the need for run-time bookkeeping. This means references cannot be null and cannot require extra work such as garbage collection to ensure memory is properly freed.

One way to implement this is through memory arenas: by allocating a group of objects together, each object can refer to any other object in the same arena without overhead, and the arena is freed as a whole, at which point accessing any of its contents is a compile-time error.

C# implements a limited form of memory arenas: it supports special references that can only point to types on the stack and cannot be stored on the heap, thus there is one arena for the stack, with compile-time checks and no run-time overhead, and one arena for the heap, without compile-time checks but with the run-time overhead of garbage collection.

Rust implements valid references through its more complex ownership model, which enables more scenarios than C#'s stack-only references but requires more developer effort. The lifetime of Rust references must often be specified explicitly.

Ranged integers. To avoid overhead, safe languages must provide integer bounds as type invariants, including bounds not known at compile-time such as the number of network cards. This enables compilers to prove the absence of division by zero.

Most languages already provide a few specific bounds that correspond to machine types, such as C's `uint16_t` for integers between 0 inclusive and $2^1$6 exclusive. This enables compilers to translate operations in these types to the equivalent machine operations, such as addition of 16-bit integers. However, some high-level languages have no ranged integers at all. In Python, for instance, all integers are mathematical integers of infinite range, thus all integer operations incur the overhead of general-purpose mathematical integer arithmetic. Even with a fast path for "sufficiently small" integers, the code has the overhead of checking if the fast path applies.

Ranged arrays. To avoid overhead, safe languages must provide arrays that encode their length, i.e., the range of valid indices, in their type. In combination with ranged integers, this enables a compiler to prove that an array access is valid at compile-time instead of requiring run-time checks.

Array ranges must be allowed to have bounds not known at compile-time, as with integer ranges. For instance, a variable must be allowed to contain the index of the network card that was last queried for new packets and be used to index an array of network cards, even though the number of network cards is machine-dependent and unknown at compile-time. Rust in particular does not meet this criterion: its array types can include their length, but only if the length is a compile-time constant.

All three of our proposed invariants enable developers to give information to the compiler explicitly and formally instead of keeping it in their head. Developers in C must keep track of which references are guaranteed to be valid and which are not, of what range each integer variable has, and of what range each block of allocated memory can be indexed with. For ranges that are not compile-time constants, this is typically done by making the lower bound 0 and storing the upper bound in a variable, as a compiler would for a ranged array or integer.

6.3 Implementation

We write safe drivers in Ada, Rust, and C#, and a baseline unsafe driver in C. To do so, we extend Rust and C# to add limited forms of the type invariants we described.

In Ada, the only issue we have is that ranged array types by default carry both upper and lower bounds, even if the latter is 0, requiring extra memory. We use a workaround suggested by the AdaCore developers [1] to store only the upper bound.

In Rust, we implement a limited form of ranged arrays by using existing compiler support for eliding checks if an access is performed with an integer of a small enough type, such as indexing an array with 256 elements using an 8-bit unsigned integer. This causes small memory overheads for arrays that do not need 256 elements but must have them to benefit from bounds check elision, which we find acceptable for a prototype.

Interestingly, Rust's "aliasing XOR mutability" model is too restrictive for device drivers. Some driver instances must share state, such as a buffer pool used by both receive and transmit queues. Both queues must be able to mutate the pool: receive queues take buffers and transmit queues give buffers back. However, in safe Rust, the queues cannot simultaneously have a mutable reference to the pool. Furthermore, drivers need to use volatile reads and writes for memory-mapped I/O, since the network card can change its data at any time. The card logically has a mutable reference to its own data. But in Rust's model, if the card can mutate itself, then nobody else should be allowed to even access it. The solution used by the Rust version of the Ixy [30] network driver is to use unsafe Rust code, forgoing safety guarantees. Instead, we write our own type of reference that internally uses unsafe code but exposes a safe interface:

```
pub struct LPtr<'a, T> {
  ptr: NonNull<T>,
  _lifetime: PhantomData<&'a mut T>,
}
impl<'a, T> LPtr<'a, T> {
  pub fn new(src: &'a mut T) -> LPtr<'a, T>;
  pub fn read_volatile(&self) -> T;
  pub fn write_volatile(&self, value: T);
  pub fn map<U, F>(&self, f: F) -> U
    where F: FnOnce(&mut T) -> U;
}
```

Our custom `LPtr` type enforces lifetime, and thus memory safety, but not ownership, leaving correctness to developers. It has a field of type `PhantomData`, a special Rust type that evaporates during compilation and only serves as a marker for data lifetime in niche scenarios such as implementing a reference. In addition, we implement an array type with lifetime but not ownership, matching our `LPtr` type.

In C#, we use the same bounds check elision trick as in Rust, with the same overhead, but we first extend the language with ranged arrays. The C# maintainers have prototyped such arrays already [22], but they are not yet part of the language. We also add support for arrays of stack-only references.

6.4 Evaluation: Completeness

There are two facets to the completeness of this chapter: how broadly applicable this is to low-level systems code, and what kind of languages can be used.

In terms of systems, while we have no solid proof that these findings generalize, we believe that network drivers are complex enough to be representative of low-level systems code that uses unsafe code for performance but would benefit from safety. The use of buffer pools in the DPDK model means the driver uses a data structure. Data structures are too complex for current automated network software verification techniques, which must therefore assume their correctness, as we saw in previous chapters. In the case of a buffer pool, the code typically accesses a single element of a buffer array at a time, preventing the use of check elision heuristics that apply to loops over entire arrays.

We also cannot be 100% complete in making low-level systems safe since some low-level operations are inherently unsafe. The drivers require unsafe operations during initialization. They must query the PCI metadata of the network card to know where the card's registers are mapped in memory, then convert the integer read from this metadata into a pointer to the block of registers. This is done by reading and writing to PCI metadata using port-mapped I/O then asking the OS to map the space of the network card's registers into virtual memory. This is unavoidable given the hardware interface, though this unsafe code can be handled in the OS instead, as in Singularity [46]. The driver must also request "pinned" memory from the OS, i.e., memory whose physical address will not change, then ask the OS for said physical address. This enables such memory to be used by the network card, which uses physical addressing. No OS interactions are needed after initialization, and performance concerns anyway discourage such interactions during packet processing.

In terms of languages, we explicitly did not choose any interpreted languages, even those that can be compiled to native code such as Python using PyPy. These languages have features with overhead by default, such as Python's unbounded "big integers". Such overhead is not a key issue for languages that are designed to be interpreted, since interpretation already sacrifices speed.

There are plenty of interesting research languages we could have used, but they are typically not designed to be as usable as mainstream languages, and they are not always maintained. For instance, we could have chosen a solver-aided language such as Dafny [69], but it requires manual effort to write proofs in addition to writing code. We could have chosen a research language such as Sing#, a C# extension used for Singularity [46], but it is not maintained thus its compiler may be hard to extend and not include modern optimizations. While Rust is also the subject of formal methods research [53] to avoid bugs in unsafe Rust, developers do not need to know this to write safe Rust.

6.5 Evaluation: Run-time performance

Do our safe drivers impose run-time overhead? We benchmark our safe drivers to ensure their performance matches our baseline in practice, since checking the exact equivalence of assembly code is not practically feasible. We use Clang 13 for C, Rust 1.58, .NET 6, and GNAT 11 for Ada. We intended to use GCC for C, but it produces a slower binary than Clang, as we describe later. For C# we use ahead-of-time compilation, which produces the same code as the default just-in-time compilation, so that we can inspect the assembly code that we run.

We use two machines in a setup based on RFC 2544 [11]: a "device under test" runs the driver under test and a "tester" runs the MoonGen packet generator [29]. Both machines run Ubuntu 18.04 on two Intel Xeon E5-2667 v2 CPUs with power-saving features disabled and have two Intel 82599ES NICs. We use only one Ethernet port per card to ensure PCIe bandwidth is not a bottleneck. We measure throughput using minimally sized packets: 64 bytes of content plus 20 bytes of Ethernet framing. We transmit such packets for 30 seconds at a configurable rate. Our drivers are single-threaded. We set the CPU frequency to 1.5 GHz instead of the default 3.6 GHz of this CPU model, because otherwise our drivers saturate the links and exhibit identical performance. We write "forwarder" programs on top of the drivers that modify the source and destination MAC addresses of each packet and forward it to the opposite card. We measure the highest throughput at which the drivers do not drop packets, then we measure the latency of packets in increments of 1 Gb/s from 0 to the maximal loss-free throughput. This is the same benchmark we used to evaluate the original TinyNF driver.

We first benchmark the drivers for the "restricted" model, in which a single driver instance combines reception and transmission to share data structures and thus minimize overhead. This model can only be used by network software that handles packets one by one without reordering them. We show the results in Figure 6.1, which we already previewed at the beginning of this paper. The results are nearly identical for the different drivers, except that the Rust version sustains more throughput. These results are consistent across different benchmark runs. The latency bump around 11 Gb/s is the same as with the original TinyNF benchmarks, and reproduces with DPDK and Ixy, thus it is likely a hardware issue.

We then benchmark the drivers for the "flexible" model, in which reception and transmission use a shared buffer pool. This model supports all network software, including those that must reorder packets. We show the results in Figure 6.2. C# is still close to C, with slightly higher latency at the highest load. Rust keeps its advantage compared to C, though it has higher latencies at lower loads. Surprisingly, the Ada version can sustain higher throughput not only compared to the other drivers of the same model but even to the drivers of the restricted model, albeit with higher latency than the restricted model. We double-checked the Ada code to ensure it performs the same tasks in the same order including volatile reads and writes. The only explanation we can find is that, to remove bounds checks from the Ada code, we use more specific types than in other languages. In particular, we use integers bounded to the batch size for reception and to the number of received packets for transmission. The Ada compiler may produce better code when using Ada's bounded integers, since they provide additional information.

We investigate further by writing drivers using the "restricted" model but with a static output count. That is, instead of determining the number of network cards at run-time, this number is known at compile-time using the preprocessor in C, constant generic parameters in Rust, and generic values in Ada. C# has no such feature. This requires recompiling the source code for each deployment, but it could be practical for network software that intrinsically has a fixed output count. For instance, a firewall could have "internal" and "external" virtual devices and be chained with a router whose number of outputs depends on the actual number of physical devices. In theory, using static output counts gives compilers more room for optimizations, such as unrolling loops. We show the results in Figure 6.3. Ada reaches the same maximum throughput as the version with the flexible model but with a lower latency, which is expected given this restricted model. Rust performs almost identically to Ada, within the noise of small variations across experiments. The C version barely improves with a static output count. Since we used language features in Rust and Ada for this experiment but had to use the more general pre-processor in C, this provides more evidence that code using specific language features may enable compilers to optimize better.

To measure the impact of run-time checks on performance, we write a C# driver without our C# extensions. The compiled assembly code of this driver thus includes run-time checks, unlike other drivers we wrote. We compare it to our C driver using different compilers, to use the performance difference between compilers of the same language as a baseline. This leads to unexpected results, which we show in Figure 6.4. First, using GCC to compile our C driver leads to a larger performance penalty than we expected. We confirm by inspecting the assembly code that GCC spills many more values to the stack than Clang does. Second, the impact of run-time checks in the C# version is lower than the impact of using a different C compiler. In fact, the C# version with run-time checks has similar performance to the C version, though the former's latency spikes near its breaking point in terms of throughput. These results suggest that run-time safety checks may have less impact on performance than the specific optimization choices made by compilers.

6.6 Evaluation: Manual effort

Does our approach lead to maintainable code? We evaluate the maintainability of our drivers qualitatively by inspecting their source code. To keep comparisons fair, we do not consider our language extensions, as they are only prototypes in unsafe code and thus require worse syntax to use than real extensions would. Overall, the answer is mixed: the code of our drivers is as

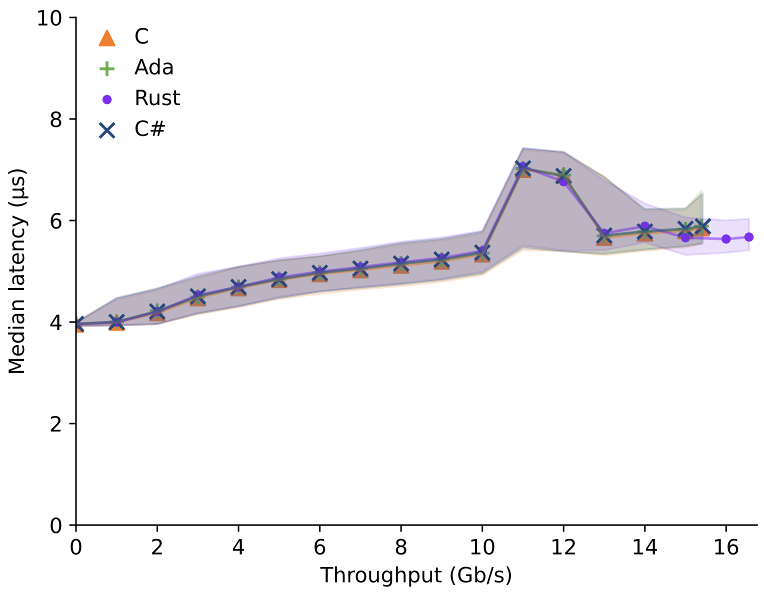


Figure 6.1: Throughput vs. latency until the drivers start dropping packets for our drivers using the “restricted” model, with a shaded 5-95% ranges for latencies.

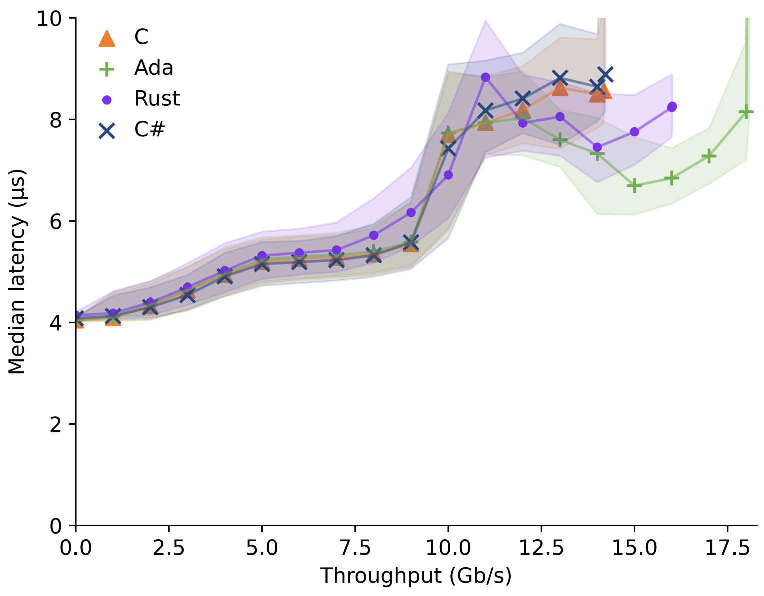


Figure 6.2: Throughput vs. latency until the drivers start dropping packets for our drivers using the “flexible” model, with a shaded 5-95% ranges for latencies.

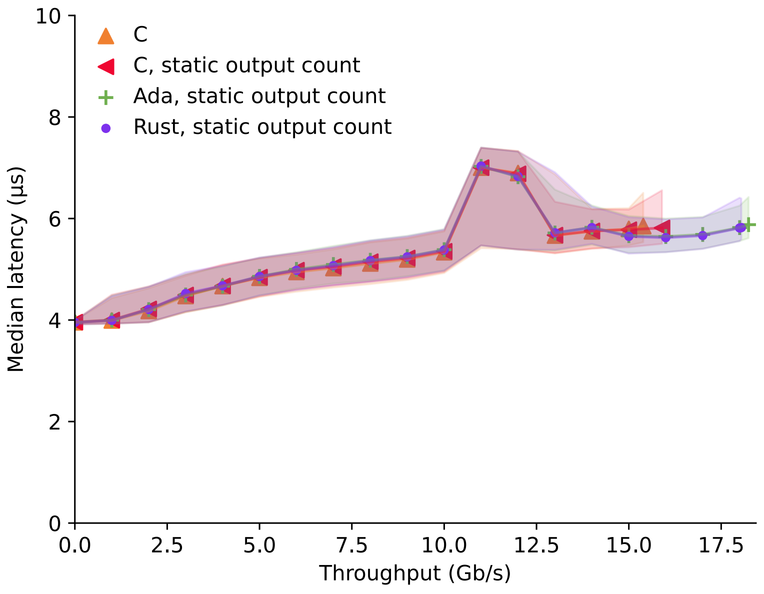


Figure 6.3: Version of Figure 6.1 using a static output count.

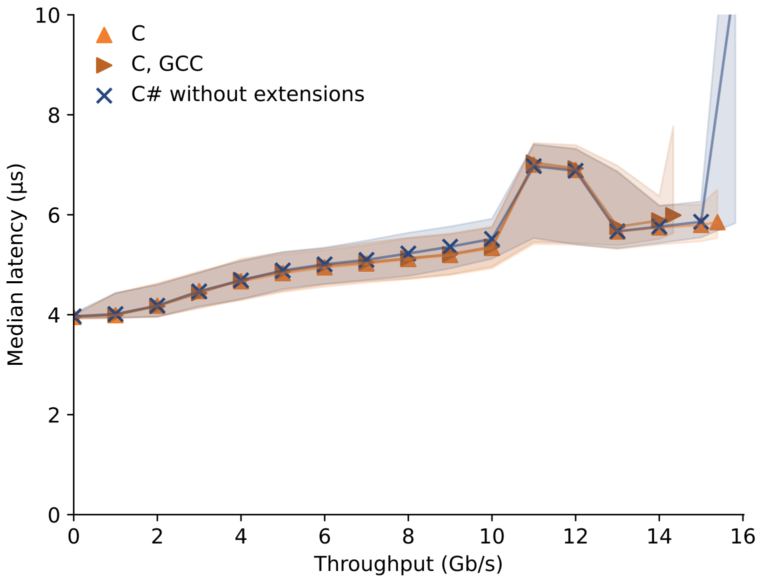


Figure 6.4: Version of Figure 6.1 using GCC for C and without our C# extensions.

easy to read as other code in the respective languages but writing and evolving it is difficult due to the lack of language and compiler support.

The source code of the final version of our drivers looks like normal code in their respective languages. Besides the use of our language extensions, there is little that would tip a reader that we carefully wrote the code to avoid overhead. It is possible to avoid overhead without having to write unusual or contorted code, but this does not mean it is easy.

Writing our drivers was difficult because current compilers were not designed to statically prove safety. Small changes in the source code cause large changes in the compiled assembly code. For instance, the GNAT Ada compiler accepts code that looks safe to developers but in fact requires safety checks at run-time due to language semantics and inserts checks without a warning nor any other means of “debugging” this behavior.

The process of writing safe code without overhead today, even in Ada, which has the necessary features already, is similar in theory to writing manual proofs of correctness but worse in practice because of the lack of tooling. Proofs typically read well once they are finished, yet slight changes in which lemmas are used and in which order can break the proof due to the proof checker being heuristic-based. Similarly, the code of our drivers reads well yet slight tweaks cause run-time checks as compilers can no longer prove the checks are unnecessary. However, whereas proof checkers enable developers to write intermediary assertions and provide information about why the proof process failed, compilers provide no way to access their internal logic. Without such information, writing safe code without overhead is tedious and slow. Developers must guess what source code might cause the compiler’s heuristics to output the desired assembly code and check that output after every change until they find a way to convince the compiler to not insert checks.

This situation is made worse by the impact on performance of minor differences in assembly code. We found such minor differences that caused performance regressions on the order of 10%. Thus, one must first spend time convincing the compiler to not insert run-time checks, then benchmark the code. If the code is not fast enough, one must convince the compiler again, this time using different code with the same semantics. Editing the assembly by hand is possible but unsafe and error-prone.

The safety of Ada, Rust, and C# did help us when writing drivers. For instance, we model network card registers as indexes in a buffer. We made copy-paste errors while translating C constants for register indexes, leaving some of them orders-of-magnitude too large. In C, such code silently does not work, as the OS maps more memory than we need. In safe languages, this fails with a descriptive error message. The type invariants we used also caught programming errors such as forgetting to initialize variables, whereas in C the lack of such invariants means that these errors cause compiler warnings at best.

6.7 Summary

In summary, we showed in this section that writing a simpler driver such as TinyNF in safe languages, or even a slightly more complex driver using a buffer pool, does not necessarily require performance tradeoffs. By enabling compilers to automatically prove the safety of the driver at compile-time, there is no need for run-time checks, and thus no run-time performance penalty.

However, it remains to be seen whether different and more complex systems can also be written in safe code without overhead, or whether more challenges await.

7 Limitations

Generalizability to other systems is the main open question and possible limitation of this thesis. We focused on software network functions and their network card drivers, which exhibit common low-level systems code patterns. However, there may be other challenges in other low-level systems that we did not run into, especially around shared-memory parallelism. Good software engineering practices encourage structured ways to deal with parallelism, such as message passing, which is more maintainable and also easier to verify, in a similar manner to how verifiability for TinyNF also had other benefits. However, there may be concurrent systems for which message passing is not fast enough and our techniques cannot handle the necessary low-level mechanisms. In the context of software network functions, shared-nothing parallelism can often work, though with the risk of skew when sharing resources such as the TCP/UDP ports a NAT can allocate.

In general, the same principles used in this thesis should lead to interesting solutions: find the right abstractions for a tool to deal with the specific kind of program under consideration, and write the code using these abstractions. For instance, while TinyNF requires packet-by-packet processing, there are likely useful abstractions to deal with packet reordering that do not require a tool to handle the general case of a buffer pool.

One potential hurdle with some low-level systems is the specific kinds of operations necessary to communicate with hardware. For instance, the operations done on a network card are quite different from those on page tables, which are themselves quite different from a CPU initialization sequence. There is a risk of creating many one-off tools that cannot talk to each other. This may be mitigated by overarching frameworks such as formally modeling assembly code in the K framework [23].

Recursion is a construct none of the techniques introduced in this thesis handle. While some recursion can be easy to rewrite as a loop, and some such loops can be handled by our summarization techniques, many algorithms are easier to write recursively. New techniques are necessary to automatically summarize recursive calls, possibly by translating them to loops inside of a tool rather than requiring the developer to do it.

The main reason we did not handle recursion is that it is typically associated with functional programming languages, and low-level systems are typically not written in these languages. Thus, handling recursion was not a priority for us. However, this does not mean recursion is a bad idea, or that low-level systems should not be written in functional languages; after all, the initial prototypes of seL4 [60] were written in Haskell.

Debuggability is a general problem across the techniques introduced in this thesis, which we did not handle in part because it requires human-computer interaction research outside of the scope of the thesis. For instance, while KLINT can always obtain a counter-example if a network function does not satisfy its specification, this counter-example may be hard to understand for developers, especially if they did not themselves write the contract that translates the data structures they use into maps KLINT sees. Similarly, debugging the reason why a compiler inserted a run-time check into our safe drivers is little more than guesswork. Intuitive answers often appear to not work for unknown reasons, since the heuristics deployed by compilers are not meant to be explainable.

Why not write more drivers? We “only” wrote two kinds of drivers for one kind of network card in this thesis. The reason we did not do more is the high engineering cost of writing a driver today, mainly due to the lack of details about hardware and the errors present in the details that are available. Even the data sheet for the Intel 82599 NIC we used, which looks rather complete at first glance, contains its fair share of typos, contradictory information, information spread across multiple seemingly-unrelated parts, and outright incorrect data. In particular, looking at the official Intel driver for this network card, one can immediately spot operations whose goal and impact are undocumented. Our driver works despite not performing these operations, but a production-ready implementation would need to understand exactly why.

Furthermore, writing a driver for the 82599 mostly consists of initializing a lot of hardware registers to specific values, and debugging typos in the data sheet or one’s own code that cause the hardware to not respond after this initialization. Thus, writing many drivers for a research project is a much larger scope than it ideally should be, which also explains why there are not many research projects writing such drivers. Most systems for fast networking, such as ClickOS [71], DPDK [27], netmap [97], SoftNIC [43], and IX [5], reuse existing drivers, which are bottlenecks on their performance. Arrakis [89] uses custom drivers but focuses on interrupt-driven I/O, which strikes a different tradeoff. Ixy [30], is the only research driver we know of besides ours. It is odd to have more research operating systems than drivers: the former are by definition more complex as they contain at least one driver.

8 Related Work

In this section, we recap the existing themes and work this thesis builds upon. We refer the reader back to Chapters 1 and 2 for more details on the most important aspects necessary for this thesis.

Symbolic execution, as described by King [59], is a program analysis technique that automatically enumerates all paths through a piece of code, by running the code with *symbolic* rather than concrete inputs. When it encounters a branch, a symbolic execution engine forks execution into two states corresponding to taking and not taking the branch. In the presence of unbounded loops, symbolic execution does not terminate in its naive form since an engine will create an infinite number of states. Not all states are *feasible*, and a solver must check whether the conditions that guard a state can all be true at the same time before reporting a code bug in the state. Well-known symbolic execution engines include KLEE [14] for LLVM [68] bitcode and angr [102] for assembly code.

Path explosion is the key problem facing symbolic execution [7]: the number of paths can be so large that, even if it is bounded, an engine cannot finish in reasonable time. However, before the work presented in Chapter 4 of this thesis, we are not aware of precise metrics to measure path explosion. Metrics such as cyclomatic complexity [72], NPATH complexity [80] and asymptotic path complexity [3] exist, and the latter can predict path explosion [6], but they remain imprecise.

SMT solvers, such as Z3 [24] and CVC5 [4], are at the core of techniques such as symbolic execution. SMT stands for "Satisfiability Modulo Theories" and is thus a superset of SAT, the satisfiability problem on Booleans. An SMT solver takes as input a Boolean formula in a given theory, such as bitvectors or quantifier-free linear integer arithmetic, and outputs either "satisfiable", meaning the solver has found an assignment for all variables such that the formula holds, "unsatisfiable", meaning the solver has proven that no such assignment exists, or "unknown". Some theories, such as quantifier-free bitvectors, are *decidable*, meaning that a correct solver will always answer either "satisfiable" or "unsatisfiable" in bounded time. This does not mean the answer will be quick, since "five years" is an example of bounded time. Some theories, on the other hand, are *undecidable*, meaning a correct solver will have to answer “unknown” for some queries. This does not mean the theory is unusable, since the solver may still be able to solve formulas given to it in practice, even if it cannot solve all possible formulas.

Automated verification is a general area of work that covers exhaustive symbolic execution and more. The goal of automated verification is to formally prove that a piece of code fulfills its specification. This specification may be complex, possibly involving multiple runs of the code, or may be as simple as “don’t divide by zero”. Because symbolic execution engines explore all possible paths in a piece of code, tools such as Vigor [119] and this thesis’s KLINT can use symbolic execution to automatically verify the correctness of a piece of code by checking that the specification holds on each individual path. The main obstacle to automated verification is the presence of code patterns that a tool cannot handle, such as arbitrary pointer arithmetic in low-level code. For instance, if a piece of code obtains an integer from the user, casts this integer to a pointer, and writes more user input to the pointer’s location, a tool must be conservative and assume that essentially anything could have happened, including behaviors that are obvious bugs such as overwriting the program stack’s return address to an arbitrary location.

Theorem provers, such as VeriFast [49] and Coq [19], let developers “hand-hold” a solver by providing lemmas, assumptions, contracts, and other annotations. This extends the reach of verification, at the cost of manual effort. It also requires expertise in verification from developers, since proof languages are at a higher level than most programming languages, and the way to make a solver understand why a proof is correct is not always obvious to beginners.

Bug finding is a major area of computer systems research, whose goal is as its name implies to find bugs in systems. Unlike verification, it is not necessary to find non-buggy behaviors, and in fact the goal of a bug-finding tool is typically to explore as few paths as possible to find a bug. KLEE [14] is a symbolic execution engine designed to find bugs, which involves guiding the engine towards paths that are more likely to contain bugs and away from paths that are more likely to re-explore already explored behavior. A more “brute force” way to find bugs is *fuzzing*, in which tools such as AFL [118] generate random inputs and check if running the code with these inputs causes a crash. Because purely random inputs are unlikely to go far, since they are probably not even valid representations of any complex format such as compressed archives, one can combine fuzzing with symbolic execution as in SAGE [9] to generate inputs in a “white box” way that takes program structure into account. Bug finding can also be done for hardware, as in MEFISTO [52].

Runtime verification is a related but distinct area of work, focusing on checking behavior at runtime. This catches bugs even in code currently beyond the reach of formal verification, for instance due to complex parallelism. Tools such as Aragog [117] trade completeness and performance for applicability. They only look at input and output packets and view network functions as “black boxes”, thus they impose no constraints on code. But they cannot guarantee the absence of bugs, and in distributed environments cannot prevent bugs that can only be detected after compiling information from different machines. They also impose runtime overheads, unlike tools such as KLINT, due to the runtime nature of checks.

Software network functions, also known as *middleboxes* [15], are a type of software that processes network packets. For instance, *firewalls*, which ensure that "undesired" packets from the outside world are not forwarded inside a private network, are a kind of network function, and so are NATs, routers, and bridges. Unlike their hardware counterparts, software network functions are more flexible since they can be updated easily and chained together in arbitrary ways thanks to virtualization [86]. Network functions are a critical part of Internet infrastructure, since they must function correctly for packets to arrive where they are meant to arrive. If a network function starts dropping packets that it should forward, or modifying parts of packets that should be left untouched, end hosts are unlikely to fulfill their objective.

Network card drivers are the lowest level in the software network function stack. A network card driver, as its name implies, drives the behavior of the network card: initializing it, configuring the necessary mechanisms to apply desired policies, and exchanging information with it to receive and transmit packets. Drivers are typically found within operating systems such as Linux or Windows, but they can also run as normal programs in user space either because the operating system is a microkernel such as seL4 [60] or because the program wishes to bypass the kernel entirely to minimize overheads as in DPDK [27]. While user space drivers must be given an entire network card without apparent isolation, the operating system may use modern hardware features to *virtualize* network cards and thus give each user space driver the impression that there is an entirely separate physical card while preserving the benefits of kernel-bypass direct access. Writing a driver from scratch is rare, e.g., the DPDK driver is an adaptation of the Linux one. One example of such a driver is Ixy [30], and another is TINYNF presented in this thesis.

Network function verification tools such as Vigor [119], Gravel [120], and Prevail [37], which require source code, all inspired our design. Bolt [48] and Pix [47] verify performance instead of correctness, and require source code, though we believe they could use our techniques to only require binaries. While verifying binaries provides key advantages, verifying source code makes debugging failed verification easier since compiler optimizations can make it hard to match what the binary does wrong to what the code does wrong.

Network verification, distinct from network function verification, deals with verifying the behavior of entire networks given *models* of individual components, as in BUZZ [34]. For instance, checking that no packet can ever enter an infinite forwarding loop is desirable.

BPF is a more applicable but weaker form of network function verification: an in-kernel "verifier" checks memory safety and crash freedom, allowing untrusted code to run in kernel mode for performance without safety risks. BPF verifiers are fast, at the cost of restricting the code developers may write. BPF code cannot contain unbounded loops, must use specific data structures, and must include explicit checks for out-of-bounds memory accesses, even if a “smarter” and slower verifier might not need these checks. Internet infrastructure companies such as Cloudflare [70] use BPF as a core part of their infrastructure. Prevail [37] showed that formal methods can help BPF verification scale. Verified interpreters [113] and just-in-time compilers [82] for BPF exist, but they make no promises about functional correctness.

Using maps to analyze programs has been proposed before, though previous approaches were not aimed at functional verification, such as the Memsight [18] memory model for symbolic execution, or the technique proposed by Dillig et al. [25] to verify memory safety. BPF, mentioned above, uses maps extensively and thus the BPF verifier must analyze them.

Performance verification is a related but distinct area of work from correctness verification. Tools such as Bolt [48] can verify that a piece of code has expected performance characteristics, either in terms of assembly instructions or in terms of cycles given hardware models. Freud [98] is another such example.

9 Conclusion and Future Work

In summary, we addressed some of the challenges exhaustive symbolic execution currently faces in the context of formally verifying software network functions, i.e., data structures and loops. Specifically, we proposed *maps* as a unifying abstraction to let a symbolic execution engine reason about data structures, including heap memory, and loops.

Using our abstractions, we were able to automatically verify the correctness of software network functions that previously required a large amount of hand-written annotations. Furthermore, future network functions can also be verified more easily without writing more annotations.

We also provided evidence that modifying software to make it automatically verifiable is not necessarily a burden since it also improves performance, safety, and maintainability. Specifically, we wrote a driver for a widely-used network card, formally verified it with symbolic execution, and showed that one can translate this driver in safe languages without compromising on performance as long as the languages support basic type invariants.

Looking ahead, we see an exciting body of work in combining these two areas of work: developers should be provided with abstractions that both make their code automatically verifiable and help compilers produce fast binaries.

Extending the definition of "network function" is a clear next step for this line of work. For instance, an HTTP REST API server could be considered a kind of network function operating on messages within the HTTP protocol rather than within Ethernet or Wi-Fi. One may then be able to use similar techniques to those proposed in this thesis to formally verify the correctness of, say, EPFL's authentication system.

Extending the NF model would likely lead to interesting challenges such as how to deal with the kind of string operations used in common protocols such as HTTP. As always, the general case is likely infeasible, but the specific kinds of operations are much more constrained in practice, and a tool could be built to handle these operations and not the general case.

Storage stacks are related but not quite similar to networking stacks in terms of structure. Indeed, when reading the documentation for modern storage hardware such as NVMe one cannot help but notice the same underlying structure of descriptor rings with requests and responses. We briefly considered them during this thesis but we did not have concrete application domains at hand for a faster or simpler storage driver.

Should a programming language enforce verifiability? A possible consequence of the techniques proposed in this thesis may be to design a new programming language that only allows developers to write code statically known to be verifiable, such as with a restricted kind of looping construct and specific type invariants to provably avoid run-time checks.

This does not preclude adding syntactic sugar on top of such a language for "please insert a run-time check here, I cannot prove this", akin to Rust's `?` macro and C#'s `?.` operator. However, any use of such syntactic sugar should be an explicit developer choice, which can be noticed and audited in a code review, rather than an implicit decision the language encourages.

Bibliography

[1] Ada RFC: Lower bound constraint (accessed jan 8, 2024). https://github.com/AdaCore/ada-spark-rfcs/blob/master/prototyped/rfc-fixed-lower-bound.rst.

[2] ANGR CONTRIBUTORS. angr issue 938: SEGFAULT libz3 (accessed jan 8, 2024). https://github.com/angr/angr/issues/938.

[3] BANG, L., AYDIN, A., AND BULTAN, T. Automatically computing path complexity of programs. In *Intl. Symp. on the Foundations of Software Engineering (FSE)* (2015).

[4] BARBOSA, H., BARRETT, C. W., BRAIN, M., KREMER, G., LACHNITT, H., MANN, M., MOHAMED, A., MOHAMED, M., NIEMETZ, A., NÖTZLI, A., OZDEMIR, A., PREINER, M., REYNOLDS, A., SHENG, Y., TINELLI, C., AND ZOHAR, Y. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (2022), D. Fisman and G. Rosu, Eds., vol. 13243 of *Lecture Notes in Computer Science*, Springer, pp. 415–442.

[5] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2014).

[6] BESSLER, G., CORDOVA, J., CULLEN-BARATLOO, S., DISSEM, S., LU, E., DEVIN, S., ABUGHARARH, I., AND BANG, L. Metrinome: Path complexity predicts symbolic execution path explosion. In *Intl. Conf. on Software Engineering (ICSE)* (2021).

[7] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[8] BORNHOLT, J., JOSHI, R., ASTRAUSKAS, V., CULLY, B., KRAGL, B., MARKLE, S., SAURI, K., SCHLEIT, D., SLATTON, G., TASIRAN, S., VAN GEFFEN, J., AND WARFIELD, A. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *ACM Symp. on Operating Systems Principles (SOSP)* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 836–850.

[9] BOUNIMOVA, E., GODEFROID, P., AND MOLNAR, D. Billions and billions of constraints: Whitebox fuzz testing in production. In *Intl. Conf. on Software Engineering (ICSE)* (2013), pp. 122–131.

[10] BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. What's decidable about arrays? In *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation* (2006).

[11] BRADNER, S., AND MCQUAID, J. Benchmarking methodology for network interconnect devices. RFC 2544, RFC Editor, 1999.

[12] BRUNELLA, M. S., BELOCCHI, G., BONOLA, M., PONTARELLI, S., SIRACUSANO, G., BIANCHI, G., CAMMARANO, A., PALUMBO, A., PETRUCCI, L., AND BIFULCO, R. hxdp: Efficient software packet processing on FPGA NICs. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[13] BRUNI, A., DISNEY, T., AND FLANAGAN, C. A peer architecture for lightweight symbolic execution. http://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf, Unpublished.

[14] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2008).

[15] CARPENTER, B., AND BRIM, S. RFC 3234: Middleboxes: Taxonomy and issues, 2002.

[16] CHIPOUNOV, V., GEORGESCU, V., ZAMFIR, C., AND CANDEA, G. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems (HOTDEP)* (2009).

[17] COOK, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1971), STOC '71, Association for Computing Machinery, p. 151–158.

[18] COPPA, E., D'ELIA, D. C., AND DEMETRESCU, C. Rethinking pointer reasoning in symbolic execution. In *ACM Intl. Conf. on Automated Software Engineering (ASE)* (2017).

[19] Coq. http://coq.inria.fr/.

[20] CORBET, J. Bounded loops in BPF programs (accessed jan 8, 2024). https://lwn.net/Articles/773605/.

[21] CORBET, J. The BPF system call API, version 14 (accessed jan 8, 2024). https://lwn.net/Articles/612878/.

[22] .NET runtime pull request: Valuearray - a compliment type to the span, which owns its data without indirections (accessed jan 8, 2024). https://github.com/dotnet/runtime/pull/60519.

[23] DASGUPTA, S., PARK, D., KASAMPALIS, T., ADVE, V. S., AND ROŞU, G. A complete formal semantics of x86-64 user-level instruction set architecture. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, p. 1133–1148.

[24] DE MOURA, L. M., AND BJØRNER, N. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[25] DILLIG, I., DILLIG, T., AND AIKEN, A. Precise reasoning for programs using containers. In *Symp. on Principles of Programming Languages (POPL)* (2011).

[26] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2014).

[27] DPDK: Data plane development kit (accessed jan 8, 2024). https://dpdk.org.

[28] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2016).

[29] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F., AND CARLE, G. MoonGen: A scriptable high-speed packet generator. In *ACM Internet Measurement Conf. (IMC)* (2015).

[30] EMMERICH, P., PUDELKO, M., BAUER, S., HUBER, S., ZWICKL, T., AND CARLE, G. User Space Network Drivers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)* (Sept. 2019).

[31] ENGELEN, R. V. Efficient symbolic analysis for optimizing compilers. In *Intl. Conf. on Compiler Construction* (2001).

[32] EQUINIX. Network edge | Equinix edge services (accessed jan 8, 2024). https://www.equinix.se/products/digital-infrastructure-services/network-edge.

[33] FARSHIN, A., BARBETTE, T., ROOZBEH, A., MAGUIRE JR., G. Q., AND KOSTIĆ, D. PacketMill: Toward per-core 100-Gbps networking. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).

[34] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2016).

[35] FLANAGAN, C., AND GODEFROID, P. Dynamic partial-order reduction for model checking software. In *Symp. on Principles of Programming Languages (POPL)* (New York, NY, USA, 2005), POPL '05, Association for Computing Machinery, p. 110–121.

[36] FLANAGAN, C., AND LEINO, K. R. M. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. on Formal Methods Europe* (2001).

[37] GERSHUNI, E., AMIT, N., GURFINKEL, A., NARODYTSKA, N., NAVAS, J. A., RINETZKY, N., RYZHYK, L., AND SAGIV, M. Simple and precise static analysis of untrusted Linux kernel extensions. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2019).

[38] GODEFROID, P., AND LUCHAUP, D. Automatic partial loop summarization in dynamic test generation. In *Intl. Symp. on Software Testing and Analysis (ISSTA)* (2011).

[39] GOODIN, D. Critical vulnerabilities in exim threaten over 250k email servers worldwide. https://arstechnica.com/security/2023/09/critical-vulnerabilities-in-exim-threaten-over-250k-email-servers-worldwide/, 2023.

[40] GRAD, P. Homeland security warns of Windows worm (archived, accessed jan 8, 2024). https://web.archive.org/web/20210516133324/https://techxplore.com/news/2020-06-homeland-windows-worm.html.

[41] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow processing and the rise of commodity network hardware. *SIGCOMM Computer Communication Review 39*, 2 (mar 2009), 20–26.

[42] HABERMEHL, P., IOSIF, R., AND VOJNAR, T. What else is decidable about integer arrays? In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures* (Berlin, Heidelberg, 2008), FOSSACS'08/ETAPS'08, Springer-Verlag, p. 474–489.

[43] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[44] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *ACM Symp. on Operating Systems Principles (SOSP)* (October 2015), ACM.

[45] HAWKINS, A. J. Waymo has 7.1 million driverless miles – how does its driving compare to humans? https://www.theverge.com/2023/12/20/24006712/waymo-driverless-million-mile-safety-compare-human, 2023.

[46] HUNT, G., AND LARUS, J. Singularity: Rethinking the software stack. *Operating Systems Review 41*, 2 (2007).

[47] IYER, R., ARGYRAKI, K., AND CANDEA, G. Performance Interfaces for Network Functions. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2022).

[48] IYER, R., PEDROSA, L., ZAOSTROVNYKH, A., PIRELLI, S., ARGYRAKI, K., AND CANDEA, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2019).

[49] JACOBS, B., AND PIESSENS, F. The VeriFast program verifier, 2008.

[50] JACOBS, B., SMANS, J., AND PIESSENS, F. The verifast program verifier: A tutorial, Nov. 2017.

[51] JAFFAR, J., MURALI, V., NAVAS, J. A., AND SANTOSA, A. E. TRACER: A symbolic execution tool for verification. In *Intl. Conf. on Computer Aided Verification (CAV)* (2012).

[52] JENN, E., ARLAT, J., RIMEN, M., OHLSSON, J., AND KARLSSON, J. Fault injection into vhdl models: the mefisto tool. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing* (1994), pp. 66–75.

[53] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang. 2*, POPL (dec 2017).

[54] KAKARLA, S. K. R., BECKETT, R., MILLSTEIN, T., AND VARGHESE, G. SCALE: Automatically finding RFC compliance bugs in DNS nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association, pp. 307–323.

[55] KAPUS, T., AND CADAR, C. A segmented memory model for symbolic execution. In *Intl. Symp. on the Foundations of Software Engineering (FSE)* (2019).

[56] KAPUS, T., ISH-SHALOM, O., ITZHAKY, S., RINETZKY, N., AND CADAR, C. Computing summaries of string loops in c for better testing and refactoring. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2019).

[57] KERN, C., AND GREENSTREET, M. R. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst. 4*, 2 (apr 1999), 123–193.

[58] KINCAID, Z., BRECK, J., BOROUJENI, A. F., AND REPS, T. Compositional recurrence analysis revisited. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2017).

[59] KING, J. C. Symbolic execution and program testing. *Commun. ACM* (1976).

[60] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., KOLANSKI, K. E. R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *ACM Symp. on Operating Systems Principles (SOSP)* (2009).

[61] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash ≈ local flash. *SIGARCH Comput. Archit. News 45*, 1 (apr 2017), 345–359.

[62] Kogias, M., Iyer, R., and Bugnion, E. Bypassing the load balancer without regrets. In *Symp. on Cloud Computing (SOCC)* (2020).

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

[64] Kozen, D. C. *Rice's Theorem.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1977, pp. 245–248.

[65] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2012).

[66] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[67] LAN/MAN Standards Committee. IEEE standard for local and metropolitan area networks: Media access control (MAC) bridges. Tech. rep., IEEE Standards Association, 2014. IEEE Std 802.1D-2004.

[68] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization (CGO)* (2004).

[69] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)* (2010).

[70] Majkowski, M. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[71] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the art of network function virtualization. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2014).

[72] McCabe, T. J. A Complexity Measure.

[73] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference* (San Diego, CA, Jan. 1993), USENIX Association.

[74] Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal* (2014).

[75] Miano, S., Bertrone, M., Risso, F., Bernal, M. V., Lu, Y., Pi, J., and Shaikh, A. A service-agnostic software framework for fast and efficient in-kernel network services. In *ACM/IEEE Symp. on Architectures for Networking and Communications Systems* (2019).

[76] MITRE Corporation. MS13-064. Available from CVE Details, CVE-ID MS13-064., 2013.

[77] MITRE CORPORATION. CVE-2014-9715. Available from CVE Details, CVE-ID CVE-2014-9715., 2014.

[78] MITRE CORPORATION. CVE-2015-6271. Available from CVE Details, CVE-ID CVE-2015-6271., 2015.

[79] MOZILLA RESEARCH. Rust programming language (accessed jan 8, 2024). https://www.rust-lang.org/.

[80] NEJMEH, B. A. Npath: A measure of execution path complexity and its applications. *Commun. ACM 31*, 2 (1988).

[81] NELSON, L., BORNHOLT, J., GU, R., BAUMANN, A., TORLAK, E., AND WANG, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symp. on Operating Systems Principles (SOSP)* (2019).

[82] NELSON, L., GEFFEN, J. V., TORLAK, E., AND WANG, X. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[83] NELSON, L., SIGURBJARNARSON, H., ZHANG, K., JOHNSON, D., BORNHOLT, J., TORLAK, E., AND WANG, X. Hyperkernel: Push-button verification of an OS kernel. In *ACM Symp. on Operating Systems Principles (SOSP)* (2017).

[84] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 327–341.

[85] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. Use of formal methods at amazon web services. https://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf, 2014.

[86] Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[87] OPEN INFORMATION SECURITY FOUNDATION. Suricata website (accessed Jan 8, 2024). https://suricata.io/.

[88] OWENS, S., MYREEN, M. O., KUMAR, R., AND TAN, Y. K. Functional big-step semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632* (Berlin, Heidelberg, 2016), Springer-Verlag, p. 589–615.

[89] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T. E., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2014).

[90] PIRELLI, S., AND CANDEA, G. A simpler and faster NIC driver model for network functions. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[91] PIRELLI, S., AND CANDEA, G. Safe low-level code without overhead is practical. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), pp. 2173–2184.

[92] PIRELLI, S., VALENTUKONYTĖ, A., ARGYRAKI, K., AND CANDEA, G. Automated verification of network function binaries. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2022).

[93] PRIMORAC, M., ARGYRAKI, K., AND BUGNION, E. How to measure the killer microsecond. In *SIGCOMM KBNets Workshop* (2017).

[94] PWNMEOW. Spring4shell explained (accessed jan 8, 2024). https://www.hackthebox.com/blog/spring4shell-explained-cve-2022-22965.

[95] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. P. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2013).

[96] RFC 3234. http://www.ietf.org/rfc/rfc3234.txt, 2002.

[97] RIZZO, L. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conf. (USENIX)* (Boston, MA, 2012), USENIX Association, pp. 101–112.

[98] ROGORA, D., CARZANIGA, A., DIWAN, A., HAUSWIRTH, M., AND SOULÉ, R. Analyzing system performance with probabilistic performance annotations. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)* (2020).

[99] RUST AUTHORS AND COLLABORATORS. Issues - rust-lang/rust (accessed jan 8, 2024). https://github.com/rust-lang/rust/issues.

[100] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-extended symbolic execution on binary programs. In *Intl. Symp. on Software Testing and Analysis (ISSTA)* (2009).

[101] SHIROKOV, N., AND DASINENI, R. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer, May 2018.

[102] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symp. on Security and Privacy (S&P)* (2016).

[103] Slaby, J., Strejček, J., and Trtík, M. Compact symbolic execution. In *Intl. Symp. on Automated Technology for Verification and Analysis* (2013).

[104] Synopsys. Heartbleed (accessed jan 8, 2024). https://heartbleed.com/.

[105] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2 (1972), 146–160.

[106] Team, I. D. V. Dpdk intel nic performance report release 20.11. http://fast.dpdk.org/doc/perf/DPDK_20_11_Intel_NIC_performance_report.pdf, 2020.

[107] PMU Tools GitHub repository (accessed jan 8, 2024). https://github.com/andikleen/pmu-tools.

[108] Trtík, M., and Strejček, J. Symbolic memory with pointers. In *Intl. Symp. on Automated Technology for Verification and Analysis* (2014), pp. 380–395.

[109] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* (01 1937).

[110] Urban, C., Gurfinkel, A., and Kahsai, T. Synthesizing ranking functions from bits and pieces. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2016).

[111] Wang, C., and Lin, F. Solving conditional linear recurrences for program verification: The periodic case. *Proc. ACM Program. Lang. 7*, OOPSLA1 (2023).

[112] Wang, J., Lévai, T., Li, Z., Vieira, M. A. M., Govindan, R., and Raghavan, B. Galleon: Reshaping the square peg of NFV, 2021.

[113] Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., and Tatlock, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2014).

[114] Wikipedia Contributors. Log4shell (accessed jan 8, 2024). https://en.wikipedia.org/wiki/Log4Shell.

[115] Xie, X., Chen, B., Liu, Y., Le, W., and Li, X. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Intl. Symp. on the Foundations of Software Engineering (FSE)* (2016).

[116] Yan, L., Pan, Y., Zhou, D., Candea, G., and Kashyap, S. Transparent multicore scaling of single-threaded network functions. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)* (2024).

[117] Yaseen, N., Arzani, B., Beckett, R., Ciraci, S., and Liu, V. Aragog: Scalable runtime verification of shardable networked systems. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[118] ZALEWSKI, M. American Fuzzy Lop (accessed jan 8, 2024). http://lcamtuf.coredump.cx/afl/.

[119] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R. R., RIZZO, M., PEDROSA, L., ARGYRAKI, K. J., AND CANDEA, G. Verifying software network functions with no verification expertise. In *ACM Symp. on Operating Systems Principles (SOSP)* (2019).

[120] ZHANG, K., ZHUO, D., AKELLA, A., KRISHNAMURTHY, A., AND WANG, X. Automated verification of customizable middlebox properties with Gravel. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2020).

[121] ZINZINDOHOUÉ, J.-K., BHARGAVAN, K., PROTZENKO, J., AND BEURDOUCHE, B. HACL*: a verified modern cryptographic library. In *ACM Conf. on Computer and Communications Security (CCS)* (2017).