Research Statement

Yifei Yuan

My research interests lie in formal methods and networked systems. I am particularly interested in developing novel formal techniques to improve the reliability of computer networks.

Today, network operators need to implement a variety of functions (firewall, load balancing, intrusion detection, etc.) in order to ensure the availability, performance and security of networks. Recent innovations (e.g. software-defined networking, network function virtualization) offer new opportunities for network operators to program even more diverse and complex functions. Unfortunately, programming these functions correctly in networks is still challenging, error-prone, and time-consuming; and programming bugs in the implementations of these functions may cause significant financial loss, disappointing user experience, and even damaged company reputation.

To address these challenges, my research investigates real-world problems, formalizes those problems, develops rigorous solutions, and finally implements tools for practical use. In particular, my research focuses on two critical research questions in the lifecycle of network operations: first, how to develop high-level programming abstractions to simplify the programming of networks and thus reduce the manual programming errors; second, how to efficiently check the correctness of implementations of networks, especially those with complex stateful functions. To answer the first question, I built two innovative programming abstractions: 1) a programming-by-example paradigm called NetEgg [1, 4], and 2) a high-level language called NetQRE [5] for specifying functions with quantitative computations. In addressing the second question, I built 1) Mikado [2], an efficient testing framework, and 2) NetSMC [3], a symbolic model checker, to check the correctness of stateful networks. In what follows, I will elaborate on those works and discuss my future research.

High-level Programming Abstractions

Programming network functions can be tedious and error-prone for network operators given today’s low-level programming abstractions. To simplify the programming of networks, I developed the following two high-level programming abstractions in my PhD thesis work.

NetEgg. In 2015, I conducted interviews with 101 network operators and engineers, where I found that more than 80% of the interviewees have to (or will have to) program networks, while many of them faced the difficulties in programming reliable networks. In practice, however, network operators often prefer to use examples such as timing diagrams to describe the desired network functions and protocols, and then generalize these examples into implementations.

Motivated by this observation, I proposed and implemented an example-based programming tool, NetEgg [1, 4]. NetEgg allows the network operators to describe representative behaviors of a desired network function in timing diagrams, and then synthesizes the implementation automatically from the examples. As part of the synthesis, NetEgg infers the state that needs to be maintained, as well as the rules to update the state.

There were two main challenges in the design of NetEgg: 1) how to concisely represent a large set of example behaviors; and 2) how to efficiently synthesize the desired program from the examples. To address the first challenge, I proposed a novel representation integrating timing diagram with
symbolic values. Responding to the second challenge, I developed custom pruning heuristics to facilitate the search for desired programs.

NetEgg can synthesize a wide range of network functions within seconds. Our user study showed that NetEgg reduced programming errors by 30% and saved 50% programming time compared to an alternative programming approach based on Python.

**NetQRE.** Networks need to be dynamically updated for traffic engineering and for timely responding to security threats. Such updates require monitoring network traffic to compute numerical quantities based on a variety of network and application-level performance metrics. For example, the detection of slowloris attacks needs to track each TCP session and compute the packet transfer rate in that session, based on the known TCP traffic patterns. Unfortunately, today’s state-of-the-art tools lack programming abstractions that capture application or session-layer semantics based on traffic patterns. As a result, network operators have to implement complex state machines to handle streaming packets, in order to reason about the interactions across layers.

A potentially better approach for such quantitative functions, thus I argue, should offer stream-level semantics where operators can directly reason about the packet stream in a static fashion, and automatically compile the specifications into correct and efficient streaming implementations. To that end, I designed and implemented NetQRE, a high-level declarative toolkit that aims to simplify the specification and implementation of quantitative network functions [5]. In order to offer a stream-level specification abstraction, NetQRE integrates regular-expression-like pattern matching at flow-level as well as application-level payloads with aggregation operations such as sum and average counts. To automatically translate specifications into correct and efficient implementations, I generalized the novel theory of quantitative regular expressions with parametric extensions, and developed the compilation algorithm of NetQRE. As a result, NetQRE not only allows natural specification of a wide range of quantitative network functions ranging from detecting security attacks to application-layer network management tasks, but also results in high performance that is comparable with that of highly optimized manually-written low-level code.

**Stateful Network Testing and Verification**

In addition to reducing manual errors in programming networks, it is also essential to check the correctness of the network with respect to high-level intents. To that end, I have also designed and implemented network testing and verification tools to efficiently check the correctness of stateful networks (i.e., networks with stateful functions).

**Mikado.** Recent years have seen a great success of network testing tools that can efficiently generate test cases for a single network policy. In practice, however, network operators need to enforce a wide spectrum of policies that a network must implement and thus they need efficient techniques for testing ensembles of those policies. Unfortunately, running tests for ensembles of policies faces fundamental conflicts between efficiency and correctness in stateful networks. For example, testing only one policy at a time would take prohibitively long time for thousands of policies to be tested, while naively running all tests in parallel may compromise correctness in testing results due to conflicts among the test cases.

To address this challenge, I designed Mikado, an efficient test scheduling framework for testing ensembles of policies in stateful networks [2]. Given test cases from one or multiple testing tools, Mikado automatically detects conflicting test cases, and then parallelizes as many non-conflicting tests as possible. In order to detect conflicts, Mikado is based on a formal model of stateful networks to check potential write-read interference on any state between two tests. As a result, Mikado not only guarantees the correctness of the schedule, but also achieves up to 90% reduction in testing time with only a few seconds overhead in generating schedules.
NetSMC. Formal verification is a promising approach to systematically check the correctness of the implementation of networks. However, a key challenge that yet to be addressed is how to build efficient verification techniques for stateful networks, and thus to offer a practical tool for broader real-world scenarios.

Developing such a verification technique for stateful networks is particularly challenging due to the immense number of network states that need to be considered. For example, a typical stateful firewall needs to maintain the set of all legitimate flows that are allowed to go through. Thus, the entire state space can have as many as \(2^{96n}\) states (64 bits for source and destination IP addresses and 32 bits for source and destination port numbers) for a network with \(n\) firewalls, and trivially applying existing verification solutions (e.g. BDD-based model checking) would not work in such a case. Although recently there are some proposals on verifying basic properties such as reachability in stateful networks, they, however, rely on general-purpose solvers to handle the large state space, and thus failed to scale to large stateful networks as well.

To address this challenge, I designed NetSMC, a custom symbolic model checker for stateful networks [3]. There are two key insights in the design of NetSMC. First, I analyzed a range of network functions and observed that most of them maintain states based on equivalent classes in the flow space. Based on this observation, a set of network states can be succinctly encoded in a fragment of existential first order logic. Second, to determine the equivalence of two sets of network states as required by a symbolic model checking algorithm, I leveraged the algorithm to the classical query containment problem in database theory, in order to efficiently check the equivalence. Putting the two insights together, NetSMC achieves 8000× speedup compared with state-of-the-art stateful network verification tools, and meanwhile, supports significantly more properties.

Future Work

Despite substantial progress in formal techniques designed and applied to networks, there remain many other challenges in the way to meet the increasing demand for networks and to keep up with the constantly evolving network technologies. I am excited to explore new research opportunities in related areas and to carry my future research in the following directions.

Network verification and synthesis. Mikado and NetSMC are my early attempts at efficiently checking the correctness of stateful networks. I plan to extend those works in the following two directions. First, I will study how to incrementally check the correctness of a stateful network when the network makes partial changes (e.g. device failures, routing changes), in order to further improve the scalability of the tools. Second, I will study how to efficiently verify quantitative properties (e.g. packets should be delivered to server A in 100 ms) to offer a formal guarantee of network performance.

Moving forward, I am also interested in incorporating my past experience in designing domain-specific programming abstractions and verification techniques for networks into the research of network synthesis. As NetEgg has already demonstrated the feasibility of synthesizing network controller programs from operators’ intents (i.e. example behaviors), I plan to expand this idea to the synthesis of network topologies, network-wide configurations, and dataplane programs.

Human interaction with programming frameworks. The emergence of programmable networks triggers a new wave of innovations in network programming frameworks. Oftentimes, these programming frameworks are driven by specific types of applications, and seldom address their interaction with network programmers. Therefore, it remains a key challenge to understand how network programmers interact with these programming frameworks. In particular, I am driven to address the following problems: how to make a programming framework accessible to network programmers; how to improve the productivity of network programmers; and how to enhance the
reliability of networks with the aid of network programming frameworks.

**Programming and verification of IoT.** Internet-of-Things (IoT) promises appealing applications via highly smart and connected devices. This new trend of innovation opens up new research opportunities in the programmability and verification of IoT implementations. I plan to explore this opportunity focusing on the following research problems.

First, smart devices in IoT typically have constrained computation and memory resources. However, smart devices need to provide enough programming flexibility in order to support personalized and custom functions, and must also meet ever-increasing performance requirements in the meanwhile. This fundamental conflict poses the question that what primitives a smart device should offer to achieve a reasonable balance between programmability and efficiency in various scenarios. Second, with the increase of smart devices both in variety and in quantity, it becomes more complex to implement high-level intents, especially for end-users in the scenario of smart homes. Therefore, a key question remains in how to design intuitive programming abstractions that can incorporate functions on a variety of devices so as to simplify the programming of IoT. Last, it is also critical to ensure the correctness of IoT implementations composed with various functions on a large number of distinct smart devices. Hence, how to efficiently check the end-to-end correctness of IoT, from network-level functions to individual programs on each smart device, remains another challenge.

**References**


