Research Statement
Nik Sultana

Introduction and Broad Vision

My research targets the programming of distributed and heterogeneous systems. Examples of such systems include the Internet, modern datacenters and envisioned 5G cell-phone networks. Many aspects of life depend on such systems, including health-care, education, and entertainment. These systems are constantly growing to meet greater demands for connectivity and better-quality service, and were essential to many people and services when they retreated online during the COVID-19 pandemic. The online shift seems to have passed a tipping point, and even after in-person meetings become safe again people might continue using remote presence for work, conferences, and health checks.

The special ingredient I bring is my unusually broad expertise in computing research. Until my PhD I worked on theoretical research, mostly on verification and automated reasoning, including their application to Security and Systems Biology during three research internships. Post-PhD I focused on Networking and Systems. My research morphed into building platforms that solve practical problems and heed realistic concerns, such as security and portability, but still invokes theory to reason about systems.

During the last five years I worked on different aspects of the programmability of distributed and heterogeneous systems. This included language and abstraction design, toolchain implementation, and semi-automated decomposition of monolithic systems into distributed systems to improve performance, utilization and security.

During the next five years I seek to push this programmability towards a much broader scope. As illustrated in Fig. 1 and detailed in the next section, my past work targets relatively small clusters of devices within larger networks. My future work targets seamless programmability of distributed systems in large, heterogeneous networks by leveraging new hardware and ideas in system autonomy and in-network computing.

The coarse and rigid structure of present-day distributed services, often decomposed only into client and server portions, is challenging to optimize jointly for multiple objectives, including privacy and performance.

My ambition is to design frameworks that can fluidly and opportunistically move parts of computations to better meet execution objectives—such as performance, security, and resource utilization—without burdening programmers with complex cross-layer reasoning. These frameworks will enable flexible runtime re-composition to better take advantage of available resources and changing demands. This can dynamically reduce impedance mismatch between the executing system and the resources it uses during different invocations of the system. Sophisticated resource utilization...
will be used to avoid waste and reduce cost while improving utilization and performance.

In Fig. 1, such a framework could move computations from $A$ to take place closer to $K$, dynamically reconfiguring the nodes to opportunistically use resources adjacent to the (possibly asymmetric) set of paths between the two peers. Here $K$ could be somebody playing a resource-intensive multiplayer game on their phone, a business using a cloud-based service to query a dataset, a remote community who can avoid crossing the Internet for local online meetings, or—if we collapse Provider+Transit+Access to a single, yet large, administrative domain—a batch computation in a localized part of a datacenter or in a peered datacenter over a WAN.

### Programming for Distribution

This section outlines my recent work and how it supports my future work. My future work will involve dynamically extracting and coordinating program fragments from a fluidly-reconfigurable system—one built in a framework that allows fine-grained recomposition of software and its dependencies to better suit available resources. These fragments are dispatched to distributed, heterogeneous targets to meet user- and vendor-defined objectives.

One of the techniques I have used in recent years, and will build on further in future research, is tool-assisted system decomposition. Building systems is often challenging, but decomposing them is hardly ever any easier. Yet decomposition can enable us to achieve various goals—such as improved resource utilization, performance, and security through compartmentalization. I have worked on different projects that use this idea in very different ways.

**Flightplan** [16] (system release) proposed the *dataplane disaggregation* problem: splitting up dataplane programs into a set of programs that execute jointly in the network. A *dataplane* consists of the packet-handling part of a network element. As dataplane programs become more complex, the dataplane programming paradigm might need to evolve towards jointly programming a collection of heterogeneous dataplanes, rather than a single device. In this project we targeted P4, a well-known domain-specific language for dataplane programming. I pitched the idea and architected a broad solution that also addressed detecting and recovering from distributed faults. I also implemented a toolchain that included different *runtimes* to support the execution of disaggregated programs, and a *planner* that ingested various information, including the network topology and objectives, to map subprograms to network elements. Fig. 1 shows Flightplan being used to distribute computing across heterogeneous nodes $A$-$C$ and the router $K$. Flightplan was developed as part of the DARPA DCOMP and GAPS programs, the latter of which I am Co-PI at Penn.

Instead of physically decomposing a system across several devices, we can separate the way that differently-behaving clients experience the same system based on the clients’ characteristics. I used this idea for Denial-of-Service (DoS) analysis and mitigation. Compared to Flightplan, this idea is less program-invasive and avoids introducing distributed failures, but it also lacks performance and resource utilization advantages. This idea was explored in a combination of systems I designed as part of the DARPA XD3 program. I devised an approach where Flowdar found DoS vulnerabilities and Hashtray sifted clients to partition their resource usage at runtime. **Flowdar** [15] (system release) generates configurable traces of complex software and analyzes it using user-defined functions. We used this to detect potential application-logic DoS activity. **Hashtray** [14] (system release) is a suite of hash data structures, including probabilistic ones. **DoSarray** [10] (system release) is an extensible evaluation system we used for hundreds of experiment runs.

I also developed another decomposition-based technique called **Compartment-Based Recomposition (CoBRe)**. This was developed
in the DARPA GAPS program, of which I am Co-PI at Penn. CoBRe transforms a monolithic program into a distributed program and provides runtime support for failure detection and handling. It automates the application of privilege separation (privsep), a well-known software-security approach that provides vulnerability-containment, so the compromise of one part of the system would not necessarily compromise the full system. I initiated and led this project, and architected a broad design that included novel program-level reasoning, distributed execution, and portability and legacy concerns. CoBRe includes a novel whole-program analysis and a simple but effective compartmentalization model. These theoretical underpinnings automate tedious and error-prone reasoning about the movement of information across compartments. I designed and implemented the Pitchfork tool that applies CoBRe to C programs: it decomposes programs into a collection of jointly-executing compartment processes which can be sandboxed separately. This work makes privsep much easier to apply in practice, to help secure more software. Fig. 1 shows Pitchfork being used both on a daemon running on A and on a client running on K. Pitchfork was evaluated using a variety of third-party systems, including server and client software, across three operating systems. Hashtray and Pitchfork in A need to be combined manually at present. My future work will apply language-based methods to avoid manual intervention.

In addition to the decomposition-oriented approaches described above, I also worked on language-oriented support for programming distributed systems. In Flick [1] I led the design of a domain-specific language (DSL) for application-level network functions as part of a larger collaboration across four universities, and implemented its compiler and reference runtime. The language was based on the Communicating Sequential Processes (CSP) algebra, and was carefully customized to its intended domain through the language’s types and execution units. Semantically, programs in this language are constrained to implementing dataflow processors composed of terminating stages. Syntactically, the language was designed to tend towards shorter programs by using stream operators. I led and helped implement the Emu [11] project which eased the programming of high- and predictable-performance network services by combining high-level general-purpose languages and reconfigurable hardware, as part of a multi-university collaboration. I also designed and formalized a debugging approach for network-attached reconfigurable hardware. Fig. 1 shows Emu being used to target heterogeneous nodes D and E from the same source code, while Flick provides transparent application-level support in C. Note that these instantiations of Flick and Emu are ad hoc—there is nothing that binds their instantiation to network-wide policies or objectives, and this complicates reasoning about performance objectives or outages. My future work seeks to systematize cross-network deployment of these aspects of distributed programs to retain a logically-complete view of program parts.

**Past → Future work.** My previous work lays the groundwork for future work in three ways.

i) Problem space exploration. Emu provides insight into heterogeneous device programming through a common interface that supported multiple programming languages—all the languages running on .NET—but did not investigate inter-device coordination. Flightplan went beyond investigating inter-device coordination to also provide a resource-aware reasoning framework for allocating the decomposed dataplane program, but its planning is confined to devices linked to a Top-of-Rack (ToR) switch. Finally, Pitchfork also introduced security as an explicit objective, where previous systems were focused on performance and correctness. ii) Language design. My experience from Flick will be vital to realize the proposed work because
the program ecosystem I envisage requires a new paradigm consisting of a context-aware, co-designed language and runtime. By design, all programs in this paradigm need to support flexible decomposition and reorganization in different ways to suit the available resources. This behavior cannot be retrofitted into existing languages. Current general-purpose languages make it too easy to calcify program structure into a form that cannot be easily reconfigured. I caught a good glimpse of this when adapting the two most widely-used web servers during my research on DoS-resilience: they tended towards being extendable but not decomposable for distributed operation. Thus, being able to design and implement a new language is a key enabler for this research, which will build on ideas from Active Networking and earlier realizations of heterogeneous distributed computing such as that pursued in the Java ecosystem with Oplets, Jini, Applets and Servlets.

Future work

I seek to develop frameworks for the fluid deployment of computation across a network, to develop systems that opportunistically use or lease infrastructure to better deliver their service. Recasting existing services in this way can realize improvements that are currently unachievable because of current systems’ rigid construction. In the envisioned system, when \( K \) seeks to use a service from Provider in Fig. 1, a policy-based negotiation would determine additional resources that \( K \) can use to optimize shared objectives. These resources could be proximal to \( K \), such as \( I \), or be leased from Access, Transit (including other tenants in an IXP), or Provider. This idea goes beyond Fog and Edge computing, which are scoped around the Access network. It re-evaluates the end-to-end principle in network design to take advantage of opportunities from heterogeneous computing devices such as GPUs and FPGAs; developments in software-defined networking and privacy-preserving computation; the ubiquity of computing devices; higher connection density of cities; and the coordination vacuum between all these to better utilize procurable resources.

My plan to develop this research is based on three thrusts that can seed separate research projects. i) Programming model for fluid computation. This project involves developing the model for building systems that can be flexibly decomposed. This involves implementing a language and runtime, learning from the experience of Flick. It will be driven by patterns of decomposition-needs from use-cases drawn from existing distributed programs and topolog-
gies. This work will combine program-level thinking to preserve system behavior \cite{17}, combined with satisfying practical constraints such as the need for debuggability \cite{18} and fault mitigation \cite{16}. ii) Resource-aware deployment and coordination. This systematizes the sporadic cooperation of network elements to jointly execute computations. Its outputs include implementing a scheduler. Unlike (i), this project takes into account policy objectives and their dynamic exchange between peers. Among other things we need to reason about multi-tenant sharing of devices, device heterogeneity and resource planning \cite{16}, and security and privacy \cite{2,6,12} for different stake-holders. This can also benefit from formal modeling \cite{5,13} to evaluate different loads and permissions. iii) Automated integration of fluid, legacy and target-specific systems. This project explores tool-assisted migration paths between existing implementations in languages like C, Go or Rust, black-box functions on specialized hardware, and our new language. This would initially explore three ideas: i) an API for existing languages to use fluid functionality; ii) using whole-program processing as in Pitchfork’s but to automate porting from existing languages; iii) Finer-grained matching of program needs with target-specific capabilities \cite{16}.

Longer term goals include collaborating with Human-Computer Interface (HCI) and economics experts to determine good interaction and pricing models for in-network resources such as $F$ and $G$ in Fig. 1, and carrying out a user study based on the deployment of a campus prototype.

Other important enablers for this research involve continuing to engage with other stakeholders by organizing community workshops, as I did for NetPL \cite{3}, and through industrial collaboration. I have experience in carrying out such collaboration in recent and ongoing projects to pursue published work \cite{4} and tech transfer over regular meetings. This involves iterative problem scoping, student on-boarding and guidance, and design and prototyping.

Past work

I transitioned to researching Systems and Networking after my PhD, which was on logic. In this section I describe work leading up to that.

As an undergrad I carried out a research project in which I designed a new graphical programming language. Programs consisted of interacting blocks on a screen. I was fascinated by John Conway’s *Game of Life* and wanted to make it useful for practical programming. Then, as now, I was interested in the pragmatics of programming languages—not only their theoretical expressiveness, but also the expressiveness that their users typically attain.

The following year I worked on abductive inference for the synchronous language Lustre for my final-year undergrad project. I built a Lustre interpreter that also accepted temporal-logic formulas that encoded desirable program properties. Abductive inference and Truth Maintenance were used to approximate the causes of observed failures in the program.

For my Masters thesis, I used a proof assistant to verify that a set of program transformations used for refactoring functional programs were indeed behavior-preserving \cite{17}. As an RA I later worked on *proof search*—i.e., algorithmically proving mathematical theorems. I wrote a system that built a proof as a term in a typed \(\lambda\)-calculus as a witness to a theorem’s provability. This knowledge surfaced later while working on my PhD \cite{7} and on Flightplan \cite{16}.

My PhD was on combining logics by interfacing the tools that implement reasoning support for those logics. I developed a tool-interfacing approach that tolerated missing information in proofs emitted by different tools. Information would be missing because a tool’s proof format was incomplete, incompletely documented, or volatile. The proof translation interface used proof search to reconstruct the missing bits of proofs. I applied this approach to different logics, embedding them in another system that would certify the translation’s correctness \cite{8}.
References


