Anirudh Sivaraman - Research Statement

Computer networks have revolutionized society by *commodifying connectivity* between disparate locations. Going beyond connectivity, I see *ubiquitous network programmability* as networking's next frontier: enabling programmatic transformations of data while it is in transit between two locations. Peering into the future, the network as a programmable platform has radical consequences: Network infrastructure can evolve much faster than it historically has, freeing it from slow protocol standardization and hardware upgrade cycles. Networked applications can compute on application data within the network as soon as such data is first available—enhancing applications on several dimensions (e.g., fairness [10], CPU overhead [9], and request latency [22, 21]).

In my work, I approach network programmability through two opposing lenses [24]: (1) **the network core**, which interconnects servers and (2) **the network edge**, which provides network access to applications. A programmable core allows us to rapidly modify high-speed network infrastructure like switches and routers, which have historically suffered a hard tradeoff between flexibility and performance. A programmable edge provides application developers with a "do-it-yourself" approach to network programmability—especially cloud tenants who can not access the network core.

A running theme through my research is what I call **smart waypoints**: small pieces of code that intercept and programmatically transform data in transit between two points. Such waypoints are embedded within communication infrastructure at all layers. They allow customization of complex and entrenched pieces of hardware and software that have traditionally been hard to modify. Examples from my research include P4 programmable routers [23, 7, 25], smart network-interface cards [15], the host network stack [16], software sandboxes embedded in network stacks [29] and Web proxies [3], and VMs interposing between other VMs [10, 9].

My research style is to identify foundational and forward-looking questions within the broader vision of programmable networks. These questions span the entire stack: hardware, operating systems, compilers, and networked applications. To find the best answers, I often collaborate with experts in allied research areas including computer architecture, compilers, program synthesis, formal methods, language design, and distributed systems. For realistic evaluations, I build hardware and software systems with a substantial engineering component to validate research ideas.

To ensure relevance and impact, I have open sourced code and replication instructions for most of my research projects. I have also worked closely with both the open-source community and startup companies (Barefoot Networks and Clockwork). This has resulted in several contributions to the P4 programming language and broader open-source adoption of the Mahimahi network emulator, the Gauntlet compiler bug-finding tool, the P4Testgen test oracle, the In-Band Network Telemetry standard, and the DC.p4 (now switch.p4) program. It has also resulted in considerable interest from industry in a few of my projects like the On-Ramp congestion-control algorithm and the PIFO hardware design for programmable scheduling. I also co-chaired the P4 Summit in 2020, co-organized a well-attended NSF workshop on programmable networks in 2018, co-organized a tutorial on networking for finance at SIGCOMM 2020, and am a member of the Network Programming Initiative.

Looking to the future of computer networks, I believe networks will continue to evolve as they have in the past—from physical network interface cards to their virtual counterparts to container network interfaces to service proxies interconnecting microservices to serverless computing where communication happens through shared storage. As a networking researcher, I plan to embrace this change by meeting application developers where they are and continuously revisiting fundamental notions of *what a network is and who it serves*. For me, this has meant continuously seeking out new ways to make the network invaluable, e.g., moving from fixed to programmable network devices and enabling network programmability for cloud tenants. In the same vein, going forward, to ensure the vitality of networking, I plan to draw inspiration from emerging application domains (e.g., networking for microservices, networking for virtual reality, inter-drone communication, and edge computing offerings) to identify how network flexibility can deliver new value to users and developers.

In many environments—most notably environments cloud tenants find themselves in—the developer of a networked application has no control over the network core and can't program the core to do their bidding. To this end, I have pursued a research agenda of programmability at the network's *edge*: an application's point of attachment to the network. My goal is to empower cloud developers to take matters into their own hands and enhance their networked applications by strategically offloading some application logic to programmable waypoints.

Network primitives for distributed systems. Distributed systems traditionally assumed nothing more of the network than best-effort packet delivery. In contrast, recent work [14, 18] has demonstrated how certain network features can accelerate distributed consensus. This work leverages network core features to sequence, multicast, and prioritize packets. Because such features are unavailable to cloud tenants, cloud tenants typically use lower performance consensus protocols instead. To remedy this, we have developed a new network primitive, deadline-ordered multicast (DOM), to help build high-performance and cloud-native distributed systems. In DOM, senders and receivers within a multicast group have synchronized clocks. A sender tags a message with a global time deadline and sends it to all receivers. A receiver only processes a message on or after the message's deadline is reached and processes multiple messages in order of their deadlines. DOM's heavy lifting of unicasting to multiple receivers in the absence of switch multicast is handled by a proxy running inside a VM waypoint that interposes between clients and servers, reducing clients' CPU overhead in the process. We have used DOM to build Nezha [9], a crash-fault-tolerant consensus protocol (Figure 1). We are using DOM to design new protocols for concurrency control and Byzantine Fault Tolerance. We are also designing an autoscaling cloud service for DOM and a "bolt-on" DOM layer within an eBPF waypoint to transparently accelerate existing distributed protocols.

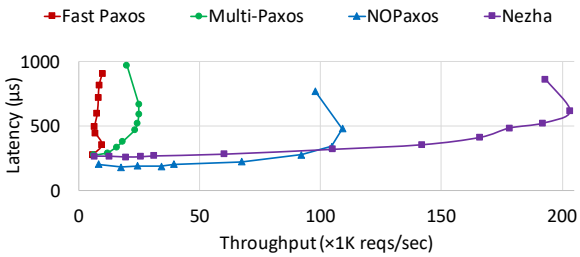

Figure 1: Nezha, which leverages our new deadline-ordered multicast (DOM) primitive, outperforms Multi-Paxos, Fast Paxos, and NOPaxos.

Cloud-native stock exchanges. Financial exchanges have begun migrating from on-premise and custom-engineered datacenters to the cloud, accelerated by a rush of new investors, cost savings from the cloud, and the desire for more resilient infrastructure. Despite its scale and elasticity benefits, the cloud's variable network can cause market unfairness: orders can be processed out of sequence, and market data can be disseminated to market participants at different times due to varying latencies between participants and the exchange. We have built CloudEx [10], a fair-access cloud exchange, which uses clock synchronization to compensate for the cloud's noisy network conditions. CloudEx uses a perimeter of trusted gateway VM waypoints that are clock synchronized to a matching engine VM. These gateways both timestamp incoming orders from untrusted participants (for execution at the engine in timestamp order) and hold outgoing market data (for simultaneous release at a time specified by the matching engine) (Figure 2). We have

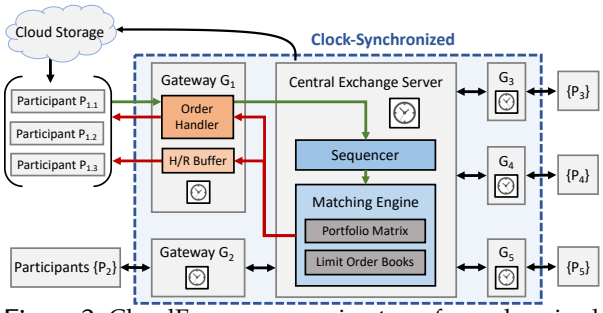

Figure 2: CloudEx uses a perimeter of synchronized gateways to hold and simultaneously release market data (H/R buffer) and to timestamp incoming orders.

deployed and evaluated CloudEx in two courses and CloudEx has already seen follow-on work in the networking community [12]. We are now adding fault tolerance to CloudEx via Nezha [9].

Optimizing packet-processing bytecode. Software sandboxes provide a safe way to extend stable software like browsers or OS kernels. A notable example is the eBPF waypoint for packet processing in Linux. An eBPF compiler has the challenging task of generating eBPF bytecode that is (1) safe (e.g., it must not access invalid memory) (2) compact (so that the kernel can check its safety quickly) and (3) performant (so that it doesn't impact throughput and latency). We built K2 [29], an eBPF compiler that employs *stochastic superoptimization* to search for such bytecode: K2 randomly samples bytecode sequences based on a cost model and then checks equivalence of these sequences to a specification eBPF bytecode using an SMT solver. K2 often outperforms eBPF's clang compiler at the O2 optimization level. We believe this approach could be valuable for bytecode formats running within other programmable waypoints, such as WebAssembly inside Web proxies [3].

Autoscaling and scheduling latency-sensitive networked applications. The cloud is increasingly home to latency-sensitive applications such as autonomous vehicles and microservices. We have applied optimization techniques to improve the cost-performance of such applications in the public cloud. These optimization techniques take as input telemetry collected by waypoints (e.g., one-way delay or throughput measurements between pairs of VMs) and use this telemetry to select, schedule, and provision VMs. COLA [22] uses a multi-armed bandit formulation to reduce queueing latency due to under-provisioned microservices by automatically provisioning enough VMs for each microservice with the goal of meeting a desired end-to-end latency target for a Web application that has multiple microservices. LemonDrop [21] uses a quadratic assignment problem formulation to reduce communication latency between different components of an application by requesting several VMs of the same VM type from the cloud provider, selecting a subset of these VMs based on their observed performance, and mapping application components onto this subset. By dropping underperforming VMs (lemons) from the initially requested VMs, we find that LemonDrop can substantially improve performance for networked applications, e.g., CloudEx, Nezha, and microservice-based applications [11]. We believe these techniques can be applied to other latency-sensitive use cases such as game engines and low-latency ad marketplaces.

Congestion control. Many datacenter congestion control algorithms need network core assistance (e.g., routers marking packets), putting these algorithms beyond the reach of cloud tenants. On-Ramp [16] is an approach to congestion control implemented solely in the Linux kernel's qdisc waypoint, and hence can be deployed with no network core support. On-Ramp can also be implemented in eBPF waypoints for safe kernel extensibility—or within a programmable NIC waypoint to save scarce end-host CPU cycles and improve On-Ramp's performance. On-Ramp is an underlay module that sits below and transparently improves an existing congestion-control algorithm in the end host by improving the algorithm's response to severe fan-in workloads. It works by holding packets at the sender whenever the one-way delay to a receiver exceeds a threshold—like how an onramp gates access to a busy freeway.

PROGRAMMING THE NETWORK CORE

Traditionally, packet-switched networks emphasized an architecture with smart end hosts and a minimal core infrastructure: routers focused solely on distributed routing protocols and packet forwarding. Over time, routers have taken on an ever-changing list of requirements including access control, load balancing, packet scheduling, and measurement. To address these important needs, router flexibility has become as important as performance. However, despite efforts in the 1990s and 2000s, programmable routers remained 10–100x slower than fixed-function routers, precluding their use in

production. Motivated by this, I have developed hardware designs for fast and programmable routers along with the associated system software.

Hardware designs for packet processing. As part of my dissertation, which won the SIGCOMM 2017 Doctoral Dissertation Award, I developed new router hardware designs to walk the tightrope between programmability and performance. The unifying theme in these designs was a focus on restricted, but important, classes of router functionality—providing programmability without losing performance. For instance, scheduling was thought to be too hard to program at line rate because the scheduler sits at the heart of a switch where digital design is most challenging. PIFO [25] was a new hardware design for a programmable packet scheduler based on a simple priority queue with programmable packet priorities. The PIFO project has seen significant follow-on work in the networking research community. Domino [23] developed an instruction set, programming model, and compiler for *stateful* packet-processing algorithms, significantly advancing over prior work that had considered largely stateless packet processing [4]. Marple [17] (best paper at SIGCOMM 2017) mathematically characterized and developed hardware designs for a family of measurement statistics that sidestepped the memory-accuracy tradeoff present in many solutions based on measurement sketches. I have also developed new architectures for programmable switches [5] and network-interface cards [15].

Solver-aided compilers. Programming routers today is rudimentary and akin to programming microprocessors in the 1970s or GPUs in the 2000s—before productive languages and good compilers were developed. On the bright side, unlike at the dawn of compiler research, we now have at our disposal high-quality solver engines for many optimization and satisfiability problems (e.g., program synthesis engines, SMT solvers, ILP solvers). Many of my recent projects have made use of such solvers to pose compiler problems declaratively as optimization or constraint satisfaction problems. Our thesis [6] is that such an approach has the potential to both (1) simplify compiler development by using solvers to do a compiler's algorithmic heavy lifting (e.g., code generation, resource allocation) and (2) improve the quality of the compiler's output via exhaustive search, which is critical for network devices that must provide high performance. Next, I illustrate research results from this approach.

Bug Type	**Status**	P4C	BMv2	Tofino
Crash	Filed	36	4	35
	Confirmed	33	4	25
	Fixed	27	4	8
Semantic	Filed	31	1	10
	Confirmed	26	1	7
	Fixed	22	1	0
Total	**96**	**59**	**5**	**32**

Table 1: Gauntlet found numerous P4 compiler bugs using fuzz testing and translation validation.

A traditional compiler *iteratively* transforms a program using code rewrite rules to progressively generate machine code. In contrast, Chipmunk [6, 7] shows the benefits of treating code generation *declaratively* as a program synthesis problem: the compiler intelligently searches over a large combinatorial space of machine code programs to find machine code that is equivalent to the source program. Gauntlet [20] is a tool to find crash and miscompilation bugs in P4 compilers. To detect crash bugs, Gauntlet uses fuzz testing. To detect miscompilation bugs, Gauntlet revives a classical compiler technique called translation validation: using an SMT solver to prove that the compiler correctly translated a program. An inability to prove indicates a miscompilation. Gauntlet found numerous new and confirmed bugs in the open-source P4C reference compiler frontend and the P4 behavioral model and Tofino backends (Table 1) and now runs as part of P4C's continuous integration pipeline. Gauntlet's semantics have also been used as a reference for subsequent research [26]. We are now extending Chipmunk to additionally perform resource allocation using an ILP solver [8]. We have also worked with Intel's compiler team on an open-source test generator [2, 19] for P4 programs that builds on ideas from Gauntlet and leverages symbolic execution powered by an SMT solver.

Multitenancy mechanisms. Network programmability would be of very limited value if only the network's owners were able to program their devices. Ideally anyone writing an application running

over a network—even one they don't own—should be able to program the network. To that end, we envision a future where cloud providers offer programmable packet processing as a service to tenants. To support this, we need mechanisms to run multiple tenant programs on a network device. Sharing is typically handled by the OS, but programmable routers lack the hardware support for an OS. In a series of papers [28, 30, 27], we designed compile-time approaches and hardware support for such sharing. One of these projects, Menshen [27], has been used by subsequent research projects [13].

THE FUTURE: A HARDWARE RPC PROCESSOR FOR MICROSERVICES

Going forward, I plan to tackle a multi-year moonshot with network programmability at its core: *a high-performance microservices platform, centred around a programmable RPC processor in hardware*. I envision a research center where we tackle the full stack of problems associated with fabricating such a processor and demonstrating the value of such a platform. Executing on this will involve expertise from multiple faculty and industry partners spanning open-source VLSI design [1], computer architecture, compilers, networking, distributed systems, programming languages, and applications—and will likely produce multiple dissertations of considerable intellectual depth. I expand below.

Web services have increasingly been factored into a loose collection of microservices, which communicate with each other using RPCs. Over time, common portions of inter-microservice communication have been factored out into waypoints called service proxies, which interpose on inter-microservice traffic and provide common functionality such as security and load balancing. Recent service proxies are extensible, allowing us to add functionality via WebAssembly bytecode; we have prototyped a new approach to distributed tracing through such extensions [3]. While programmable service proxies represent an exciting opportunity, they significantly increase RPC latency and consume additional CPU cycles [31].

Our goal is to eliminate these overheads by moving the service proxy to a specialized in-line hardware waypoint: the RPC processor. In initial studies, we have found that designing such an RPC processor requires not just designing equivalent hardware for proxy functionality that is currently in software, but rather fundamentally rethinking a proxy's abstractions to make them more hardware-offload friendly, e.g., using message-oriented transport instead of reliable bytestreams, and simplifying header formats to make them amenable to high-speed hardware parsing.

Our end-to-end vision is a platform where each microservice can be pinned to one or more CPU cores. A core communicates with other cores either on the same or a different machine by passing data to a low-latency RPC interface (e.g., by writing to a special CPU register). The cores are responsible only for sending and receiving RPCs. The programmable RPC processor handles all other aspects of communication in hardware: serialization, compression, encryption, congestion control, authorization, authentication, load balancing, and reliable data delivery. Additionally, the RPC processor will enforce isolation between the processing of RPCs belonging to different microservices—important in multi-tenant environments—and leave just the right number of programmable knobs for application-specific customization of in-line RPC processing.

Such a processor could form the basis of a platform for high-performance distributed computing in the broadest sense—providing underlying technology support for diverse use cases such as microservices, actor-based programming models, programming models based on communicating sequential processes, and frameworks like MPI.

⁂

Computer networks have transformed society by turning connectivity between disparate locations into a commodity. My long-term goal is to similarly make network flexibility a commodity—so that programmability is the norm, rather than a research curiosity.

REFERENCES

[1] chipIgnite Shuttle 2204C | Efabless. https://platform.efabless.com/chipignite/.

[2] Contribute the P4Tools platform and P4Testgen. https://github.com/p4lang/p4c/pull/3495/.

[3] Jessica Berg, Fabian Ruffy, Khanh Nguygen, Nicholas Lee, Taegyun Kim, **Anirudh Sivaraman**, Ravi Netravali, and Srinivas Narayana. Snicket: Query-Driven Distributed Tracing. In *ACM HotNets*, 2021.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[5] Sharad Chole, Andy Fingerhut, Sha Ma, **Anirudh Sivaraman**, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. dRMT: Disaggregated Programmable Switching. In *ACM SIGCOMM*, 2017.

[6] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, **Anirudh Sivaraman**, and Srinivas Narayana. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *ACM HotNets*, 2019.

[7] Xiangyu Gao, Taegyun Kim, Michael Dean Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, **Anirudh Sivaraman**, Srinivas Narayana, and Aarti Gupta. Switch Code Generation using Program Synthesis. In *ACM SIGCOMM*, 2020.

[8] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, **Anirudh Sivaraman**, Srinivas Narayana, and Aarti Gupta. High-level synthesis for packet-processing pipelines. https://arxiv.org/abs/2211.06475, 2022.

[9] Jinkun Geng, **Anirudh Sivaraman**, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. In *VLDB*, 2023.

[10] Ahmad Ghalayini, Jinkun Geng, Vighnesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and **Anirudh Sivaraman**. CloudEx: A Fair-Access Financial Exchange in the Cloud. In *ACM HotOS*, 2021.

[11] Google. Online boutique application. https://github.com/GoogleCloudPlatform/microservices-demo.

[12] Prateesh Goyal, Ilias Marinos, Eashan Gupta, Chaitanya Bandi, Alan Ross, and Ranveer Chandra. Rethinking Cloud-Hosted Financial Exchanges for Response Time Fairness. In *ACM HotNets*, 2022.

[13] Hejing Li, Jialin Li, and Antoine Kaufmann. SimBricks: End-to-End Network System Evaluation with Modular Simulation. In *ACM SIGCOMM*, 2022.

[14] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX OSDI*, 2016.

[15] Jiaxin Lin, Kiran Patel, Brent E. Stephens, **Anirudh Sivaraman**, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *USENIX OSDI*, 2020.

[16] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and **Anirudh Sivaraman**. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *USENIX NSDI*, 2021.

[17] Srinivas Narayana, **Anirudh Sivaraman**, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*, 2017.

[18] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX NSDI*, 2015.

[19] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtěch Havel, Rob Sherwood, Vlad Dubina, Volodymyr Peschanenko, Nate Foster, and **Anirudh Sivaraman**. P4testgen: An extensible test oracle for p4. https://arxiv.org/abs/2211.15300, 2022.

[20] Fabian Ruffy, Tao Wang, and **Anirudh Sivaraman**. Gauntlet: Finding bugs in compilers for programmable packet processing. In *USENIX OSDI*, 2020.

[21] Vighnesh Sachidananda. *Scheduling and Autoscaling Methods for Low Latency Applications*. PhD thesis, Stanford University, 2022.

[22] Vighnesh Sachidananda and **Anirudh Sivaraman**. Collective Autoscaling for Cloud Microservices. https://arxiv.org/abs/2112.14845, 2022.

[23] **Anirudh Sivaraman**, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM*, 2016.

[24] **Anirudh Sivaraman**, Thomas Mason, Aurojit Panda, Ravi Netravali, and Sai Anirudh Kondaveeti. Network Architecture in the Age of Programmability. In *SIGCOMM Computer Communication Review Editorial*, 2020.

[25] **Anirudh Sivaraman**, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, 2016.

[26] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: A Practically Usable Verification System for Production-Scale Programmable Data Planes. In *ACM SIGCOMM*, 2021.

[27] Tao Wang, Xiangrui Yang, Gianni Antichi, **Anirudh Sivaraman**, and Aurojit Panda. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *USENIX NSDI*, 2022.

[28] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, **Anirudh Sivaraman**, Dan R. K. Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud*, 2020.

[29] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and **Anirudh Sivaraman**. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *ACM SIGCOMM*, 2021.

[30] Hang Zhu, Tao Wang, Yi Hong, Dan R. K. Ports, **Anirudh Sivaraman**, and Xin Jin. NetVRM: Virtual register memory for programmable networks. In *USENIX NSDI*, 2022.

[31] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting Service Mesh Overheads. https://arxiv.org/abs/2207.00592, 2022.