Summary. After my PhD at UCLA and my postdoc at MIT, I started as an assistant professor at UCR in January of 2017. My research interests are at the intersection of two areas: (1) verification and synthesis, and (2) concurrent and distributed computing. I develop specification, verification and synthesis techniques and tools to build reliable, secure and efficient computing systems, in particular, subtle concurrent and distributed systems. My research results have been published in 35 papers in diverse venues that include[1]: 2 POPL, 2 CAV, 1 ICFP, 3 OOPSLA including a distinguished paper award, 1 CACM, 3 DISC, 1 PODC, 1 PPoPP, 1 CONCUR, 1 best paper at ISSRE, 1 ICBC, and 1 FSE. I received the SIGPLAN[2] Research Highlight [16] in 2019. I obtained four NSF grants including the NSF CAREER award. In addition to external review committees, I served for the program committees of five conferences including POPL, OOPSLA and ECOOP, and an NSF proposal review panel.

1 Research Projects

I present an overview of my recent projects.

Secure and Resilient Information Flow. *Inter-organizational systems* where subsystems with partial trust need to cooperate are common in finance, healthcare and military. In the face of malicious Byzantine attacks, the ultimate goal is to assure end-to-end policies for the three aspects of trustworthiness: *confidentiality, integrity and availability.* In contrast to confidentiality and integrity, provision and validation of availability has been limited. This project presents a security-typed object-based language, a *partitioning transformation*, an operational semantics, and an *information flow type inference system* for partitioned and replicated classes. The type system enforces *end-to-end policies* simultaneously for all the three aspects of trustworthiness. It provably guarantees that well-typed methods enjoy *noninterference*, and that their *types quantify their resilience to Byzantine attacks.* Further, type inference results in *automatic placement and replication* of field objects and methods. The synthesis tool can adjust the degree of replication to the resiliency strength of specifications and result in *trustworthy-by-construction* distributed systems.

Automatic Analysis and Synthesis of Distributed Systems. Distributed system replication is widely used as a means of fault-tolerance and scalability. Traditional strong consistency maintains the same total order of operations across replicas. This total order is the immediate source of multiple desirable consistency properties: *integrity, convergence and recency.* However, maintaining the total order has proven to inhibit availability and performance. Weaker notions exhibit responsiveness and scalability; however, they forfeit the total order and hence its favorable properties. This project revives these properties with as little coordination as possible. It presents tools (Hamsaz and Hampa) that, given a sequential class with the declaration of its integrity and recency requirements, automatically *synthesizes a correct-by-construction replicated class* that simultaneously guarantees the three properties. The approach is based on *novel sufficient conditions* for integrity, convergence and recency that require certain orders between conflicting and dependent operations, and constrain the set of pending operations. To decide the validity of these *coordination-avoidance* conditions, the tools apply automatic solvers to analyze both the given class statically and the operations dynamically. It then *reduces coordination to classical minimization problems*, and uses the results to instantiate *novel parametric coordination protocols.* The results been published in POPL'19 [3] and CAV'20 [12]. We are building replication synthesis tools for Amazon Web services. Further, we are investigating automatic replication that preserves information flow security policies.

Verification of Distributed Systems. Distributed systems are critically used in geo-replicated data stores, replicated controllers for planes, cars, power plants, and digital currencies. However, they have proven to be complicated due to node and network failures and combinatorially large state-spaces. Thus, they repeatedly suffer data, currency and service loss. Further, the modern efficient data stores provide weaker notions of consistency such as eventual and causal consistency which make client code prone to bugs. This project aims at building *mechanically verified (also known as certified) distributed systems.* Our first framework, Chapar, supports *modular verification of causal consistency for replicated key-value store implementations and their client programs* in the Coq theorem prover. It includes a novel operational semantics which serves as a specification for implementations and a guarantee for clients, verification of clients based on this specification, and a proof technique that is successfully applied to verify two implementations. The implementations is extracted from Coq to OCaml to built executable stores. However, Chapar, did not

consider general distributed components and their composition. Network middleware has been traditionally composed as layered stacks of components. Our recent work presents a novel approach to *compositional verification of distributed system stacks*. The goal is to modularly verify each component based on only the implementation of the component itself and the specification of lower components. This project presents *TLC (Temporal Logic of Components)*, a novel temporal program logic that offers intuitive inference rules for verification of both safety and liveness properties of functional implementations of distributed components. TLC is proven sound with respect to a novel operational semantics for stacks of composed components in partially synchronous networks. We are developing a Coq framework for TLC to build certified middleware. The results of this project have been published in POPL'16 [7] and ICFP'20 [2].

Domain-specific Languages and Type Systems. This project designs principled languages and type systems for specific domains. In particular, we have brought type-safety to two different domains: interaction safety for biochemistry programs, and initialization before use for the Linux kernel. This project introduced *BioScript, a domain-specific language for programmable biochemistry*. We designed the core syntax and type checking and inference systems for BioScript. The *type system* ensures that certain types of errors, specific to biochemistry, do not occur, including *the interaction of chemicals that may be unsafe*. The paper was published at OOPSLA'18 [13] and received the distinguished paper award, and later, was recognized as the SIGPLAN Research Highlight in 2019. Further, the project was invited and accepted to appear in the Communications of the ACM (CACM). Every year, a committee representing SIGPLAN's major conferences and elected officials select 2 to 4 papers that are of high quality and broad appeal as SIGPLAN Research Highlights from all the 18 sponsored conferences. Similarly, we designed a qualifier type system to type-check initialization before use for C functions in the Linux Kernel. The results appeared in FSE'20 [18].

Automatic Fence Insertion. In the interest of performance, processors feature relaxed memory models that reorder instructions. However, the correctness of concurrent programs is often dependent on the preservation of the program order of certain instructions. Thus, the instruction set architectures offer memory fences that prevent certain classes of reordering. Using fences is a subtle task with a performance and correctness trade-off. This project presents a high-level programming model and a tool to *capture the orders that the user requires*, and *algorithms that insert the minimum number of fences* to preserve the specified orders. In particular, it presents a greedy and polynomial-time optimum fence insertion algorithm for the class of programs with structured branch and loop statements, in addition to NP-hardness results for other classes. The results of this project appeared in my dissertation [4], OOPSLA'15 [1], PODC'17 [5], DISC'19 [17]. As a direct impact, these algorithms have been *integrated into LLVM* compiler infrastructure by independent developers [14].

Concurrency Testing and Verification. This project presented testing and verification techniques and tools for transactional memory (TM) and concurrent data structures. The results of this project appeared in CAV'14 [9], DISC'13 [10], DISC'14 [11], NFM'19 [6], and CONCUR'12 [8].

In return for the simpler programming model, TM algorithms are complicated and error-prone. To *test concurrent algorithms* for common bugs, I developed a tool called Samand [10] that given a concurrent algorithm and the specification of a *particular error at the interface* of the algorithm, checks if there is a concurrent execution that exhibits that error, and produces an example trace. Thus, it makes regression testing possible for concurrent algorithms. It translates *concurrent execution to constraints* and applies constraint solvers to search for possible executions. I used Samand to show the *incorrectness of DSTM and McRT* TM algorithms. They suffered from the write-skew and write-exposure anomalies respectively. These results were surprising as previous work had claimed verification of these algorithms.

Opacity is a correctness condition for TM algorithms. I defined [11] a *decomposition of opacity* called markability as a conjunction of separate intuitive invariants, and proved that markability is required and sufficient for opacity. Separation has obvious benefits of *modularity and scalability for verification*. The proofs of markability of TM algorithms can mirror the algorithm design intuitions about the location and order of read and commit points. I defined a *program logic* called synchronization object logic (SOL) [6] that supports reasoning about the *execution order and linearization orders* of calls on the primitive synchronization objects. It provides inference rules that axiomatize the properties and interdependence of these orders, and the properties of common synchronization object types. SOL is proved sound based on a denotational semantics. I formalized SOL in the PVS proof assistant and used it to machine-check the *markability of the TL2* TM algorithm.

During my internship at Oracle labs, I defined a framework [8] in PVS for *verification of TM algorithms based on I/O Automata*. I specified both the specifications and implementations using I/O automata and proved simulation relations between them. I used the framework to prove the correctness of the NOrec TM algorithm. The proofs of new algorithms can leverage the forward and backward simulation lemmas and the hierarchy of simulations in the framework.

Mainstream programming languages offer libraries of concurrent data types that guarantee linearizability. Linearizability supports *horizontal composition*. An execution may involve method calls on multiple objects and each method call appears to take effect atomically. However, they usually don't support *vertical composition*: An operation that calls multiple methods on a data structure are not atomic by default. I presented [9] an automatic and modular *verification technique for atomicity of vertical composition*. I presented a novel sufficient condition called condensability and a tool called Snowflake that generates proof obligations for condensability of composing methods in Java and discharges them using constrain solvers. Snowflake could successfully verify a large fraction of an existing suite of compositions from several open-source applications.

Blockchain Transactions. A cross-chain transaction *exchanges assets between multiple parties across multiple blockchains*. It can be represented as a directed graph, and may include a sequence of exchanges at each blockchain. Further, it may have off-chain steps and hence may not be strongly connected. If all parties conform to the protocol, all the assets should be transferred. If any party deviates from the protocol, the conforming parties should not experience any loss. Further, if the source parties pay, the sink parties should be paid. This project presents a protocol that guarantees the above properties for general cross-chain transactions with sequenced and off-chain steps when a few certain parties are conforming. It presents a tool called XChain that given a high-level description of a cross-chain transaction, can automatically generate smart contracts in Solidity. The first result of this project was published in ICBC'20 [15].

2 Grants

I received four NSF grants and I am the PI for three of them. The first one, "CRII: SHF: Certified Byzantine Fault-tolerant Systems" (PI, $174,999) is a single PI grant. It investigates verification of reliability and security of distributed systems where faulty or malicious nodes exhibit arbitrary or misleading behavior. These systems have applications ranging from finance to aircraft control. The second grant that Professor Song and I received is titled "SaTC: CORE: Small: Practical Whole Kernel Memory Safety Enforcement" (co-PI, $474,399, equal shares). This research project develops new techniques to eliminate critical memory access vulnerabilities from commodity operating system kernels. It includes both provably sound type systems and efficient implementation techniques. The third grant is "FET: Small: Stochastic Synthesis of Peptides and Small Molecules" (PI, $500,000, equal shares) with my co-PIs, Professors Brisk and Grover. This project accelerates the development of new drugs. It bring the level of automation inherent in computing disciplines to the life sciences. This project applies program synthesis principles to automate the discovery of new drugs. Given a drug specification, the synthesis process automatically identifies a ligand that complies with the specification, can be safely synthesized and strongly binds to the desired proteins. The fourth grant is "CAREER: Distributed System Synthesis on Certified Middleware" (PI, $523,801). This proposal addresses programmer productivity and reliability of distributed systems that spans both the client applications and the supporting distributed middleware. It includes both novel automatic synthesis techniques for client applications and novel verification techniques for distributed middleware.

3 Service

I was a member of the PC (Program Committee) for POPL'20 (14 papers), PC for OOPSLA'20 (16 papers), PC for DisCoTec'20 (2 papers), PC for ECOOP'18 (8 papers), PC for CPP'17 (8 papers), ERC (External Review Committee) for POPL'17 (6 papers), and an external reviewer for OOPSLA'19 (1 paper) and ESOP'20 (1 paper). I was a panel member of an NSF SHF^3 program in 2018 (8 proposals). Further, I was a committee member in Academic Panel of the Doctoral Symposium at OOPSLA'18. In the day long symposium, we advised students for progress in their dissertations. I also chaired a session at OOPSLA'18, two sessions at POPL'20 and a session at ICBC'20.

References

[1] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 367–385, 2015.

[2] Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. Tlc: temporal logic of distributed components. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020.

[3] Farzin Houshmand and Mohsen Lesani. Hamsaz: replication coordination analysis and synthesis. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[4] Mohsen Lesani. *On the Correctness of Transactional Memory Algorithms*. PhD thesis, University of California, Los Angeles, USA, 2014.

[5] Mohsen Lesani. Brief announcement: Fence insertion for straight-line programs is in P. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 97–99. ACM, 2017.

[6] Mohsen Lesani. Transaction protocol verification with labeled synchronization logic. In *NASA Formal Methods Symposium*, pages 280–297. Springer, 2019.

[7] Mohsen Lesani, Christian J Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 357–370, 2016.

[8] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *International Conference on Concurrency Theory*, pages 516–530. Springer, 2012.

[9] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Automatic atomicity verification for clients of concurrent data structures. In *International Conference on Computer Aided Verification*, pages 550–567. Springer, 2014.

[10] Mohsen Lesani and Jens Palsberg. Proving non-opacity. In *International Symposium on Distributed Computing*, pages 106–120. Springer, 2013.

[11] Mohsen Lesani and Jens Palsberg. Decomposing opacity. In *International Symposium on Distributed Computing*, pages 391–405. Springer, 2014.

[12] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *International Conference on Computer Aided Verification*, pages 324–349. Springer, 2020.

[13] Jason Ott, Tyson Loveless, Christopher Curtis, Mohsen Lesani, and Philip Brisk. Bioscript: programming safe chemistry on laboratories-on-a-chip. *Proc. ACM Program. Lang.*, 2(OOPSLA):128:1–128:31, 2018.

[14] sconstab. [X86] Add support for load hardening to mitigate load value injection (LVI). `https://reviews.llvm.org/rG8ce078c7503`.

[15] Narges Shadab, Farzin Houshmand, and Mohsen Lesani. Cross-chain transactions. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*, pages 1–9. IEEE, 2020.

[16] SIGPLAN. Sigplan research highlights. `http://www.sigplan.org/Highlights/`.

[17] Mohammad Taheri, Arash Pourdamghani, and Mohsen Lesani. Polynomial-time fence insertion for structured programs. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[18] H. Zhang D. Wang C. Song Z. Qian M. Lesani S. Krishnamurthy P. Yu Y. Zhai, Y. Hao. Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the ESEC/FSE'20 (The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, FSE '20, New York, NY, USA, 2020. ACM.

Notes

[1]POPL, PLDI, OOPSLA and ICFP are in the top four conferences in programming languages.
CAV is one of the top two conferences in logic and verification.
FSE is one of the two top conferences in software engineering.
PODC and DISC are the top two conferences in distributed computing.
Conference abbreviations:
CACM: Communications of the ACM
CAV: International Conference on Computer-Aided Verification
CPP: ACM SIGPLAN and SIGLOG conference on Certified Programs and Proofs
ECOOP: European Conference on Object-Oriented Programming
ESOP: European Symposium on Programming
FSE: The ACM Symposium on the Foundations of Software Engineering
CONCUR: International Conference on Concurrency Theory
DISC: The International Symposium on DIStributed Computing
DisCoTec: International Federated Conference on Distributed Computing Techniques, Program Committee
ICBC: IEEE International Conference on Blockchain and Cryptocurrency
ICFP: ACM SIGPLAN International Conference on Functional Programming
ISSRE: International Symposium on Software Reliability Engineering
OOPSLA: ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and
PODC: ACM Symposium on Principles of Distributed Computing
PLDI: ACM-SIGPLAN Symposium on Programming Language Design and Implementation
POPL: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Applications
PPoPP: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming
[2]SIGPLAN: ACM Special Interest Group on Programming Languages
[3]SHF: Software Hardware Foundation