Building Certified Systems Software

RESEARCH STATEMENT

Ronghui Gu (ronghui.gu@columbia.edu)

My research goal is to make safety-critical systems software truly trustworthy through *formal verification*. Safety-critical systems software, such as OS kernels, hypervisors, distributed systems, and blockchain systems, form the backbone of today's computing hosts. Despite their importance, these systems software are complicated and highly concurrent, and all it takes is a single weak link in the code—one that is virtually impossible to detect via traditional testing—to leave a system vulnerable to hackers. Theoretically, formal verification offers a solution to this problem by proving that systems software behaves correctly with respect to its specifications, and protects data security under all circumstances. While formal verification dates back to the 1960s, complete formal proofs of uniprocessor OS kernels have only become feasible recently, most notably demonstrated by seL4 in 2009. While the seL4 example was encouraging, it was still prohibitively expensive to verify—the verification of its 9,000 lines of code (LOC) took 11 person-years to develop. Furthermore, verification of concurrent systems software remains out of reach.

My research is focused on **designing compositional verification theories and scalable frameworks to formally verify the correctness and security of concurrent safety-critical systems software**. I hypothesize that the above challenges are caused by ignoring the *layered structure* in the programming language theories. While modern systems are implemented with well-designed layers, this layered structure has not been exploited by the verification techniques like program logic: even with the most advanced methods, modules across different layers and multiple threads have to be reasoned about at the same abstraction level. This makes systems verification difficult to untangle and costly to extend.

My work is among the first to address these challenges by creating a novel class of specifications named *certified abstraction layers*, and a modular verification approach named *microverification*. In theory, these concepts uncover the insights of layered design patterns and are rich enough to fully characterize a system's functionality. In practice, they are "live" enough to connect with actual implementations, enabling a modular approach to building trustworthy systems software stacks without the need to verify every line of code. This layered approach can decompose an otherwise prohibitive verification task into many simple and independent tasks, which can then be verified at proper abstraction levels that are convenient for manual and automated proofs. I have also developed data-driven verification techniques and push-button verification frameworks to show that after a layered decomposition, many verification tasks such as invariants learning, and even the entire system, can be verified automatically. These pioneering advances have made possible a series of innovations that stand as milestones toward building trustworthy systems software, including: CertiKOS (the first verified multiprocessor OS kernel); SeKVM (the first verified commodity cloud hypervisor); and Giallar (the first push-button verification framework for the Qiskit quantum compiler). My work in building certified systems software has received three Amazon Research Awards, an OSDI Jay Lepreau Best Paper Award (2021), an SOSP Best Paper Award (2019), and a CACM research highlight (2019). The certified systems developed by my research team have been used in high-profile DARPA programs [CRASH, HACMS, V-SPELLS] and has been a core component of the DeepSpec NSF Expeditions in Computing project [DeepSpec], being widely considered "a real breakthrough" toward creating hacker-resistant systems [YaleNews, ColumbiaNews]. I have also started the technology transfer by founding CertiK, a Web3 security unicorn startup valued at $2 billion and backed by Insight Partners, Tiger Global, Sequoia, Coatue, Goldman Sachs, and Softbank Vision Fund [CertiKNews]. More than $300 billion worth of crypto assets are secured by products and services adopting my research results. My work has also been covered widely in popular media [TWSJ, CNBC, Bloomberg, Forbes]. Below I describe five major thrusts of my research.

Certified Abstraction Layers

Modern systems software consists of a multitude of abstraction layers (e.g., OS kernels, hypervisors, distributed protocols, and blockchain systems), each of which defines an interface that hides the implementation

details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Despite their obvious importance, abstraction layers have mostly been treated as a system concept; they have almost never been formally specified or verified. This makes it difficult to scale program verification across multiple layers.

I address the above challenges by introducing a novel language-based account of abstraction layers, named *certified abstraction layers* and show that they correspond to a strong form of abstraction over a particularly rich class of specifications, named *deep specifications* [POPL'15]. Abstraction over deep specification is characterized by an important implementation independence property: any two implementations of the same deep specification must have contextually equivalent behaviors. This allows program verification to be done over a more abstract interface that is implementation-independent. Each module is verified at its proper abstraction level by showing a contextual simulation between the implementation at that level and the deep specification at a higher level. Once this proof is done, the deep specification forms a new layer interface, called a *certified abstraction layer*. Any client program of this module can be reasoned about using this more abstract layer specification alone, without looking at the actual implementation.

To apply this layer-based verification approach to real systems, I developed the CertiKOS framework to specify, verify, and compose certified abstraction layers in the Coq proof assistant. Using CertiKOS, I have developed multiple certified uniprocessor OS kernels. The most realistic one is called mCertiKOS, which consists of 37 certified abstraction layers, took less than one person-year to develop, and can boot a version of Linux as a guest. Besides the power to build certified systems from scratch, my layer-based approach also makes it efficient to introduce new system features. Examples are my mCertiKOS extensions for verifying its device drivers [PLDI'16b, JAR'18], security properties [PLDI'16a, SOSP'19a], and scheduling properties [POPL'20]. An extended version of mCertiKOS was used by the DARPA CRASH program.

Certified Concurrent Abstraction Layers

Moving from the sequential kernel verification to the concurrent one is not a straightforward process and requires a more robust compositional theory. A concurrent kernel allows arbitrarily interleaved execution of kernel/user modules across different layers of abstraction. Many researchers even believe that the combination of concurrency and the kernels' functional complexity makes the formal verification intractable, and even if possible, the cost would far exceed that of verifying a uniprocessor kernel.

I hypothesize that the key to untangling this "intractable" complexity is rooted in the contextual property of deep specifications [PLDI'18, Patent'20a]. A deep specification in the concurrent setting must ensure behavior preservation not only under any client context but also under any *concurrent context* with any interleaving. A concurrent context corresponds to a particular execution of the kernel that encapsulates the behaviors of the rest of the CPUs (or threads), as well as their interference. The *certified concurrent layer* is then parameterized over this concurrent context. Given a particular concurrent context, the interleaving is determined, and a concurrent layer is reduced to a sequential one—this allows me to apply sequential verification techniques for building new concurrent layers. A newly introduced concurrent layer becomes "deep" only if its simulation proof holds for all valid concurrent contexts. To ease this step, I lift the concurrent machine model (which allows arbitrary interleaving at any point) to an abstract local machine model, where all interference from the concurrent context is restricted to certain specific points, i.e., just before the invocations of synchronization primitives. This machine lifting enables the *local reasoning* for multiprocessor programs.

Using this concurrent layer-based approach, I developed and verified a practical multiprocessor OS kernel in Coq [OSDI'16, Thesis'16]. This certified multiprocessor kernel is written in 6,500 LOC and runs on stock x86 multiprocessor machines. This is the first fully verified multiprocessor OS kernel with fine-grained locking [ASPLAS'17]. This work was selected as a CACM Research Highlight [CACM'19] and has been a core component of the DARPA HACMS program and the DeepSpec NSF Expeditions in Computing project. To demonstrate the extensibility of my concurrent layer-based approach, I also developed the first formally-verified, fine-grained, concurrent file system [SOSP'19b]. My work on verifying concurrent systems has been recognized as "a milestone that ... could lead to a new generation of reliable and secure systems software" [YaleNews].

Microverification of Commodity Systems Software

While the recent success shows that it is feasible to verify simple systems software, commodity systems software continues to elude verification due to their concurrent design and sprawling codebase. All previous verified systems, including my CertiKOS work, are far simpler than their commodity counterparts. For example, it took seL4 11 person-years to verify 9K LOC and CertiKOS three person-years to verify 6.5K LOC. By comparison, the Linux kernel is more than 2M LOC. It remains unknown how desired properties of such a vast system may be verified in its entirety.

To address this problem, I introduce *microverification*, a new approach for thoroughly verifying the security properties of large, multiprocessor commodity system software without needing to prove the correctness of all components of the overall system [S&P'21, Security'21, Patent'20b]. The hypothesis behind microverification is that it is tractable to verify existing commodity software with only minor adaptations to decompose the system into a small verifiable core which works with a larger set of untrusted services so that we can reason about the properties of the entire system by verifying the core alone. To verify the multiprocessor core, I introduce security-preserving layers to modularize the proof without hiding information leakage. This enables us to prove each layer of the implementation refines its specification, and the top layer specification is refined by all layers of the core implementation. We can then prove noninterference at the top layer specification and guarantee the resulting security properties hold for the entire system implementation.

Using microverification, I was able to formally verify for the first time guarantees of VM data confidentiality and integrity for a retrofitted version of the Linux KVM hypervisor, named SeKVM. I showed that microverification only required modest KVM modifications and proof effort, yet resulted in a verified hypervisor that retains KVM's extensive commodity hypervisor features and performance. I also showed that my SeKVM proofs hold on Arm relaxed memory hardware by introducing VRM, a new framework that makes it possible for the first time to verify concurrent systems software on Arm relaxed memory hardware [SOSP'21, Patent'22]. Using this microverification approach, I also built the security and correctness proof for Realms, a new abstraction for confidential computing to protect the data confidentiality and integrity of virtual machines, introduced by the Arm Confidential Compute Architecture [OSDI'22a]. The verified Realms firmware will be shipped with the Armv9 chips. My SeKVM work is the first machine-checked proof for a commodity multiprocessor hypervisor. SeKVM is used in the high-profile DARPA VSPELL program, has won two Amazon Research Awards, and is considered to have "[laid] the foundation for future innovations in system software verification" [ColumbiaNews].

Data-driven Invariants Learning

The layered verification approach makes it possible to decompose a complex verification task into many small and simple verification ones, which can then be verified at proper abstraction levels using different techniques to reduce proof effort. One of the main challenges behind automating these verification tasks is the inference of invariants. To scale systems verification, I have developed data-driven techniques to automatically infer invariants for loops and distributed protocols as described below.

Verifying systems software with loops requires determining loop invariants. Automatically inferring loop invariants, especially for nonlinear ones, is challenging due to the potential for overfitting on a small number of samples, and the large space of possible nonlinear inequality bounds. To address the challenges, I introduced Gated Continuous Logic Network (G-CLN), a novel neural architecture for learning loop invariants directly from *data*, i.e., program execution traces [ICLR'20, PLDI'20]. Unlike existing neural networks, G-CLNs can learn precise and explicit representations of formulas in Satisfiability Modulo Theories (SMT). I developed a sound and complete semantic mapping for assigning SMT formulas to continuous truth values that allows G-CLNs to be trained efficiently. To address overfitting that arises from finite program sampling, I introduced fractional sampling—a sound relaxation of loop semantics to continuous functions that facilitates unbounded sampling on the real domain. I also designed a new activation function for naturally learning tight inequality bounds. I evaluated G-CLN on the standard benchmark and showed that it solves more loop invariants than prior work, with an average runtime of 53.3 seconds.

Finding the inductive invariant of distributed protocols is a critical step in verifying the correctness of

distributed systems, but it is a time consuming process, even for simple protocols. To address the challenges, I developed DistAI, a data-driven automated system for learning inductive invariants for distributed protocols [OSDI'21, OSDI'22b]. DistAI generates data by simulating the distributed protocol at different instance sizes and recording states as samples. Based on the observation that invariants are often concise in practice, DistAI operates in formula space, starts with small invariant formulas, and enumerates all strongest possible invariants that hold for all samples. Because DistAI starts with the strongest possible invariants, if they are not inductive, DistAI knows to monotonically weaken them, repeating the process until it eventually succeeds. I proved that DistAI is guaranteed to find the inductive invariant that proves the desired safety properties in finite time if one exists. The evaluation shows that DistAI successfully verifies 26 common distributed protocols automatically, including various versions of Paxos. It also shows that DistAI outperforms alternative methods both in the number of protocols it verifies, and the speed at which it does so, including solving Paxos more than an order of magnitude faster than any previous method. DistAI received an OSDI Jay Lepreau Best Paper Award (2021) and an Amazon Research Award.

Verification of Quantum Software

My verification techniques can build not only trustworthy classical systems software but also trustworthy quantum software. Quantum compilers are my first verification target because their role in bridging quantum applications to hardware makes them essential components in the quantum software stack. However, they are hard to implement correctly due to nonintuitive quantum mechanics rules, and heavy optimizations to fit quantum programs into real quantum devices of limited qubit lifetime and connectivity. These challenges make quantum compilers error-prone. For example, Bugs4Q finds 27 bugs in the Qiskit compiler, which is the most widely-used open-source quantum compiler with more than 100K users from 171 countries. Undetected bugs in the Qiskit compiler can corrupt the computation results of the millions of simulations and real runs on quantum devices. To address the problem, I developed Giallar, a toolkit that can automatically verify that a compiler pass preserves the semantics of quantum circuits in a *push-button* manner [PLDI'22]. While the efficient equivalence checking for general quantum circuits is still beyond reach, Giallar shows that the output quantum circuit can be obtained from the input circuit through a sequence of equivalent rewrites. With Giallar, I implemented and verified 44 (out of 56) compiler passes in 13 versions of the Qiskit compiler, during which three bugs were detected and confirmed by the Qiskit team. The evaluation shows that most Qiskit compiler passes can be automatically verified in seconds, and verification imposes only a modest overhead on compilation performance.

My second verification target is to verify the error bound in quantum programs. Practical error analysis is essential for the design, optimization, and evaluation of Noisy Intermediate-Scale Quantum (NISQ) computing. However, bounding errors in quantum programs is a grand challenge, because the effects of quantum errors depend on exponentially large quantum states. To address this problem, I developed Gleipnir, a novel methodology toward practically computing verified error bounds in quantum programs [PLDI'21]. Gleipnir introduces the $(\hat{\rho}, \delta)$-diamond norm, an error metric constrained by a quantum predicate consisting of the approximate state $\hat{\rho}$ and its distance δ to the ideal state ρ. This predicate $(\hat{\rho}, \delta)$ can be computed adaptively using tensor networks based on Matrix Product States. Gleipnir features a lightweight logic for reasoning about error bounds in noisy quantum programs, based on the $(\hat{\rho}, \delta)$-diamond norm metric. To understand the scalability and limitation of Gleipnir, I conducted case studies using two classes of quantum programs that are expected to be most useful in the near-term—the Quantum Approximate Optimization Algorithm and the Ising model—with qubits ranging from 10 to 100. The measurements show that, with 128-wide MPS networks, Gleipnir can always generate error bounds within 6 minutes. For small programs ($\leq$ 10 qubits), Gleipnir's error bounds are as precise as the ones generated using full simulation. For large programs ($\geq$ 20 qubits), Gleipnir's error bounds are 15% to 30% tighter than those calculated using unconstrained diamond norms, while full simulation invariably times out after 24 hours.

Future Directions

I plan to continue working on the five major thrusts of my research for scaling systems verification. Below, I provide two examples of ongoing works that are in keeping with this direction.

Programming certified systems software directly. My existing projects on systems verification, including CertiKOS and SeKVM, all require (manually) writing the actual implementation in a C-like language and a formal specification in a proof assistant language. Lacking the support of writing certified programs directly makes the certified software systems difficult to develop and maintain. I will explore how to bridge this enormous gap between low-level system programming and high-level specification reasoning through a uniform framework that enables programming certified software directly. This goal is ambitious but still promising. Because deep specifications precisely capture the contextual functionality of their implementations, why not write only the layer specifications and then generate the whole system from the layers automatically? When focusing on a single verification task between two layers, the gap between the implementation and its deep specification, as well as the gap between two adjacent specifications, are both relatively small. I aim to exploit this uniform framework in more depth by attempting: (1) to generate a complete kernel from layer specifications following the line of the program synthesis work; and (2) to link adjacent layers automatically by taking advantage of the recent progress in the artificial intelligence.

Build trustworthy systems software in new domains Besides OS kernels, hypervisors, distributed systems, and quantum compilers, I have also applied formal verification techniques in other domains, such as blockchain smart contracts [WS'22], software dependency updates [OSDI'22c], preventing correlated failures in the cloud [OOPSLA'17], and concurrency sampling [CAV'18]. I will further investigate more powerful and more compositional verification frameworks to build more certified systems software:

- I will continue to build a zero-vulnerability system stack consisting of verified components, such as database systems, file systems, network stacks, operating systems, and distributed systems. Although each of these components has been actively studied, there is a high demand to link all their guarantees together to form a trustworthy system stack. My success with certified abstraction layers and microverification reveals a promising way to address these challenges. I can provide "customized" machine models for different systems by exposing an interface that abstracts a particular set of system features. This will enable a domain-specific approach to building certified layers for each system separately. The major remaining issue is linking together all of these layers with different reliability requirements and system features using a single compositional theory.

- I will explore how to build high-confidence CPS with real-time assurance. Due to the physical consequences, proving that a CPS behaves as required regarding both functionality and timing is highly desirable. We have built an embedded system (a variant of mCertiKOS) that is proved to be functionally correct and temporally isolated, and can run on a real drone. However, the most difficult problem is how to formally model time in the theory. My proposal is to abstract the flow of time as a list of special events, which will be indicated by a timing oracle that is enriched from concurrent context. I can apply the concurrent verification techniques to prove real-time properties beyond the temporal isolation with this model.

- I will explore how to establish a general and efficient specification-based testing framework. For concurrent software (e.g., user-level apps) without high-reliability requirements, the complete formal verification is unnecessary and too expensive, while the random testing has difficulty detecting concurrency bugs caused by certain interleavings. I plan to solve this research problem by utilizing deep specifications to intelligently generate the test cases in the concurrent setting. Due to the "live" property mentioned earlier, I can run these tests over the specifications directly, without waiting for the responses from heavy-workload operations.

References

[ASPLAS'17] J. Kim, V. Sjöberg, R. Gu, and Z. Shao. Safety and liveness of MCS lock—layer by layer. In *Asian Symposium on Programming Languages and Systems (ASPLAS 2017)*, pages 273–297. Springer, 2017.

[Bloomberg] Hackers seize $80 million from qubit in latest defi attack. https://www.bloomberg.com/news/articles/2022-01-28/hackers-seize-80-million-from-qubit-in-lastest-defi-attack.

[CACM'19] R. Gu, Z. Shao, H. Chen, J. Kim, J. Koenig, X. N. Wu, V. Sjöberg, and D. Costanzo. Building certified concurrent OS kernels. *Communications of the ACM (CACM 2019)*, 62(10):89–99, 2019.

[CAV'18] X. Yuan, J. Yang, and R. Gu. Partial order aware concurrency sampling. In *International Conference on Computer Aided Verification (CAV 2018)*, pages 317–335. Springer, 2018.

[CertiKNews] Goldman Sachs joins other investors in $88m round for web3 and blockchain security firm CertiK. https://techcrunch.com/2022/04/07/goldman-sachs-joins-other-investors-in-88m-round.

[CNBC] More than $320 million stolen in latest apparent crypto hack. https://www.cnbc.com/2022/02/02/320-million-stolen-from-wormhole-bridge-linking-solana-and-ethereum.html.

[ColumbiaNews] Columbia engineering team builds first hacker-resistant cloud software system. https://www.engineering.columbia.edu/news/first-hacker-resistant-cloud-software-system.

[CRASH] Clean-slate design of resilient, adaptive, secure hosts (CRASH). https://www.darpa.mil/program/clean-slate-design-of-resilient-adaptive-secure-hosts.

[DeepSpec] DeepSpec: The science of deep specifications. http://deepspec.org/.

[Forbes] Blockchain smart contracts: More trouble than they are worth? https://www.forbes.com/sites/shermanlee/2018/07/10/blockchain-smart-contracts-more-trouble-than-they-are-worth/?sh=70144f9623a6.

[HACMS] High-assurance cyber military systems (HACMS). http://opencatalog.darpa.mil/HACMS.html.

[ICLR'20] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *Proceedings of the 8th International Conference on Learning Representations (ICLR 2020)*, 2020.

[JAR'18] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. *Journal of Automated Reasoning (JAR 2018)*, 61(1-4):141–189, 2018.

[OOPSLA'17] E. Zhai, R. Piskac, R. Gu, X. Lao, and X. Wang. An auditing language for preventing correlated failures in the cloud. volume 1, page 97. ACM, 2017.

[OSDI'16] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 653–669, 2016.

[OSDI'21] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021)*, pages 405–421, 2021.

[OSDI'22a] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 465–484, 2022.

[OSDI'22b] J. Yao, R. Tao, R. Gu, and J. Nieh. Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 485–501, 2022.

[OSDI'22c] Y. David, X. Sun, R. J. Sofaer, A. Senthilnathan, J. Yang, Z. Zuo, G. H. Xu, J. Nieh, and R. Gu. UPGRADVISOR: Early adopting dependency updates using hybrid program analysis and hardware tracing. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 751–767, 2022.

[Patent'20a] Z. Shao, R. Gu, V. Sjoberg, J. Kim, and J. Koenig. Systems and methods of formal verification, Dec. 10 2020. US Patent App. 16/767,569.

[Patent'20b] S.-W. Li, L. Xupeng, R. Gu, and J. Nieh. Systems, methods, and media for trusted hypervisors, Dec. 31 2020. US Patent App. 16/916,051.

[Patent'22] R. Gu, J. Nieh, and R. Tao. Systems, methods, and media for proving the correctness of software on relaxed memory hardware, Jan. 20 2022. US Patent App. 17/376,120.

[PLDI'16a] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 648–664. ACM, 2016.

[PLDI'16b] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 431–447. ACM, 2016.

[PLDI'18] R. Gu, Z. Shao, J. Kim, X. N. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661. ACM, 2018.

[PLDI'20] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, pages 106–120, 2020.

[PLDI'21] R. Tao, Y. Shi, J. Yao, J. Hui, F. T. Chong, and R. Gu. Gleipnir: toward practical error analysis for quantum programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*, pages 48–64, 2021.

[PLDI'22] R. Tao, Y. Shi, J. Yao, X. Li, A. Javadi-Abhari, A. Cross, F. Chong, and R. Gu. Giallar: Push-button verification for the qiskit quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2022)*, pages 641–656, 2022.

[POPL'15] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608. ACM, 2015.

[POPL'20] M. Liu, L. Rieg, Z. Shao, R. Gu, D. Costanzo, J.-E. Kim, and M. Yoon. Virtual timeline: A formal abstraction for verifying preemptive schedulers with temporal isolation. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL 2020)*. ACM, 2020.

[Security'21] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, 2021.

[SOSP'19a] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242. ACM, 2019.

[SOSP'19b] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 259–274. ACM, 2019.

[SOSP'21] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, 2021.

[S&P'21] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A secure and formally verified linux kvm hypervisor. In *2021 IEEE Symposium on Security and Privacy (S&P 2021)*, pages 1782–1799. IEEE, 2021.

[Thesis'16] R. Gu. *An Extensible Architecture for Building Certified Sequential and Concurrent OS Kernels*. PhD thesis, Yale University, 2016.

[TWSJ] How hackers target bridges between blockchains for crypto heists. https://www.wsj.com/articles/how-hackers-target-bridges-between-blockchains-for-crypto-heists-11649151001?mod=markets_minor_pos1.

[V-SPELLS] Verified security and performance enhancement of large legacy software (v-spells). https://www.darpa.mil/program/verified-security-and-performance-enhancement-of-large-legacy-software.

[WS'22] S. Lin, X. Sun, J. Yao, and R. Gu. Scivik: A versatile framework for specifying and verifying smart contracts. In *Memorial Volume for Shoucheng Zhang*, pages 403–437. World Scientific, 2022.

[YaleNews] CertiKOS: A breakthrough toward hacker-resistant operating systems. http://news.yale.edu/2016/11/14/certikos-breakthrough-toward-hacker-resistant-operating-systems.