Teaching Statement

Kaiming Huang

I was born in the hometown of Confucius, where education and teaching have long been deeply valued. A familiar Chinese saying captures the goal I bring to my own teaching: *"Give a person a fish and you feed them for a day; teach them how to fish and you feed them for a lifetime."* In computer science, I interpret this principle as helping students develop both the ability and the curiosity to continue learning beyond a particular assignment, programming language, or course. My own experience has shown me how powerful that curiosity can be. As an undergraduate, my program offered no course on Capture the Flag (CTF) competitions; nevertheless, over four years I learned a great deal about CTF challenges outside the classroom, driven by my own interest and supported by faculty and fellow students who shared their knowledge and enthusiasm. That experience taught me that effective teaching can do more than transfer knowledge: it can give students the tools and motivation to explore questions on their own. My goal is therefore not only to show students how to solve a problem, but to help them understand why a solution works and what assumptions it relies on, while leaving them curious and prepared to tackle problems they have not seen before.

Make difficult ideas accessible. Two lessons from my Ph.D. advisor, Prof. Trent Jaeger, have strongly influenced how I approach teaching. The first is that *any technical talk should be accessible to a second-year graduate student*. To me, this means that technical depth should come from the substance of the material, not from unnecessary complexity in how it is presented. I therefore try to build explanations from the problem and its intuition toward the underlying mechanism, introducing terminology and technical details only after students understand why they are needed. To put this principle into practice, I also draw on a lesson from my Ph.D. co-advisor, Prof. Jack Sampson: use analogies to connect unfamiliar concepts with familiar ones, then refine them as greater precision becomes necessary. When students struggle with a concept, I first ask whether I have found the right abstraction, analogy, example, or sequence of explanation rather than simply repeating the same explanation.

Build an approachable classroom. The second lesson is even simpler: *be nice to people*. I have found this advice especially important in teaching, because students often come to us with not only technical questions, but also frustration and self-doubt. During one semester as a teaching assistant, a student came to my office hours visibly anxious about the course. Before discussing her assignment, I first talked with her until she felt comfortable enough to work through the problem with me. Later in the semester, she did not perform as well as she had hoped on an exam and became very upset. I told her, "It is just an exam." One difficult exam did not define what she was capable of learning or accomplishing. That experience has stayed with me because it reminded me that being approachable is part of teaching, not something separate from it. Students are more willing to ask questions, admit confusion, and keep working after a setback when they know that their instructor sees them as more than a grade. I want my students to know that I will hold them to high standards, but that they can always approach me when they need help.

Turn explanation into practice. I had the opportunity to put these principles into practice as an instructor for a Python course at Penn State. During the fifteen-week semester, I taught two 75-minute recitation sessions each week and independently designed the material and exercises for those sessions. I viewed recitation not as a repetition of lecture, but as a bridge between understanding a concept and being able to apply it. I therefore organized each session around concepts that students were likely to find difficult and paired explanations with exercises that required them to use the ideas immediately. I also held three hours of office hours each week, which often became informal round-table discussions with ten or twenty students. Rather than simply correcting a student's code, I would ask them to explain what they expected the program to do, compare that expectation with its actual behavior, and identify the assumption that led to the difference. When several students encountered related problems, I could turn an individual question into a discussion from which the entire group learned. These experiences reinforced my belief that the most useful help is to show the reasoning that can be applied to the next one.

Encourage independent reasoning. At the same time, I recognize that students do not all arrive at the same understanding through the same path. Two students who produce the same incorrect answer may have arrived there for completely different reasons, so I first try to understand how a student is reasoning rather than simply correct the final answer. I also encourage students to explain that reasoning confidently even when they are unsure whether it is

correct. An incorrect answer can still be productive if a student can explain why it seemed reasonable, because it gives us something concrete to examine together. This requires trust in both directions: students should feel comfortable questioning an expected answer, including mine, and know that their reasoning will be taken seriously. Everyone can be wrong; what appears to be a student's mistake may instead reveal an ambiguity in the problem, an incorrect answer key, or an instructor's mistaken judgment. I therefore want students to support their conclusions with reasoning rather than defer automatically to authority, while remaining willing to revise those conclusions when the evidence points elsewhere.

Design assessments for understanding. Beyond the Python recitations, I have served as a teaching assistant for both Software Security and System Programming. I have also delivered guest lectures in Software Security on topics including reference monitors and memory safety. My role in System Programming gave me additional experience in course design. In addition to grading and helping students, I developed exam questions and project components. This experience made me think carefully about what an assignment or exam question is actually asking students to demonstrate. A difficult question is not necessarily a good question: if students fail because the wording is confusing or because they have to guess what the instructor intends, the assessment tells us little about whether they understand the underlying concept. I therefore try to design questions that make the intended concept clear while still requiring students to reason through the problem. Similarly, for programming projects, I want students to succeed because they understand the system they are building, not because they happened to find a combination of changes that passes the tests. This experience taught me that assignments and exams should not only measure learning, but also reinforce students' ability to think independently.

Cultivate research independence. I extend the same teaching philosophy to research mentoring. I have mentored three undergraduate students through honors theses in software and systems security and have also worked closely with junior Ph.D. students entering new research areas, with several of these collaborations contributing to top-tier conference papers. These experiences have strengthened my ability to mentor students at different stages. My goal in research mentoring is to gradually transfer ownership of the problem from myself to the student. At the beginning of a project, I help students turn a broad idea into concrete research questions and tractable technical tasks. As they gain experience, I increasingly ask them to interpret unexpected results, compare alternative explanations, and propose the next experiment themselves. I also encourage them to treat failed experiments as useful evidence by asking what hypothesis was ruled out and what should be tested next. Ultimately, I want students to progress from asking, "What should I do next?" to explaining what they believe the next step should be and why. As a faculty member, I plan to extend this mentoring model across undergraduate, M.S., and Ph.D. researchers. Well-scoped implementation and evaluation tasks can provide accessible entry points, while more advanced students can progressively take ownership of problem formulation, analysis design, and research direction. This progression from guided participation to independent research judgment is, to me, the research counterpart of teaching someone how to fish.

As a faculty member, I would be excited to teach undergraduate and graduate courses in systems and security, including Systems Programming, Operating Systems, System and Software Security, and Program Analysis. I would also like to develop an advanced course on **Software Security and Program Analysis**, covering topics such as static and dynamic analysis, symbolic execution, fuzzing, memory safety, and binary analysis. A second course, **Verification and Security for AI-Enabled Systems**, would introduce formal verification, adversarial robustness, probabilistic guarantees, and the opportunities and limitations of generative models in security analysis. Across these courses, I would emphasize the same progression that guides my teaching: begin with the underlying property or invariant, examine how real systems violate it, and then reason about what a proposed technique can guarantee.

Ultimately, I want students to leave my classroom with more than knowledge of particular tools or systems. I want them to remain curious about questions beyond the syllabus, confident enough to articulate and defend their own reasoning, and intellectually humble enough to revise that reasoning when evidence shows that they are wrong. I also hope they learn to approach difficult problems—and the people working through those problems with them—with patience and respect. Whether I am teaching an introductory programming student, explaining a systems concept to a graduate class, or mentoring an undergraduate researcher through an open-ended project, my goal is the same: **to make difficult ideas accessible, encourage students to think and speak independently, and help them develop the curiosity, confidence, kindness, and judgment to continue learning on their own.**