Artificial intelligence (AI) is now highly proficient at coding and deeply integrated into software development. While this greatly boosts productivity, it also marks a profound shift in cybersecurity: AI is reshaping how vulnerabilities are created, detected, exploited, and fixed. This dual nature, AI as both a vulnerability amplifier and a security enabler, has significant implications for how we build and secure software. Cybersecurity is also a central concern for AI safety, as underscored by both the White House and OpenAI's Preparedness Framework.

My research seeks to transform AI from a source of vulnerabilities into a cornerstone of trustworthy software, and more broadly, of AI safety. To realize this vision, I pursue two complementary directions: (i) **systematically benchmarking AI's cybersecurity capabilities and risks**, and (ii) **developing secure-by-design methods for AI-based code generation**. The first direction establishes a quantitative basis for understanding the evolving AI-cybersecurity landscape, while the second builds on these insights to proactively address the emerging risks of AI-generated vulnerabilities.

To systematically benchmark AI in cybersecurity, I decompose the problem into *different roles* AI can play in the vulnerability lifecycle. Acting as *attackers and defenders*, AI can assist in key tasks like vulnerability reproduction and discovery. Our benchmark, CyberGym [1], reveals a rapid increase in AI's capabilities on these tasks within just months. Underscoring its practical relevance, **CyberGym was adopted by Anthropic in the Claude Sonnet 4.5 System Card**. Acting as an *insecure programmer*, AI can inadvertently introduce and propagate vulnerabilities at scale. To measure this new risk, we created BaxBench [2] to evaluate the security and correctness of AI-generated backend code, revealing that even state-of-the-art models frequently produce vulnerable code. BaxBench was featured as a *Spotlight paper at ICML 2025*.

To mitigate AI-generated vulnerabilities, my approach moves beyond traditional post-hoc vulnerability detection and patching, advancing a *secure-by-design* paradigm. I developed the first techniques that embed security directly into models' training and inference processes, *enabling AI to generate secure code from the outset*. Our security-aware fine-tuning [3] teaches large language models (LLMs) to follow best security practices encoded in historical security patches while generating functional code. **This work won a Distinguished Paper Award at ACM CCS 2023**, and **its impact is evident in multi-million-dollar initiatives like Amazon Nova AI Challenge**. Moreover, our type-constraining method [4] enforces formal typing constraints during inference, guiding LLMs to produce type-safe code that eliminates entire classes of vulnerabilities and errors. This work sparkled significant discussion in the broader tech community, *reaching #1 on Hacker News*.

My research is grounded in three complementary domains: (i) *Security*: to reason about vulnerabilities, e.g., [1, 2]; (ii) *Machine Learning*: to analyze and build AI models, e.g., [3]; and (iii) *Programming Languages*: to leverage formal methods for safety guarantees, e.g., [4]. This integration enables a holistic approach to addressing challenges at the intersection of AI and cybersecurity that spans evaluation, training, and inference-time enforcement. I developed this cross-disciplinary expertise throughout my academic journey, as depicted in Figure 1. My PhD work built a foundation in program analysis and machine learning (with application to security), integrating formal reasoning with data-driven methods. As LLMs pushed general AI coding tools from research prototypes to industry practice, I recognized that security and reliability would become the next critical frontier. This insight shaped my transition toward AI and cybersecurity, a path I initiated in my late PhD, expanded during my postdoc, and continue to pursue. My line of research bridging these fields was recognized with the **ETH Medal for Outstanding Doctoral Thesis**.

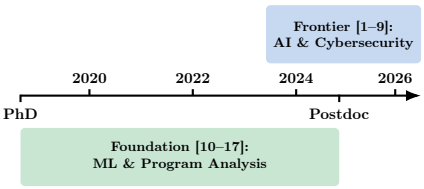

Figure 1: My research journey.

Benchmarking the Cybersecurity Capabilities and Risks of AI

Cybersecurity benchmarks serve diverse stakeholders: researchers use them to characterize emerging behaviors, frontier AI labs to improve their models, practitioners to deploy AI tools safely, and policymakers to design

evidence-based governance. Yet, current evaluation practices remain narrow in scope and scale. For example, OpenAI's evaluation (e.g., on GPT-5) relies on roughly a hundred capture-the-flag (CTF) problems. These tasks, originally designed for educational or competition purposes, do not capture the breadth and complexity of real-world security challenges. My research bridges this gap by developing cybersecurity benchmarks across different stages of the vulnerability lifecycle, as visualized in Figure 2.

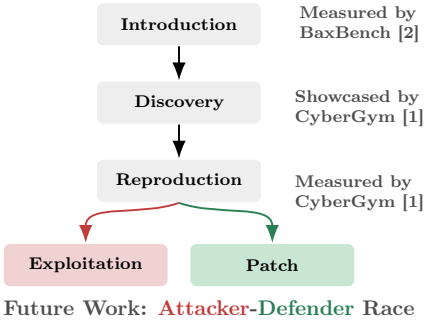

Figure 2: My research on benchmarking the vulnerability lifecycle.

CyberGym: Quantifying AI's Rapidly Advancing Cybersecurity Capabilities To capture both attackers and defenders, CyberGym primarily targets the task of *vulnerability reproduction*: given a textual vulnerability description and the corresponding codebase, the agent must generate a proof-of-concept test that reproduces the vulnerability. This task is challenging, requiring statically locate the flaw in large repositories and effectively trigger it at runtime. CyberGym couples this task with a diverse corpus of 1,507 real-world vulnerabilities, drawn from OSS-Fuzz, Google's continuous fuzzing platform. Together, these elements establish CyberGym as a graded ladder of difficulty to effectively measure the rapidly evolving cybersecurity landscape. Evaluations with CyberGym show that Anthropic's Claude models nearly tripled their success rate in just 7 months (11.9% on v3.7 and 28.9% on v4.5). Moreover, test-time scaling, a technique easily accessible to attackers, amplifies these capabilities: running 30 trials boosts v4.5's performance from 28.9% to 66.7%. Beyond reproducing historical flaws, the agents also *discovered 35 previously unknown vulnerabilities*, many of which had remained undetected after years of OSS-Fuzz's continuous fuzzing.

BaxBench: Revealing the Pervasive Security Risks in AI-Generated Code BaxBench [2] evaluates how AI *introduces vulnerabilities*, a threat emerging as AI becomes a "pair programmer". It tasks LLMs with generating standalone backend applications, a domain that is practically relevant, functionally complex, and security-critical. The benchmark contains 392 instances, built from a matrix of 28 realistic development scenarios and 14 diverse implementation frameworks. BaxBench assesses both functional correctness and security: each generated backend is checked with unit tests and concrete exploits. These tests and exploits are written by experts and tailored for each scenario, ensuring rigorous and reproducible evaluations. Experiments on BaxBench expose that even when models produce correct code, they often introduce security flaws. For instance, Anthropic's Claude models, despite their strong coding reputation, consistently produce vulnerable code in 30%-40% of its correct outputs. This silent failure mode is particularly concerning because it gives the appearance of correctness while embedding exploitable weaknesses.

Near-Term Future Work: Benchmarking the Attacker and Defender Race As depicted in Figure 2, CyberGym and BaxBench illuminate the early phases of the vulnerability lifecycle. Beyond these stages, the path diverges into a race between attacker exploitation and defender remediation. I plan to extend CyberGym to measure this race, answering the key question: *which side benefits more from AI?* Rather than developing isolated benchmarks, I will unify the attacker and defender paths within a single, comprehensive framework that enables direct, side-by-side comparison.

I will first focus on the attacker path, evaluating how AI can execute end-to-end exploitation. Real-world exploitation is complex, often chaining multiple escalating steps. For instance, evolving an initial buffer overflow vulnerability into manipulated memory layout, arbitrary read/write, hijacked control flow, and ultimately system compromise. These tasks are challenging for AI, as they require reasoning over low-level program states, not just high-level source code. My approach is to decompose this "full kill chain": my framework will first benchmark individual escalation steps, then compose them into a multi-stage evaluation. I will develop sandboxed environments with robust validators to ensure safe and reproducible evaluation, while adhering to

ethical standards to prevent harmful outcomes. On the defender side, my framework will measure how AI can repair the same set of vulnerabilities used for the attacker path. Benchmarking repair must consider multiple factors: a correct patch must remove the vulnerability, preserve functionality, and avoid introducing new flaws. Our defender benchmark will address this by providing comprehensive functional and security tests that are either expert-written or generated by fuzzers and AI with validation.

Developing Secure-by-Design AI for Code Generation

BaxBench and CyberGym reveal a concerning trend: as AI accelerates software development, it simultaneously amplifies the introduction and discovery of vulnerabilities. This raises a key question: *How can we harness AI not as a source of risk, but as an builder for secure and reliable systems?* Traditional software security is largely reactive, relying on detecting and patching flaws. This post-hoc model, built on the assumption that code is human-written and human errors are often inevitable, is suboptimal for the speed and algorithmic nature of AI. Rather than reacting to AI-generated flaws, we have an opportunity to redesign AI itself, engineering it to internalize security principles and generate secure code from the outset. This approach prevents classes of vulnerabilities upfront, reducing attack surfaces and reliance on costly post-hoc mitigation. My research drives the shift from post-hoc defense to a secure-by-design paradigm for LLM-based code generation, as illustrated in Figure 3. I achieve this by embedding security directly into LLMs' training [3] and inference [4] processes.

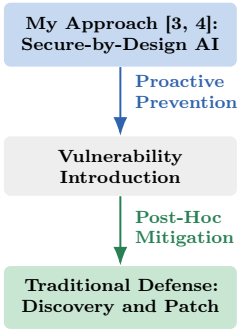

Figure 3: My proactive approach vs. post-hoc defense.

Security-Aware Fine-Tuning: Teaching LLMs to Generate Secure Code Humans learn from past vulnerabilities not just to become better patchers, but to write more secure code. My work [3] applies this principle by fine-tuning LLMs on historical patches so they internalize secure coding practices. We leverage patched code pairs as a security preference signal, guiding models to favor the fixed, secure code over the insecure original. To make this signal more explicit, we fine-tune the model to imitate the secure edits in the diff regions, i.e., the exact parts of code modified by human developers to remove vulnerabilities. To preserve general coding fluency, we apply regularization on the unchanged code regions against the original pretrained model. Recognizing that prior datasets are limited in data quality, we curated a high-quality fine-tuning dataset covering nine critical MITRE Top-25 weaknesses. Our approach is effective across LLMs of different families and sizes. For instance, applied to a 2.7B CodeGen model, our approach boosts the ratio of secure generations from 59.1% to 92.3%, all while maintaining general coding utility.

Type-Constrained Code Generation: Enforcing Safety at Inference Time Fine-tuning teaches the model to prefer secure patterns, but it is a probabilistic approach. To complement this, I also explore providing formal guarantees at inference time. In type-constrained code generation [4], we enforce correctness and safety based on the classic principle that "well-typed programs do not go wrong". Our key idea is to integrate an incremental type checker directly into LLMs' decoding loop. At each generation step, this checker maintains the typing environment of the current program prefix and prunes any token that would violate the language's typing rules. This process guarantees that the model can only generate type-safe continuations, eliminating entire classes of bugs captured by the type system. We build such a checker using a *prefix automaton* and a *type reachability* search, formalizing it on a simply typed language and implementing it on a significant subset of TypeScript. Evaluations across various coding tasks and LLMs show that type-constrained decoding reduces compilation errors by over 50% while improving functional correctness (up to 86.7% relatively).

Near-Term Future Work: Semantic Security Reasoning and Stronger Language-Level Safety My future research will advance secure-by-design AI for code generation by continuously enhancing both model capabilities and inference-time enforcement. First, I will work to enable semantic-level security understanding

and reasoning in frontier LLMs. While my prior work [3, 6] showed that LLMs can learn local security patterns, today's advanced reasoning models hold the potential for a semantic-level, generalized security understanding through multi-step thinking and self-refinement. However, our findings from BaxBench [2] reveal a critical gap: even currently the best reasoning models lack this deep reasoning ability and still frequently produce vulnerable code. I will therefore develop new methods to unleash this deep understanding, creating AI models that can genuinely reason about security and validate their own decisions. Second, building on my type-constraining work [4], I plan to explore enforcing richer language-level safety guarantees at inference time. Examples include guarantees from Rust's type system (to prevent memory safety issues and data races) and information-flow type system (to prevent data leaks and injection). Simultaneously enforcing these constraints during generation can achieve a stronger level of provable safety. This approach is well suited to AI, as models are not constrained by the human learnability or ergonomic barriers that often limit developer adoption of these stricter features.

Long-Term Research Roadmap

In the previous sections, I have detailed my *near-term plans* that build directly on my current thrusts. Looking further, my *long-term research roadmap* is driven by an overarching goal: creating AI systems that can reliably assist in developing software systems that are verifiably secure, correct, and trustworthy. This goal addresses not only a productivity challenge but also a core issue in AI safety. Below, I outline the three synergistic research thrusts that build toward this long-term vision.

I aim to deepen secure code generation for different security-critical domains. As AI-assisted coding gets adopted across diverse fields, in-depth specialization becomes critical for security, especially for high-stakes domains such as cryptography (e.g., avoiding API misuses), networks (e.g., secure configurations), and finance (e.g., transaction integrity). However, this specialization is challenging, as it must overcome the scarcity of in-domain data, subtle semantic flaws (e.g., timing leaks), and high dependency on usage context. In collaboration with domain experts, my research will first establish benchmarks defining domain-specific security challenges and then develop adapted secure-by-design generation techniques. One direction involves designing an expressive domain-specific language that enables domain experts to define their security constraints.

I will advance from secure code generation toward secure system synthesis. Beyond code-level errors, modern vulnerabilities often emerge from insecure system-level interactions, such as improper data flows or access control. My goal is to develop an AI system that reasons about and avoids these system-level vulnerabilities. This requires overcoming the "architectural blind spot" of current LLMs, which are trained on isolated code snippets, not system-wide design principles. To this end, I will extend my secure-by-design paradigm [3, 4] to operate on system representations, not just code. By embedding global security reasoning into the synthesis process, I aim to enable AI to construct secure and reliable systems. BaxBench already demonstrates the feasibility of scaling evaluation from individual functions to full backend applications; I will extend this approach to larger, multi-component systems.

I will generalize from security to building trust in broader programming. While security is a primary concern, the broader lack of trust in AI-assisted programming fundamentally stems from the lack of verifiable guarantees in its outputs. My work on type-constrained generation [4] achieves language-level guarantees, but general trust requires reasoning about richer properties. To this end, I will advance the "verifiable code generation" paradigm that we recently explored [5, 18]: developing an AI that generates not only code but also its formal specification and a machine-checkable proof. This approach, however, hinges on the "specification problem": a program is only as good as its specification, and developers must ultimately validate if the specification aligns with their intent. I will address this by creating an AI "auto-informalizer" that translates complex formal specifications back into text explanations. This process improves security, correctness, and transparency, making AI a trusted partner for dependable software.

References

[1] Zhun Wang*, Tianneng Shi*, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI Agents' Real-World Cybersecurity Capabilities at Scale. *arXiv.2506.02548*, 2025. **Adopted by Anthropic's Claude Sonnet 4.5 System Card**.

[2] Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanovic, Jingxuan He, and Martin Vechev. BaxBench: Can LLMs Generate Correct and Secure Backends? In *ICML*, 2025. **Spotlight Paper**.

[3] Jingxuan He and Martin Vechev. Large Language Models for Code: Security Hardening and Adversarial Testing. In *ACM CCS*, 2023. **Distinguished Paper Award**.

[4] Niels Mündler*, Jingxuan He*, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-Constrained Code Generation with Language Models. *Proc. ACM Program. Lang.*, (PLDI), 2025.

[5] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal Mathematical Reasoning: A New Frontier in AI. In *ICML*, 2025. **Spotlight Paper**.

[6] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction Tuning for Secure Code Generation. In *ICML*, 2024.

[7] Slobodan Jenko, Niels Mündler, Jingxuan He, Mark Vero, and Martin Vechev. Black-box Adversarial Attacks on LLM-Based Code Completion. In *ICML*, 2025.

[8] Kazuki Egashira, Mark Vero, Robin Staab, Jingxuan He, and Martin Vechev. Exploiting LLM Quantization. In *NeurIPS*, 2024.

[9] Kazuki Egashira, Robin Staab, Mark Vero, Jingxuan He, and Martin Vechev. Mind the Gap: A Practical Attack on GGUF Quantization. In *ICML*, 2025.

[10] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *NeurIPS*, 2024.

[11] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *ICML*, 2021.

[12] Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. On Distribution Shift in Learning-based Bug Detectors. In *ICML*, 2022.

[13] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. Learning Fast and Precise Numerical Analysis. In *PLDI*, 2020.

[14] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. Learning to Explore Paths for Symbolic Execution. In *ACM CCS*, 2021.

[15] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *ACM CCS*, 2019.

[16] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting Debug Information in Stripped Binaries. In *ACM CCS*, 2018.

[17] Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin Vechev. Learning to Find Naming Issues with Big Code and Small Supervision. In *PLDI*, 2021.

[18] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation. *arXiv:2505.23135*, 2025.