I work broadly in computer security and privacy. The most important problem in my field is the ever-growing size and complexity of software systems that are rife with vulnerabilities. Patches and defenses are continuously deployed, but the software attack surface is extremely large and attackers invariably find ways to gain a persistent foothold. An effective way to end the arms race between vulnerabilities and defense tools is to isolate the software using trusted hardware. The first incarnation of such hardware, called trusted execution environments (TEEs), proposed to isolate parts of kernel code. They have been around for three decades but haven’t been fully adopted or widely available. The latest incarnation of TEEs was introduced in the last 5 years and they directly isolate user applications instead of isolating small parts of the kernel.

Given these timely advances in hardware capabilities, my research asks the following questions: what is the least amount of code that needs to be bug-free to securely run user applications? At the moment, even after using TEEs, this number can be upwards of a few million lines of code. Can we do any better?

I have studied the problem of securing user applications against potentially buggy software in-depth. I have two key results. First, I showed a principled way of using TEEs to securely execute Linux applications while only trusting 20K lines of code [8]. This is within the realm of full formal verification. Towards this goal, I proved formally verified guarantees over a large-subset of this code [1]. Second, to see how we can improve this further, I constructed new TEE designs where one only has to trust 2K lines of code [4, 11]. Overall, these implementations point to a new way of executing secure applications with a few thousand lines of code [5, 2]. This is in sharp contrast to the state-of-the-art with a million lines of code. My research has led to a direct impact at various companies (Intel, Google, Microsoft, Qualcomm, SAP Labs, Seagate, ISPs) and three start-ups. Broadly, my research encompasses trusted hardware [8, 9, 5, 2, 11, 4, 7], program analysis [14, 13, 12, 10, 3], formal methods [6, 1], and cryptography [15].

Main Contributions
My main thesis work has been on TEEs. I showed how to build a systematic solution by using TEEs to safeguard applications against large and potentially buggy software [7]. Contemporary TEEs provide a new capability to isolate applications, called Enclaved TEEs (e.g., Intel SGX). It was initially intended to isolate and run tiny pieces of secure functionality (e.g., key management). However, on a closer look, TEEs offers something more powerful—the hardware isolates user-applications. It is similar in concept to, but strictly more secure than, the software-based isolation of containers (e.g., Dockers). This observation has led several researchers, including me, to show how we can run full-blown expressive programs in these TEEs. If this can be done, we can enable application-virtualization with strong hardware isolation, thus removing the threats of the entire OS. One way to execute rich functionality in these hardware-isolated regions is to import the entire operating system (OS) functionality inside the isolated region. This defeats the purpose of TEEs by executing millions of lines complex and potentially vulnerable code alongside sensitive application code.

My insight was to delegate the system functionality to the untrusted stack but thoroughly vet the computed results before using them. This design significantly reduces the amount of trusted code from millions to a few thousand lines while ensuring compatibility. This is within the scope of prior verification efforts. As a step toward full verification, I defined and verified the filesystem interface to provide verified security guarantees. Once TEEs provide compatibility for rich applications with small and verified trusted code, I showed how they enable new use-cases. As a better alternative to commodity TEEs, I showed a more mindful way of constructing better and new TEEs designs ground-up. Lastly, I demonstrated the unexpected security repercussions and mitigations under the new threat model where the applications do not trust the rest of the software, such as the OS.

Compatibility. Instead of importing the support for rich functionality inside the application, I argue that TEEs can securely delegate these functions to an underlying untrusted OS. To demonstrate this, I built Panoply [8] on the design principle of delegate-instead-of-emulate. Panoply implements a thin layer inside the application, intercepts the unsupported functionality calls, and tunnels them safely to the untrusted OS. Thus, it seamlessly executes existing rich-functionality code inside enclaved TEEs while preserving their security and execution semantics. This approach reduced the trusted code from a few million lines to 20K, which is well within the realm of formal verification. Panoply runs modern Linux applications such as full-fledged web servers, databases, cryptographic protocols, and anonymous communication services.

Impact. Panoply has seen two major deployment success stories. Anquan Private Blockchain uses it for hardware root of trust to yield a throughput of thousands of transactions per second. Anqlave, a south-east Asian startup, has adopted Panoply as a framework for applications with hardware-root of trust.
**Verified Security Guarantees.** Delegating tasks to an untrusted OS is a tricky prospect because the OS can tamper the results of a requested operation. Thus, securing this unsafe interface is of paramount importance. There are two potential mitigation options: moving the functionality inside the applications to avoid unsafe calls or ad-hoc checks on the returned results. The former option increases the trusted code to millions of lines. Panoply, which takes the latter option, dedicates 20K lines of code to check the returned results. However, without rigorous scrutiny, it can still be susceptible to OS attacks. A principled solution, that addresses the root problem, is to use formal verification to guarantee that the checks employed by the application are sufficient. With this principle, I built BesFS—the first formally verified POSIX-compliant filesystem interface for TEEs [1]. BesFS has a machine-checked interface and implementation that is proven safe. More importantly, it has a composition theorem which allows proving the safety of high-level system interfaces that chain syscalls. This degree of expressiveness allows TEEs to run unmodified applications.

**Impact.** BesFS showed that the software provided by Intel, Google, and Microsoft for TEEs use unsafe interface and are susceptible to attacks. BesFS is integrated into Panoply and is deployed at Anqlove.

**New Use-cases.** First, I showed how to use TEEs on the network middleboxes to execute an auditable filtering mechanism [5] against distributed denial of service (DDoS) attacks. Second, I showed secure marketplaces for offering pre-trained neural network models on a pay-to-infer basis [2].

**Impact.** One of the biggest regional Internet service providers (ISPs) in Asia-Pacific has agreed to test our auditable filters. The neural network use-case is being tested at Microsoft.

**New TEE Designs.** The first part of my work addresses the deficiencies in existing TEE designs. However, it is an after-the-fact measure from a hardware standpoint. Now that we know the limitations and gaps in current designs, we can build a ground-up solution mindful of these pitfalls. To this end, I have built two hardware designs for TEE support—PodArch and Keystone. I lead the efforts for PodArch [11] before Intel announced its TEE in 2013. PodArch augments the CPU instruction set to support application-level isolated execution to protect them against a malicious operating system. Although similar to Intel TEE in many aspects, the key difference in PodArch design is its in-built support for backward compatibility with existing binaries. Next, I built the first open-source framework for architecting new TEE designs, called Keystone [4]. The main departing insight in Keystone is to re-use the natively available and standard hardware features for TEE building blocks. It showcases how TEEs can be customized to easily leverage the diverse set of existing hardware chips, cache controllers, and memory controllers. Keystone adds only 2K lines of code while ensuring compatibility with a large body of existing tools and applications.

**Impact.** PodArch has influenced designs at Qualcomm. Keystone is being used at Seagate and Baidu X-Lab.

**Uncovering & Defending New Attack Surface.** The new threat model posed by enclaved TEEs brings out new challenges. My work showed that new side-channels come into play in unanticipated ways [9]. I demonstrated an attack wherein the attacker selectively starves the application memory. When the application attempts to access a memory location, the attacker sees the failure and the memory address. Thus, although the attacker cannot directly access the application memory, it can still learn a significant amount of information just by observing page faults delivered to the OS. It is one of the earliest works to show that this is not merely a theoretical attack, but is devastating enough to compromise hardened cryptographic libraries. My follow-up work showed its disastrous effect on neural nets to steal the sensitive image labels inferred using a secret model [2]. Defending this attack surface is non-trivial because obvious solutions either cost performance or require heavy rewriting of the applications. My solution is **contractual execution**—a hardware extension that lets the untrusted operating system to perform its traditional job of memory management without gaining any sensitive information about the application. The insight here is to use the hardware as an arbiter to establish a contract between the operating system and the application. The contract stipulates that the operating system will always provide a certain amount of memory to the application, the hardware checks if this contract holds during the execution, and informs the application of any violation. This puts the application back in control for mitigating any leakage before the attacker learns any information. This simple yet powerful insight allows the application to protect itself from leakage with minimal hardware changes.

**Impact.** Intel has integrated my proposed enhancement prototypes as a defense against the page-fault side-channel attack. This is a direct outcome of my summer internship at Intel in 2017, where I worked closely with their hardware security team to enable it.
Research Beyond TEEs

I briefly summarize several collaborative projects that have produced important results in the areas at the intersection of systems security with programming languages, web security, and applied cryptography.

**New Class of Attacks.** Memory vulnerabilities (e.g., buffer overflow) have been a long-standing security problem. We showed that existing defenses for these vulnerabilities suffer from definitional weaknesses, they only check the correctness of program control-flow. Thus, the attacker can still violate the data-flow and go undetected. We demonstrated the full potential of exploiting these definitional gaps with a principled attack called data-oriented programming (DOP) [10]. With DOP, the attacker gains maximum expressiveness capability i.e., it is Turing-complete. DOP remains an open problem because of the highly expressive attack surface. It trivially bypasses all software defenses as well as Intel’s hardware mechanisms. **Impact.** Intel took note of the severity of DOP attacks and is working on adding support to prevent them.

**Information Flow Analysis, Vulnerability Detection, and Auto-patching.** Our work on estimating quantitative information flow over strings [14] and neural networks [6] used principled techniques such as generating functions and approximate model counting respectively, they scale enough to be useful for real-world security use-cases. We have improved a well-established vulnerability detection technique called symbolic execution. We use neural networks to eliminate common bottlenecks such as loops, complex path constraints, and state explosion [3]. In our work on auto-patch generation for JavaScript [13, 12], we analyzed 1000 popular websites to detect vulnerabilities and then systematically synthesized program patches by extending a blueprint that calls safe parsing functions. **Impact.** Our auto-patching work has resulted in a startup in south-east Asia called Dexecure, it was accepted into the prestigious YCombinator Fellowship.

**Secure Execution Primitives.** We showed an alternative technique to secure server applications by using cryptographic primitive of partially homomorphic encryption along with TEEs [15]. **Impact.** It has influenced products at SAP Labs.

**Impact Summary**

As an immediate impact, my work has affected existing products and hardware. It has established novel solutions at startups. It has drawn attention to open problems that are important in the long-term.

**Industry & Start-ups.** My TEE line of work has resulted in impact at various companies including Intel, Google, Microsoft, Qualcomm, Seagate, and Internet service providers. It has been commercialized as startup products at Anquan and Anqlave. I have advised these startups from their early stages. My other work has influenced products at SAP Labs and resulted in a startup called Dexecure.

**Academic.** After BesFS, researchers disclosed a wide-scale manual analysis of unsafe interface in 8 TEE systems and found 35 vulnerabilities. My work not only anticipated a super-set of these attacks and demonstrated examples of them in 4 systems but also provided a formally verified interface as a defense. Keystone has received a resounding welcome and wide press coverage as the open-source TEE initiative[4]. It has picked up a lot of commercial interest in a short time. It has already been used in top-tier and award-winning papers from research groups at University of Washington, Columbia University, Microsoft Research, KU Leuven, and University of Birmingham. Several undergraduates, at Berkeley and other international universities, have expressed research interest in both Keystone and BesFS, and actively contribute to the project. I am advising these students in Berkeley and Singapore on new research projects. Our two workshops on open-source TEEs have been popular and well-received both in academia and industry with over 100 participants. They have created TEE awareness in the immediate systems security circles as well as researchers in machine learning, formal methods, and architecture community.

**Future Research Directions**

I build large-scale secure systems with long-term impact. I plan to expand my research into three problems: **End-to-end Formally Verified TEEs.** TEEs use interfaces and functionality from the layers between the application and the hardware, where it is important and non-trivial to safeguard them. I will continue to enhance the line of certification-based solutions towards an end-to-end verified hardware and software stack. How far can we stretch the boundaries of minimal trusted code in TEEs with a formally verified stack ground-up? The key
challenge is to identify and capture the right set of guarantees expected from each layer in the TEE execution stack and then systematically formalize their expected behavior.

**Hardware-software co-design for TEEs.** Computer architecture is rapidly evolving, with custom accelerators and configurations on the rise. This shift is motivated by use-case demands such as better networking (e.g., software-defined networks) and faster machine learning (e.g., tensor processing units). Such hardware-software co-design allows one to customize their computation stack for specific tasks. These custom stacks are equally susceptible to attacks from a bloated software; worse yet, their specific usage and interface can make the attack surface larger. Do new attacks come into play because of the hardware assumptions made by these use-cases? I plan to explore isolation primitives for these emerging designs and use-cases.

**Mitigating Side-channel Leakage.** In the past couple of years, there has been a surge of side-channel attacks such as Specter and Meltdown that exploit speculative execution; TEEs are not immune to them. The concept of contractual execution that I laid out for the TEE memory side-channel can be extrapolated for defending speculative execution attacks. If the hardware can securely establish and enforce a contract between the TEE and the operating system for shared resources (say, branch predictors), the TEE can detect and mitigate leakage via these channels.

Being a systems security research, I am excited to continue working broadly at its intersection with trusted computing, program analysis, and applied cryptography.

**References**