Research Statement: Building Secure Systems using Language-based Techniques

Anitha Gollamudi

Abstract
At a high level, my research primarily focuses on enforcing strong application-specific information security guarantees against powerful attackers using language-based security mechanisms. A key part of my PhD research uses Trusted Execution Environments (TEEs)—architecture protection mechanisms such as Intel’s Software Guard Extensions (SGX), ARM TrustZone and Sanctum—to enforce confidentiality and integrity against privileged attackers (e.g., OS kernels).

1 Introduction
Trusted Execution Environments such as Intel SGX, ARM TrustZone and Sanctum use hardware techniques to provide confidentiality and integrity in the presence of privileged software and thus have the potential to protect an application from such low-level attacks. They do so by enabling an application to execute its code and data in protected containers called enclaves. Even the privileged software (OS kernel) cannot access an enclave’s contents. TEEs also provide integrity guarantees for the code running inside an enclave. This can be useful to attest that prior to launch of an enclave, the contents haven’t been modified, for instance, by an untrusted cloud service on which the enclave is running. This is exciting as they can now enable a secure execution of code even on untrusted machines (e.g., internet).

2 Research during PhD
Though TEEs provide isolated execution, they are not a panacea. Specifically, they cannot prevent code from intentionally leaking secrets. Also, using them is tricky and requires low-level architecture-specific expertise that a high-level application developer may not have.

Gollamudi and Chong [2] leverages both information flow control mechanisms and enclaves, to enforce a variety of confidentiality policies (e.g., policies that change during the execution of the program) against powerful kernel-level attackers that are able to arbitrarily change the code outside the enclaves. Our work partitioned the application written in a C-like language and automatically placed the security-critical parts of an application to run inside an enclave. The partitioning algorithm guarantees that the resulting placement preserves the noninterference property, a stronger security condition that says that publicly observable data isn’t influenced by secret inputs, against a low-level attacker (e.g., malware) that can arbitrarily modifying the code executing outside the enclave. Applications processing sensitive data (e.g., credit cards, medical records, certificate authorities etc.) can directly benefit from these strong confidentiality guarantees. Specifically, an authentication module wishing to forget the user’s password is now guaranteed to enforce the security policy.

Extending these guarantees to a distributed setting is challenging. Lack of trust between entities limit the kinds of tasks a decentralized application can perform since parties can now act maliciously by corrupting the computation. Gollamudi et al. [5] uses TEEs in a distributed setting to obtain security (noninterference) even when the computation is run on an untrusted node. It proposes DFLATE, a programmable calculus that supports distributed execution, communication channels, concurrency, and TEEs. In DFLATE, users can explicitly state their
trust in TEEs using delegation of authority. The type system statically tracks all delegations (including those from the attackers) and ensures that attackers do not learn secrets or influence the execution of principals that do not trust them. DFLATE enforces security in distributed settings where mutually distrustful parties compute over aggregated data (e.g., Multi-party computations, sealed bid auctions).

One of the key insights of DFLATE is that, thanks to TEEs, an entity can now express its trust directly in the code, rather than using code signatures. Gollamudi and Chong [3] generalize this idea by defining computation principals, principals representing computation, and introduces COAL, an authorization logic that enables entities to express their trust in the computation principals. Using COAL, an entity can now specify an access control policy that directly refers to the intensional properties of the code. For instance, a policy can specify that Alice delegate her authority to any piece of computation that has been verified to be memory safe.

3 Future Directions

Secure Composition of TEE Systems. A typical TEE subsystem consists of an SDK for managing enclaves and their security features (e.g., hardware-protected keys and remote attestation), some supporting libraries (e.g., cryptography), and a user application. The SDK is written using a mix of C/C++ and X64 assembly (e.g. for SGX instructions). Any bug in code running within the enclave can lead to serious vulnerabilities. For example, poorly designed library could be exploited to leak enclave secrets, such as their signing keys. Verifying TEE subsystems is not only important but also attractive as the library code to bootstrap and run enclaves is fixed and small.

Gollamudi and Fournet [4] propose a methodology to gradually verify the TEE subsystem. It addresses two challenges. First, to verify the TEE code, it proposes using $F^*$ and Vale, the high-level and assembly-level verification languages respectively [1]. Second, it uses memory containers (referred to as sandboxes hereafter), to isolate the untrusted user application. This ensures that the composition is secure. Figure 1 shows the summary of composition inside a TEE.

A key insight of the above work is that sandboxing a component can yield a policy that can be precisely specified using a refinement type (e.g., using $F^*$ types). Such a specification enables safe composition of untrusted components. Generalizing this methodology, I am proposing to verify TEE subsystems by employing a sandbox for each untrusted component. As components are verified for their functional correctness, they are removed from sandboxes and get merged with the main component. Another advantage of such an approach is that it allows incremental deployment of verified components that is more practical in the real world.

Verifying an Embedded TEE OS. TEE operating systems (e.g., OP-TEE, Trusty TEE, RISC-V Enclaves) provide support for applications to use TEE features for enhanced confidentiality and integrity guarantees targeting mobile and embedded devices. A typical Trustzone based OS has two worlds, normal and secure, controlled by rich and secure OS, respectively. These OS kernels have small footprint (in the order of few kilobytes) and are designed to be auditable for security. As such, providing for spec-
ification for the kernel interfaces and verifying the functional correctness of such kernels is extremely appealing. However, it is still a tall order. As a comparison, seL4 microkernel is only 200KB and still took more than a decade to verify the full functional correctness of the kernel. In order to ensure that the research is both successful and useful, I propose the following breakdown with tangible outcomes for each milestone, with first two being the focus during the postdoctoral fellowship at UCSD CSE.

Rethinking Kernel’s interface This entails re-designing the kernel interface for offering better security guarantees to the application. In Trustzone based solutions, this amounts to designing the interaction between rich OS and secure OS. One way forward might be to model a kernel by providing precise semantics to syscalls (e.g., seccomp) and the transitions between normal and secure worlds. Proving that the transitions are secure may require giving precise types to the syscalls. For example, the type of seccomp may express that it only accepts a block of code that does not invoke any syscall and returns the result of running the code. seccomp may also be repurposed to invoke other isolation mechanisms such as an SFI sandbox,¹ SGX enclave, RISC-V enclaves etc. In this case, the return type is even more interesting: these mechanisms can return a certificate proving that the code was run unmodified (code attestation) in the specified memory locations.

Formal Memory Model a TEE OS leverages the isolation mechanisms (hardware and software) to provide confidentiality guarantees to its applications. Modeling isolation requires modeling the underlying memory. A fundamental prerequisite of a TEE memory model is to capture the disjointness of the TEE address spaces. In this aspect, it is more similar to a separation kernel. Investigating the connection between TEE OS and separation kernel may throw insights on how to leverage the past research on separation kernel.

Refining the Model The final and the most challenging step is to have an implementation of a TEE OS that refines the above models. That is, the set of behaviors of a concrete TEE OS is a subset of the set of behaviors of the OS model. By proving that the TEE OS refines the OS model, we are verifying that the actual implementation respects the specification, and is thus functionally correct. This is a top-down verification approach and is commonly employed to write provably correct operating systems. This is a long-term goal that I am not necessarily aiming at during my time at UCSD.

References


¹SFI stands for software fault isolation. SFI sandboxes are built using software-only mechanisms whereas TEEs use hardware-only mechanisms.