Research Statement

Klaus v. Gleissenthall

Computer systems are often incredibly complex; to get them right, programmers have to make scores of minute implementation choices, each of which has the potential to compromise the safety and security of the entire system. I want to make systems building easier by providing methods that help practitioners write correct and secure code while keeping additional programmer effort low.

Even though programming languages and verification research has made great progress along this path, e.g., by verifying software security and correctness through mathematical proofs [1, 3, 7, 17], today’s techniques still suffer from two major weaknesses: First, they focus on software in isolation, that is, they abstract hardware through simple, idealized execution models. Real hardware is however incredibly complex due to ubiquitous fast-paths and performance optimizations and therefore may diverge from the simple model’s behavior in subtle ways. Because of this mismatch, security and correctness guarantees for software operating on an idealized hardware model may not carry over to the real-world. Second, traditional verification techniques require tremendous user effort, which severely limits their adoption. To move towards a world where programming languages and verification techniques enable not only specialists but normal programmers to provide end-to-end guarantees for real world systems, we need to address both challenges.

Current Research & Impact I have worked on several projects towards this end. First, I have built a method for verifying that a hardware design indeed provides the guarantees that cryptographic software requires in order to avoid leaking secrets through timing variability. Our method requires minimal user interaction, and has been able to prove absence of timing variability (and find timing violations) for existing hardware designs including RISC-processors, FPU’s and crypto cores. This work raised my interest in the impact of speculative execution on security guarantees for software. In short, speculative execution allows the processor to guess the outcomes of branches, jumps or address calculations before their final values are known. As a result, cryptographic software may violate guarantees that were given for a simple hardware model without speculation. I have addressed this mismatch through an operational semantics that precisely describes actual hardware behavior, and a method to automatically insert minimal synchronization to provably remove speculative execution induced bugs. During this work, we discovered speculative execution bugs in popular crypto libraries like Open SSL and libsodium and even verified libraries like HACL* [17]. Our semantic modeling also lead us to discover a new Spectre variant called Spectre-MOB (for memory order buffer) which affects Intel i7-5600U (Broadwell) and 230 i7-6700K (Skylake) processors. Finally, I have built a framework that improves the usability of verification techniques for distributed systems by exploiting symmetry and commutativity. My work on distributed systems has already influenced research in other areas: a recent paper in approximate computing uses our approach to apply techniques for proving reliability guarantees for approximate computing in sequential programs to distributed systems [6]. I will now describe these projects in more detail and then discuss future work.

Security: Bridging the Hardware/Software Gap

Eliminating Timing Channels in Hardware (Security’19) Hardware often serves as the root of trust in computer systems. Unfortunately, recent developments have increased doubt about whether this trust is well earned. For example, even hardware security features like trusted platform modules and hardware enclaves have been recently shown to leak their secrets via timing side channels [4, 11] which allow an attacker to learn enough information to guess secrets simply by observing how long a computation takes to execute.
My recent research has focused on turning trust into certainty. I have built a method called IODINE [15] that helps devise – and automatically verify – conditions under which a hardware design is constant time, i.e., free of timing channels. Verifying constant time execution of hardware poses a technical challenge: unlike cryptographic software, computations in hardware are concurrent as register state is shared by multiple pipelined computations that mutually influence each other. My work addresses this challenge through a new notion of timing that’s suitable for hardware. IODINE tracks for each register, which computations are still live, i.e., active in the current cycle, and requires that the same computations are live, independently of which secrets the computation operates on. IODINE can be applied to existing hardware designs with little overhead: we have used IODINE to prove absence of timing variability (and find timing violations) for a number of existing hardware designs – including RISC-processors, FPUs and crypto cores, taken from GitHub and OpenCores.

Foundations for Speculative Execution (Under Submission) Timing side-channels also lie at the heart of a recent revolution in security: speculative execution attacks like Spectre [8], Meltdown [9] and Fallout [10] have overturned existing ways of writing secure systems and guaranteeing their correctness. For example, my work showed that even cryptographic libraries like HACL* which have been proven free of timing side channels do in fact contain timing channels that exploit speculative execution. At its core, this mismatch between proof and reality comes from considering incomplete foundations. While traditional models assume that programs are executed sequentially, modern hardware internally performs speculative execution – a kind of concurrent computation that allows the processor to guess outcomes of branches, jumps or address calculations before their final values are known. I have worked on building foundations that capture the effects of speculative execution and allow for security guarantees in the presence of speculation, in particular, an operational semantics [5]. Such a semantics can serve to define a basic notion of correctness under speculation that analysis tools can build on. We have, for example, used our semantics to build a symbolic execution analysis that managed to discover vulnerabilities in widely used crypto libraries including Open SSL and libsodium. Our modelling of speculative execution semantics also lead to other unexpected benefits: our semantics lead us to uncover a new Spectre variant, Spectre-MOB (for memory order buffer) which depends on the processor erroneously forwarding a secret value from a store to a subsequent load due to a wrongly predicted address. We have tested it to work on Intel i7-5600U (Broadwell) and 230 i7-6700K (Skylake) processors.

Fixing Speculative Execution Bugs (Under Submission) These discoveries leave us an in an unfortunate position, as many of our of core cryptographic algorithms are vulnerable. Yet, there are no satisfying fixes. Speculation can be stopped by inserting memory fences (akin to adding synchronization to a concurrent computation), but current methods insert fences either via heuristics that provide no guarantees for the resulting fix, or exhaustively, after every memory load, which leads to unacceptable performance cost. I have built and implemented a method called BLADE [16] that automatically fixes speculative execution bugs in cryptographic software. BLADE builds on the insight that, rather than prohibiting speculation altogether, it suffices to cut the data-flow from expressions that speculatively introduce secrets to those that leak them through the cache. To cut the data flow, BLADE introduces a new abstract primitive called protect, which turns expressions that may contain secrets into innocuous ones, and which can be implemented via existing architectural mechanisms. BLADE relies on a type system to automatically synthesizes a minimal number of protect calls (via type inference) and prove that the program is indeed secure (via type checking). Finally, we also prove that our fixes are correct with respect to our speculative execution semantics, i.e., that type-safe programs do not leak more than they would if they were run sequentially. I have used Blade to fix previously vulnerable implementations of the Signal crypto protocol and crypto algorithms in HACL* and CT-Wasm and shown that it inserts fewer fences (by two orders of magnitude) than current compiler mitigations (CLANG) and thus enables secure computation with negligible performance overhead.

Simpler Proofs For Distributed Systems (POPL’19 & OOPSLA’17)

My second line of work concerns asynchronous distributed systems, which form the core of today’s cloud infrastructure. These systems are infamously complex and error prone, but we can gain more confidence
in their functioning by equipping them with mathematical proofs. Unfortunately, proofs come at tremendous cost: for example, the verified key-value store in Microsoft’s Ironfleet project took 3.7 person years to complete. During my post-doc, I developed an idea called pretend synchrony [2, 13], which aims to reduce verification effort by soundly treating distributed cloud programs as if they were executing in lock-step on a single machine. This drastically reduces the number of relevant behaviors – and with it proof complexity. Internally, this reduction is enabled by using language based techniques to ensure that the distributed system is well-structured, and then exploiting the symmetry and commutativity that is inherent in such well-structured systems in order to simplify verification. We have compared the verification effort against the state-of-the-art and showed that synchrony reduces the number of manually specified invariant annotations (by 6x) as well as the time taken to check the programs (three orders of magnitude). My work has also inspired new research in the field of approximate computing: a recent paper uses our approach to apply techniques for proving reliability guarantees for approximate computing in sequential programs to distributed systems [6].

Future Research

There are many obvious next steps for the above projects, but I want to give an idea of which types of projects I like to work on by outlining some promising new directions. My research methodology is to pick an application domain where high level correctness and confidentiality guarantees are hard to maintain manually, and find techniques from PL and formal methods that help to automatically enforce these properties. Once I have an idea of how to approach the problem, I implement a prototype and apply it to real world examples. I find that this often helps to guide and refine the theory. In hindsight, my projects often turn out to be about concurrency, sometimes in disguise.

**Side-channel Free Hardware Enclaves** Hardware enclaves like Intel SGX and ARM TrustZone promise a means of outsourcing computation to an untrusted provider while maintaining data confidentiality and integrity. But recent attacks like Foreshadow [4] highlight the need for formal guarantees. Building on my work on hardware verification [15] and side channels [5, 16], I want build automated techniques for verifying isolation and side channel-freedom in hardware enclaves. I have initiated a collaboration with the group of Dawn Song to build a side-channel hardened and verified version of the Keystone enclave.

**Verifying Hardware Software Contracts** The main lesson behind the Spectre vulnerabilities is that hardware and software need to share a common set of assumptions, beyond the granularity of the instruction set architecture. Using my experience with language based techniques [2, 16] and hardware verification [15], I propose to make these assumptions explicit by designing a language and verification method that describes the interface between hardware and the compiler and allows verifying that their composition indeed guarantees the intended correctness properties.

**High-level/Functional Language For Secure Hardware** The end of Moore’s law brings about a need for custom hardware accelerators. Yet, programming abstractions of hardware description languages like Verilog make it hard to write correct, performant and secure hardware. Building on my experience in applying language based techniques to distributed systems and side-channel attacks [2, 16] and my background in verification [12, 13, 14, 15], I want to build a high-level language abstraction and verification support for hardware.

**Proving Universal Composability of Cryptographic Implementations** Proving universal composability (UC) of a cryptographic protocol provides strong guarantees that the protocol is secure, even if composed with an arbitrary attacker, and embedded inside a larger distributed system that is interacting with it. I want to use my experience in verifying distributed systems to build a framework that simplifies writing formal proofs for UC, and allows to produce executable, performant protocol implementations.

**Programming Abstractions for Non-Volatile Memory** Non-volatile memory has emerged as a fast, low-energy alternative to traditional storage. However, maintaining data-consistency in the presence of failures makes its use challenging. Building on my work on language based techniques for verifying distributed
system and side-channel freedom [2, 16], I want to devise programming abstractions and verification tools that simplify the correct use of non-volatile memory.

**Pretend Synchrony for Serverless Computing** Pretend Synchrony [13] provides a means to turn verification methods that have been developed for sequential programs into verification methods for distributed systems. This opens up a range of research opportunities, e.g., by applying information flow control, differential privacy, or robustness notions to distributed programs. One particularly interesting application is to use information flow control to ensure confidentiality in serverless computing.

**Regulating Network Access Via Types** The Rust programming language has revolutionized systems programming by providing a rich type system that guarantees memory safety of any well-typed program. In a similar way, using my experience from [2, 16], I want to build a type system that provides high level guarantees by regulating access to the network, building on notions like borrowing and ownership.

References


