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. The goal of my research is to make systems building easier by providing methods that help practitioners write correct code while keeping additional programmer effort low.

Even though programming languages and verification research has advanced significantly along this path e.g., by verifying correctness of distributed systems and security critical code through mathematical proof \([1, 3, 7, 18]\), most existing techniques still suffer from two weaknesses: First, they focus on software in isolation, that is, they abstract hardware through simple, idealized execution models. Real hardware is however full of fasts paths and performance optimizations which may cause it to diverge from the simple model’s behavior in subtle ways, \(e.g.,\) through secret dependent timing or microarchitectural side-channels. Because of this mismatch, security and correctness guarantees for software operating on an idealized model may not carry over when run on real-world hardware. Second, traditional verification techniques require tremendous user effort, which severely limits their adoption. My research aims to address both challenges with the goal of shifting towards a world where programming languages and verification techniques enable not only specialists but everyone to provide end-to-end correctness and security guarantees for real world systems.

Current Research & Impact

I have worked on several projects towards this end. First, I have built a method called IODINE which allows to devise and verify usage conditions under which a hardware design does not leak secrets through timing variation, \(i.e.,\) indeed behaves like its simple abstraction. By enforcing these conditions in software, verification methods can provide true end-to-end guarantees for security critical code like cryptographic algorithms without having to worry about the peculiarities of the actual hardware they are running on. IODINE can be easily applied to existing hardware designs: we have used IODINE to find and verify usage conditions for a number of open-source hardware designs including RISC-processors, FPUs, and crypto cores. If usage conditions are hard to obtain, \(e.g.,\) when source code is unavailable, software mitigations have to embrace the hardware’s original complexity: Through my work on IODINE, I became interested in mitigating the effect of speculative execution. In contrast to simple sequential execution, speculative execution allows the processor to guess the outcomes of branches, jumps or address calculations before their final values are known. As a result, sensitive software like cryptographic algorithms may violate guarantees that were given for the simple sequential model when executed under speculation. I have addressed this mismatch through two contributions: first, an operational semantics that precisely describes the behavior of actual hardware under speculation and second, a method called BLADE which provably removes speculative execution induced bugs by automatically inserting a minimal number of speculation barriers. This work also led us to discover speculative execution bugs in popular crypto libraries including Open SSL and libsodium and even in verified libraries like HACL* \([18]\). Our semantic modeling also revealed a new Spectre variant called Spectre-MOB (for memory order buffer) which affects Intel i7-5600U (Broadwell) and 230 i7-6700K (Skylake) processors. Finally, in order to make verification methods easier to use, I have built a framework that reduces the proof burden in 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 avenues for future research.
**Verification: Eliminating Timing Channels in Hardware (Usenix Security’19)** Hardware often serves as the root of trust in computer systems. Unfortunately, this trust is not always well-earned. For example, recent attacks like Foreshadow [4] and Tpm.fail [11] have shown that even hardware security features like trusted platform modules and hardware enclaves leak their secrets via timing side channels, which allow an attacker to learn enough information to guess secrets simply by observing how long a computation takes to execute. These leaks are particularly devastating, since hardware security features often form the last line of defense. IODINE helps to prevent such leaks by devising – and automatically verifying – conditions under which a hardware design is constant time, i.e., free of timing channels. IODINE is easy to use: it can be applied to existing hardware designs with little effort beyond specifying usage assumptions.

The core technical challenge that IODINE had to solve to make easy verification possible is, perhaps surprisingly, about concurrency: modern hardware simultaneously processes hundreds of pipelined instructions that may mutually influence one another. But how can we measure the timing of such a computation? My work addresses this challenge through a new notion of timing that is suitable concurrent hardware computations. 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.

In the future, I want to scale IODINE to large real-world systems like commercial processors (e.g., ARM cores, which already offer a data-independent timing mode) and work towards hardware security features we can trust by verifying open source hardware security platforms like OpenTitan and Keystone.

**Foundations for Speculative Execution (Under Submission)** Another issue at the interface between hardware and software has recently overturned existing ways of writing secure systems: Speculative execution attacks like Spectre [8], Meltdown [9] and Fallout [10] threaten confidentiality guarantee of well-trusted and even verified crypto libraries. 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]. We used our semantics to define a basic notion of correctness under speculation that analysis tools can build on. We used this correctness notion as the basis of a symbolic execution analysis which discovered 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 the vulnerability 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. My work on BLADE [16], offers a solution to this predicament by propsing a general technique to automatically and provably eliminate speculative execution bugs in cryptographic software. BLADE builds on a simple insight: Rather than prohibiting speculation altogether, it suffices to cut the data-flow from expressions that speculatively introduce secrets to those that leak them. To cut the data flow, my work introduces a new primitive called protect, which turns expressions that may contain secrets into innocuous ones, and which can be implemented via existing architectural mechanisms. BLADE implements this approach via a type system that automatically synthesizes a minimal number of protect calls (via type inference) and proves that the program is indeed secure (via type checking). In the future, I want to extend this line of work along two lines: first, I want to extend mitigations to additional classes of programs, for example, non-cryptographic, general purpose code like web-servers, and second, I want to build foundations that ensure end-to-end agreement between hardware and software to rule out timing (and other) side channel attacks in the presence of speculation and other modern processor features.

**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. Even though these systems are infamously complex and error prone, 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. My work has 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. In general, my research methodology is to pick an application domain where end-to-end correctness and confidentiality guarantees are hard to maintain manually, and find techniques from PL and verification 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. This often helps to guide and refine the theory.

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 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 mitigations of side-channels [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 their composition. In particular, this will involve connecting hardware mitigations like [17] and software mitigations like Blade.

High-level/Functional Language For Secure Hardware Instead of mitigating the shortcomings of existing hardware, we can write new, correct by construction circuits. This is particular interesting for custom hardware accelerators, which are becoming more prevalent due to the end of exponential growth in hardware performance. Unfortunately, existing 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 [12–15], 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, by exploiting restrictions to a “well-structured” language fragment, and which 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.
References


