Research Statement

Klaus v. Gleissenthall

Research Interests

Computer systems are often incredibly complex; to get them right, programmers have to make scores of minute, seemingly arbitrary 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 developing methods that help practitioners write correct and secure code, where my focus is on maintaining high-level safety and confidentiality properties while keeping programmer effort low. Building these methods often requires distilling core insights from the application domain: while my work draws from Programming Languages and Formal Methods, my post-doctoral research has focused on tailoring techniques from these fields to problems from Security and Distributed Systems. I have published my work in top conferences, including POPL, PLDI, USENIX SECURITY, OOPSLA and CAV.

Timing Leaks and Speculative Execution

Much of my recent research focuses on how to mitigate the impact of hardware on the security of software systems.

Eliminating Timing Channels in Hardware (Usenix Security’19) Hardware often serves as the ultimate root of trust in computer systems. For example, trusted platform modules provide the basis for digital signatures and attestation. But also software implementations of cryptographic algorithms rely on basic hardware instructions to implement their functionality. Unfortunately, this trust has proved to be unfounded. For instance, recent work has shown that trusted platform modules leak their secrets via timing side channels [7], i.e., an attacker can learn enough information to guess the private key, simply by observing how long a computation takes to execute. I have built a method called IODINE [9] that helps devise – and automatically verify – conditions under which a hardware design is free of such timing channels. On a technical level, the main innovation behind IODINE is a new notion of timing that’s suitable for hardware where register state is shared by concurrent computations due to pipelining. In short, 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. We used our method to prove absence of timing variability (and find timing violations) for a number of existing hardware designs – including RISC-processors, FPUs and crypto cores, which we took from online source repositories.

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 [4], Meltdown [5] and Fallout [6] have overturned existing ways of writing secure systems and guaranteeing their correctness. For example, we found that even cryptographic libraries that 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 form considering the wrong 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. Recently, I have worked on building foundations that allow for security guarantees in the presence of speculation, in particular, an operational semantics [2]. 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 Open SSL and libsodium. A careful modelling of semantics can lead to other unexpected benefits: our semantics lead us to uncover a new Spectre variant, Spectre-MOB (for memory order buffer), which we have disclosed to Intel.

**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 easy fixes in sight. 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 costs. I have built and implemented a method called BLADE that automatically fixes speculative execution bugs in cryptographic software. BLADE builds on the insight that, to stop speculative execution attacks, it suffices to cut the data-flow from expressions that speculatively introduce secrets to those that leak them through the cache, rather than prohibiting speculation altogether. BLADE enforces this property through a static type system that (1) types each expression as either transient, i.e., possibly containing speculative secrets or as being stable, and (2) prohibits speculative leaks by requiring that all sink expressions are stable. An expression that contains secrets can be turned into an innocuous expression through the use of a new abstract primitive called protect which can be implemented via existing architectural mechanisms. Our type system can automatically synthesize a minimal number of protect calls needed to ensure the program is secure. Finally, we also proved that our fixes are correct with respect to our speculative execution semantics, i.e., that the fixed programs do not leak more than as 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 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 lie at the heart of 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 – the gold standard for ensuring their correctness. But, unfortunately, proofs come at tremendous cost: for example, the verified key-value store in Microsoft’s Ironfleet project reportedly took 3.7 person years to complete. During my post-doc, I developed an idea called pretend synchrony [1, 8], which aims to reduce verification effort by soundly treating distributed cloud programs as if they were executing in lock-step on a single machine, thereby slashing the number of relevant behaviors – and with it proof complexity. Internally, this reduction is enabled by exploiting symmetry and commutativity that is inherent in well-structured distributed systems. 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 [3].

**Future Research**

I will now discuss some directions for future work. 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, I like problems that have strong real world motivation but require new technical insights. More specifically, 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 a basic approach, I implement a prototype and apply it to real world examples. I find that this often guides the theory and
ensures its practical relevance. 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. Yet, recent attacks like Foreshadow have shown the need for formal guarantees. Building on my work on hardware verification, I want to 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. 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 safe hardware. Based on [8, 9], 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 system that is interacting with it. I want to use my experience from [9] and [8] to build a framework that simplifies writing formal proofs for UC, and produces 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 verifying distributed system, 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 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 [1], 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**


