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 [1,3,7,17], 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 fast 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 lead us to discover speculative execution bugs in popular crypto libraries including Open SSL and libsodium and even in verified libraries like HACL* [17]. 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.

Security: Bridging the Hardware/Software Gap

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 [10] 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.

The core technical challenge that IODINE had to solve to make this 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 for 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.

Foundations for Speculative Execution (PLDI'20) Another issue at the interface between hardware and software has recently overturned existing ways of writing secure systems: Speculative execution attacks like Spectre [8] and Fallout [9] threaten confidentiality guarantees 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 an operational semantics [5] that captures the effects of speculative execution and therefore forms a basis for security guarantees. We used our semantics to define a basic notion of correctness and used it as a basis for a symbolic execution analysis which discovered vulnerabilities in widely used crypto libraries including Open SSL and libsodium.

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 [15], offers a solution to this predicament by proposing 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).

Simpler Proofs For Implementations of Distributed Systems (POPL'19 & OOPSLA'17)

Writing correct distributed systems is notoriously difficult. Engineers not only have to come up with correct algorithms, they also need to ensure that no additional mistakes are introduced when the algorithm is implemented. To make it easier to write implementations that are free of protocol and implementation errors, verification researchers have devised methods that verify correctness via mathematical proof. But unfortunately, these methods are costly, which limits their adoption. For example, Microsoft's verified key-value store took 3.7 person years to complete. I have worked on simplifying verification by exploiting the following observation: while it is hard to verify the correctness of an arbitrary system, the systems that are written in practice are *well-structured* and therefore a lot easier to verify. I developed this idea in a method called Pretend Synchrony [2, 12], 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 reducing the number of relevant behaviors – and with it proof complexity.

Research Methodology and Future Research

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. In future work, I plan to explore the two lines of research outlined above – security at the interface of hardware and software, and exploiting structure in distributed systems – using this methodology.

Side-channel Free Hardware Enclaves Hardware enclaves like Intel SGX, ARM TrustZone promise a means of outsourcing computation to an untrusted provider while maintaining data confidentiality and integrity. Often they are formed by an amalgam of basic hardware features and the software that's driving them. In such a setting, software drivers need to be aware of the peculiarities of the hardware features they rely on, as illustrated by recent attacks like Foreshadow [4]. I want to build on my work on hardware verification [14] and side channels [5, 15], to create automated techniques for devising usage conditions, and verifying isolation and side channel-freedom in hardware enclaves.

Verifying Hardware Software Contracts More generally, hardware and the software that utilizes it 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 [15] and hardware verification [14], 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 [16] 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, 15] and my background in verification [11–14], I want to build a high-level language abstraction and verification support for hardware.

Pretend Synchrony for Serverless Computing Serverless computing allows distributing a given functionality transparently across servers. But since each server is potentially shared by a large number of computations from different origins, it becomes crucial to ensure that secrets cannot be leaked from one computation to another. I think that a promising path towards this goal is to follow the idea of Pretend Synchrony by enforcing that computations are well-formed in that each distributed computation corresponds to an equivalent sequential one. This idea can be applied more generally: any verification method that has been developed for sequential programs can be turned into verification methods for distributed systems via Pretend Synchrony. This is interesting for information flow control, differential privacy, or robustness notions and many others.

Proving Universal Composability of Cryptographic Implementations A similar idea can be applied to simplify proofs of cryptographic protocols. The gold standard for proving such protocols secure is universal composablity (UC): proving universal composablity of a protocol ensures that, even if the protocol is 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. I want to explore how to make well-structured use of non-volatile memory in code, guide programmers to write such code, and finally exploit structure for verification.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*.

[2] Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. In *OOPSLA*, 2017.

[3] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security*, 2017.

[4] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Usenix Security*, 2018.

[5] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. Towards constant-time foundations for the new spectre era. In *PLDI'20*, 2019.

[6] Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. Verifying safety and accuracy of approximate parallel programs via canonical sequentialization. In *OOPSLA*, 2019.

[7] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *SOSP*, 2015.

[8] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Security and Privacy*, 2019.

[9] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading kernel writes from user space.

[10] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. Tpm-fail: TPM meets timing and lattice attacks. In *USENIX Security*, 2020.

[11] Klaus v. Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, 2016.

[12] Klaus v. Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *POPL*, 2019.

[13] Klaus v. Gleissenthall, Boris Köpf, and Andrey Rybalchenko. Symbolic polytopes for quantitative interpolation and verification. In *CAV*, 2015.

[14] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. Iodine: Verifying constant-time execution of hardware. In *USENIX Security*, 2019.

[15] Marco Vassena, Klaus v. Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. Automatically eliminating speculative leaks with blade. In *Under Submission*, 2019.

[16] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *MICRO*, MICRO.

[17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *CCS*, 2017.