Research Statement

Klaus v. Gleissenthall

Research Objectives As we trust computers with our money, health records, pictures and videos, with our most intimate thoughts in the form of search histories and private conversations—we need to make sure that our data is well protected. Unfortunately, today's computer systems are incredibly complex, and getting them right means making scores of seemingly irrelevant implementation choices, every one of which might derail the security of the entire system. My research agenda revolves around building tools that help practitioners write ***correct and secure code*** using techniques from ***programming languages*** and ***formal methods***.

While we have already made great progress along this path, most existing techniques still suffer from two weaknesses: First, they fail to model critical aspects of real systems. For example, in software security, we often represent hardware through simple, idealized models. But real hardware is brimming with fast paths, caches, buffers and performance optimizations like speculative execution. Failing to account for these features allows bugs to slip through and leads to systems that fail to keep our data safe. Second, today's verification techniques are extremely hard to use for non-experts. This means they are not adopted as widely as we'd like and therefore fail to fulfil their promise.

To make a step towards a world where programming languages and verification techniques enable not only specialists but everyone to provide correctness and security guarantees for real systems, we need to address both challenges.

Prior Research & Impact

I have made several first steps towards this end. My first line of research wants to reliably capture the behavior of real hardware to prevent side-channel leaks.

Modeling Hardware Leakage My interest in hardware security started with my work on IODINE which discovers actual ground-truth conditions under which a hardware design is free from timing leaks. Unfortunately, Iodine was hard to use for non-experts. To address this problem, I next worked on Xenon [7], which drastically reduces user-effort. My line of attack was the insight that solver and user should work hand-in-hand. For this, I developed an algorithm that pairs the solver's ability to infer proofs for straight-forward yet tedious arguments, with the user's domain knowledge via an interactive verification loop. A user study showed that Xenon had a significant effect on verification time, and, in some cases, **helped non-expert users succeed** while the control group wasn't able to finish at all.

Iodine and Xenon faithfully model real-world hardware, however, they both express their models in terms of the internal processor state. To make it easier to reason about software security, I have built a verification method that allows these assumptions to be expressed at a higher level of abstraction–at the level of the instruction set architecture (ISA) [10]. This paper recently received the **Distinguished Paper Award at CCS**.

Leakage Semantics This work builds on my earlier research in [4], where me and my collaborators proposed one of the first leakage descriptions in form of an ISA semantics for speculative execu-

tion [4], and used it to give the first definition of speculative security—a notion called speculative non-interference—which has since then been used widely. This work has received an **honorable mention** at the **Intel hardware security award**, discovered several speculation vulnerabilities in cryptographic code including Open SSL and libsodium, and predicted a theoretical vulnerability, which was later discovered in a real processor.

Fixing Speculative Attacks My work on leakage semantics for speculative execution led me to think about how one could defend against speculative execution attacks in software. This led me to develop a method called Blade which removes speculative execution induced bugs by automatically inserting a minimal number of speculation barriers. Blade certifies the correctness of these fixes independely via an information-flow type system. This work has received a **POPL Distinguished Paper award**, and was featured on the **SIGPLAN PL Perspectives** blog [1]. This year, I have also received an **ERC Starting Grant**—a personal grant of 1.5 million Euro—for this line of work. ERC personal grants are the largest and most selective personal grants in Europe. In 2023, **only 22** starting grants were awarded in computer science throughout Europe—only two in the area of formal verification.

Verifying Distributed Systems My second line of work on correctness proofs for distributed systems is also motivated by concerns about usability. Proofs can give us high confidence in the correctness of our core infrastructure. But they often come at tremendous cost: Microsoft's Ironfleet project which implemented a verified key-value store, took 3.7 person years to complete. To make proving distributed systems easier, I developed an idea called pretend synchrony [2, 6], which aims to reduce verification effort by soundly treating distributed cloud programs as if they were executing on a single machine. Fortunately, this idea proved to be effective: our experiments showed that the technique reduced the number of manually specified annotations, **by a factor of six**, and reduced checking time by **three orders of magnitude**.

Fuzzing While I like clean mathematical models, I'm not afraid to delve into the messy details of real systems. My interest in distributed systems led me to work on **fuzz-testing** of Byzantine fault tolerant protocols. Our method, ByzzFuzz [12], discovered a previously **unknow bug in the Ripple blockchain** production implementation, which violates termination, *i.e.*, allows an attacker to stall the system indefinitely. This bug has been confirmed by the developers, and received a payout via the bug bounty program. Our work also received an **ACM SIGPLAN Distinguished Paper Award**.

This work led me to apply fuzzing in another setting, that is, compilers [5]. Sanitizers help uncover memory vulnerabilities in software by flagging undefined behavior at runtime. Compilers, on the other hand, exploit undefined behavior for performance. This mismatch spells trouble. Compilers may remove undefined behavior or sanitizer checks, which can then resurface due to minor modifications such as a different optimization level. Indeed, this is a real-world problem, as our experiments show. We discovered **19 new bugs in Linux Containers, libmpeg2, NTFS-3G, and WINE** that had previously been masked by compiler optimizations.

Future Research

There are many obvious next steps for the above projects, but I want to give an idea about the 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. I find that this often helps to guide and refine the theory. My projects often involve reasoning 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 attacks like Foreshadow [3] highlight the need for formal guarantees. Building on my work on hardware verification [8,10] and side channels [4,9], I want to build techniques for verifying isolation and side channel-freedom in hardware enclaves.

High-level/Functional Language For Secure Hardware My work on hardware security often left me scratching my head about Verilog. Indeed, existing hardware description languages often make it hard to write correct, performant and secure hardware. I want to change this by adding high-level language abstraction and verification support. For this, I want to augment functional hardware language like CLASH with refinement types for verification. As a benchmark I want to build verified hardware security extensions, such as support for memory capabilies (*e.g.*, CHERI) or mechanisms for secure speculation. I have already taken first steps in this direction [11] together with my former master's student Robin Webbers, who has now started a PhD with me.

Counterexamples for Refinement Types My work on usable verification in Xenon and pretend synchrony led me to think about the usability of general purpose verification methods. Many such methods are based on refinement types, which provide a convenient way to statically check properties of programs. Indeed, verification with refinement types is great if our code is accepted by the checker; it becomes much more unpleasent, if verification fails. In that case, the type-checker will often give little to no information as to what went wrong. Regrettably, this is not purely an implementation issue; indeed, we're lacking a good theoretical understanding of what a counterexample to a typing derivation should look like. I want to formalize a mathematical notion of refinement type refutation that can serve as counterexample, and implement a search procedure for such refutations in the Liquid Haskell verifier.

Composable Security for Hardware A universal composablity (UC) proof can provide strong evidence that a cryptographic protocol is secure, even when composed with an arbitrary attacker, and when embedded inside a larger distributed system. Importantly, the UC framework allows careful modeling of the acceptable leakage of a protocol, *e.g.*, what exactly is leaked through side-channels. I want to use insights from universal composability to build a theory of composable security for hardware. One can think of each hardware module as a participant in a larger distributed protocol that makes up the core. The theory should enable practical modeling and verification of leakage descriptions for hardware. Importantly, modularity will help scale this to real-world cores, and deal with performance optimization such as speculation. Similar to UC security, this will require proving a theorem stating that security of each component implies the security of the overall system.

Patching Memory Errors Via Verified Sanitizers Despite decades of research, memory errors are still the most common cause for security vulnerabilities. Often, even after a vulnerability is discovered, considerable time passes until a patch is available; this may be even harder for legacy systems, where developers are no longer available. Memory errors often coincide with undefined behavior, which in turn can be detected by sanitizers. However, current sanitizers are huge libraries with unclear semantics. I want to build a compiler pass in Low* that provably detects certain types of undefined behavior according to a formal semantics. This sanitizer can then be used to patch memory vulnerabilities via targeted sanitization.

References

[1] `https://blog.sigplan.org/2021/04/21/automatically-eliminating-speculative-leaks-from-cryptographic-code-with-blade/`.

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

[3] 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.

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

[5] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. Don't look ub: Exposing sanitizer-eliding compiler optimizations. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

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

[7] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. Solver-aided constant-time hardware verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 429–444, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

[10] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. In *CCS*, 2023.

[11] Robin Webbers and Klaus v. Gleissenthall. Refinement types for hardware. In *LATTE*, 2022.

[12] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.