Software is everywhere, but *correct* software is not. One approach is full formal verification: mathematically proving that a program will not go wrong. In practice, though, most software engineers do not, and cannot, verify their software. Instead, they rely on testing and unsound static analyses. My career goal is to make verification, not testing, the first choice for a serious software developer.

Two observations about developers inform my approach to this grand challenge: (1) developers already use simple verification tools whose output is easily understood, such as type systems; and (2) developers commonly use unsound bug-finding tools, like FindBugs or Infer, that run quickly and issue few false alarms. The popularity of these techniques suggests the criteria that will make a verification approach attractive to developers: usability (explainability and predictability), precision (few false alarms), and speed (fast feedback and scalability). These principles motivate my research: verification techniques that are usable, precise, and scalable on real programs. To evaluate my work with respect to these principles, I often collaborate with industry by deploying tools to real developers.

A common theme throughout my work is decomposing complex problems into cooperating sets of simple analyses. Simple analyses are fast and developers can easily understand them, which makes them scalable and predictable. Simple analyses can also be expressive: we have shown how to combine simple analyses to solve complex problems that have troubled the verification community for decades, such as preventing out-of-bounds array accesses [1] and resource leaks [2].

However, a simple analysis is not necessarily simple to *design*. In fact, it is often more difficult to find elegant solutions to real, complex problems. The key insight behind my approach to designing verification techniques is to design explicitly *for simplicity*: the other desirable properties of verifiers—soundness, usability, precision, and speed—are often consequences of the simplicity of the right abstraction.

Array bounds. We designed a set of seven cooperating verifiers to prove that array accesses are in-bounds [1]. This shows how expressive simple analyses can be when combined in the right way. Prior monolithic approaches attempted to reason about all the complexity of bounds-checking using one complex abstraction, such as systems of arbitrary linear inequalities over program variables. Our abstractions decompose the complexity into multiple novel dependent type systems, simple linear inequalities, and more. The way they interact was also novel: the analyses are carefully staged to avoid mutual dependence except where necessary for precision, which we limited to one case in the rely-guarantee style. Our system is usable: it has a lower annotation burden than Java's generics. It is precise: similar to the best extant monolithic approaches based on abstract interpretation. And it is fast: it analyzes large programs in minutes rather than hours.

Accumulation analysis. *Typestate* protocols are finite-state machines that describe the operations that are legal and illegal in an object's various states. For example, a file in an Open state might have legal Read and Close operations, but a file in a Closed state might only have a

legal Open operation. In general, sound typestate checking is computationally expensive due to the need to reason about aliasing. In practice, this cost has been a barrier to the adoption of typestate verification. However, some typestate properties are *monotonic*: the set of operations that are legal only grows as the object transitions through typestates. We discovered that these monotonic typestates can be checked soundly and modularly, without needing to reason about aliasing, using a family of simple analyses that we call *accumulation analyses*. An accumulation analysis conservatively under-approximates the set of typestate transitions that have definitely occurred and forbids *goal transitions* until the analysis estimates that all of their *enabling transitions* must have occurred. The next two paragraphs give examples of monotonic typestate properties that we have verified soundly and modularly using accumulation analyses.

Malformed object construction. When an object is constructed, some set of logical parameters must be provided. For example, a geometric point object might require both x and y values, but its color might be optional. The popular builder design pattern—where each logical argument has its own method on a "builder" object, and the final object is only created when the builder's "build" method is called—enables programmers to avoid defining exponentially-many constructors. The builder pattern is convenient for programmers, but using it does cost some compile-time safety. Without the builder pattern, the programmer would not have written a constructor that took no "x" value; with the builder pattern, the programmer might forget to call a logically-required setter method, such as "setX()" in the point example, before calling "build". We designed an accumulation analysis whose goal transition is "build" and whose enabling transitions are exactly the methods that set the required logical parameters. This restores compile-time safety when using the builder pattern [3]. In a user study of AWS developers, those using our approach were about 50% faster and about 50% more likely to correctly update all call-sites. Security vulnerabilities can also result if the missing parameter was necessary for safety. In 9.2 million lines of code, our tool found 16 real security bugs with just 3 false positives (84% precision), but needed just 34 manually-written annotations (1 per 250,000 lines of code).

Resource leaks. After a program allocates a programmer-managed resource, such as a file descriptor, a network socket, or a database connection, the program must release the resource on all paths. Failing to do so causes a *resource leak*, which can cause resource starvation or denial-of-service, especially in long-running applications. Our key insight is the monotonicity of the property: all resources must be closed at least once. Our approach to solving this problem combines three simple analyses: (1) a taint analysis that tracks which expressions might contain objects that need to be closed, (2) an accumulation analysis whose goal transition is "a resource goes out of scope" and whose enabling transition set is { close() }, and (3) an analysis that compares the previous two when an expression may go out of scope [2]. Our approach is sound, fast, and precise: it outspeeds traditional approaches that track all aliasing by orders of magnitude and is competitive with unsound bug-finders on precision—on the benchmarks we tested, our analysis improved slightly upon the precision of the analysis built into the Eclipse IDE (29% vs 25%) while dominating on recall (100% vs 13%). Our approach is also usable: it required only 286 manually-written annotations in over 400,000 lines of code (about 1 for every 1,500 lines) in distributed-systems infrastructure that made heavy use of resources.

Compliance. Another way to make verification more attractive to developers is to automate a manual task that developers already have to do. An example is *compliance*, a process common in industry whereby an external auditor affirms that a company's systems properly handle sensitive data. For example, credit card companies require that companies holding credit card data must follow the PCI DSS (Payment Card Industry Data Security Standard), which has requirements like "credit card data be encrypted while it is stored." In practice, these audits involve manual examination of code by the auditor. We realized that many of these properties could be expressed as simple refinement type systems, and we designed and deployed them [4]. Auditors at an industrial partner accepted the output of our verifiers, obviating the need for manual audits of those properties, and they presented the results at a developer conference [5]. Developers preferred our approach—using the typechecker was less work for them than a single manual audit—as did the auditors, because our sound checks eliminated the possibility of human error. We ran our analyses on 76 million lines of code and found 173 true violations with only 1 false positive (99% precision) while requiring only 23 manually-written annotations (~1 per 3.3 million LoC). We also compared our analyses to extant unsound bug finders on an existing benchmark: our tools found all the errors (i.e. had 100% recall, vs. 88% for the next best tool) with comparable precision (our tools had 97% precision vs. 100% for the best unsound bug-finder).

Future work. I am currently exploring what other monotonic typestate-like properties can be verified using simple accumulation analyses rather than heavy-weight typestate analyses. One example is authorization, i.e. verifying a rule like "only access trusted data after the appropriate credentials have been provided." Another example is object initialization beyond the builder pattern. For example, to prove that all fields of an object are set to non-null values before the object is used, we can use an accumulation analysis that tracks the set of fields that have definitely been initialized so far, rather than the set of methods that have been called on an object (in fact the literature describes bespoke analyses that do operate this way, without identifying the underlying principle [6]).

Longer-term, my goal is to continue to make verification a better choice for working developers. I will continue to work with my industrial collaborators to find problems that developers regularly encounter, and continue to solve them by combining simple analyses. I will also work on improving existing analyses. One current drawback of modular verifiers is that sometimes they require human intervention in the form of annotations at module boundaries. *Annotation inference* promises to lessen or eliminate this burden. I see promise in bootstrapping local type inference to whole-program type inference. Unlike prior type inference approaches, this approach reuses the flow-sensitive local inference already present in practical verifiers rather than a separate type inference algorithm, and is therefore applicable to new verifiers with little or no modification. Another promising thread is techniques that help developers take advantage of the modularity in fast verifiers. Developers often make small improvements to their code when they are already changing it—for example, by adding missing documentation to a method they are updating. Could working developers be convinced to use the same technique to modularly verify a codebase?

[1] **Martin Kellogg**, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. "Lightweight verification of array indexing." In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[2] **Martin Kellogg**, Narges Shadab, Manu Sridharan, and Michael D. Ernst. "Lightweight and Modular Resource Leak Verification." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[3] **Martin Kellogg**, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. "Verifying object construction." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.

[4] **Martin Kellogg**, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. "Continuous compliance." In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[5] Chad Woolf, Byron Cook, and Tom McAndrew. “Automate Compliance Verification on AWS Using Provable Security.” https://www.youtube.com/watch?v=BbXK_-b3DTk. 2019.

[6] Xin Qi and Andrew C. Myers. "Masked types for sound object initialization." *ACM SIGPLAN Notices (POPL)* 44, no. 1, 2009.