Research Statement

Andrew K. Hirsch

My work focuses on the semantics and design of programming languages for writing decentralized software [5–9, 11, 17]. Thus, my work allows programmers to write and reason about programs with components which may be mutually distrusting, may represent different threads of computation, may represent different nodes in a distributed system, or more. My general approach is to include trust and communication as explicit primitives in the languages that I design. I abide by a key design principle: communication should only happen along trust lines. This principle allows us to design languages where programs and proofs are *correct by construction*: communication never leaks secrets, nor does it cause errors such as deadlock.

My research in this space can be divided into three lines. First, I have developed **new semantic foundations for language-based security**, which provide a toolbox for *designers* of such languages. Language designers often have to prove theorems—such as *noninterference*, the main security theorem for information-security policies—for every language they design. By taking advantage of semantic techniques, we can prove such theorems once, and then apply them to a new language simply by noting what language features it has and does not have. Second, I have designed logics with **trust as a first-class citizen** to state and reason about security policies. Treating trust as something that can be manipulated in the policies themselves allows us to reason about more-realistic policies, but makes the metatheory much more difficult to develop. Finally, I am working on **functional deadlock-free-by-construction concurrent programming**. This allows a programmer to program in a familiar functional style and extract several programs—for example, one program for each node in a distributed system. These programs provably communicate according to a shared protocol.

Not only is my work unified by motivations involving decentralization, communication, and trust, it also shares a foundational technical core: *modal logic*. In my work, I represent different decentralized software components using *modalities of truth*, a concept which goes back at least to Aristotle and which had been used extensively by the logic and computer-science communities since the 1960s. As a result, I can address the problems of decentralized software using decades of work on the semantics and proof theory of modal logic.

In the following three sections, I explain each of these lines of research in more detail. In particular, I describe the research I have done in each line and how it has addressed important gaps in the research literature.

New Semantic Foundations for Language-Based Security

Language-based security strives to design programming languages which enforce security in any program written in those languages. Most commonly, they enforce information-security guarantees using *information-flow control*; languages that implement information-flow control are called information-flow languages. In their simplest form, these languages divide data into secret and public using a *modal* type system in which modalities of truth are interpreted as information-security policies. The languages then use these type systems to ensure *noninterference*, which states that secret data cannot affect an adversary's observations.

Almost all previous work on information-flow languages has focused on building languages for new settings. In contrast, **my goal is to develop a semantic toolbox for the designers of information-flow languages**. In particular, I have built models of information-flow languages and exploited those models to prove general theorems which can apply to future languages. This can make noninterference—along with other information-security guarantees—easier to prove in future works.

I have begun to build that toolbox in three ways. First, I designed FLAFOL, an authorization logic that inclues information-security policies [9]. Since authorization decisions are based on data which might be subject to information-security policies, care must be taken to not leak information about such data when enforcing authorization policies. Authorization-policy enforcement corresponding to FLAFOL proofs is guaranteed to comply with these information-security policies, with no additional proof burden on the implementer.

Second, I provided a semantics of *program-counter labels*, a common aspect of information-flow–control type systems, via a *monadic* semantics of *effects* [5]. Noninterference arguments for languages that use program-counter labels are famously complicated. My semantics enabled me to prove a theorem called *noninterference half-off*. This guarantee enables simpler proofs of noninterference, since language designers can use monadic semantics to reason about effects rather than program-counter labels. In particular, they need only prove noninterference for simpler *pure* languages, which do not need program-counter labels.

Third, I provided a Coq library based on *Interaction Trees* for implementing and reasoning about information-flow programming languages [17]. Interaction trees represent possibly-nonterminating programs in Coq as trees of interactions with their environment, with a branch for every possible response from the environment. The original interaction-trees library reasoned about interaction trees using *bisimilarity*, which states that programs interact with their environment in the same way. I added an *indistinguishability* relation, which tells us when an adversary cannot see a difference between programs via their interactions with their environments. This allowed me to instantiate the indistinguishability model of information flow [14], and thus use the interaction-trees library to build simple proofs of noninterference. In particular, I was able to show that a compiler preserved noninterference. Notably, formal proofs of secure compilation are an active area of current research [2].

In current work, I am working on building indistinguishability models of particular languages which combine functional programming with *declassification* using logical relations [11]. Declassification allows secret data to influence public data in certain cases; thus, noninterference is no longer an appropriate security guarantee. Instead, more-subtle guarantees limit the influence private data can have on public data. By understanding what those more-subtle guarantees are and how they can be proven, I am stretching the limits of indistinguishability models. Thus, while building models of particular languages does not result in theorems that can be applied to new languages, I plan to use the lessons learned from this work to build better general models.

Trust as a First-Class Citizen in Logic

Systems have certain resources that not every component should have access to. Policies about what system components may access what resources are known as *authorization* policies. For instance, imagine a key-card access system for an apartment complex where users may fill any combination of three roles: residents, who may access their homes; staff, who may access the building they work in; and security auditors, who may see access logs. Access logs map key-card–id numbers to building accesses, while security auditors are not allowed to know which key-card–id numbers are associated with which names. An auditor, Alice, may see a card access both the building that her friend Bob works in and the building he lives in causing her to believe that the number belongs to Bob. Based on this, Alice may take certain actions; for instance, she may ignore when that key-card id accesses buildings that she believes Bob should have access to. If another guard, say Dave, trusts Alice, he will be willing to believe Alice when she tells him that the key-card id belongs to Bob.

Previous work used modalities to reason about *beliefs* such as Alice's. Combining modal logics with trust resulted in logics called *authorization logics*. Authorization logics have successfully been used not only to reason about authorization policies [1], but also to enforce such policies [16, 18].

While directly using logic to enforce authorization policies ensures security, doing so also breaks the assumptions used in arguments for the correctness of such logics. These arguments rely on *models* of the logic, which interpret statements in the logic as being true about some mathematical structure. Because the systems use logical formulae directly to represent beliefs, systems no longer fit these models, since the models do not contain logical formulae. In order to bridge this disconnect, I developed FOCAL—an authorization logic similar to that found in the Nexus operating system [16, 18]—and built a *belief* semantics for it that also uses formulae to represent beliefs [6]. Belief semantics inspired later work in models for social-network privacy policies [13], while the design of FOCAL influenced the design of a later authorization logic used in software-defined networks [19].

While FOCAL provided correctness arguments for practical authorization logics, no practical authorization logic was known to enjoy *noninterference*, which is also the name of the main security theorem for authorization logics. Noninterference for authorization logics says that a principal's beliefs do not affect what another principal believes, as long as the second principal does not trust the first. However, knowing who trusts whom when trust can be manipulated by logical connectives is highly nontrivial. When I designed FLAFOL [9], which refined FOCAL with information-security policies, I adopted proof-theoretic techniques to state and prove noninterference even in the face of these difficulties.

Through FOCAL and FLAFOL, I have managed to solve real impediments to real-world use of authorization logics. However, both theoretical and practical issues remain to be addressed. On the theoretical side, I am currently working to scale the techniques I used in the proof of noninterference of FLAFOL to handle new logical features. On the practical side, I am interested in developing secure proof-search techniques in order to help prove that enforcement mechanisms do not violate security.

Deadlock-Free-By-Construction Higher-Order Programming

So far I have described security aspects of decentralized programming. However, decentralization also requires writing concurrent software, allowing code to be distributed between different processes or even computers. I am particularly interested in concurrent programs where processes communicate via messages, as happens at every level of concurrency, from distributed systems to modern processors. A major goal of many works on message-passing concurrency is proving *deadlock-freedom*—that the different processes in the code never find themselves waiting on each other. However, this focuses only on the concurrency model, while care must also be taken when choosing a programming model for writing concurrent software. Usually, software engineers prefer higher-order languages such as object-oriented or functional languages, since higher-order abstractions allow more and better-behaved code reuse. Combining these features with features from concurrency models makes deadlock-freedom even more difficult to reason about. Much work uses increasingly-complicated *session types* to tame communication [see e.g., 10, 15].

Choreographic programming offers a promising alternative to session types: instead of writing each process's code separately, we write one program for our entire system using an Alice-and-Bob–style notation [12]. However, previous formal work only considered lower-order choreographic programming, which prevented serious code reuse. Despite this, unverified implementations of higher-order choreographic programming, via an object-oriented language called Choral, found real-world use based partially on deadlock freedom [4].

I solved this problem by designing Pirouette, the first (higher-order) functional choreographic language [7]. Pirouette's design is parameterized over a single-process language; Pirouette messages are values in this language. Moreover, I showed how we can lift a type system for such a single-process language to a modal type system for Pirouette. Here, modalities of truth represent a process holding a value or performing a computation. Thus, modal logic forms an essential organizational backbone for Pirouette.

Endpoint projection translates a Pirouette program into a program for each process expressed in a standard concurrent λ calculus. That is, each process's program is expressed in a functional programming language with conventional send and receive primitives. While programs in this language can easily deadlock, projections of Pirouette programs are deadlock free. This guarantee is formally proven in Coq, as are all of Pirouette's other guarantees. The corresponding increase in confidence of these guarantees is especially important given that major theorems in concurrency theory have recently been disproved [3].

Pirouette only begins the exploration of higher-order choreographic programming. While it provides safe higher-order concurrent programming, the data structures expressible in Pirouette are limited to those that can be stored locally. Moreover, Pirouette does not allow communication of types nor type-level dynamics. In current work, I am extending Pirouette with all of these features. I plan to continue this work for a long time, lifting functional-programming features to safe distributed programming.

Future Work

I believe that programming languages and logics with strong guarantees—such as deadlock-freedom and security—will serve an indispensible role in the future of decentralized computing. However, not every situation calls for the same langauge. As we explore new domains and learn from the past, our languages will have to evolve. Thus, we must deeply understand how our languages deliver their guarantees. Such an understanding requires not only continuing the threads of research I described above, but also using their shared modal foundations to combine them. Moreover, I plan to design concrete languages, abstract models, and verification techinques. Each of these styles contributes directly to this understanding and inspires more developments in the others.

Concrete languages allow us to explore the design space of languages and logics while also holding the possibility of impact via adoption. Going forward, I plan to continue to build new logics and programming languages with strong guarantees. For instance, I am currently working on building and proving secure an authorization logic which contains the logical features necessary to state practical security policies. Moreover, I plan on extending Pirouette in several directions, such as by adding the ability to interact with *client* processes which are not controlled by the program author.

Using abstract models of languages, we can develop libraries of theorems which can be applied to new concrete language designs. So far, my work in this direction has focused on information-flow control, but this is not fundamental. In fact, I have built abstract models focusing on other issues in programming-language design, specifically in combining producer and consumer effects [8]. Going forward, I plan to ease the design of new authorization logics and choreographic languages by building abstract models of these languages as well, just as I have for information-flow languages.

Finally, verification techniques give a toolbox for programmers who want to prove their programs secure and correct. Coq libraries such as the interaction-trees library are a promising method for applying abstract models to verification. This style also holds promise for authorization policies: Coq libraries and metaprogramming facilities could allow realistic methods for providing authorization-logic proofs that systems enforce security policies. Language design also has a significant role to play here. In particular, I plan to use the underlying similarities between authorization logics and choreographic programming to build a tool for verifying concurrent programs.

References

[1] Martín Abadi. Access control in a core calculus of dependency. In *International Conference on Functional Programming (ICFP)*, 2006. doi: 10.1145/1159803.1159839.

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation, 2019. URL `https://arxiv.org/abs/1807.04603`.

[3] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peresotti. Choreographies in coq. In *Types for Proofs and Programs (TYPES)*, 2019. URL `http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_27`.

[4] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects, 2020. URL `https://arxiv.org/abs/2005.09520`.

[5] Andrew K. Hirsch and Ethan Cecchetti. Giving semantics to program-counter labels via secure effects. In *Principles of Programming Languages (POPL)*, 2021. doi: 10.1145/3434316.

[6] Andrew K. Hirsch and Michael R. Clarkson. Belief semantics of authorization logic. In *Computer and Communication Security (CCS)*, 2013. doi: 10.1145/2508859.2516667.

[7] Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. In *Principles of Programming Languages (POPL)*, 2022.

[8] Andrew K. Hirsch and Ross Tate. Strict and lazy semantics of effects: Layering monads and comonads. *International Conference on Functional Programming (ICFP)*, 2018. doi: 10.1145/3236783.

[9] Andrew K. Hirsch, Pedro de Amorim, Ethan Cecchetti, Owen Arden, and Ross Tate. First-order logic for flow-limited authorization. In *Computer Security Foundations (CSF)*, 2020. doi: 10.1109/CSF49147.2020.00017.

[10] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asyncrhonous session types. *Journal of the ACM*, 63(1):1–67, 2016. doi: 10.1145/2827695.

[11] Jan Menz, Andrew K. Hirsch, Peixuan Li, and Deepak Garg. Logical relations for higher-order where declassification. In *In Preparation*, 2022.

[12] Fabrizio Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013. URL `https://www.fabriziomontesi.com/files/choreographic_programming.pdf`.

[13] Raúl Pardo. *Privacy Policies for Social Networks—A Formal Approach*. PhD thesis, Chalmers University of Technology, 2017. URL `https://research.chalmers.se/publication/252991`.

[14] Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. In *European Symposium on Programming (ESOP)*, 1999. doi: 10.1007/3-540-49099-X_4.

[15] Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. In *Principles of Programming Languages (POPL)*, 2019. doi: 10.1145/3291638.

[16] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus Authorization Logic (NAL): Design rationale and applications. *Transactions on Information and System Security (TISSEC)*, 14(1), 2011. doi: 10.1145/1952982.1952990.

[17] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Security semantics with interaction trees. In *In Submission to SPLASH*, 2022.

[18] Emin Gün Sirer, Willem De Bruijin, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Symposium on Operating Systems Principles (SOSP)*, 2011. doi: 10.1145/2043556.2043580.

[19] Christian Skalka, John Ring, David Darais, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffan Smolka, and Nate Foster. Proof-carrying network code. In *Computer and Communication Security (CCS)*, 2019. doi: 10.1145/3319535.3363214.