Personal Statement

Kristopher Micinski, Syracuse University

During my first three years at Syracuse, I have consistently published at top-tier venues, chaired several workshops, raised $914k in federal grant funding, and been actively involved in community mentorship efforts. In my research, I have branched out into novel directions relating to scaling program analyses engines to unprecedented levels of throughput and expressivity. Significant innovations include novel all-to-all communication algorithms (HPDC '22), scaling control-flow analysis up to 800 threads of a supercomputer (CC '21), lattice-oriented programming (in Rust) which beats current systems (Flix) by 50× (CC '22), and orders of magnitude (hours to seconds) runtime improvements for logic programs which manipulate tree-shaped data. I have also continued to focus on reverse engineering (USENIX '20) and formal methods for security foundations (CSF '22). My long-term goal is to use these tools to revolutionize reverse-engineering by enabling high-fidelity decompilation, instrumentation, hardening, and exploit generation for stripped binaries, powered by our high-performance engines and logical innovations.

In the rest of this document I include a research statement, a teaching statement, and conclude with a statement of career goals and perspective. I have also included five selected papers. Each of these was produced during my first three years at Syracuse:

- HPDC '22. "Optimizing the Bruck Algorithm for Non-Uniform All-to-All Communication." [4] Work I did in our collaborative PPoSS grant to optimize all-to-all for parallel relational algebra—this came up in our implementation and we mentored Sid's student Ke to build new algorithms to optimize all-to-all for nonuniform comm workloads.
- CC '22. "Seamless Deductive Inference via Macros" [14]. My PhD student's first paper introducing Ascent, which got all strong accepts at CC.
- CC '21. "Compiling Data-Parallel Datalog" [5]. Our first paper on data-parallel Datalog, excluding language-level innovations we have developed but detailing our parallelization approach.
- CSF '20. "Abstracting Faceted Execution." [10]. My paper on abstract interpretation for faceted execution, a dynamic reference monitor. This approach can be seen as a new perspective on program analysis for information flow which achieves high precision due to a novel representation.
- USENIX '20. "An Observational Investigation of Reverse Engineers' Processes." [15]. Collaborative work (with Dan Votipka) where we interviewed reverse engineers to understand and categorize a technical exposition of pain-points and workflows that drove their reverse engineering process. This empirical work motivates my excitement to apply static analyses (and machine-learning-based tools) to reverse engineers in the future.

I also include a draft:

- "Higher-Order, Data-Parallel Structured Deduction" [6] This work details an (a) state-of-the-art program analysis / deductive inference system, (b) a novel language implementation strategy for Datalogs using structured data which (c) allows order-of-magnitude improvements compared to state-of-the-art engines.

1 Research

In my research, I design and implement formal systems for reasoning about computer security at an unprecedented scale of computational and expressive power. Throughout my career, I have published top papers in a wide breadth of areas, many inspired by an overarching goal: *achieve orders-of-magnitude algorithmic breakthroughs necessary to apply rich formal systems in reasoning about the security of production software systems.* During my PhD, I worked on interaction-based security, collaborating on foundational work (e.g., HyperLTL [3]) and using techniques such as abstract interpretation and symbolic execution to check security properties of production Android applications [9, 11, 12]. At Syracuse, I have built a large and diverse research group (consisting of three PhD students, four MS students, and over 10 undergraduates) which works to build state-of-the-art reasoning engines for expressive logical analytics tasks including program analysis, model checking, and security auditing. My students have done foundational work and published at strong venues, and I have raised $914k in extramural research funding from DOD, DARPA, and NSF.

```
if secret%2 == 0:
  output("even")
else:
  output("odd")
```

Logic, Verification, and Program Analysis for Security A primary research focus of mine is language-based security, a field which uses programming language semantics and logic to provide rich formal guarantees of the security of a software system. For example, consider the program on the right: it is insecure (assuming the output is publicly visible) because observing the output allows us to know secret's evenness. This so called implicit flow is a violation of *noninterference*, which says that no secret input may influence a program's public outputs. Noninterference was first formalized by Goguen and Meseguer in 1982 [7], but has proven extremely challenging to scale to production systems. This is because noninterference is a *hyperproperty*, a statement that quantifies over multiple executions of the program simultaneously (in this case, relating all input-output pairs to other input-output pairs). Unfortunately, while hyperproperties are known to characterize a broad class of security properties, they face a serious algorithmic challenge: checking hyperproperties often involves reasoning about pairs (or, in general, k-products) of program states, leading to state explosion for program analyses based upon hyerproperties.

During my PhD, I collaborated to build model checking algorithms for hyperproperties. In this work, we defined new temporal logics for security, notably HyperLTL [3]. HyperLTL extends the well-known linear temporal logic with *trace quantifiers*, which explicitly bind traces so that multiple traces may be mentioned simultaneously. For example, we may express noninterference via the HyperLTL formula $\forall\pi_0.\exists\pi_1.\big(G\ (\mathsf{input\_empty}\ \pi_1)\big) \wedge \big(\pi_0 =_L \pi_1\big)$, which reads: "for all traces π_0, there exists π_1 such that globally (always, G) π_1's input is empty, and π_1's output is low-equivalent to π_2 (i.e., their public views agree, $\pi_0 =_L \pi_1$). HyperLTL is strictly more expressive than LTL, as LTL's temporal quantifiers globalize the notion of a trace and thus preclude mentioning two traces at once. I codesigned HyperLTL and implemented the model checker for HyperLTL for labeled transition systems, the first model checker capable of verifying general hyperproperties based on temporal logics. Our paper "Temporal logics for hyperproperties" has been cited over 200 times, and HyperLTL has spawned numerous efforts in verification for security properties [8, 2, 1].

Our work on HyperLTL, while a rich and expressive result, was unsatisfactory to me personally because of its high algorithmic overhead: our model checking algorithm uses an explicit cross-product construction on the state space, incurring state explosion and inhibiting its application to nontrivial software systems. I became inspired by faceted execution, a dynamic enforcement mechanism for noninterference wherein program values may be *faceted*, representing a decision tree of views based upon the observer. For example, the facet $\langle \text{Alice} : 1 \diamond \bot \rangle$ would signify a value which is 1 when viewed by Alice, and $\bot$ (null) when viewed by anyone else. After implementing facets as a set of Scheme macros with an undergraduate advisee (at Haverford [13]) I came to see facets as powerful, but far too slow to apply to production languages (paying a penalty at every indirect control transfer). This led me to an insight: I could use facets in an *abstract* execution of the program, soundly approximating which values may be faceted via a novel program analysis. I elaborated this idea in my CSF '20 paper, "Abstracting Faceted Execution," which presents a novel approach to noninterference by combining abstract interpretation with faceted execution [10].

Next-Generation Program Analysis and Data Analytics Engines After spending a decade working on formal methods for security, I became disheartened by the fact that so many of the rich formal systems I studied were infrequently applied to ensure security for production systems. Upon joining Syracuse, this gulf between theory and practice inspired me to shift my focus to the design and high-performance implementation of expressive declarative languages for program analysis and similar formal systems. I have led the development of two complementary declarative languages targeting the implementation of formal systems, Slog (an MPI-based data-parallel Datalog) and Ascent (for large unified-memory machines), both of which achieve orders-of-magnitude performance improvements compared to the state of the art. My work on Slog and Ascent has been funded by DARPA V-SPELLS ($400k) and NSF PPoSS (planning, $83k).

Our work was initially motivated by several major deficiencies inherent to Datalog, a sub-Turing logic programming language based on Horn-SAT. For example, the Datalog program on the right computes the transitive closure of edge by asserting that (a) every edge is a path, and (b) any path from x to y may be extended by an edge from y to z. Datalog has attracted significant recent interest due to highly-efficient implementations (e.g., Soufflé) which power applications such as ddisasm (the most precise disassembler currently available) and DOOP (which scales context-sensitive analysis of Java to thousands of lines).

```
path(x, y) :- (edge x y)
path(x, z) :- (path x y),
              (edge y z)
```

```
ret(av, k) :- ret(av, $Frame(e, env)),
              store($Frame(e, env), k).
```

While Soufflé achieves best-in-class performance for graph analytics tasks such as transitive closure and triangle counting, our experiments revealed orders-of-magnitude inefficiencies when using Soufflé for the implementation of formal systems which rely upon structured data such as lists, trees, and abstract syntax trees. To understand the problem, consider the code on the left which comes from an abstract interpreter and uses the constructor $Frame to represent a stack frame. This rule performs a join between the ret and store relation; crucially, however, the join matches the common value $Frame(e,env). Unfortunately, Soufflé's compilation methodology does not allow indexing for structured data, and instead uses a select-then-filter approach,

implicitly materializing a cross-product and inducing superlinear algorithmic complexity. This inefficiency is a symptom of a general problem inherent to conventional Datalogs: structured data is *second-class*, compared to facts proper.

We hoped to use Datalog to write highly-scalable implementations of formal systems, which would perform whole-program static analysis on large codebases and scale to hundreds of threads while retaining the succinctness of declarative languages. However, the algorithmic and scalability deficiencies we observed in modern Datalog solvers pushed us to create Slog, a data-parallel declarative language designed to enable scaling formal systems up to hundreds of threads on large unified machines and leadership-class supercomputers. Our approach has three main parts: (1) a key semantic extension to Datalog, *subfacts* and a *subfact-closure* property, that is (2) implemented uniformly via ubiquitous fact interning, supported within relational algebra operations that are (3) designed from the ground-up to automatically balance their workload across available threads, using MPI to address the available data-parallelism directly. Slog was designed in collaboration with Thomas Gilray and Sidharth Kumar (UAB), and inspired a wide array of problems (e.g., new innovations in all-to-all communication algorithms, which we published at HPDC '22 [4]). Slog's subfact-closure property lifts all subfacts to the top level, allowing them to be indexed just as relations in vanilla Datalog. Table 1 (right) illustrates the importance of Slog's subfacts, revealing the exponential gap between Slog and Soufflé on a control flow analysis (k-CFA): as program size (n) increases, analysis complexity (control-flow points, third column) grows exponentially, but the superlinear overhead induced by Soufflé quickly makes the analysis intractable (e.g., five seconds in Slog versus 45 minutes in Soufflé). Slog's Datalog subset was published at CC 2021 [5]—demonstrating scalability of whole-program control-flow analysis up to 800 threads on the Theta supercomputer; the full language and runtime system are currently in submission to several top conferences.

Table 1: 3-k-CFA: Slog vs. Soufflé

n	Iters	Cf. Pts	64 Processes	
			Slog	Soufflé
8	1,193	98,114	0:02	00:15
9	1,312	371,010	0:03	02:56
10	1,431	1,441,090	0:05	45:49
11	1,550	5,678,402	0:16	☃
12	1,669	22,541,634	1:07	☃
13	1,788	89,822,530	5:08	☃

While Slog represents a major innovation in declarative language design and scales to hundreds of threads, its distributed nature (a) imposes sequential overhead (most significant at low core counts) and (b) forbids computations that would require materializing an entire relation on a single thread. To complement Slog, we designed Ascent—a macro-embedded language (in Rust) which beats Soufflé on a single thread and allows programming with *non-powerset lattices*. For example, the Ascent program on the right computes the shortest path between pairs of nodes. At each iteration, the Ascent implementation uses the lattice's join operator (min, in this case) to store *only* the shortest path. This is impossible to express in vanilla Datalog, which would force enumeration of *all* paths. Lattices are crucial for achieving acceptable algorithmic complexity for many program analysis and graph analytics tasks. Ascent's parallel, Rust-based implementation beats all competitor systems, e.g., beating Flix (a lattice-oriented program analysis language) by 50–100×. Our single-threaded Ascent implementation appeared at CC '22 [14]; we are currently adding parallelism (beating Soufflé at all core counts) and integration with SMT solvers to enable parallel symbolic execution, model checking, and property-directed reachability.

```
lattice shortest_path(i32, i32, Dual<u32>);
relation edge(i32, i32, u32);
shortest_path(*x, *y, Dual(*w)) <-- edge(x, y, w);
shortest_path(*x, *z, Dual(w + l))
  <-- edge(x, y, w), shortest_path(y, z, l);
```

Building and Analyzing Binary Corpuses: Assemblage My long-term goal is to apply program analysis (particularly for security) to large corpuses of production software. To support this, I have spent the past two years leading a team of (masters and undergraduate) students to build Assemblage, a software system which scrapes all of GitHub and automatically builds Windows PE (currently 600k) and Linux executables (2M) in a diverse set of build configurations. Crucially, Assemblage also builds a rich source-to-binary mapping. We are currently using Assemblage to benchmark a wide array of binary analysis and malware classification tools, and are preparing a submission to the VLDB Scalable Data Science track. Assemblage is collaborative work with the US Laboratory for Physical Sciences ($430k).

2 Teaching

Passion for teaching motivates every aspect of my scholarship. Since completing my PhD, I have taught 13 courses at the undergraduate and PhD level, consistently achieving top teaching reviews and producing freely-available content (on YouTube) which has gained significant popularity (thousands of views) outside of Syracuse. Central to my approach is a project-focused style, which pairs hands-on programming exercises with challenging projects that demand students to cultivate skill in debugging and software engineering. I consistently innovate in course delivery, leveraging my computing expertise to build highly-interactive courses. However, I also realize that effective teaching requires humility, compassion, and communication. I check in with students frequently and personally, pushing them to achieve as much as I know

they can. And I reflect on the ways in which I fall short, learning from my failures and helping me become better at these things.

Teaching History and Goals I have taught a broad array of courses including introductory computing (in Python), programming languages (at Maryland, Haverford, and Syracuse), compilers, and graduate seminars in program analysis and security. After my PhD, I further committed myself to teaching by accepting a teaching-focused postdoc (visiting professorship) at Haverford College. In each of these experiences, I have observed a unifying challenge: undergraduate students have weak end-to-end problem-solving skills, reifying as an inability to successfully identify the proximate source of a bug. This has inspired the following overarching goal: *equip students with the the problem-solving (i.e., debugging) skills they need to write correct, robust programs on their own.* To reach this goal, I design engaging programming projects which tackle foundational CS concepts and build upon collaborative in-class exercises. For example, during my module on the λ-calculus, students build a Church encoder, compiling a significant subset of Scheme into just the λ-calculus; students say this allows them to viscerally experience the Turing-completeness (of the λ-calculus) in a familiar language. While I design these projects to reinforce course topics, an unstated goal is to force students to grow practical debugging skills.

Flipped-Classroom and Course Delivery Early in my career, I focused on developing thoughtful lectures, hoping to inspire students in the same way I was inspired to love computing. As I gained experience, I came to understand the myriad learning styles my students have. However, I observed one common issue: *once a student gets lost, they will quickly feel defeated and become disinterested.* I structure my courses to directly confront this problem in several ways. First, I developed my undergraduate course as a series of 15–30 minute video lectures. While this required serious time investment in filming, editing, and production, students universally report that engaging, well-produced videos encourage them to repeatedly absorb the most challenging concepts. I am proud of these videos, and (if you are so inclined) I invite you to watch one (e.g., "L5: Recursion over Lists") to get a sense of my teaching style. During class proper, I begin by recapping the video, giving a slightly different perspective. However, I also integrate in-class worksheets and programming exercises. These exercises have significantly helped my course outcomes: instead of students becoming cumulatively more lost through lecture, exercises force students to "check in," and force me to devote more time to topics that students find most confusing.

Building a Growth-Positive Classroom Effective instruction requires motivating students to push themselves beyond their limits. I want my students to confront failure as rapidly as possible, understanding these failures as a necessary vehicle for growth rather than a rebuke of their efforts or person. In the beginning of my career, I often wasted my time perfecting content that was useful only to the highest-achieving students, and failing to help lower-achieving students meet their potential. There is no technical solution this problem, because it is a human problem. As such, I *use technology to automate repetitive teaching tasks to enable an unprecedented level of personal attention to students.* I am always available in office hours, Zoom, Slack, and email. I directly email students who are lagging on projects, asking them when they plan to finish and emphasizing that I want them to succeed. My teaching reviews reflect this: students *often* say that I am the most supportive instructor they have ever had, and praise my commitment to fairness and student support.

3 Goals and Perspective

My current career ambition is to continue to grow a productive research group that consistently produces top-quality work. This includes mentoring and successfully graduating high-quality PhD students, along with mentoring more junior (MS and undergraduate) students in my research lab. I will continue to compete for grant money (focusing on NSF), and am currently planning $1.1 million of a 5-year PPoSS Large to be submitted in January. I will also put together a competitive application for NSF career and will submit each year until I go up for tenure. I see myself as a strong candidate for CAREER, having spent the last several years working to build my research output (after a teaching-focused postdoc) in preparation to compete for the CAREER grant.

In terms of concrete outputs, I plan to submit to multiple top venues every year in areas such as security (USENIX, CCS), programming languages (POPL, PLDI, ICFP, SPLASH), and formal methods (CAV). I will also submit to several lower-tier venues each year with MS and undergraduate students, using these papers as training exercises with the long-term goal of building a student's research productivity. However, I have found that the quality if feedback from lower-tier conferences is much lower, and I am most eager to submit extremely strong papers to top-tier venues.

In terms of research direction, I hope to use my pre-tenure years to establish myself as a researcher who has made

foundational contributions to binary analysis, reverse engineering, and formal methods for security. My students and I have engineered unique tools, datasets, and techniques for analyzing large (millions of binaries) binary corpuses. These systems-building projects have taken a long time, and we must capitalize on that effort to produce novel results using our group's collective knowledge and perspective.

Last, I hope to continue to be more active in various areas of teaching and service. I see teaching as a strength of mine that does not require a large amount of effort, though I will continue to work hard to deliver top-quality courses and to maintain high engagement with my students. I also have chaired several workshops, and this has been a positive and enriching experience for me. However, one area of necessary improvement I see is involvement on PCs at top-tier conferences. I see this as something that will naturally come as I continue to routinely submit to top-tier venues, but I will work to improve this. Last, I need to continue to improve mentorship and diversity efforts. Chairing PLMW is a start in this direction, but I am working to brainstorm a "reverse engineering / security" mentorship program that seeks to appeal to both industry and academic participants (e.g., undergraduates seeking to go into security, or even professionals without formal credentials seeking jobs as reverse engineers). I hope to flesh out these plans as part of my NSF CAREER proposal.

References

[1] Erika Ábrahám and Borzoo Bonakdarpour. Hyperpctl: A temporal logic for probabilistic hyperproperties. In Annabelle McIver and Andras Horvath, editors, *Quantitative Evaluation of Systems*, pages 20–35, Cham, 2018. Springer International Publishing.

[2] Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of hyperltl. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.

[3] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer Berlin Heidelberg, 2014.

[4] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. Optimizing the bruck algorithm for non-uniform all-to-all communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 172–184, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Thomas Gilray, Arash Sahebolamri, Sidharth Kumar, and Kristopher Micinski. Higher-order, data-parallel structured deduction, 2022.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[8] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 94–112, Cham, 2021. Springer International Publishing.

[9] Jinseong Jeon, Kristopher Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*, pages 3–14, Raleigh, NC, USA, October 2012.

[10] Kristopher Micinski, David Darais, and Thomas Gilray. Abstracting faceted execution. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF '20)*, pages 184–198, 2020.

[11] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S. Foster, and Michael R. Clarkson. Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution. In *European Symposium on*

Research in Computer Security (ESORICS '15), volume 9327 of *Lecture Notes in Computer Science*, pages 520–538, Vienna, Austria, September 2015.

[12] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Jeffrey S. Foster, and Michelle L. Mazurek. User Interactions and Permission Use on Android. In *Conference on Human Factors in Computing Systems (CHI '17)*, 2017.

[13] Kristopher K. Micinski, Zhanpeng Wang, and Thomas Gilray. Racets: Faceted execution in racket. In *Scheme Workshop at ICFP 2018 (Scheme '18)*, 2018.

[14] Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 77–88, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Daniel Votipka, Seth M. Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle M. Mazurek. An observational investigation of reverse engineers' processes. In *Proceedings of the 29th USENIX Security Symposium*, Proceedings of the 29th USENIX Security Symposium, pages 1875–1892. USENIX Association, 2020. (Acceptance rate: 16.1%).