Project Summary

CAREER: An Incremental Approach to Declarative Reverse Engineering

Overview:

Reverse engineering (RE) is a ubiquitous aspect of software understanding. For example, in binary reverse engineering, an expert analyst interactively discovers ("reverses") properties of a low-level software artifact such as a Windows PE or ELF binary. Recent observational studies of RE professionals show that they employ iterated rounds of hypothesis formation and falsification, assisted via a variety of automated and ad-hoc methods. Beyond binary RE, programmers employ RE skills throughout their career, in debugging their own code, understanding their collaborators' code, or integrating with APIs and frameworks.

Unfortunately, because RE tools (such as decompilers, symbolic executors, and binary analysis systems) are extremely complex software systems, they are often very brittle and do not readily facilitate ad-hoc experimentation or integration with downstream tools. This project proposes a new methodology for reverse engineering, declarative reverse engineering, wherein the artifact under analysis is decompiled to an arbitrarily-higher-level representation via declarative rules. I will lead the development of REVERSI, the first multi-target, multi-language declarative decompiler that scales to production-sized x86-64 binaries. The project's thesis is that the use of modern logic programming techniques will directly tackle the challenges faced by RE tools. The project's goal is to revolutionize the way we design decompilers, radically increasing efficiency, reusability, and scalability by cleanly isolating the logical specification of the decompilation task from low-level matters such as data-structure implementation, binary parsing, and parallelization.

Keywords: Reverse engineering; Decompilation; Datalog; Logic Programming

Intellectual Merit:

This project develops new foundations for software reverse engineering, extending an active body of work in logic-defined disassembly to logic-defined decompilation. The project's approach is to evolve (bottom-up) a forest of candidate decompilations, iteratively applying (in parallel) a set of decompiler passes; each pass is implemented via logic-defined rules specifying the conditions under which a low-level fragment may be lifted to an incrementally-higher-level representation. To successfully tackle inevitable state-space complexity, we leverage efficient representations (E-graphs), high-performance operationalization (GPUs, supercomputers), and novel search strategies (aggressive path pruning). While numerous prototypes of logic-defined compilers exist for domain-specific languages (e.g., smart contracts), the proposal targets large Linux ELF / Windows PE x86-64 and ARM binaries. Successfully scaling the principles of declarative decompilation to general-purpose languages (C, C++, Rust) and production binaries will require new data structures, programming abstractions, and formalizations. To ensure a rich formal basis, the project (de)compiles binaries to the verified compiler toolchain CompCert—explicating sources of potential unsoundness by ubiquitously tracking provenance of each decompiled program fragment.

Broader Impacts:

The project will develop an RE summer school aimed at undergraduates looking to develop their skills in systems, compilers, and RE. Supporting my long-term teaching goals in compiler design, I will develop a course module, "Essentials of Decompilation." Additionally, I will develop a graduate-level seminar, "Decompiler Design." The project will also support the continued development of ASSEMBLAGE, a successful dataset curation project (over 1M Windows PE binaries) actively used by several government agencies and labs. Last, I will maintain active connections with industry players, maintaining a blog "Declarative Reverse Engineering in Practice" and releasing demo-focused YouTube videos for the general public which teach an iterative, debugging-focused approach to problem solving. All educational material, datasets, research artifacts, and publications will be made publicly-available via a project website.

Mentoring Plan

This project requests funding for one graduate student paid full-time throughout time the duration of the project. I have mentored several PhD students, with my first student (Arash Sahebolamri) successfully defending his dissertation during the summer of 2023. In this plan, I describe the strategy I currently use to mentor my PhD students; above all, I work to understand my students as individuals, continuously working to identify a mutually-harmonious collaboration style that helps the student achieve their career goals.

Setting and Communicating Expectations When each student joins my group, we have a long meeting where I discuss my responsibility as their advisor. My role is to support the student as they become a researcher: I teach them advanced technical material relevant to the domain, teach them how to identify potentially-fruitful directions, and I teach them how to communicate their results by collaboratively writing papers that appear at competitive, well-regarded venues. Each year, the student and I will have a career status update meeting. These meetings begin with me asking how the student feels the last year went, what they enjoyed, what was painful or tedious, and how they would like to adjust course for the next year. I then give the student my feedback, discussing both positive habits I notice along with areas where the student needs to grow. I work to ensure that the student understands the root of my concern, and work to collaboratively brainstorm what systemic changes can be made to ensure productive outcomes over the next year.

Mentoring Events and Programs I will recommend the student to participate in research community mentoring programs, especially SIGPLAN's Programming Langauges Mentoring Workshop (PLMW) and the SIGPLAN-M mentoring network; these help students get exposure to the broader research community.

Meetings and Communication I meet for roughly an hour with each student once a week, either in person or via Zoom (if one of us is traveling). I ask students to send me a bullet-point list of progress and suggested review material (code, sections of a paper) the day before we meet so that I have time to prepare for the meeting: preparing for meetings is crucial to avoiding time-wasting distractions. When I meet with students, I work to rapidly move into the most challenging technical details, either bugs in code or confusion about specific technical formalism: I found that high-level discussion is often a waste of time, and that students get the most out of collaboratively working on hard, focused examples apropos the problem domain. At the end of each meeting, I have students write down a bulleted list of immediate next steps for the next week.

Flexibility and Communicating Negative Feedback Graduate school is often a stressful time, where students rapidly confront failure as they learn to do top-quality work. I work to ensure that when I give students negative feedback, I do so in an emotionally-neutral way, and take a growth-oriented perspective. I work to be a flexible advisor, and allow my students extreme flexibility in terms of working times, hours per week, vacation, and working location. When I notice consistently low engagement over the course of weeks or months, I work to be direct with students and identify the root of the issue, making clear that my role is to help them to start making consistent progress while being mindful of the issues impeding their efforts.

Lab Culture and Environment My lab is very collaborative, and I facilitate the lab culture in multiple ways. I hold group meetings with my students once a month where we eat and discuss topics, both work and non-work related. We routinely host reading groups on relevant papers, and I give students detailed, low-level feedback to build their presentation skills. Last, I take the lab to dinner several times a year to celebrate accomplishments (dissertation or proposal defenses, paper acceptances) and have fun.

Data Management Plan

This project will lead to the following types of materials: (a) research results in the form of scholarly publications arising out of the research tasks; (b) data and software artifacts arising out of the research tasks; and (c) educational/curricular materials produced by the project. **All of the project's results and artifacts will be made available and accessible through a dedicated project website.**

Managing of raw data and metadata: The project will use a range of open access/publicly available data. Any public data that will be used to test and benchmark our main research products will be appropriately referenced from our publications, ArXiV reports, and website, for reproducibility and testing by the broader research community. This also applies to any synthetic data that we generate in-house (e.g., synthetic graphs and model executions).

Data privacy: Our work-in-progress internal software will be maintained via private GitHub version control software repositories. This service also provides backups and archives of the repository.

Private / Medical Data: **No** private medical or personally-identifying information of *any* kind will be used as part of this project.

Dissemination Plan

Scholarly publications: Any significant research results arising from the project will be documented in the form of ArXiV reports, and sent to conferences for peer-review. We will make all those publications available/accessible from our website, either with links to the publication venues' digital archives or directly hosting with permissions from the publishers.

Software sharing: All the software developed as part of this project will be released to the public domain using an open-source license such as the GNU General Public License, version 3 (GPL-3.0) or the BSD license. This licensing allows for free availability to third-party non-profit sectors, while also provides the freedom to modify and commercialize specialized/customized versions of the software to corporate sectors. Software will be released only after it has been thoroughly tested and a peer-reviewed publication resulting from the use of the software has been published. This is to guarantee the software's scientific validity and utility to the user-base. We will, however, ensure that the software is made available as soon as the first peer-reviewed paper is accepted so that the broader community has immediate access to the software even as our project is still in progress. Software versioning will reflect the state of the different features supported and patches incorporated. All open source software products will be made available through BitBucket or GitHub so that it is easily available upon search on the web. Software will be accompanied with necessary user manuals, training videos, and tutorials about different functions and software features. In addition to the above venues for software dissemination, we will also develop demo toolkits that we will use for short courses and conference tutorials. These materials will also be made available on the respective open source publication sites.

Plans for Reproducible Research: PI Micinski is personally committed to reproducible research. All the software artifacts, data, and tools including architecture and system designs, models, simulation code, evaluation data, and application benchmarks will be made open-source and freely available, as described earlier in this plan. We will also maintain open-source repositories (e.g., github) for source code of algorithms and simulation models. We will create tutorial style materials on how to use the code and models, and how to extend them for relevant research in this problem space. We will use these materials while presenting tutorials and teaching short courses. We will ask independent researchers (graduate students and faculty) to test the usability and reproducibility of our results via coda worksheets and tutorial materials containing the step by step procedures. We will also use durable archival services such as Zenodo as appropriate to share living, reproducible artifacts for each of our research papers.

Project Description

1 Prologue: Reverse Engineering, Today and Looking Forward

This project develops new foundations, methodologies, algorithms, tools, educational material, and workforce development initiatives for software reverse engineering (henceforth RE) [28]. RE is the process by which we iteratively infer properties of source code. For example, in the course of debugging, we iteratively discover properties of code we wrote, often by falsifying hypotheses. By contrast, in binary reverse engineering, we discover properties of an object-level artifact, such as an ELF binary or a Java bytecode file [101]. The goal of this project is to build new foundations for RE, whose applications span binary reverse engineering to IDE-integrated code analysis used ubiquitously by developers. The central hypothesis of this project is that the tight loop of reverse engineering workflows may be unified via declarative interfaces and implemented via modern, high-performance techniques from databases and logic programming.

RE is an important and challenging problem due to the extremely unstructured nature of the task; for example, in binary RE, it is not even clear what a *procedure* is [30]. Traditional invariants (*e.g.*, calling conventions) may be impossible to predict a-priori (*e.g.*, due to packed code [77]); this severely hinders the application of traditional tools such as symbolic executors [16, 15] to the RE process, which is a shame because such tools are often explicitly motivated by RE. To address this gap, various *lifters* exist, but the heavyweight encodings employed by these tools often obfuscate analysis results and significantly degrade precision [63]. Because of the challenge in applying automatic tools, practitioners often spend significant amounts of time interacting with an interactive disassembly tool; their work often consists of iterative rounds of hypothesis formation and falsification, guided by their specialized domain knowledge [64, 100, 99, 66].

I argue that a proximate source of complexity, human labor, and frustration present in current RE workflows is due to the fact that current RE tools lack a crucial insight: RE is a kind of *reverse programming*, and thus the process of reversing should be guided by code. From this perspective, we can envision decompilation as a program synthesis problem: synthesize a source program P which, when compiled, is contextually-equivalent to object file O. Making this statement rigorous is a challenging task—in fact, it is so challenging, it is an explicit non-goal of this proposal, due to the need for a formal semantics of an arbitrary high-level language (C or C++) and the challenges of full abstraction. However, soundness is often a secondary goal in RE, which often necessitates iterated exploration across a variety of intermediate representations (henceforth IRs): "what if this function is invoked with NULL?" Instead, this project focuses on developing novel methodologies and tools to help expert users rapidly form and iteratively falsify hypotheses, using logic-programming to directly confront the nondeterminism inherent to the RE process.

In this project, I will harmonize my career-long expertise in binary analysis with my recent success in designing state-of-the-art Datalog engines for program analysis. RE occupies an exciting, challenging, and important space: it intertwines expert-guided human reasoning with high-performance semi-automated tools in an adversarial setting. Unfortunately, while current RE tools are extremely complex software artifacts, their designs often fundamentally limit robustness and impede incremental, on-demand reasoning or sharing results across tools. There is an accumulating amount of evidence that points to logic programming serving as a solution [35, 83, 41, 44]. For example, GrammaTech's ddisasm has recently appeared as the first truly viable option for reassembleable disassembly—assembly which can be reassembled to produce an equivalent object file. This is an exciting development, because reassembleable disassembly is a prerequisite for any higher-level analysis or instrumentation task. However, ddisasm is batch-oriented, not integrated into the RE process; in an ideal world, the full power of logic programming would be available to the user as they performed the RE process. This vision will be realized in the form of REVERSI, the first multi-target, multi-language declarative decompiler capable of scaling to production Windows PE x86-64 binaries.

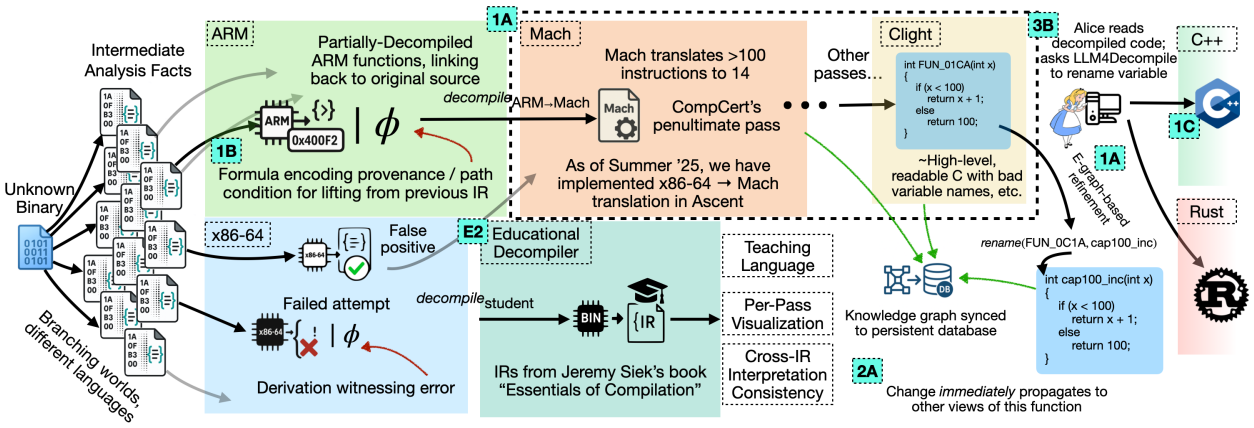


Figure 1: An end-to-end example of the future of reverse engineering if the project is successful; labels indicate (e.g., **1A**) apropos tasks. Reversing a binary (left) yields a forest of candidate decompilations organized into a knowledge graph—higher-level IRs refine low-level IRs, adding additional structure and tracking provenance for efficient querying / transformation.

Enter Alice, a Reverse Engineer Alice[1] works for DeviceKo, a smart device company which purchases OEM IoT devices and integrates their custom firmware for remote sensing and other services. On Monday morning, Alice receives a new Jira ticket titled HVAC-FW-AUDIT-25Q3; attached to the ticket are (a) **hvac_ctrl_v3.2.5.bin**, a 17 MB compressed ARM ELF firmware image that the vendor hosts on their OTA update server (b) **`bootloader.map`**, a linker-map excerpt supplied by the vendor's field-apps engineer, and (c) a two-page marketing PDF touting "mil-grade encryption and password-free local diagnostics." The ticket notes: "*Reverse ASAP—need go/no-go by next Friday.*" Alice downloads the bundle from the internal artifact store, verifies the SHA-256 digest against the hash her boss pasted into Slack, then stages the binary on her analysis workstation. She is given no source code and no compiler metadata—just the raw image.

Alice's first step is the same as most professional reverse engineers: she drops the stripped ARM ELF firmware into Ghidra. After the auto-analysis phase finishes, Alice is confronted with thousands of undefined function names, dubious stack-variable guesses, and a decompiler listing littered with warnings: "could not recover variable type." She spends half her morning manually promoting likely code pointers to functions, correcting mis-identified jump tables, and renaming stack variables so that subsequent data-flow views are readable. But even with these edits, Ghidra's decompiler is still optimistic about calling conventions; when Alice renames or re-types a parameter, downstream analyses are *not* automatically re-run and do not reuse partial results, so she must trigger expensive re-decompilation passes by hand.

Frustrated, Alice pivots to `angr` [102] to explore a suspected auth-bypass. She writes a Python script:

```
proj = angr.Project("hvac_ctrl_v3.2.5.bin", load_options={"auto_load_libs": False})
cfg  = proj.analyses.CFGFast()
state = proj.factory.blank_state(
    addr=cfg.kb.functions.function(name="auth_check").addr)
simgr = proj.factory.simulation_manager(state)
simgr.explore(find=is_bypass_state, avoid=is_crash_state)
```

Listing 1: angr script to perform exploration of `auth_check()` in hvac_ctrl_v3.2.5.bin

[1] Alice's story is not firsthand, but a collection of relevant stories from REs gathered in my collaboration with Votipka et al. [99]

Unfortunately, within minutes the analysis hits state-space explosion: the symbolic execution inherits every over-approximation and missed function boundary from Ghidra's CFG import. Alice tries tighter exploration strategies—concolic execution, pre-constrained inputs, dynamic slicing—but each tweak requires restarting the analysis from scratch: angr is unaware of the fixes Alice performed in Ghirda. Alice's day ends as a fragile patchwork: a partially-corrected Ghidra project, several one-off angr scripts, and an intuition that the auth-bypass is exploitable; but the disintegration between her RE work, Ghidra's analysis, and angr leave her stranded with only ad-hoc notes and screenshots—no reproducible, auditable evidence chain.

A Vision for Declarative Reverse Engineering Alice's predicament is emblematic of contemporary practice: every manual correction she makes is valuable, yet each tool she invokes treats that insight as an isolated, one-shot event. The project's thesis is that...

> *Reverse engineering should be an **incremental, query-driven** workflow in which user edits, automated analyses, and symbolic reasoning are all expressed in the **same declarative vocabulary**—and recomputed **incrementally, on demand** via rich provenance-guided reasoning.*

Figure 1 sketches my vision for the future of reverse engineering. Alice loads her firmware binary into REVERSI, either using a command-line interface or via a plugin in Ghidra. She then specifies (a) the binary file, (b) the compiler command she wishes to invoke (in this case decompcert), and (c) parameters which control state-space exploration; for example, if Alice expects there is no Rust code in the binary, she can de-prioritize Rust analysis. REVERSI then builds an initial set of input tables, possibly using external tools such as ddisasm or libcapstone. While ddisasm produces a fully-disassembled program by default, in our preliminary work we have developed a modified version of ddisasm which exposes all candidate disassemblies which we use as input to REVERSI—performing superset disassembly in REVERSI per se is a stretch goal (ddisasm is several thousand lines of Soufflé Datalog). In Figure 1, we observe the various nondeterministic choices (left) for the disassembly yield various disassemblies into ARM (top) and x86-64 (bottom), of which the bottom includes both failed attempts and also false positives. The blue box **1B** indicates that augmenting decompiled programs with path conditions and SMT formulas is part of task **1B**. A superset of decompiled, provenance-preserving programs moves on to decompilation into higher-level IRs, starting with Mach—CompCert's penultimate IR [60]. Alice can then query, inspect, and transform the lifted code in real time: changes she makes to decisions force the enumeration of subsequent decompilation. The proposed corpus of work addresses Alice's pain points through four tightly-coupled cross-cutting concerns:

- **Unified Logic Core.** Every analysis—control-flow recovery, type reconstruction, calling-convention inference—is encoded once as a set of stratified logic rules. Whether triggered by an automated pass or by Alice renaming a variable, the rule engine derives a consistent new world-view of the binary.
- **Incremental View Maintenance.** Edits propagate in milliseconds, not minutes. A change to one function's prototype invalidates only the affected fragments; downstream facts are refreshed via incremental Datalog evaluation and E-graph rewriting, preserving prior work.
- **Cross-Tool Provenance.** Each derived fact carries a lightweight certificate linking back to its origin—be it a byte, heuristic, or Alice's own assertion. Symbolic states, disassembly edges, and high-level IR nodes are all integrated into the same queryable knowledge database.
- **Composable Interaction.** GUI gestures map to declarative updates; scripts and notebooks issue the same predicates. Alice can pivot fluidly: a Jupyter cell that spawns concolic execution uses the very CFG she just corrected in the graphical front-end, and the results feed directly into later logic queries.

2 Objectives, Goals, and Deliverables

The project's goals are to develop novel methodologies to drive the development of new RE tools, and to expand knowledge of RE principles to undergraduates, graduate students, industry professionals, and the broader public; a representative set of milestones is sketched in Figure 1. With respect to intellectual merit, this project will develop new foundations for building incremental abstract interpreters whose results may be introspected upon to facilitate rich, on-demand user querying and refinement in the context of an IDE or RE tool. To achieve state-of-the-art performance and scalability, I build upon my recent work in designing the highest-performance Datalog engines for static analysis, graph analytics, and provenance. The result is a set of contributions which collectively build a long-term foundation for my career in building interactive program understanding tools built on logic programming engines. In sum, this proposal will target three major research thrusts:

T1 Develop the field of declarative decompilation, a new methodology for rapidly constructing scalable, extensible decompilers using logic-defined rules. This thrust will develop REVERSI, a unifying system which will be incrementally constructed throughout the course of the project.

T2 Support incremental, on-demand reverse engineering, and stream-based processing of binary corpuses.

T3 Unify declarative and generative RE, harmonizing the methods from thrusts **T1** and **T2** with emerging LLM- and language-agent-based decompilation techniques. Support the ongoing ASSEMBLAGE corpus construction project and government-wide LLM4Binaries efforts.

Intellectual Merit This project proposes the first serious attempt to rigorously define and implement the idea of superset decompilation via provenance-grounded refinement: I leverage my current skills in Datalog semantics and implementation to build new languages for performing RE, which rely upon technical formalisms I will develop in task **1A** (my largest and most ambitious task, spanning all years). Additionally, I decompile stripped binaries into SMT formulas, permitting a novel kind of automated exploit generation (**1B**) and leverage my declarative infrastructure to enable multi-language lifting (**1C**). Thrust **T2** tackles items relevant to making RE tools truly interactive and scalable, while thrust **T3** harmonizes modern LLM- and agent-based decompilation techniques with the project's novel declarative decompilation approach.

2.1 Broader Impacts

My education plan (§7) addresses the lack of academically-grounded content on RE, despite strong industry demand [53]. My goal is to expand the U.S. pipeline of RE-informed professionals and promote the "RE mindset" for debugging. Planned activities include a summer school, undergraduate and graduate courses, public-facing blog posts, and beginner-friendly videos on RE and debugging. The research and teaching plans are tightly integrated: thrust **E2** leverages my methodology from **1A** to teach undergraduates (and the broader public of professionals) how to construct correct, testable decompilers in a systematic manner. Additionally, my work on ASSEMBLAGE is the largest-publicly-available dataset of Windows PE binaries for binary analysis; ASSEMBLAGE is used by a wide array of researchers in government, industry, and academia, and will continue to have impact as the highest-quality baseline of provenance-annotated binaries. Last, because the proposed targets foundational aspects of binary RE, the project has the potential to further enhance national security, especially the pipeline of qualified RE professionals.

$$
\begin{aligned}
&\ldots \\
\mathit{MemoryAlias}(x,w) \;&\leftarrow\; \mathit{dereference}(y,x),\\
&\qquad\quad \mathit{ValueAlias}(y,z),\\
&\qquad\quad \mathit{dereference}(z,w).\\
\mathit{ValueAlias}(x,y) \;&\leftarrow\; \mathit{ValueFlow}(z,x),\\
&\qquad\quad \mathit{MemoryAlias}(z,w),\\
&\qquad\quad \mathit{ValueFlow}(w,y).\\
\mathit{ValueFlow}(y,x) \;&\leftarrow\; \mathit{assign}(y,x).\\
\mathit{ValueFlow}(x,x) \;&\leftarrow\; \mathit{assign}(x,y).
\end{aligned}
$$

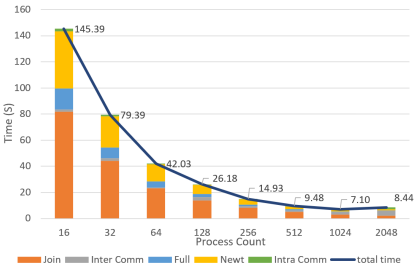


Table 1: GPULog vs. Soufflé

Prog.	Time (s)		Speedup
	GPULog	Soufflé	
Httpd	**1.33**	49.48	**37.2x**
Linux	**0.39**	13.44	**34.5x**
Postgres	**1.27**	57.82	**44.9x**

Figure 2: (left) context-sensitive points-to analysis in Datalog, (center) scaling analysis of the Linux kernel to 1k cores of the Theta supercomputer via SLOG, (right) same analysis in GPULOG (ASPLOS '25), beats optimally-configured Soufflé by 45×.

3 Background, Preparation, and Preliminary Results

I have spent the bulk of my career focused on scaling program analysis, often to large-scale object-level artifacts, including Android (Dalvik), Linux ELF, and Windows PE binaries. Throughout that time, I have observed the frustrating reality of the effort required to build production program analyses which may be readily applied to large software systems. As part of my PhD, I built a recompiler of Dalvik bytecode (for Android), and included traditional control and data-flow analysis for type recovery and rewriting [51]. Since my PhD, I have shifted to building the next generation of program analyses for solving nuanced problems meant to help give us insight into the behavior of code [39, 91, 87, 79], especially for security [70, 88, 100].

The World's Fastest Datalog Engines on Servers, GPUs, Supercomputers, and GPU Clusters For the past several years, *I have led the design of the fastest and most scalable Datalog engines on CPUs, GPUs, and supercomputing clusters [90, 79, 87, 84, 38, 89].* Datalog is a logic programming language which enables solving complex reasoning by writing declarative rules which sketch an acceptable space of solutions; intuitively, Datalog extends SQL with efficient recursive rules. The growing excitement in Datalog—a language which has become the de-facto standard for complex problems in static analysis (the context-sensitive Java analysis DOOP), security (ddisasm), ontological reasoning (the industrial RDFox)—is due to its unique price point, being expressive enough for solving complex problems (*e.g.*, whole-program static analyses, drug discovery, and graph analytics) in a concise, obviously-correct manner while simultaneously exposing massive potential for parallelization. Unfortunately, current-generation CPU-based Datalog solvers reach their best performance around just 8 cores. My team and I have made several distinct orders-of-magnitude improvements in state-of-the-art Datalog engines, publishing key advancements in its expressivity [38, 79, 40] and parallelization (SLOG, VLDB '25) on modern hardware [90, 87]; our most recent engine (VFLOG, AAAI '25), a columnar GPU-based Datalog, beats Soufflé by up to 200× [89]!

As a motivating application demonstrating the power and scalability of Datalog, Figure 2 gives a representative set of rules from a context-sensitive program analysis; the center and right of the figure show two of my systems, SLOG and GPULOG applying this analysis to the Linux kernel (center) and scaling to leadership-class supercomputers and cutting-edge GPUs (the NVIDIA H100). Compared to traditional static analysis systems (*e.g.*, WALA, consisting of >100k lines of code), Datalog enables implementing rich, context-sensitive analysis of production languages in orders-of-magnitude less code (*e.g.*, DOOP handles all of Java with just hundreds of rules). The declarative, logic-defined nature of Datalog is a perfect match for applications which involve logical reasoning, including symbolic AI (*e.g.*, provenance and neurosymbolic AI), binary analysis (e.g., the SOTA Datalog-based disassembler ddisasm), business analytics, and even medical reasoning[37].

My group has developed the most scalable Datalog engines to date, and has applied our engines to

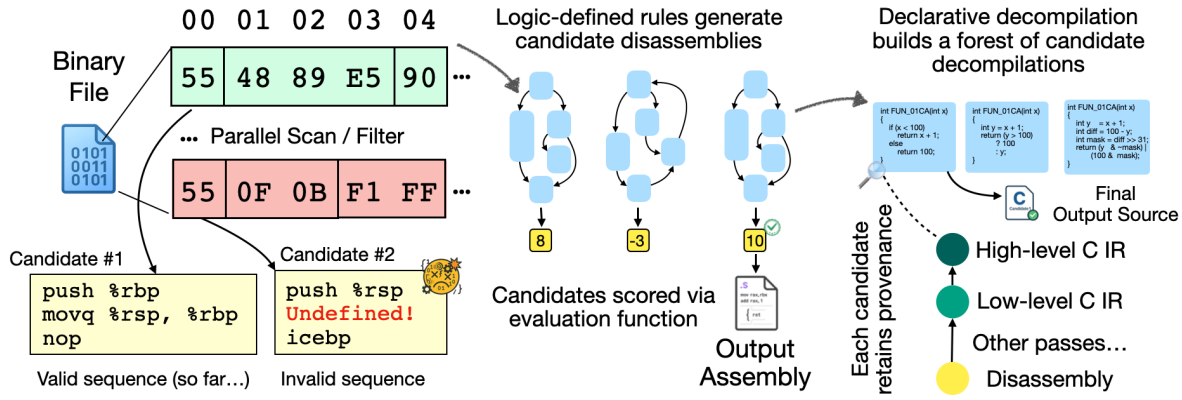


Figure 3: Superset disassembly (left) processes a binary, enumerating a superset of candidate disassemblies, which are ranked to produce a representative disassembly (center). Declarative decompilation (right) builds a forest of candidate decompilations.

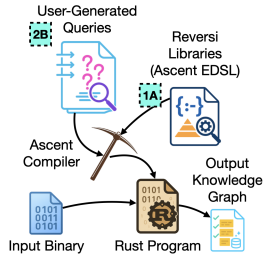
achieve the fastest and most scalable static analyses in history. The center of Figure 2 shows results of SLOG, the first data-parallel Datalog engine built on MPI, which we built using innovations in all-to-all communication [57, 33]. My former PhD student Arash implemented an award-winning CPU-based Datalog engine, Ascent, embedded in Rust via procedural macros. Ascent includes the "Bring Your Own Data Structures" technique, enabling Datalog engines to harmoniously cooperate with user-provided data structures; this work won distinguished paper at OOPSLA '23. Ascent beats competitor systems in both speed and scalability, and was used to implement (collaboratively with Galois) the yapall analysis of LLVM; our experiments of yapall scaled a 1-context-sensitive analysis of httpd to 32 threads of a large server.

4 Research Thrust T1: Declarative Decompilation

Decompilers are extremely complex software systems which automatically translate binary code back into source code—often C or C++—and are a critical component of professional-grade RE tools such as Ghidra, IDA Pro, and BinaryNinja [3, 46, 97]. Compared to compilation, the task of decompilation is inherently more challenging due to fact that the input is an unstructured array of bytes, rather than a structured abstract syntax tree [95]. While this difference may appear rather superficial, it is emblematic of a central issue, which all decompilers must address in some manner: there are *infinitely-many* programs which are contextually-equivalent to the original source program. This ambiguity exists at several levels; first, there are generally infinitely-many programs equivalent to the original source due to, *e.g.*, renamings of variable names, choices of loop construct (e.g., do...while vs. for), or similar ambiguities. However, when dealing with arbitrary stripped binaries there is an even more prescient concern: in absence of assumptions on the compiler version / flags, ABI, calling convention, and similar factors, it is practically *impossible* to decompile the file accurately [75, 44, 50]. Thus, all real tools (Ghidra, IDA Pro, etc.) operate based on best effort, providing the user a UI to selectively edit incorrect decisions (e.g., missed types, variable names) on demand. Unfortunately, because these post-decompilation changes are not properly integrated into the decompilation process per se, their implementation is often ad-hoc and confusing [99, 66].

In this thrust, I will develop the field of declarative decompilation, a systematic methodology for rapidly constructing industrial-grade decompilers in a manner which anticipates subsequent transformation, querying, incremental updating, and high-performance deployment. My proposed approach builds upon state-of-

```
% DP step of weighted interval scheduling
wis_memo(0,0,0).
wis_memo(I,Value,Pred) :-
    I       = IPrev + 1,
    Take    = Take0 + W,
    Value   = @max(Leave,Take),
    Pred    = @max(Leave,Take,IPrev,P).
    wis_memo(IPrev,Leave,_),
    unresolved_interval_order(I,_,_,_,W),
    wis_prior(I,P),
    wis_memo(P,Take0,_),
```




```
// Apply transformation to C_light in "pretty" mode
module my_C_light extends C_light [#:when "pretty"] {
  use known_constant from C_light.analysis;
  rule if_assign_to_ternary {
    // `if (c) v = a; else v = b;` |-> `v = c ? a : b;`
    @rewrite(if_expr, $IfThenElse(C,A,B))
      <-- if_expr = $If(C, $Assign(v,a), $Assign(v,b));
    // `if (c) c0; else c1` |-> `c0` when c definitely false
    @rewrite($If(Guard,Body0,Body1), Body0)
      <-- ?(known_constant Guard !false);
  } }
```

Figure 4: (Left) interval-weighted scheduling in ddisasm; (center) example compilation pipeline; (right) REVERSI rule

the-art methods for disassembly, namely superset disassembly [8, 108, 109, 71]. Superset disassembly directly tackles the complexity of stripped binaries (without entrypoints) by exploring *all possible* entrypoints building a superset of possible worlds, which are recursively explored to either (a) discover more reachable paths or (b) an invalid instruction (or similar obvious error) [8]. Leveraging my history in binary analysis and high-performance Datalog engines, I will extend the ideas of superset disassembly to superset decompilation. Compared to superset disassembly—which transforms relatively flat, first-order data—superset decompilation produces a *forest* of possible tree-structured decompilations, enabling portions of the binary to be lifted to an arbitrarily high level for querying, introspection, and (as a stretch goal) instrumentation.

To ground technical investigations in a concrete deliverable, my students and I will build REVERSI, a multi-architecture, multi-language declarative decompiler framework written in Datalog which will interface with major reverse engineering tools (Ghidra, IDA Pro, etc.). To keep the scope of REVERSI reasonable, it will be built in stages, with the first year focusing on batch-oriented decompilation to primitive C (mirroring CompCert) and subsequent years reaching out to higher-level languages (C++ and Rust), multiple architectures, and more ambitious technical explorations (e.g., E-graphs). Each year, I propose increasingly-ambitious evaluations to compare the project's results to SOTA competitors.

Task 1A: Generalizing Superset Disassembly to Declarative Decompilation, Building REVERSI Declarative decompilation is *an approach to decompilation which iteratively lifts low-level IRs to higher-level IRs via logic-defined rules.* I now describe a preliminary formal framework for declarative decompilation; a full formal treatment is left to the project. Our goal is to generalize superset disassembly, and specifically Datalog disassembly, as embodied in GrammaTech's ddisasm [35, 34]. Figure 3 sketches an end-to-end example: on the left, we illustrate the basic ideas of Datalog disassembly on a stripped binary input. ddisasm's C++ front end linearly decodes every byte via Capstone, exporting instruction, data, and invalid relations to Soufflé. Candidate basic blocks are identified via known root markers (such as push %rbp), and are then iteratively grown via Datalog rules that follow local control flow. ddisasm next performs a variety of program analyses on each candidate block, recursively building blocks by iteratively exploring control-flow edges to discover larger blocks (Candidate #1) and marking known-bad blocks (Candidate #2).

This process leaves us with a superset of candidate disassemblies (center), differing on their choices for entrypoints, etc. To project this set of candidate disassemblies to a single "final" disassembly, ddisasm employs weighted interval scheduling [34]. In Figure 4 (left), we show a relevant snippet of Souffé Datalog code that uses aggregation to build a memo table: wis_memo is a three-arity relation encoding the dynamic-programming recurrence for weighted interval scheduling: for each ordered interval I, it compares the weight of excluding it with the weight of including it (TakeWeight, equal to the interval's own weight plus the optimal value at the last non-overlapping prior interval P). The code stores the maximum of these two in Value, and via the aggregator @max sets Pred to either I - 1 or P, yielding a pointer chain that later

reconstructs the optimal, non-overlapping block set. Duplicating the equality (I = IPrev+1 and IPrev = I – 1) permits Soufflé to index the rule in both join orders, avoiding a materialization pass. Because the @max aggregator is evaluated inside the rule head, the winner is decided without a second pass over ties, and the whole recurrence executes in linear time. In sum, ddisasm performs a variety of recovery passes, including instruction boundary identification, symbolization, control-flow graph construction, interpretational register- and stack-value propagation, jump-table detection, and many others. Candidate blocks are then heuristically scored and fed to a weighted-interval-scheduling solver that selects a maximal, non-overlapping set before projecting the complete CFG, symbols, and metadata to a reassembleable (.s) file.

Proposed Work: From Superset Disassembly to Provenance-Grounded Superset Decompilation

A program is an algebraic structure (typically represented as an abstract syntax tree) written in a language L. Let $\mathcal{L} = \{\text{Assembly}, \text{C}, \text{C++}, \text{Rust}, ...\}$ be a set of languages, ordered by $\sqsubseteq$ such that $L \sqsubseteq L'$ whenever there exists a function $compile_{L \to L'} : L \to L'$; notice that $\sqsubseteq$ is transitive, as we can compose compilers. We generally wish the compilation of L into L' to be *sound* in the sense that we wish $p : L$ to have the same behavior as $compile_{L \to L'}(p) : L'$. There are a variety of formal routes to contextual equivalence—step-indexed logical relations, interaction-based bisimulations, among others—each balancing expressiveness against mechanization effort [2, 32]. In this project, we will instead adopt a straightforward inductive notion of soundness used in CompCert [60]. We require an *interpreter* for each language $L \in \mathcal{L}$, *i.e.*, a function $interp_L : L \times \Sigma \to \Sigma$ where Σ is a type of input/output (*e.g.*, list of strings read from and printed to stdin/out). The correctness of $compile_{L \to L'}$ may then be written as $\forall p \in L, \forall input \in \Sigma.\ interp_{L'}\big(compile_{L \to L'}(p)\big) = interp_L(p)$. A decompiler, $decompile_{L' \to L}$ is a function which transforms lower-level languages into higher-level languages. A decompiler is then correct whenever $\forall p \in L, \forall input \in \Sigma. interp(p) = interp\Big(decompile_{L' \to L}\big(compile_{L \to L'}(p)\big)\Big)$. As with compilers, we can chain decompilers; the correctness of a set of (de)compiler passes is the correctness of their composition [4, 29, 98].

In this task, I will build a Datalog-based superset decompiler from x86-64 code to each intermediate representation in the CompCert certified compiler toolchain [60]. It is explicitly **not** a goal to formally verify the correctness of each decompiler pass; while trustworthy, semantics-preserving decompilation *is* a goal of this project, it would not be prudent to tackle fully-verified decompilation, given that CompCert consists of over 100k lines of Rocq code. Instead, I will use a technique I call *provenance-guided superset decompilation* (PGSD). PGSD makes several key changes compared to traditional decompilers:

- Instead of *rewriting* a low-level input $p : L'$ into a higher-level L, a decompiler returns a *refined* program $p' \mid \phi(p) : L$. p' represents a lifted program which is *witnessed* by $\phi(p)$.
- $\phi(p)$ represents the provenance of the decision to decompile p into p', represented by an element of a semiring [42, 49], permitting arbitrarily-granular provenance tracking in a well-founded manner.
- Instead of mutably evolving a single program, PGSD represents a *superset lattice* of programs, enabling programs $p : L'$ and $p' : L$ from different languages to coexist in $L \sqcup L'$.
- Decompiling a program $p : L'$ into $decompile_{L' \to L}(p) = p' : L$ does not mutably destroy p, but results in $p \sqcup p' \mid \phi(p) : L \sqcup L'$: the monotonic extension of p to include the refinement p' of p.

Decompiler Passes are Collections of Rules In PGSD, compiler passes are written as Datalog-style rules [18]. An example REVERSI module is presented on the right of Figure 4, written in a proposed EDSL which compiles to Ascent Datalog [80, 79]; the proposed EDSL will incorporate the innovations of my preliminary work, particularly Datalog$^{\exists!}$ [40] and high-performance compilation to servers and GPUs. The first

line declares a module (set of transformations) which apply to programs in the C_light language: PGSD tags source trees with metadata, so it is efficient to apply the rewritings only to C_light source-trees. The [:when "pretty"] annotation refers to a runtime parameter; in this case, the "pretty" variable allows the user to control if pretty printing is turned on. The use known_constant line ensures that the results of the known_constant analysis are available; this analysis is exported by C_light and uses lattices (a common Datalog extension) to perform constant propagation. The first rule rewrites conditionals (whose bodies are simple assignments) to ternary statements, using $ to refer to syntax objects of the underlying C_light language, the rewrite syntax extends the knowledge graph (using a Rust trait defining ⊔, not pictured here) and updates the current view (shown in the CLI / Ghidra) to refer to the rewritten variant; the second rule simplifies if statements when the known_constant analysis determines the guard to be !false (a macro expanding to the abstraction of false).

E-Graphs and Semantics-Preserving Transformations Although fully-verified decompilation is out of scope of the project, I chose CompCert explicitly due to the fact that each CompCert IR is equipped with a formal semantics: this allows us to formally prove the equivalence of fragments at every IR level. I will use CompCert to formally verify the correctness of a set of semantics-preserving transformations, which will be available to users of REVERSI. Unfortunately, it is well known that reasoning about equivalence using explicit representations (as in Slog [40]) is prone to complexity-theoretic overhead. To ameliorate complexity-theoretic overhead, I am currently working with my students to incorporate E-graphs into Ascent (following Egglog [112]); I will use this machinery to power equality saturation in REVERSI.‘

Task 1B: Decompilation to Staged Symbolic Execution, Declarative Exploit Generation The rest of the tasks are united by the following theme: along with decompilation into a high-level language, we may decompile the program up to any semantically-meanigful object we wish—so long as it may be given a meaningful interpretation function. *In this task, we will extend* REVERSI *(and its underlying Datalog engine implementation) to incorporate reasoning with first-order satisfiability modulo theories (SMT)*, as in angr [102], KLEE [15], and Formulog [9, 10]. We will extend the EDSL language (used to write REVERSI transformations) with the ability to construct and check the satisfiability of SMT formulas; in the extended design rule bodies will be able to include SMT formulas, which generate speculative candidate paths until checking is demanded (either by a system-defined scheduler or user). The system will be bottom-up, pushing forward symbolic paths (representing partial decompilations) and pruning unrealizable paths via calls to the SMT solver. The combination of Datalog and SMT is not unique per se (*e.g.*, SeaHorn [43], spacer [55], and Formulog [9]), but the novelty of the proposed work lies in the specific ability to build symbolic provenance-grounded candidate decompilations and perform symbolic queries on those paths.

Our goal will be to make significant breakthroughs in automated exploit generation (AEG) [7, 104, 45, 20], a field which has seen perennial excitement, but whose progress is bottlenecked by monolithic symbolic-execution pipelines and hand-tuned heuristics that entangle memory modeling, constraint solving, and payload synthesis. However, recent success in AEG for JavaScript (PLDI ’25 [65]) has achieved significant improvements using a Datalog-based approach. Our approach treats each stage of vulnerability search as a first-class, queryable artifact: symbolic paths, memory effects, and objects of interest (*e.g.*, gadgets) materialize as Datalog facts with explicit provenance, allowing inspection, pruning, and user guidance.

Task 1C: Multi-Language, Multi-Target Decompilation The vast bulk of effort in decompilation has focused on decompilation to C, but there is an increasing interest in RE for higher-level languages, such as C++ and Rust [46, 56, 22, 107, 36, 31]. Unfortunately, although the advent of infrastructure such as LLVM and MLIR [6] has ensured that compilers are being written at the highest rate ever, decompiler construction

is still overwhelmingly complex [17]. *I argue that a proximate reason for the relatively slow progress in decompilation for higher-level languages is due to the high barrier to entry of extending an industrial-strength decompiler.* In this task, I will extend REVERSI to support multi-language, multi-target decompilation. Targets will include x86-64 and ARM (at minimum), starting with the ARM-based infrastructure currently present in ddisasm. We will initially work to replicate the work of OOAnalyzer, a Prolog-based prototype tool to identify C++ class hierarchies [83]. OOAnalyzer is part of the Pharos binary analysis framework [19], with which we will directly compare—our goal is to see a significant speedup versus OOAnalyzer, which is known to take days on relatively small binaries due to its relatively ad-hoc Prolog implementation [96]. Based on my experience studying OOAnalyzer, I have observed that a significant portion of its logic is subsumed by analysis facts which exist as part of ddisasm's disassembly. To tackle Rust, I will begin by replicating the work of RustBound [31]. Completely-sound decompilation of C++ and Rust is beyond the scope of the project, and unlike for C (where we decompile to CompCert's IRs), we will not decompile to any formally-modeled IR. Instead, our exploration of higher-level languages will be validated by test suites, fuzzing, and symbolic execution (leveraging task **1B** to find bugs).

5 Research Thrust T2: Interactive, and Incremental, and Scalable RE

Thrust **T1** proposes a new methodology for reverse engineering, a risky but ambitious goal that I hope will pay dividends to RE long-term by unlocking the potential of declarative decompilation. However, the goal of thrust **T2** is to study the foundations of interactive, incremental, and scalable RE in tandem with my work on REVERSI as a software artifact. In my collaboration with Votipka et al. (USENIX '20 [100]), one-on-one interviews with reverse engineers revealed that professional reverse engineering requires significant, hypothesis-driven exploration. We often saw RE professionals leveraging static analysis tools (e.g., angr) to augment their workloads. However, we found that the lack of incremental operation and poor tool interoperability caused significant amounts of frustration and wasted time among practitioners. In this thrust, I work to support key efforts in making fast, interactive binary analysis a reality.

Initial Progress: Declarative, Demand-Driven Reverse Engineering, D^{3}RE One of the main usages for RE tools is to perform beacon identification, using the tool's API to write a query which identifies an interesting part of the code. Search results for "Ghidra scripting," (and similar) along with popular GitHub repositories for Ghidra scripts, show that many users are writing such scripts. My hypothesis was that these commonly-used analysis scripts face significant performance issues because they often implement querying over Ghidra's database in an ad-hoc fashion, unable to take advantage of join planning and indexing. To evaluate this hypothesis, my students and I built D^{3}RE, a prototype Ghidra extension which enables users to perform querying on the binary and calculate query results using Soufflé Datalog, rather than Ghidra's Python-based scripting engine. In D^{3}RE, a user uses Datalog rules to build a query and identify parts of a binary of interest for further analysis. To run the analysis, D^{3}RE calls out to Soufflé and executes the relevant Datalog analysis script, which writes a set of output files to CSV; our Ghidra extension then loads these CSVs and renders them to overlay properties of interest on the binary.

Task 2A: Incremental Context-Sensitive Program Analysis (Collaboration with Sandia) Incremental program analysis is receiving increasing attention due the popularity of IDE-integrated code analysis tools such as language-server protocol (LSP) [25] rust-analyzer [23], and other tree-sitter–based [24] code-intelligence frameworks, all of which demand millisecond-level recomputation to provide continuous, interactive feedback during development. While serious initial efforts exist—including *IncA* [93, 74], CodeQL [92], and others [114]—*there is still no general-purpose approach to incremental, context-sensitive*

program analysis. In this thrust, I will collaborate with Sandia National labs, specifically Denis Bueno (and by extension his collaborator Arlen Cox at IDA/CCS) to study foundational approaches to context-sensitive program analysis applied to large-scale RE workloads in the CTADL static analysis tool [14]. CTADL is a high-performance, industrial-grade static analysis engine written in Soufflé offering multi-language support (including Ghidra's P-Code and Java) and a hybrid approach to context sensitivity.

We will perform fundamental research in incremental program analysis with the long-term goal of creating a fully-incremental version of CTADL for integration into Ghidra or VSCode. Our basic approach will be to take inspiration from differential dataflow and DBSP, two abstractions for writing incremental computations over streams [13, 67, 72]. Unfortunately, although a rich body of research prototypes exist, production-ready incremental parsers that expose deltas—rather than a fully-materialized AST—remain rare; even tree-sitter, the de-facto incremental work-horse, returns an updated full syntax tree, which leads downstream tools such as rust-analyzer to fall back on re-parsing the entire file after every keystroke. Initially, we will design an abstraction for incremental AST changes, which we will integrate into a fully-incremental parser as a stretch goal. Our primary focus will be on using stream-based abstractions such as $\mathbb{Z}$-sets (which track *changes* to a relation rather than simply additions) to build fast, scalable, incremental program analyses. A key research challenge will be how to retain properties such as productivity and termination in streaming-based contexts like DBSP, which eschew termination by design.

Task 2B: Accelerating Inter-procedural Binary Analysis on GPUs Binary analysis workloads are among the most important, large-scale, and challenging program analysis workloads currently available, with large analysis engines enumerating millions to billions of analysis points [35, 86, 14]. In my background research, my students and I have unlocked massive performance gains (up to 200× versus optimally-configured multi-core Soufflé) in GPU Datalog. *In this task, I apply my ongoing work in SOTA Datalog engine design to design the most scalable binary analysis engine to date.*

While there have been some initial efforts in using GPUs for (inter-procedural) program analysis [68, 110, 111], there has been relatively little application of GPUs to binary analysis. I will do this by extending our current GPU Datalog prototypes (including GDLOG and FVLOG)—which are currently C++-based and do not support a full Datalog language—to support the practical features necessary to scale to practical binary analysis tasks. I will tackle this thrust incrementally: in the first year, I will focus on translating the small core analysis of CTADL (roughly 200 lines of Datalog code). In subsequent years, I will continue to expand the ambition and scope of the research tasks, integrating the GPU-based features into REVERSI. Key research questions will focus on how to effectively leverage the GPU to meaningfully augment the RE task given the high host-device bottleneck and limited VRAM of consumer-grade GPUs.

Task 2C (Stretch Goal): Streaming Binary Corpus Analysis As a stretch goal, we will develop the field of *streaming binary corpus analysis*. In binary corpus analysis, our goal will be to analyze large collections of streaming binary corpuses. Specifically, our goal will be to rewrite ASSEMBLAGE (described in the next section) so that its architecture may make use of the incremental program analysis products developed in tasks **2A** and **2B**. ASSEMBLAGE's is currently designed as a task-parallel distributed system implemented in Python, with workers communicating to a central coordinator node via RabbitMQ. While ASSEMBLAGE's design is acceptable for the task-parallel job of building binary corpuses, this task will extract data parallelism from large binary corpus analysis problems, while enabling incremental updating.

6 Research Thrust T3: Unifying Declarative and Generative RE

Large language models, language agents, and agentic AI have taken the world by storm, with everyone from novices to experts reaching to AI services for software understanding generally [59, 47, 61, 78, 12], and reverse engineering specifically [58, 52, 21, 94, 106]. Given the success of modern LLMs, no project on symbolic program analysis would be complete without a thorough comparison against an LLM-based competitor. From 2020–2025, I led the development of the largest publicly-available corpus (nearly 1M) of benign Windows PE binaries: ASSEMBLAGE (NeurIPS '24 D&B [62, 82]). The goal of the ASSEMBLAGE project is to provide a massive amount of richly-labeled training data for the next generation of AI [62, 82]. In that project, I developed significant expertise and presence in the LLMs for binaries space: for two years in a row, the NSA's (unclassified) two-week-long LLMs4Binaries workshop have featured my team's datasets, and numerous academic and government labs use our datasets as a crucial part of their AI workloads. While that project was funded by the US Laboratory for Physical Sciences, that project is winding down and projected to end in early 2026. *In this task, I will continue the* ASSEMBLAGE *project, harmonize generative AI with declarative RE and* REVERSI*, and develop benchmarks comparing symbolic and generative RE.*

Task 3A: Maintaining, Enriching, and Supporting ASSEMBLAGE LLMs are extremely hungry for data, but unfortunately there is a severe lack of high-quality publicly-available benign Windows PE executable files[2]. Since 2021, ASSEMBLAGE has produced millions of benign Windows PE and Linux ELF binaries and has become the de-facto standard in open-source corpuses with provenance-annotated metadata [62]. ASSEMBLAGE is a distributed system that reproducibly builds binary corpuses, scaling across VMs to support different platforms / toolchains. We have two flagship datasets, both of which are actively used by numerous government and academic labs: (a) a large >1.5M dataset of benign Windows PE and Linux binaries and (b) a smaller, hand-curated custom of high-quality repositories with "deep histories:" diff-annotated metadata that details repository changes across versions.

In this task, I will ensure that we continue to operate, maintain, and enhance ASSEMBLAGE. While this task does not involve new algorithmic developments, my team has been successful in publishing influential papers on ASSEMBLAGE in "D&B" tracks for the past few years [62, 82], with several upcoming submissions in the pipeline. We will continue to develop new datasets, continually identifying new, exciting data we can capture and make available to the binary analysis community. Additionally, as part of this effort, I plan to take over hosting (yearly, as interest dictates) the "LLMs4Binaries" workshop—either affiliated with a major conference or a self-hosted event at Syracuse.

Task 3B: Unifying Agentic and Declarative Decompilation Language agents have rapidly gained popularity as a way of leveraging modern LLMs in the context of more traditional symbolic processes [105, 76]. Specific to decompilation, there have been a plethora of boutique models and neural-integrated approaches targeting decompilation [94, 58, 48]. However, vanilla LLM-based approaches did not prove to generate sufficiently-correct code as to be usable—instead, recent efforts have trended towards agentic decompilation, wherein a language model is in a loop with a traditional symbolic program (in this case REVERSI) [103, 113, 5]. In this task, I will extend REVERSI with support for features such as Model Context Protocol (MCP) [73], and will compare REVERSI versus SOTA agent-based decompilers.

Task 3C: Objective Comparisons of LLMs, Agents, and Symbolic Decompilation While modern innovations in neural decompilation are moving at an unprecedented rate, there are unfortunately too few

[2]Malicious executables are not in short supply, as malware authors do not assert their copyright.

Figure 5: RE Summer School at Minnowbrook—Example Curriculum

Class	Lecture	Lab
1	x86-64 Assembly, binary layout, Ghidra intro	Stack smashing (source provided)
2	The stack, calling conventions, and ABI intro	Windows PE binary cracking
3	Memory protections: ASLR, W⊕X, CFI	Return to libc attacks
4	The heap, dynamic allocation, virtual method calls	Reversing C++ binaries
5	Return-oriented programming, data-oriented programming	ROP Chain Exploitation

Figure 6: YouTube Lecture Screencap

papers that perform objective comparisons between neural, agent-based, and traditional (symbolic) decompilation. *In this task, I will develop a set of objective benchmarks to permit apples-to-apples comparisons for decompilation tasks.* Tasks will vary in nature, but will plan to stress program understanding tasks, including indirect control-flow (*e.g.*, ability to resolve icalls). In performing this task, I will leverage data from ASSEMBLAGE (**3A**), and will publish my results in a datasets and benchmarks track of a suitable venue.

7 Education Plan

Education is deeply important to me personally, and I have repeatedly demonstrated a commitment to teaching and mentorship throughout my career. Before my tenure-track position at Syracuse, I was a visiting assistant professor (teaching postdoc) at Haverford College, where I practiced my teaching intentionally and received targeted feedback from professors who had made teaching the cornerstone of their careers. In my time at Syracuse, I have continued innovating in my teaching, developing open-source course curriculum and publicly-available YouTube videos which have been received nearly 100k views in aggregate; I routinely receive emails from students across the world telling me they study using my course videos.

I have noticed a general trend: students lack debugging skills and self-honesty about bugs in their code. Throughout my career, I have focused on teaching students the principles of reverse engineering their own code, teaching them to rapidly form and falsify hypotheses, debugging and validating their code to produce high-quality, bug-free software. My principles for teaching debugging mirror my strategy in teaching software exploitation (which I have done multiple times, at both the graduate and undergraduate level): think adversarially, be open to being wrong, and build your mental model by iterated experimentation; observational studies of professional reverse engineers shows that they use these same skills in their work [100, 64]. In the context of this project, my education plan will focus on four groups:

- US-based undergraduates looking to build their skills in reverse engineering, with a focus on building a diverse pipeline of students intending to leverage RE as a part of their future careers.
- MS and PhD students at Syracuse University (and other universities by proxy of open-source content), offering a graduate-level seminar, entitled "Principles of Reverse Engineering."
- Industry professionals, by collaborating with professional analysts and publishing a blog, "Declarative Reverse Engineering," communicating research from the project in a practically-useful way.
- The broader public (HS students, new programmers, etc...) by providing a new YouTube-based course, "Reverse Engineering Basics," teaching CS principles in an accessible but problem-focused setting.

Thrust E1: Reverse Engineering Summer School at Minnowbrook (Undergraduate-Focused) There is a crucial lack of educational investment by US universities in RE, at every level [27]. Instead, professional

development in RE is often through expensive industry-sponsored courses, selling for up to $8.5k for a six-day course [81]. These courses are very popular, with security firm Kaspersky reports that RE is the most in-demand skill by IT professionals among their clientele in 2022 [53]. While it is perhaps unsurprising that there are few dedicated RE courses, this lack of adequate undergraduate-focused curriculum represents a failure to capitalize on the intellectual potential of RE to help students integrate their knowledge of the modern software stack. As part of this project, **I will develop and run an "RE summer school" for US-based undergraduates.** This summer school fills the gap between the "how" and the "why" of RE: we will teach students hands-on RE skills (Ghidra, ROP) in the context of rigorous, CS-focused presentation.

The summer school will include 12–20 students, and will be held at Syracuse University's Minnowbrook Conference Center[3]. Recommended course prerequisites include an undergraduate systems class, along with discrete math. I will oversee the admissions process, and will aim to admit US-based students who meet the prerequisites (as judged by a short programming-focused quiz) and claim to be interested in exploring a career path in security or RE. A sample breakdown of the curriculum is shown in Table 5: Each day will include roughly four hours of lectures, covering a breadth of topics in systems, language runtimes, and security. These lectures will teach core computing concepts at a fast pace to fill the voids and teach the cross-cutting concerns necessary to deeply understand binary RE. In the afternoons, students will work together in groups to solve RE challenges; we will also have several social events. I will recruit qualified instructors from my personal network to prepare novel content, and will prepare dedicated time for the instructors to run a "mini Dagstuhl" among themselves. Participant fees of $5k per year are budgeted for the RE summer school. These participant fees will partially cover the cost to students; I am to cap total out-of-pocket cost (to students) at $1k. At the end of the summer school, students will receive a certificate of participation.

Thrust E2: "Essentials of Decompilation: an Incremental Approach" There is currently *no* formal course material teaching decompilation, aside from scant course notes. Given the complexity and relevance of decompilation in understanding systems programming, lack of rigorous educational material on decompilation represents a strategic weakness to training the next generation of RE professionals. *In this thrust, I will construct a four-week module teaching the construction of decompilers targeting advanced undergraduates and graduate students.* My course will follow Jeremy Siek's "Essentials of Compilation," tasking students with reversing increasingly-complex languages, building a decompiler in an incremental fashion.

Thrust E3: Graduate Seminar in "Decompiler Design" Although several universities offer a graduate-level RE course [1], there is a severe lack of publicly-available material on RE at the graduate level. I will create publicly-available material for a new graduate seminar, "Decompiler Design," offered at the MS and PhD level at Syracuse University. Compared to other security-focused courses, my course will teach RE from the perspective of program analysis, with an emphasis on its operationalization using logic-programming-based tools. The course will include four projects, covering tools such as Ghidra, Soufflé, BAP [11], and Angr [102]. To my knowledge, this will be the first graduate course focused on decompilation.

Building the pipeline of RE professionals: "Binary RE Meetup." RE is a niche field, which often experiences severe job shortages [26, 53]. To facilitate networking in this space, I will host a "Binary RE Meetup" on Zoom. The meetup will consist of three things: (a) RE demos and hands-on tutorials, (b) longer, invited talks from RE experts, and (c) short (unpaid) advertisements from companies looking to hire REs. I will build upon my professional connections in industry, government labs, and government agencies to recruit. My collaborator Daniel Votipka will help advertise to his network of REs used in his studies.

[3] In June 2025, I organized the "Minnowbrook Logic Programming Seminar" [69]

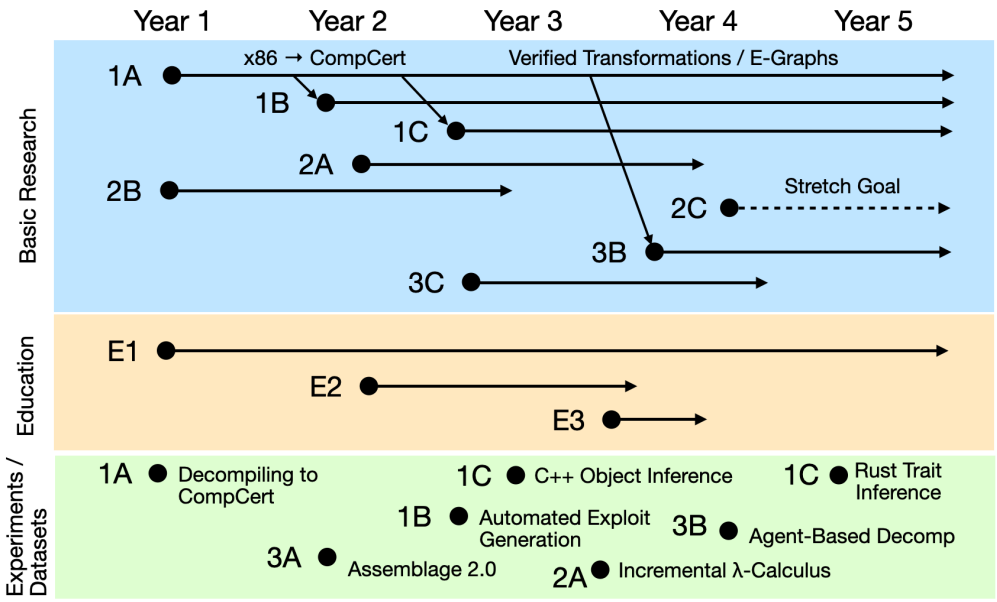


Figure 7: Rough timeline for the proposed work.

The Broader Public: "Basics of RE" YouTube Lectures. I will develop a well-produced three-hour video series teaching the basics of RE. My videos will be focused at the introductory level, and will use Jupyter notebooks [54]. Compared to all videos on RE, my material will not be primarily focused on hacking or exploitation: instead, I will teach low-level computing concepts (e.g., assembly, the stack, and function calls) in the context of low-level code. Each lecture will driven entirely by an end-to-end demo, and will develop a Jupyter notebook. The purpose of these videos is to give computing novices a deep understanding of the cross-cutting concerns that govern the modern software and hardware stack.

8 Results from Prior NSF Support

My most closely related NSF award is (CCF-#1234223) is "Collaborative Research: PPoSS: Large: A Full-stack Approach to Declarative Analytics at Scale." That award has produced papers so far at AAAI '25 [89], VLDB '25 [40], and ICS '25 [85]. The PPoSS award has funded my recent work on GPU-based Datalog engines. That project includes scaling declarative disassembly as a task (an explicit non-goal of this proposal—we leverage ddisasm), but declarative decompilation is explicitly out of scope. This project will leverage open-source artifacts such as engines and libraries produced by the PPoSS project in pursuit of the technical thrusts **T1**–**T3**. Compared to PPoSS, this project focuses specifically on the foundations of RE.

9 Timeline, Experiments, and Deliverables

Figure 7 visualizes a detailed timeline of research (top) and educational thrusts (center), along with listing a subset of proposed experiments and datasets produced by the project (bottom). The project's central task is **1A**, which runs the duration of the project. Tasks in thrust **T2** are orthogonal of **T1** by design, and run on an independent timeline. Thrust **T3** focuses on evaluation, datasets, and integration of our methods into the broader AI-based decompilation ecosystem. In the first year of REVERSI, we will focus on decompilation to CompCert's IRs, subsequently branching to work on verified transformation and symbolic execution. We are tentatively planning to run the summer RE workshop each year, assuming sufficient interest / funding.

References

[1] Reverse engineering and vulnerability analysis — 695.744. `https://github.com/GrammaTech/cgc-cbs/tree/master/cqe-challenges/CROMU_00038`. Accessed: 2020-01-10.

[2] Beniamino Accattoli, Adrienne Lancelot, Giulio Manzonetto, and Gabriele Vanoni. Interaction equivalence. *Proceedings of the ACM on Programming Languages*, 9(POPL):1627–1656, January 2025.

[3] US National Security Agency. Ghidra. https://ghidra-sre.org/, 2022.

[4] Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. *A Symmetric Approach to Compilation and Decompilation*, page 296–331. Springer-Verlag, Berlin, Heidelberg, 2002.

[5] Axelle Apvrille and Daniel Nakov. Malware analysis assisted by ai with r2ai. *arXiv preprint arXiv:2504.07574*, 2025.

[6] The LLVM authors. Multi-level ir compiler framework. `https://mlir.llvm.org/`. Accessed: July 9, 2024.

[7] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 57(2):74–84, feb 2014.

[8] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Network and Distributed System Security Symposium*, 2018.

[9] Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for smt-based static analysis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[10] Aaron Bembenek, Michael Greenberg, and Stephen Chong. Making formulog fast: An argument for unconventional datalog evaluation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1219–1248, 2024.

[11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 463–469. Springer, 2011.

[12] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712*, 2023.

[13] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. Dbsp: Automatic incremental view maintenance for rich query languages. *arXiv preprint arXiv:2203.16684*, 2022.

[14] Denis Bueno and CTADL Team. CTADL: Compositional taint analysis in datalog, May 2025. Release v0.11.2.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[16] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, volume 10, pages 1180405–1180445. Citeseer, 2006.

[17] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 491–502, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[19] CERT Division, Software Engineering Institute, Carnegie Mellon University. Pharos Binary Analysis Framework. `https://github.com/cmu-sei/pharos`, 2017. Accessed 01 Jul 2025.

[20] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[21] Guoqiang Chen, Huiqi Sun, Daguang Liu, Zhiqi Wang, Qiang Wang, Bin Yin, Lu Liu, and Lingyun Ying. ReCopilot: Reverse engineering copilot in binary analysis. *arXiv preprint arXiv:2505.16366*, 2025.

[22] Cristina Cifuentes and Keith Gough. Decompilation of binary programs. *Software—Practice Experience*, 25(7):811–829, 1995.

[23] Ferrous Systems contributors. Rust analyzer. `https://rust-analyzer.github.io/`, 2024. Accessed: July 1, 2024.

[24] Tree Sitter Contributors. The tree-sitter incremental parser generator. `https://tree-sitter.github.io/tree-sitter/`. Accessed: July 12, 2024.

[25] Microsoft Corporation. Language server protocol (lsp) specification, version 3.17. `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/`, 2023. Accessed 10 Jul 2025.

[26] Jennifer Cowley. Job Analysis Results for Malicious-Code Reverse Engineers: A Case Study. 6 2018.

[27] William Crumpler and James A Lewis. *Cybersecurity Workforce Gap*. JSTOR, 2022.

[28] Eldad Eilam. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[29] Daniel Engel, Freek Verbeek, and Binoy Ravindran. BIRD: A binary intermediate representation for formally verified decompilation of x86-64 binaries. In *Proceedings of the 17th International Conference on Tests and Proofs (TAP 2023)*, Lecture Notes in Computer Science, pages 3–20. Springer, 2023.

[30] Daniel Engel, Freek Verbeek, and Binoy Ravindran. On the decidability of disassembling binaries. In Wei-Ngan Chin and Zhiwu Xu, editors, *Theoretical Aspects of Software Engineering*, pages 127–145, Cham, 2024. Springer Nature Switzerland.

[31] Ryan Evans, William Hawkins, and Boyang Wang. Rustbound: Function boundary detection over rust stripped binaries. In Xiali Hei, Luis Garcia, Taegyu Kim, and Kyungtae Kim, editors, *Security and Privacy in Cyber-Physical Systems and Smart Vehicles*, pages 237–256, Cham, 2025. Springer Nature Switzerland.

[32] Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. Do you even lift? strengthening compiler security guarantees against spectre attacks. *Proceedings of the ACM on Programming Languages*, 9(POPL):Article 31, 1–30, January 2025.

[33] 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.

[34] Antonio Flores-Montoya, Junghee Lim, Adam Seitz, Akshay Sood, Edward Raff, and James Holt. Disassembly as weighted interval scheduling with learned weights, 2025.

[35] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[36] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.

[37] Aleksandra Foksinska, Camerron M Crowder, Andrew B Crouse, Jeff Henrikson, William E Byrd, Gregory Rosenblatt, Michael J Patton, Kaiwen He, Thi K Tran-Nguyen, Marissa Zheng, and et al. The precision medicine process for treating rare disease using the artificial intelligence tool medikanren. *Frontiers in Artificial Intelligence*, 5, Sep 2022.

[38] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 23–35, 2021.

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

[40] Thomas Gilray, Arash Sahebolamri, Yihao Sun, Sowmith Kunapaneni, Sidharth Kumar, and Kristopher Micinski. Datalog with first-class facts. *Proc. VLDB Endow.*, 18(3):651–665, November 2024.

[41] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019.

[42] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, page 31–40, New York, NY, USA, 2007. Association for Computing Machinery.

[43] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[44] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1667–1680, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 763–779, USA, 2018. USENIX Association.

[46] Hex-Rays. Ida pro. https://hex-rays.com/ida-pro/, 2022.

[47] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024.

[48] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, volume 267622140, 2024.

[49] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216, 2011.

[50] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: a machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 20–30, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Jinseong Jeon, Kristopher K. 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 *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, page 3–14, New York, NY, USA, 2012. Association for Computing Machinery.

[52] Linxi Jiang, Xin Jin, and Zhiqiang Lin. Beyond classification: Inferring function names in stripped binaries via domain adapted llms. In *Network and Distributed System Security Symposium (NDSS)*, 2025.

[53] Inc. Kaspersky. Reverse engineering is most on-demand skill among infosec specialists in 2022. `https://www.kaspersky.com/about/press-releases/2022_reverse-engineering-is-most-on-demand-skill-among-infosec-specialists-in-2022`, 2024. Accessed: July 3, 2024.

[54] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pages 87–90. IOS press, 2016.

[55] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV*

2014), volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014. Introduces the SPACER Constrained Horn-clause engine now shipped with Z3.

[56] Kamil Kratkiewicz, Petr Zemek, and Michal Rolínek. Retdec: An open-source machine-code decompiler based on llvm. In *IEEE Working Conference on Reverse Engineering (WCRE)*, pages 353–362, 2017.

[57] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In *International Conference on High Performance Computing*, pages 288–308. Springer, 2020.

[58] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[59] Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R. Lyu. A unified debugging approach via LLM-based multi-agent synergy. *arXiv preprint arXiv:2404.17153*, 2024.

[60] Xavier Leroy. Formal verification of a realistic compiler. In *Proceedings of POPL '06*. ACM, 2006.

[61] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, et al. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[62] Chang Liu, Rebecca Saul, Yihao Sun, Edward Raff, Maya Fuchs, Townsend Southard Pantano, James Holt, and Kristopher Micinski. Assemblage: Automatic binary dataset construction for machine learning. *Advances in Neural Information Processing Systems (NeurIPS '24)*, 37:58698–58715, 2024.

[63] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1100–1119. IEEE, 2022.

[64] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.

[65] Filipe Marques, Mafalda Ferreira, André Nascimento, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. Automated exploit generation for node.js packages. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[66] James Mattei, Madeline McLaughlin, Samantha Katcher, and Daniel Votipka. A qualitative evaluation of reverse engineering tool usability. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 619–631, New York, NY, USA, 2022. Association for Computing Machinery.

[67] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[68] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 47(8):107–116, 2012.

[69] Kristopher Micinski. Minnowbrook logic programming seminar (supercut w/ extras). https://www.youtube.com/watch?v=3ec9VfMUVa8, 2025. YouTube video.

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

[71] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic Disassembly . In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198, 2019.

[72] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[73] OpenAI. Gpt-4o system card, 2024. https://openai.com/research/gpt-4o-system-card.

[74] André Pacak, Sebastian Erdweg, and Tamás Szabó. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[75] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP)*, pages 833–851. IEEE, 2021.

[76] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[77] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1), jul 2013.

[78] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[79] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. Bring your own data structures to datalog. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[80] Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. Seamless deductive inference via macros. CC 2022, page 77–88, New York, NY, USA, 2022. Association for Computing Machinery.

[81] Inc. SANS. For610: Reverse-engineering malware: Malware analysis tools and techniques. https://www.sans.org/cyber-security-courses/reverse-engineering-malware-malware-analysis-tools-techniques/, 2024. Accessed: July 3, 2024.

[82] Rebecca Saul, Chang Liu, Noah Fleischmann, Richard Zak, Kristopher Micinski, Edward Raff, and James Holt. Is function similarity over-engineered? building a benchmark. *Advances in Neural Information Processing Systems (NeurIPS '24)*, 37:21636–21655, 2024.

[83] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 426–441, New York, NY, USA, 2018. Association for Computing Machinery.

[84] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. Towards Iterative Relational Algebra on the GPU. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1009–1016, 2023.

[85] Ahmedur Rahman Shovon, Yihao Sun, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. Multi-node multi-gpu datalog. In *Proceedings of the 2025 ACM International Conference on Supercomputing (ICS '25)*. ACM, 2025.

[86] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 245–251, Berlin, Heidelberg, 2011. Springer-Verlag.

[87] Y. Sun, S. Kumar, T. Gilray, and K. Micinski. Communication-avoiding recursive aggregation. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 197–208, Los Alamitos, CA, USA, nov 2023. IEEE Computer Society.

[88] Yihao Sun, Jeffrey Ching, and Kristopher K. Micinski. In *NDSS Workshop on Binary Analysis Research (BAR '21)*, 2021.

[89] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. Column-oriented datalog on the gpu. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(14):15177–15185, Apr. 2025.

[90] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Optimizing datalog for the gpu. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 762–776, New York, NY, USA, 2025. Association for Computing Machinery.

[91] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. Modern datalog on the gpu, 2024.

[92] Tamás Szabó. Incrementalizing production codeql analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1716–1726, New York, NY, USA, 2023. Association for Computing Machinery.

[93] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1–15, New York, NY, USA, 2021. Association for Computing Machinery.

[94] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

[95] Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Zuchen Gao, Siyi Wang, Qi Luo, Jing Li, and Yuqun Zhang. Decompile-bench: Million-scale binary-source function pairs for real-world binary decompilation, 2025.

[96] u/TheRealIronWolf. Pharos\OOAnalyzer – How long is it supposed to run? `https://www.reddit.com/r/ghidra/comments/ldaq1m/pharosooanalyzer_how_long_is_it_supposed_to_run/`, February 2021. Discussion thread on r/ghidra, posted 5 Feb 2021. Accessed 22 Jul 2025.

[97] Vector35. Binary ninja. `https://binary.ninja/`. Accessed: July 15, 2024.

[98] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 934–949, New York, NY, USA, 2022. Association for Computing Machinery.

[99] Daniel Votipka, Mary Nicole Punzalan, Seth M. Rabin, Yla Tausczik, and Michelle L. Mazurek. An investigation of online reverse engineering community discussions in the context of ghidra. In *2021 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 1–20, 2021.

[100] 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 Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[101] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391. IEEE, 2018.

[102] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[103] Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1841–1864, 2025.

[104] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium*, pages 781–797, Baltimore, MD, USA, 2018.

[105] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[106] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[107] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.

[108] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and Xiangyu Zhang. D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2391–2408, 2023.

[109] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2709–2725, Boston, MA, August 2022. USENIX Association.

[110] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Yao. Gpu-based static data-flow analysis for fast and scalable android app vetting. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 274–284, 2020.

[111] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Yao. Gpu-based static data-flow analysis for fast and scalable android app vetting. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 274–284, 2020.

[112] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages*, 7(PLDI):468–492, 2023.

[113] Muqi Zou, Hongyu Cai, Hongwei Wu, Zion Leonahenahe Basque, Arslan Khan, Berkay Celik, Jing Tian, Antonio Bianchi, Ruoyu Wang, and Dongyan Xu. D-lift: Improving llm-based decompiler backend via code quality-driven fine-tuning. *arXiv preprint arXiv:2506.10125*, 2025.

[114] Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):424–448, 2022.

Facilities, Equipment and Other Resources

Facilities Facilities for this project will include only office and computer lab space needed to conduct the research. This includes faculty office and graduate student office space furnished to faculty and graduate students at Syracuse University. These resources are provided by Syracuse University for the PI and graduate student to use throughout the duration of the proposed research.

Equipment This project will make use of Syracuse University's research computing virtual cloud and high throughput machines. Research computing at Syracuse University is a collaborative effort between the campus research community and technology groups from across campus. The technology groups include both distributed information technology teams as well as support from Information Technology Services (ITS) which provides research computing infrastructure resources. This project will use the new (as of 2020) Zest cluster to enable the demanding high- performance needs of the proposed work. Zest is a High Performance Computing (HPC) cluster for campus researchers which supports tying together multiple compute nodes at SU's datacenters for research work that cannot be split into smaller components or fit within a single machine. To facilitate this, Zest compute elements are interconnected with low-latency networking that pass information between nodes many times faster than traditional network technologies. Zest is made up of over 1000 cores and 6TB of memory across 31 interconnected nodes. Additionally, the SUrge is a cluster of over 300 GPUs, including A100s. Last, Dr. Micinski used his startup funds to purchase two dedicated servers for the lab which will be utilized for testing and evaluation: one has a 6000 Ada GPU and an AMD Threadripper, and the other (an AMD EPYC server) has 512GB of RAM with 128 threads.

Other Resources There are no other relevant resources to mention.