Research Statement

Brent Pappas
University of Central Florida, Department of Computer Science
brent.pappas@ucf.edu | www.pappasbrent.com

My research objective is to strengthen the security of **software supply chains**, that is, the components, libraries, and tools for developing, building, and distributing software. Although invisible to software users, software supply chains underpin virtually all modern software, and have become the target of a growing number of cyberattacks. The recent SolarWinds compromise impacted 18,000 public and private sector organizations [1], and was accomplished via malicious code injected into the Orion software build process. The XZ Utils backdoor infected millions of devices across the globe, and was achieved by modifying the project's build system to inject malicious code, masquerading as test case files, into the XZ Utils' liblzma library [2]. The AI agent hackerbot-claw exploited supply chain vulnerabilities to achieve remote code execution in open-source repositories from Microsoft and DataDog [3], illustrating that with the rise of AI technology, software supply chain attacks will only continue to proliferate.

Prior work has mostly tried to help developers preempt software supply chain attacks with documentation and guidelines, but little attention has been given to software verification. The CISA's SBOM [4] and VEX [5] documents provide developers with an inventory of a software's dependencies and vulnerabilities, but do not help developers check whether a software project has been compromised by a direct modification to its build system, as in the case of SolarWinds. Meanwhile, academic researchers have contributed taxonomies for classifying software supply chain attacks and mitigation techniques for preventing them [6], [7], but these documents cannot be used to continuously audit code for supply-chain-related compromises. A more complete approach to hardening software supply chains necessitates that developers constantly verify software projects as they continue to evolve. To do so, developers need automated tools for specifying, analyzing, and verifying the behavior of projects' build systems. Such tools can even be retrofitted to legacy software which may have been designed without supply chain security in mind.

My research secures software supply chains by developing tools and techniques for hardening their individual components. I have developed the first static code analyzer for identifying C preprocessor macros behaviorally equivalent to C variables, enums, and functions [8]; and have led a team of undergraduate students to design an automated tool for translating C-equivalent macros to C code. This enables C translators, such as c2rust for translating C to Rust [9], to translate C programs to safer languages while preserving their macro abstractions. I have implemented a technique for specifying file access permissions for individual software build phases, and mechanisms for detecting violations of such permissions [10]. This approach, called *build phase isolation* directly protects against pipeline poisoning attacks, which modify a software build system to perform some nefarious action such as malicious code injection. I recently directed a large team of undergraduate students to study the degree of natural isolation present in several hundred open-source code repositories, and found that a significant portion of projects are already highly isolated, having most files accessed exclusively by a single build phase. These results indicate that, for many projects, hardening security with build phase isolation is highly feasible.

I plan to continue researching new methods and tools for securing the manifold components of the software supply chain. Securing each component of the software supply chain, and the passage of information between them, will result in safer software world-wide.

1 Research Contributions

My PhD research has focused on strengthening software security by securing the individual components of the software supply chain. My first research publication was a paper in the 46th International Conference on Software Engineering (ICSE '24) [8]; in this work I analyzed the semantics of C preprocessor macros to assist C translation tools in converting macro-laden C code to safer languages, such as Rust, while preserving macro abstractions. Another paper of mine (co-authored by two undergraduate research assistants in a successful research-mentoring collaboration) on actually translating macros to C code is currently under review at the 41st IEEE/ACM International Conference on Automated Software Engineering. This work advances the state of translation research by contributing the first set of semantic-aware rules for translating compile-time macros to run-time functions.

In addition to my work on analyzing and translating C Preprocessor code, during my PhD I have also conducted research on strengthening software build systems. This began with an ICSE '26 New Ideas and Emerging Results (NIER) paper on identifying malicious build system behavior by comparing build phases' file access patterns to specifications detailing their permitted access patterns [10]. After publishing this work I led a team of seven undergraduate students to study the feasibility of using file access patterns to secure build systems by studying the natural isolation between real-world software build phases, resulting in a paper currently under review at ICSE '27. This work advances software security research with novel and performant methods for identifying vulnerabilities in build automation systems, along with analyses supporting their applicability.

1.1 Semantic-aware Analysis and Translation of C Preprocessor Macros

Society's digital critical infrastructure is written in millions of lines of unsafe C code. To make such programs safer, governmental [11] and academic researchers have proposed translating C code to safer languages, such as Rust. But real-world C programs are not written purely in C code, and also make use of **macros**, which are named code fragments and not part of the C language proper. Macros are heavily used in real code, with C programs containing one macro invocation per every three lines on average [12]. Yet despite their prevalence, state of the art C translation lack support for macros, and instead resort to preprocessing C code first before translating it, replacing all macros with their corresponding code fragments first and then translating the resultant C code to the target language. The problem with this approach is that translators end up translating a completely different version of the C programs than the original, producing translations devoid of the original C program's macro abstractions. To solve this problem, my research has presented novel techniques for analyzing macro semantics [8] and for translating macros to C code constructs. My research contributions assist C translation tools in converting C code to target languages while preserving C programs' original macro abstractions.

1.1.1 Semantic Analysis of Macro Usage

In my work on Maki [8], I designed a macro analysis tool for identifying the ease of porting macros to C code by decomposing macro behavior into 26 fine-grained syntactic and semantic properties. Downstream C translation tools can use these properties to determine how to translate macros in a way that retains their original abstractions. Using Maki to analyze real-world code, I discovered that more than a third of macros (37%) are easy to port, requiring developers to only make slight changes to a macro's definition and invocation sites in order to turn it into C code. With Maki's output as a guide, I submitted patches to Linux kernel maintainers converting 11 macros to C code; 9 of these were accepted, and one of them fixed a bug hidden in the macro.

1.1.2 Semantic-aware Macro Translation

After developing Maki, I recruited a team of two undergraduate students to help me produce MerC, the first semantic-aware macro translation tool capable of converting C preprocessor macros into C variables, enums, and functions. To evaluate MerC's effectiveness, I also designed MacroBench, the first benchmark specifically tailored to assessing a tool's macro translation ability. Using MacroBench, we compared MerC's ability to perform macro-to-C translations to that of various LLMs, and found a trade-off: MerC, though fast and correct, only translates a limited number of macros; whereas LLMs translate more macros, but, being non-deterministic, sometimes get translations wrong. We discovered that one can reap the benefits of both MerC and LLMs, while minimizing their downsides, by combining the two tools into a translation tag team which first uses MerC to translate many macros quickly and correctly, and then calls on LLMs to translate the more complex remaining macros. A publication associated with MerC has been conditionally-accepted to the 41st IEEE/ACM International Conference on Automated Software Engineering. The undergraduate students who assisted me on this project have since leveraged their research experience to advance their own careers, with one student being admitted to the Sound and Music Computing Program at Pompeu Fabra University, and the other acquiring a role as a software engineer at Moog, Inc.

1.2 Analyzing and Securing Software Build Systems

Modern software developers use build system technology to automatically configure, compile, and test their software, in order to rapidly deploy it to users. Build system technologies, which include tools such as Make, CMake, and Maven, are intricate programs for orchestrating and executing the phases of the software build process. This coupling of power and complexity make build systems an enticing target for *pipeline poisoning*, which is an attack technique that secretly modifies a project's build system to perform malicious actions like injecting backdoor code into compiled binaries. To protect against pipeline poisoning, in my research I have investigated new techniques for specifying software build phase permissions, performant mechanisms for ensuring that permissions are followed, and analyses of the practicality of using such approaches to secure real-world code.

1.2.1 Detecting Pipeline Poisoning Attacks

In 2025 I developed Foreman, an automated tool for checking that software build phases follow developer-specified file access permissions [13]. Software build phases are the steps that constitute a software build pipeline; common build phases include configuration, compilation, and testing. Pipeline poisoning alters build phases to access files that are normally only used by other phases; for instance, the XZ Utils backdoor modified the project's compilation phase to extract code from poisoned test files, files which normally would only be used by the project's testing phase [2]. Such malicious file actions can be considered as violations of *build phase isolation*. Build phase isolation is a property that build phases exhibit when they each operate under a unique set of well-defined inputs and outputs. Build phase isolation is violated when one phase accesses the inputs of another, as in the case of the XZ Utils backdoor modifying the compilation phase to read files belonging to the testing phase. Foreman detects violations of build phase isolation by executing a sequence of software build phases, recording which files each phase accesses, and finally comparing the recorded file access patterns to a user specification of phases' file access permissions. In a 2026 ICSE NIER paper [13], I demonstrated that Foreman was capable of detecting the two poisoned test files in the XZ Utils backdoor.

1.2.2 Analyzing Real-world Build Phase Isolation

Most recently, I have studied the feasibility of enforcing build phase isolation to secure real-world code, and have found that many existent programs already exhibit high levels of natural isolation, making them amenable to isolation enforcement. Build phase isolation hardens build systems against pipeline poisoning by executing build phases in isolated environments for preventing illegitimate file accesses. As my work on Foreman demonstrates, build phase isolation enforcement is capable of detecting real-world attacks [13]. But build phase isolation enforcement is only most effective when project files are exclusively accessed by a single build phase, since more shared files between build phases creates a wider attack surface for injecting malicious file accesses into a build system. If files in real-world codebases are shared by many phases, then isolation enforcement mechanisms will need to be more lenient, potentially enabling malicious accesses to sneak past. On other hand, if files are exclusively accessed by a single build phase, then isolation can be enforced more strictly, reducing the attack surface.

To determine the feasibility of applying build phase isolation to existing codebases, I conducted an empirical evaluation of the *natural isolation* present in real-world build systems. Natural isolation is the degree of isolation between a project's build phases, without any changes being made to the project's build system. To study natural isolation, I recruited a team of seven undergraduate students to help me create PhaseBench, the first benchmark of executable software build phases, which consists of 1035 cataloged build phases from 368 software projects. To track the files each phase accesses, I created a Filetrace, a new and blazingly-fast tool for tracking file access patterns across build phases. With Filetrace, I collected the file access patterns for all PhaseBench's build phases, and discovered that natural isolation is high, with 75% of studied programs having more than half of their files accessed by only a single build phase. This work advances research on build system security with new and promising findings about the applicability of build phase isolation enforcement. A publication for this research currently under review at ICSE '27. Moreover, the undergraduate researchers who assisted me with this project have used this experience to obtain internships at software companies such as Keystone Strategy and Bogen Communications.

2 Future Research Agenda

My future research will continue to harden codebases against software supply chain attacks. To achieve this goal, I will expand my current tooling to trace a greater variety of build system access patterns, use these tools to develop systems for automatically inferring secure build specifications, and apply these systems to triage attack vectors in real-world code.

2.1 Exhaustively Tracing Build System Access Patterns

My automated tools Foreman and Filetrace are capable of tracing build systems' file access patterns, but there are many more types of access patterns build systems exhibit which could belie malicious behavior. A URL to access a dependency could be hijacked to instead fetch malware. A test binary could be modified to execute malicious code invoking previously-unused system calls. A configuration script could be tweaked to exfiltrate or poison an environment variable. Tracking such network, file, and environment variable access patterns will require both the novel application of existing tools, and the development of new access tracing instrumentation. For instance, the Linux command line utility `ss` could be used to trace accesses to network sockets; the `strace` utility is capable of tracing system call invocations; and the Bash shell itself could be patched to record all reads and writes to environment variables. All these tools fall under the broad category of systems software, aligning perfectly with my teaching expertise. I plan to capitalize on this synergy, and will

recruit from my classes undergraduate students interested in research to assist me with developing and evaluating these tools.

2.2 Automatically Inferring Secure Build Specifications

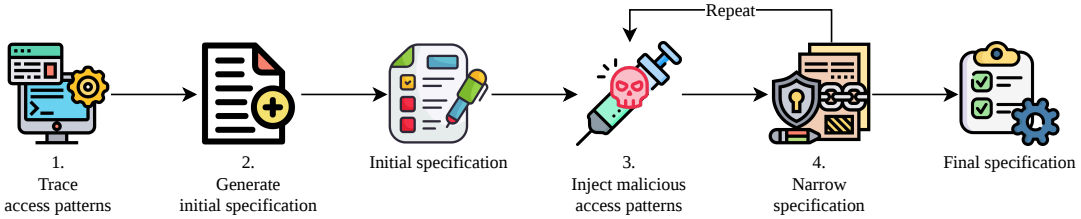

Figure 1: Automated system for inferring secure build specifications.

My tool Foreman currently detects malicious file access patterns by comparing a build system's actual access patterns against a manually-specified file containing the system's permitted access patterns. Manually specifying access patterns is not only error-prone, but flat-out untenable for medium-to-large build systems, such as that of the Linux kernel which spans over 200,000 lines of code [14], [15]. To solve this problem, I plan to develop an automated system for inferring secure build specifications, which is shown in Figure Figure 1 and follows six steps. In Step 1, the build system is ran with instrumentation for tracing its normal access patterns. This instrumentation would include both the previously developed tools Filetrace, as well as newly-developed tools described in Section 2.1. Step 2 will use these traced behaviors obtained from Step 1 to automatically generate an initial specification reflecting the program's normal execution. This initial specification is intended to be sufficient for executing the analyzed build system, but does not secure the system against attacks; this responsibility is delegated to Steps 3 and 4.

To harden the produced permission set against attacks, Step 3 first injects into the build system malicious access patterns. These malicious access patterns will include patterns obtained found in real historical attacks, mutations of the legitimate access patterns obtained in Step 1, and attacks synthesized by AI LLM technology. Step 4 will then compare these injected malicious access patterns to the program's set of legitimate access patterns, and narrow the final permission set to prohibit the illegitimate access patterns, thus removing the attack vectors they pose. In practice, this step may be performed manually by a human, automatically with an AI tool, or semi-automatically with a hybrid approach. Steps 3 and 4 can then be run repeatedly to progressively harden the build system against more and more kinds of attacks. The loop can be terminated after reaching a fixed point, or until the resulting permission set is deemed by the user to be satisfactory.

To assist with the development of this system, and with the comparison of permitted flows against malicious flows in Step 4, I plan to recruit from my classes undergraduate students interested in systems software and AI technology.

2.3 Triaging Attack Vectors in Real-world Code

After developing the system described in Section 2.2, I plan to apply it to real-world codebases in order to identify what attack vectors they are most vulnerable to, while simultaneously constructing specifications capable of protecting against those vectors. To do this, I will lead a team of undergraduate students to first collect a dataset of open-source software. My team will then apply our permission specification system to detect the attack vectors the collected codebases are vulnerable to, and create permission specifications for defending against them. If we discover any real exploits present in the studied software, we will take the responsible course of action, and notify the software owners about them first before disclosing them to the public.

2.3 References

[1] MITRE | ATT&CK, "SolarWinds Compromise." [Online]. Available: https://attack.mitre.org/campaigns/C0024/

[2] L. Collin, "xz." [Online]. Available: https://github.com/tukaani-project/xz/

[3] V. Sharma, "hackerbot-claw: An AI-Powered Bot Actively Exploiting GitHub Actions – Microsoft, DataDog, and CNCF Projects Hit So Far." [Online]. Available: https://www.stepsecurity.io/blog/hackerbot-claw-github-actions-exploitation

[4] The Cybersecurity and Infrastructure Security Agency, "2025 Minimum Elements for a Software Bill of Materials (SBOM)." [Online]. Available: https://www.cisa.gov/resources-tools/resources/2025-minimum-elements-software-bill-materials-sbom

[5] The Cybersecurity and Infrastructure Security Agency, "Minimum Requirements for Vulnerability Exploitability eXchange (VEX)." [Online]. Available: https://www.cisa.gov/sites/default/files/2023-04/minimum-requirements-for-vex-508c.pdf

[6] Z. Pan *et al.*, "Ambush From All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 1, pp. 403–418, Jan. 2024, doi: 10.1109/TDSC.2023.3253572.

[7] S. Hamer, J. Bowen, M. N. Haque, R. Hines, C. Madden, and L. Williams, "Closing the Chain: How to reduce your risk of being SolarWinds, Log4j, or XZ Utils." [Online]. Available: https://arxiv.org/abs/2503.12192

[8] B. Pappas and P. Gazzillo, "Semantic Analysis of Macro Usage for Portability," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 229–240. doi: 10.1145/3597503.3623323.

[9] Immunant inc., "C2rust." [Online]. Available: https://github.com/immunant/c2rust

[10] B. Pappas and P. Gazzillo, "Build Code is Still Code: Finding the Antidote for Pipeline Poisoning," in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering*, in ICSE-NIER '26.: Association for Computing Machinery, 2026, pp. 1–5. doi: 10.1145/3786582.3786799.

[11] DARPA, "Translating All C TO Rust (TRACTOR)." [Online]. Available: https://sam.gov/opp/1e45d648886b4e9ca91890285af77eb7/view

[12] M. Ernst, G. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002, doi: 10.1109/TSE.2002.1158288.

[13] B. Pappas, "Foreman." [Online]. Available: https://github.com/appleseedlab/foreman

[14] J. Oh, N. F. Yıldıran, J. Braha, and P. Gazzillo, "Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 893–905. doi: 10.1145/3468264.3468578.

[15] The Linux Kernel Organization, "The Linux Kernel." 2024.