Securing the Software Supply Chain: Research, Outreach, Education

Laurie Williams[*], Yasemin Acar[†], Michel Cukier[‡], William Enck[*],
Alexandros Kapravelos[*], Christian Kästner[§], Dominik Wermke[*]
[*]North Carolina State University, Raleigh, NC, USA
[†]Paderborn University, Paderborn, Germany and George Washington University, DC, USA
[‡]University of Maryland, College Park, MD, USA
[§]Carnegie Mellon University, Pittsburgh, PA, USA

ABSTRACT

Recent years have shown increased cyber attacks targeting less secure elements in the software supply chain and causing fatal damage to businesses and organizations. Past well-known examples of software supply chain attacks are the SolarWinds or log4j incidents that have affected thousands of customers and businesses. The goal of this response is to aid the Open-Source Software Security Initiative (OS3I) in **identifying focus areas for prioritization** by sharing the outcomes of research, outreach, and education by the NSF-sponsored Secure Software Supply Chain Center (S3C2) since its inception in 2022. The S3C2 experiences and outcomes to date provide the authors with a unique purview of open challenges in securing the software supply chain.

1 INTRODUCTION

The modern world relies on digital innovation in almost every human endeavor, including critical infrastructure. Digital innovation has accelerated substantially as software is increasingly built on top of many layers of reusable abstractions, including libraries, frameworks, cloud infrastructure, artificial intelligence (AI) modules, and others, giving rise to software supply chains where software projects depend on and build upon other software projects. Software developers did not anticipate how the software supply chain would become a deliberate attack vector. The software industry has moved from passive adversaries finding and exploiting vulnerabilities contributed by honest, well-intentioned developers to a new generation of software supply chain attacks where attackers aggressively implant vulnerabilities directly into open-source software components and infect build and deployment pipelines.

Focused research is needed to develop fundamental principles, techniques, and tools to close the attack vectors in the software supply chain. Plummeting trust in the software supply chain may decelerate digital innovation if software organization feel the need to divert resources to reduce their own supply chain risks, with the potential to fragment the software ecosystem.

The goal of this response is to aid the Open-Source Software Security Initiative (OS3I) in **identifying focus areas for prioritization** by sharing the outcomes of research, outreach, and education by the NSF-sponsored Secure Software Supply Chain Center (S3C2) [1] since its inception in 2022. The S3C2 experiences and outcomes to date provide the authors with a unique purview of open challenges in securing the software supply chain.

2 NSF-SPONSORED SECURE SOFTWARE SUPPLY CHAIN CENTER (S3C2)

The Secure Software Supply Chain Center (S3C2) is a large-scale, multi-institution research enterprise funded through the National Science Foundation Frontier program in 2022. The center consists of researchers at North Carolina State University (NCSU), Carnegie Mellon (CMU), and the University of Maryland (UMD), with a close collaborator at Paderborn University in Germany.

S3C2 has the following vision: **The software industry can rapidly innovate with confidence in the security of its software supply chain.**

We will achieve this vision through three goals.

- **Research Goal:** To aid the software industry to re-establish trust in the software supply chain through

[1]https://s3c2.org/

the development of scientific principles, synergistic tools, metrics, and models in the context of the human behavior of supply chain stakeholders.

- **Workforce Goal:** To aid the software industry to create a diverse workforce of technical leaders and practitioners educated and trained in secure software supply chain methods through research, hands-on education, and outreach initiatives with academic, government and industry partners.
- **Community Goal:** To aid the software industry to foster community-wide adoption of evidence-based secure practices through feedback cycles with industry and government, cross-organizational communication, technology transfer, and hands-on education.

To achieve these goals, we have structured our work into four thrusts: code dependencies of software products; build infrastructure; impact on the software industry; and education and outreach. In the following subsections, we summarize the work we have done to date on each of these thrusts and our perspective associated open challenges based on our experiences.

3 S3C2 THRUST 1: CODE DEPENDENCIES OF SOFTWARE PRODUCTS

This thrust is focused on the attack vector through which attackers inject vulnerabilities into upstream open-source components such that they can be leveraged at scale to attack upstream projects.

Modern software products commonly have tens to hundreds of direct and transitive code dependencies. Ensuring code dependencies are free of known vulnerabilities is labor-intensive, requiring dedicated personnel. Simultaneously, malicious code dependencies have become increasingly common due to typo-squatting, dependency confusion, and project take-over attacks. Our work seeks to aid practitioners with useful and high-fidelity information to ensure product security.

3.1 What We Have Learned

3.1.1 Low SBOM quality, produced but not consumed. Much attention has been focused on the Software Bill of Materials (SBOM) since the release of the US Whitehouse Executive Order 14028 [2] in May 2021, which specified that organizations wishing to sell to the US government must provide an SBOM. Later that year, we conducted a series of software supply chain security summits (see Section 5). One of the five key challenges raised by the groups was, "Leveraging the SBOM for Security" [8]. The group acknowledged that generating SBOMs is technically hard. However, they felt the deeper challenge would be making SBOM production more than a compliance checkbox. After two years and enormous attention from the industry, many tools to produce SBOMs have been created with widely different outputs – there is no agreed expectations on the quality of SBOMs, and currently, SBOMs remain largely a compliance checkbox.

Currently, SBOMs are produced but largely not consumed. How might they be consumed to aid security? Consider an application A with an SBOM $SBOM_A$ and a vulnerability database feed VDB that contains information about known vulnerabilities in software and software dependencies. In an ideal world, a software consumer runs a function $f(SBOM_A, VDB)$ to learn if application A is vulnerable to a known vulnerability, likely in one of its dependencies. Ideally, this function would not just identify whether the dependency with the vulnerability was used but also how it was used and whether this specific use is exploitable. Using this information, the software consumer can put in place mitigations or update to a newer version of the application A – or attest that the vulnerability is not security-relevant for the application, such as through a Vulnerability Exploitability eXchange (VEX) [4] statement. Mitigations, such as selective sandboxing of dependencies, can be particularly effective if the source of the application is unavailable, patches are not available, or deployment of patches is logistically challenging. The developer of application A can also run a similar process for all of the application A's dependencies and proactively update dependency versions or apply patches as needed.

Unfortunately, this ideal world is not reality. The current process is messy and error-prone, exacerbated by the low quality of SBOMs and missing data. As a result, even if the software consumer attempted to consider $f(SBOM_A, VDB)$, taking action on the results would cause significant manual effort for both themselves and the developer of application A, while not actually improving security in most cases.

3.1.2 Missing information for vulnerability analysis. The data quality problems are two-fold. First, many SBOMs are created from the final binary application.

Source Composition Analysis (SCA) tools producing these SBOMs use a range of heuristics to determine which libraries were used to build the application, often by looking for signatures of files in containers. This approach is error-prone and perhaps counter-intuitive, given that the SBOM is being provided by the company that has the source code for the application and is performing the build. Build processes often contain extensive models of dependencies. More accurate SBOMs can be created by deriving them during the build process when more complete information is available. Many practices like copying files into a project (a process commonly known as “vendoring”) might provide obstacles that can be overcome with more comprehensive build tools and provenance tracking. Generally, an SBOM created at build time will have more complete information than an SBOM created based on binary artifacts. SBOMs should also include information for both direct and transitive dependencies, a practice that is not ubiquitous.

Second, public vulnerability databases are severely lacking. It is common for practitioners to be given false security reports claiming their application is vulnerable to known vulnerabilities in software dependencies. The existence of a vulnerable software dependency does not mean that the corresponding vulnerable code is actually being used in a way that makes the application vulnerable if that vulnerable code is being used at all. Therefore, practitioners are frequently asked to spend unnecessary time fixing problems that are not there, creating *notification fatigue*. Vulnerability Exploitability eXchange (VEX) [4] has emerged to address this challenge. A developer can include a VEX statement to communicate to a customer that their application is not vulnerable to the vulnerability in the vulnerable dependency. However, this process is currently manual in discovering, in stating, and in consuming.

The VEX process can be made more automated with static and dynamic analysis tools (possibly integrated into the build process), but this relies on having reliable and detailed information about vulnerabilities and their specifics, for example, where the vulnerability is located in the dependency and what kind of data may trigger it. Only with such information can tools detect whether the vulnerability is reachable, exploitable, or whether mitigations such as additional input sanitation or local sandboxing can help.

3.1.3 Automated tools can improve vulnerability information for downstream analyses. For example, many vulnerabilities are fixed without being assigned CVE IDs or similar vulnerability identifiers. SCA companies commonly curate their own “enhanced” vulnerability databases, adding additional vulnerability fixes that they discover. In academia, a line of research seeks to discover “silent vulnerability fixes.” For example, we recently published an approach called Differential Alert Analysis (DAA) [7] that uses commercial-of-the-shelf Static application security testing (SAST) tools to identify when a developer commits code that fixes a vulnerability. With limited resources, we identified 111 silent security fixes in the NPM, Go, PyPI, and Maven ecosystems. We also showed that despite low-precision SAST tools, DAA produces high-precision results and that increasing SAST tool recall increases DAA recall. Our ongoing efforts have shown how Large Language Models (LLMs) such as CodeBERT can further discover silent vulnerability fixes.

Public vulnerability databases are not just lacking in their number of vulnerabilities but also in the information available for the included vulnerabilities. A critical piece of information is the code commit that fixes the vulnerability, often called the “vulnerability fixing commit” (VFC) or simply the “patch link.” VFCs help practitioners mitigate vulnerabilities by enhancing SCA tools, enabling patch presence verification, and new state-of-the-art techniques such as enabling few-shot bug repair. Unfortunately, we found that 63% of security advisories in the GitHub Security Advisory Database (GHSA) for open-source software do not have patch links, severely limiting downstream analyses. Given that GHSA is often seen as one of the best, if *the* best, vulnerability database for having complete and accurate information, research is needed to identify novel methods of recovering patch links for known vulnerabilities. To this end, we have created a tool called VFCFinder [6], which uses machine learning to backfill VFCs for a given security advisory. VFCFinder provides a list of the Top 5 potential VFCs. The VFC is in that list over 96% of the time. VFCFinder’s top choice is accurate 80% of the time, and if there are fewer than 15 commits between releases, over 90% of the time. We used VFCFinder to backfill patch links for over 300 security advisories on GHSA, all of which have been accepted by the GitHub security team.

3.1.4 *Sunsetting dependencies.*

Developers often adopt open-source packages as dependencies, implicitly assuming that in addition to the free code, they will receive free support and maintenance of the package in perpetuity. However, when a software package is no longer maintained, it turns from a free resource empowering rapid innovation into a liability. Developers often use signals such as project popularity to select packages to depend on, but this is no guarantee for continued maintenance. Maintainers regularly disengage from open-source projects for many reasons, including changing interests and life circumstances [11].

With a constantly increasing number of open-source software packages, the amount of packages that need to be maintained increases. The number of open-source maintainers is also still growing, but it is unclear that growth will keep up. Despite heavy investments in open source and research in open-source sustainability (mostly on how to keep projects alive and recruit new maintainers), it is not clear that there is a long-term viable model either with volunteers or with raising funds to pay maintainers. In practice, it is likely unavoidable that some, even popular projects will become abandoned. For example, we detected over 3000 abandoned npm packages among the most popular packages with over 10,000 monthly downloads [10].

When developers face abandoned dependencies, they often have little support on navigating the situation [10]. They could, among others, fork the dependency, somehow encourage others to maintain it, or move to an alternative if it exists – often at substantial cost. More research and development should focus on navigating inevitable situations where projects are no longer maintained, whether through deliberate sunsetting strategies or community-based efforts to help developers *cope* with abandoned dependencies [10].

3.1.5 *Limited reliable information on malicious dependencies.*

Beyond accidentally introduced vulnerabilities in dependencies, a severe concern in software supply chain security is the ability of threat actors to intentionally publish new malicious packages or malicious updates to existing packages (if they gain access to that package). Such malicious updates may intentionally introduce backdoors to attack any system that adopts the dependency or updates its existing dependencies. Past attacks have been widely discussed, especially malicious package updates, which are seen as a severe problem as they critically undermine the trust placed in software packages after initial adoption. In practice, almost no developer does or even would have the capacity to audit all updates to all dependencies they are using critically.

Reported numbers of deliberate supply chain attacks are high and reported to be increasing rapidly. However, we have limited reliable information. Package ecosystems tend to delete malicious packages once reported, and researchers have limited access. Reports also do not distinguish different kinds of attacks, for example distinguishing typosquatting from malicious updates to popular packages from packages taken down simply as spam (a rapidly increasing category). Our initial explorations of packages removed as malicious from npm reveal that many of them are copies of the same package under different names, and almost all of them are new dependencies rather than updates to existing ones. Most are taken down within minutes of their release. More detailed reporting and automated classification are needed to provide a more reliable understanding of security trends in software supply chains beyond the flashy numbers counting all attacks common in today's discourse. Encouraging operators of package managers to report details and share removed artifacts with researchers could provide a much better understanding.

3.1.6 *Metrics to aid in component choice.*

A meaningful quantitative security health score could aid in selecting secure components. The OpenSSF Scorecard[2] project automated the provision of such a score based upon measures of the use of software security practices. However, little research has been done to determine whether the use of security practices improves package security, particularly which security practices have the biggest impact on security outcomes. We developed five supervised machine-learning models for npm and PyPI packages using the OpenSSF Scorecard security practices scores and aggregate security scores as predictors and the number of externally reported vulnerabilities as a target variable [19]. Our models found that four security practices (Maintained, Code Review, Branch Protection, and Security Policy) were the most important practices influencing vulnerability count. However, we had low R2 (ranging from 9% to 12%) when we tested the models to predict vulnerability counts indicating

[2]https://github.com/ossf/scorecard

better metrics and models are needed to inform component choices quantitatively.

3.2 Open Challenges

Updating the version of a software dependency is not a trivial matter. Updates occasionally break functionality (e.g., by changing APIs) or introduce new bugs. Furthermore, most software is both a producer and consumer in the software supply chain, and making updates can affect downstream projects. Therefore, many projects spend significant manual effort triaging dependencies with known vulnerabilities to determine if the vulnerable dependency is used in an exploitable way. VEX helps with these decisions, but the current process is manual. Novel techniques are needed to make the production of VEX automated, accurate, and trustable. Doing so requires significantly higher quality vulnerability information.

Influencing the choice of software and library dependencies is equally important and challenging. Dependencies may be abandoned, poorly maintained, or even malicious. It is unreasonable to expect developers to manually research each dependency and its transitive dependencies before use. Research is needed to identify salient features, automatically extract them from projects, and how best to present the resulting information to developers to inform their choices.

4 S3C2 THRUST 2: BUILD INFRASTRUCTURE

The process and tooling that turns the code of multiple software projects into the production software product is just as important as the code in the software projects. This importance was highlighted with the December 2020 SolarWinds supply chain attack, where the build process was compromised to inject malicious code into the end product. Unfortunately, build systems have seen relatively little attention compared to software analysis. Our work seeks to significantly advance the maturity of build system security by creating novel analysis tools, identifying strong security properties, and understanding how practitioners can adopt them.

4.1 What We Have Learned

4.1.1 CI/CD Vulnerabilities. Modern software development has increased in complexity and several parts of it, such as building, testing and deploying, require automation. This is why Continuous Integration and Continuous Deployment (CI/CD) pipelines have become essential to the development process. Yet these pipelines introduce new code and dependencies that may introduce security bugs, posing new risks to software projects. We studied if code injection vulnerabilities are prevalent in Github Actions by building a static taint analysis system called Argus. We focused on Github because it is the most popular platform to host code and offers, via Github Actions, its CI/CD framework, automation tasks, called workflows, that have become popular.

There are significant challenges in automatically analyzing CI/CD pipelines for security problems. First, these pipelines, also known as Github Workflows, consist of jobs, each containing a sequence of steps, leading to a non-linear execution model. In addition to that, a job can depend on one or more other jobs, creating a complex interwoven build system. Automatically analyzing this new execution model for code injection vulnerabilities requires a new Control Flow Graph (CFG) that considers the non-linear execution semantics of workflows. A workflow can also reference third-party Actions and can participate in sensitive operations, such as access to untrusted input and passing data to dangerous sinks. GitHub also supports three types of Actions: JavaScript, Composite (combine multiple workflow steps within one action), and Docker. This poses significant challenges in analyzing GitHub Actions.

Our team has developed ARGUS [12], a framework designed to systematically examine code injection vulnerabilities in GitHub Workflows and Actions through staged static taint analysis. ARGUS effectively pinpoints potential threats by tracking the flow of untrusted data from sources to sensitive sinks. An essential feature of ARGUS is its Workflow Intermediate Representation (WIR), which addresses the non-linear execution semantics of workflows. We conducted a large-scale evaluation involving 2,778,483 Workflows (1,014,819 repositories) that utilized 31,725 Actions. ARGUS detected security problems in 27,465 Workflows from 16,003 repositories during this evaluation. We subjected 5,643 of these workflows to manual verification, and our findings confirmed the existence of code injection vulnerabilities in 5,298 of them. Among these vulnerabilities, 4,307 were of high and medium severity, posing a significant risk of compromising the respective repositories. Additionally, we identified 80 vulnerable Actions, which rendered any Workflow that uses them vulnerable.

4.1.2 Reproducible Builds. The Solarwinds attack in 2020 highlighted the large amount of trust placed in build systems, with attackers injecting malicious logic into the product binary signed with Solarwinds' official code signing keys. Reproducible builds provide a strong foundation to build defenses for arbitrary attacks against build systems by ensuring that given the same source code, build environment, and build instructions, bitwise-identical artifacts are created. We conducted a series of 24 semi-structured expert interviews with participants from the Reproducible-Builds.org project [9].

From the interviews, we learned that commonly encountered obstacles to reproducible builds include build directory name inclusion and cryptographic signatures on the technical side, as well as the need for patience and good social communication on the interaction side. Many interviewees mentioned positive interactions with upstream projects and other developers, although some specifically noted that upstream communication required patience. As for helpful factors, most mentioned were being self-effective, (being determined, possessing the skill-set to progress reproducible builds), and having good communication with other developers. For transitive dependency problems, concrete technical documentation could be achieved by the pervasive use of SBOMs to indicate all software included in building an artifact, so the transitive dependencies could be traced over a dependency graph. Some interviewees suggested that the overall awareness and buy-in for reproducible builds were lacking and that even with the increase in the prevalence of software supply chain attacks, reproducible builds are not yet widespread.

4.2 Open Challenges

Software threat models need to include all code, as vulnerabilities can also be found in the CI/CD codebase. However, assessing risk in build code can be challenging. For example, GitHub Actions can execute Actions in containers, which can introduce unique vulnerabilities, including privilege escalation and container escape scenarios. Current research has been limited to GitHub Actions, and there is a need to expand tools to other CI/CD ecoystems and cross-ecosystem analysis. Such tools will allow practitioners to proactively mitigate risks and ensure the robustness of their CI/CD pipelines.

Even if the build code is free of vulnerabilities, the build servers themselves could be compromised. Reproducible-Builds provide a strong primitive for protecting against such subversion. Unfortunately, the code that builds software has many sources of non-determinism that prevents Reproducible-Builds from being wide-spread. While there has been significant effort in making core dependencies reproducible, a significant effort is needed to take Reproducible-Builds the last mile.

5 S3C2 THRUST 3: IMPACT ON THE SOFTWARE INDUSTRY

To achieve the S3C2 vision and to impact society, S3C2 must continuously interact with the software industry. Our central approach for engaging with industry is to annually conduct three **Secure Software Supply Chain Summits.** The goal of the Summits is to enable sharing between industry practitioners having practical experiences and challenges with software supply chain security; to help form new collaborations between industrial organizations and researchers; and to identify research opportunities. The Summits are conducted under Chatham House Rules.

Between 2021 and 2023, we have conducted six Secure Software Supply Chain Summits: four with industry practitioners and two with US government practitioners. In the four industry summits, 51 different practitioners have participated, 4 of these participated twice; 37 organizations were represented, 13 of these organizations participated twice. In the two government summits, 26 different practitioners have participated, only 1 of these participated twice; 13 agencies were represented, 4 of these participated twice. In summary, 77 practitioners from 37 industrial organizations and 13 US government agencies have participated. We have published summaries of the Summits [3, 5, 8, 18].

5.1 What We Have Learned

In the following nine subsections, we summarize the discussions from the three 2022-2023 Summits.

5.1.1 Executive Order (EO). The participants had a range of reactions to the EO, particularly because not all of the participant's organizations sell software to the US government. Some practitioners stated that the EO has been a catalyst for a security focus. Consistent with the 2021 Summits, the participants had an overwhelming sentiment that they did not want the EO to turn into a compliance/checkbox exercise but to drive real change toward producing more security software products. However, adequate funding for these new efforts had not been provided in some organizations.

Several industrial participants raised concerns about the vagueness of the EO and the challenges they encountered in trying to comply with it. In the government Summit, some participants acknowledged that when it comes to operationalizing the EO, a lot is still "in flight".

5.1.2 Software Bill of Materials. Industry practitioners stated optimistic benefits from inventory disambiguation and believe that SBOMs can help establish customer trust by providing increased transparency and integrity of their deliverables. Customers can look for extraneous content and identify unwanted third-party packages.

However, most of the discussions about the SBOM revolved around concerns. There were comments and concerns that the EO only states that SBOMs should be created and does not describe what should be done with that data or an articulated problem statement for which SBOMs are a solution. One of the biggest hurdles of SBOMs is the overall immaturity of tooling relative to SBOM consumption. Most industrial participants felt the current state of SBOM was of a "compliance-check-the-box". A participant recently did an audit of available SBOMs. They noted that nearly none met the NTIA minimum requirements [15].

5.1.3 Supply Chain Standards, Guidelines, and Frameworks. Software security standards, guidelines, and frameworks, including 800-218 Secure Software Development Framework (SSDF) [13], NIST 800-161 [14], Supply-chain Levels for Software Artifacts (SLSA) [16], and Software Supply Chain Consumption Framework (S2C2F) [17]), have emerged to guide what organizations can do to reduce software supply chain risk. Compliance with the EO requires attestation to most SSDF tasks, making the SSDF an important standard. The other most mentioned framework used by participants was SLSA [16] security framework. Some practitioners still have difficulty deciding what to be guided by with the numerous standards, guidelines, and frameworks available.

5.1.4 Choosing Dependencies. Every dependency introduces value and risk, and once it is incorporated into a project, it is often hard to replace. Participants discussed a range of strategies for choosing dependencies, and it was apparent that there were no good metrics. The participants discussed the use of OpenSSF Scorecard security health metrics. Only one or two used Scorecard as a metric while others evaluated how they might incorporate it. One participant said that they looked at Scorecard scores and measured whether or not packages were less likely to have vulnerabilities if their score was higher and found no relationship. This participant's observations were consistent with our own studies of the relationship between Scorecard scores and vulnerability count [19].

The government participants discussed what it meant to use software dependencies from foreign countries, i.e., anything not developed in the United States. There was discussion about efforts to identify and manage when software comes from embargoed countries. However, there was also an awareness that adversaries can find ways around those mechanisms.

5.1.5 Updating Vulnerable Dependencies. Companies commonly rely on different strategies and tools when updating vulnerable dependencies. Practitioners mentioned scanning for vulnerabilities by using Software Composition Analysis (SCA) tools. Multiple practitioners mentioned feeling overwhelmed by the number of vulnerabilities identified by these tools in direct but also transitive dependencies. After updating all vulnerabilities, a tool run the next day will may identify more new vulnerabilities. This makes it extremely hard for practitioners to catch up and forces them to triage and prioritize vulnerabilities. Tooling to automatically patch dependencies (e.g., Dependabot's automated pull requests) is available. However, auto-patching without human intervention is often not an accepted practice. Organizations need a process for staying up to date. A participant suggested that having a good automated test suite can enable a company to be more confident when updating a dependency.

5.1.6 Detecting Malicious Commits. Actors of past software supply chain attacks (e.g., SushiSwap) use malicious commits to submit unauthorized changes to the source repository. Detecting and discerning these malicious commits is not always straightforward as attackers often use obfuscated code, steal authentication credentials, or use impersonation strategies to deceive and put malicious code changes through the system. Multiple practitioners believe a closer look at the committer's behavior might help discern malicious behavior. For instance, a committer's activity, reputation, and uncommon behavior (e.g., big vs small changes, critical vs ancillary fixes) can signal suspicious or malicious commits. Machine Learning (ML) could be applied to create appropriate tooling. Overall, the participants did not have good solutions for detecting malicious commits.

5.1.7 Self-Attestation and Provenance. The EO requires government contractors to attest to (1) conformity with secure software development practices; and (2) the integrity and provenance of open-source software used within any portion of a product. In the software supply chain context, provenance refers to not only the identity that created each dependency and transitive dependency but also the process through which each software component was built. For example, provenance is a key part of the SLSA framework [16], and systems such as in-toto[3] can be used to capture and communicate provenance information. Industrial participants shared concerns about ambiguity in self-attestation requirements. However, many participants agreed that the SSDF and attestation are far more foundational to security than the SBOM.

5.1.8 Secure Build and Deploy. Build platforms and CI/CD tools have the potential to enhance software build integrity by establishing documented and consistent build environments, isolating build processes, and generating verifiable provenance. Most practitioners feel comfortable in securing the deployment process and are more worried about the build process. Overall, companies seek guidance on secure build and deployment from the SLSA framework [16]. Most practitioners believe reproducible builds may not be very feasible to check whether a build has been tampered with as there still seem to be many challenges and concerns. As of now, only 20% of builds match bit-to-bit.

5.1.9 Large Language Models. Within the last year, Large Language Model (LLM)-based systems, such as ChatGPT, are increasingly used for automated code generation. A participant expressed concerns that the public's accelerated use of LLM can lead to large-scale data exfiltration. LLM system users regularly pull LLM output into their product and contribute their own proprietary data into training data through their queries.

[3]https://in-toto.io/

5.2 Open Challenges

Per the Summits, some open questions remained.

- How can attestations be made more automated?
- Can the EO be more than a compliance checkbox?
- What is the problem statement for which SBOMs is a solution? Can tools be developed to aid in the actionability and consumption of SBOMs?
- Can the current qualitative feel used to choose dependencies be turned into a trustworthy quantitative metric?
- Are vulnerabilities in transitive dependencies many layers deep less risky than vulnerabilities in a direct dependency?
- How can we better educate computer science students about securing builds?
- How can LLMs be used to aid in software supply chain security?

6 S3C2 THRUST 4: EDUCATION AND OUTREACH

Our Broadening Participation in Computing (BPC) efforts focus on summer camps and monthly workshops during the school year.

6.1 What We Have Learned

We have created a one-week non-residential summer camp for middle-school students that was offered in 2023 at NCSU (July) and UMD (August). At NCSU, 24 students participated (16 males, 8 females). At UMD, 19 students participated (11 males, 7 females, and one gender fluid). The campers were introduced to cybersecurity, cryptography, DDoS, and ethical hacking. Campers worked in teams on a security project. The camps included daily demos, a campus tour, and a field trip.

We also have developed summer 2023 REU projects for four REU students (two at NCSU, two at CMU). All four students worked on projects related directly to the S3C2 thrusts (managing abandoned dependencies, measuring reproducible builds, measuring trust, measuring and analyzing typosquatting, and discovering patterns of malicious software) and were jointly advised across institutions with joint meetings that include at least one Co-PI at NCSU and one Co-PI at CMU (and also often Ph.D. students as mentors from both institutions). Two of the 4 students were from demographics underrepresented in computing research.

6.2 Open Challenges

One important goal of the BPC activities and the National Cyber Workforce and Education Strategy [1] is the development of a more diverse cyber workforce. We made a concerted effort to recruit from underrepresented populations for our activities. In the future, we hope to increase the number of students from demographics underrepresented in computer science in the summer camps and the monthly workshops. We have observed that where the outreach activities are advertised might impact the student composition. We will ensure that organizations focusing on diversity know about these outreach activities. Creating a network of students and families should also help recruit students from various backgrounds.

Another challenge is the integration between the summer camps and the monthly workshops. Students can attend either one or both. Thus, we plan to offer two one-week long summer camps back to back at NCSU and UMD in 2024. These two camps will allow to more easily combine new students and students who have participated in the previous year summer camp and/or in the monthly workshops. So the first week would mainly include new students, and the second week would include returning students.

7 FOCUSED RESEARCH FOR PRIORITIZATION

We make the following recommendations for government priorities for additional research in securing the open-source software supply chain.

1. A public, non-proprietary vulnerability database with community contributions. While CVE and NVD are valuable sources of vulnerability information, they (a) lack high-quality data, and (b) have not scaled to meet industry demand. Many software security companies maintain their own proprietary higher-quality vulnerability feeds; however, the manual curation necessitates that they are a pay-for service. Simultaneously, academic research is creating methods of automating the discovery of vulnerability information; however, it is difficult to share with practitioners. A public, non-proprietary vulnerability database that allows individual contributions from the community has the potential to bring the open source philosophy to vulnerability data. The Cloud Security Alliance's Global Security Database (GSD) is one potential model.

2. Dependency selection as a first-class decision. Developers frequently give little thought to which library dependencies the incorporate into their projects. Dependencies may be abandoned, poorly maintained, or even malicious. New threats are also being introduced as LLMs suggest library dependencies. As such, it is extremely important for developers to give consideration to each and every dependency they include. However, this can only be possible with more automated collection and processing of information about dependencies and their transitive dependencies. While a single numeric score may not be achievable, there is significant room for improvement in how we communicate dependency risk to developers.

3. Make the production of VEX automated, accurate, and trustable. Vulnerability Exploitability eXchange (VEX) has a lot of potential to help practioners by annotating when a dependency is vulnerable to a given vulnerability. Currently, VEX statements are largely written manually and often in natural language. Automating VEX saves time for all parties. However, if the algorithms producing VEX statements are not sound, they will lead to a false sense of security and vulnerable software being used in production.

4. Automate attestation. Our summits with industry and government stakeholders surfaced concerns around attestation and self-attestation. Historically, attestations have been a paper-and-pen exercise. However, not only will such an approach not scale, but security will also be sacrificed due to interpretability. Instead, self-attestation should be automated to the extent possible. One summit participant noted that "Companies are starting to do this already. The first there will define the landscape."

5. Encourage declarative build specifications. The code building software is just as complex as the software itself. While we have written automated tools to analyze the build code of popular platforms (e.g., GitHub Actions), our tools are limited by opaque build processes (e.g., Docker containers) and inline scripts (e.g., complex Bash code). By formalizing the build process into well-defined abstractions, verification tools can prove the correctness (and integrity) of software builds.

6. Incentivize determinisitic (a.k.a. reproducible) builds. Reproducible-Builds can mitigate a large portion of the software supply chain attack surface. While it is infeasible to expect all software to be built many times to verify build integrity, specific software projects exist where the security benefit is worth the computation, e.g., the Tor project. However, end projects *cannot* have reproducible builds if their many dependencies do not already have reproducible builds. By normalizing reproducible builds, stakeholders can more easily choose when they need this added security.

7. Securing supply chains using LLMs. The popularity of LLM-based systems for code generation causes a security concern of adversarial models, resulting in tainted training data such that vulnerable code is generated and integrated into software systems. In addition to current research efforts to mitigate that attack vector, research should also focus on how LLMs can be used to enhance the security of the software supply chain.

8 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation Grant Nos. 2207008, 2206859, 2206865, and 2206921. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] July 31, 2023. National Cyber Workforce and Education Strategy. https://www.whitehouse.gov/wp-content/uploads/2023/07/NCWES-2023.07.31.pdf.

[2] May 12, 2021. Executive Order 14028: Improving the Nation's Cybersecurity. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity.

[3] William Enck Yasemin Acar, Michel Cucker, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. June 2023. S3C2 Summit 2023-06: Government Secure Supply Chain Summit. *https://arxiv.org/abs/2308.06850* (June 2023).

[4] CISA. 2022. Vulnerability Exploitability eXchange (VEX). *https://www.cisa.gov/sites/default/files/publications/VEX_Use_Cases_Document_508c.pdf* (2022).

[5] Trevor Dunlap, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. February 2023. S3C2 Summit 2023-02: Industry Secure Supply Chain Summit. *http://arxiv.org/abs/2307.16557* (February 2023).

[6] Trevor Dunlap, Elizabeth Lin, William Enck, and Bradley Reaves. 2023. VFCFinder: Seamlessly Pairing Security Advisories and Patches. arXiv:2311.01532 [cs.CR]

[7] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 489–505.

[8] William Enck and Laurie Williams. 2022. Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Security & Privacy* 20, 2 (2022), 96–100. https://doi.org/10.1109/MSEC.2022.3142338

[9] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. 2023. It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security. In *In 44th IEEE Symposium on Security and Privacy.*

[10] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. 2023. "We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (San Francisco, CA). ACM Press, New York, NY.

[11] Courtney Miller, David Widder, Christian Kästner, and Bogdan Vasilescu. 2019. Why Do People Give Up FLOSSing? A Study of Contributor Disengagement in Open Source. In *Proceedings of the 15th International Conference on Open Source Systems (OSS)*. 116–129. https://doi.org/10.1007/978-3-030-20883-7_11

[12] Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. 2023. ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions. /projects/argus/. In *Proceedings of the USENIX Security Symposium.*

[13] NIST. 2022. NIST Special Publication 800-218 Secure Software Development Framework (SSDF). *https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf* (2022).

[14] NIST. May 2022. NIST Special Publication 800-161 Rev 1 Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations. *https://csrc.nist.gov/pubs/sp/800/161/r1/final* (May 2022).

[15] NTIA. July 21, 2021. The Minimal Elements of a Software Bi,ll of Materials. *https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf* (July 21, 2021).

[16] OpenSSF. 2023. Supply-chain Levels for Software Artifacts (SLSA). *https://slsa.dev/* (2023).

[17] OpenSSF. July 2023. Secure Supply Chain Consumption Framework (S2C2F). *https://github.com/ossf/s2c2f* (July 2023).

[18] Mindy Tran, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. Sept 2022. S3C2 Summit 2022-09: Industry Secure Supply Chain Summit. *http://arxiv.org/abs/2307.15642* (Sept 2022).

[19] N. Zahan, S. Shohan, D. Harris, and L. Williams. 2023. Do Software Security Practices Yield Fewer Vulnerabilities?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, Los Alamitos, CA, USA, 292–303. https://doi.org/10.1109/ICSE-SEIP58684.2023.00032