Response to **Request for Information:** Open-Source Software Security— Areas of Long-Term Focus and Prioritization Docket: ONCD-2023-0002

Contents

GrammaTech, Inc. welcomes the opportunity to respond to the Request for Information on Open-Source Software Security: Areas of Long-Term Focus and Prioritization, Docket: ONCD-2023-0002. GrammaTech's responses regarding focus areas and sub-areas provide information based on our expertise in cybersecurity and software quality. We believe that it is urgent to foster the adoption of memory-safe programming languages and strengthen the software supply chain, as these actions will have widespread impact on reducing the impact of existing software vulnerabilities. However, it is also important to foster large scale changes to the open source community and developer ecosystem that may happen over a longer timeline, such as incorporating systematic and transparent quality assurance techniques and improving developer education.

1 Area: Secure Open-Source Software Foundations

1.1 Sub-Area: Fostering the Adoption of Memory-Safe Programming Languages

Background. The original goal of designing systems programming languages, such as the still widely used `C`, was to empower programmers to manipulate the machine running the program, including its memory. As programs became more complex, it became clear that this type of low-level machine manipulation was unsuitable for implementing more complex computational problems, such as using algorithms or mathematical structures. The program state could easily become inconsistent with the developer's intent: the program exhibited *bugs*. Subsequent systems languages, such as `C`'s extension `C++`, offered capabilities to program at a higher abstraction level (e.g., via a Standard Libraries), allowing the programmers to manipulate complex algorithms and structures without needing to implement every detail themselves. The capability of low-level memory manipulation, however, was often maintained, in the name of backward compatibility.

Memory errors have proven to be the worst kinds of software bugs: the program may crash with a diagnostic message; but it may also hang or silently open vulnerabilities in a code location seemingly unrelated to where the memory violation occurred. This elusiveness likely makes those errors a main contributor to the immense cost associated with fixing bugs, which, in 2002, NIST estimated to be in the order of tens of billions of dollars per year [8]. The situation is exacerbated for open-source software (OSS): (a) software developers may incorporate OSS into their own code with little scrutiny, since it is easy to obtain and typically free of charge, and (b) code vulnerabilities are much easier to identify *by an expert adversary*, and hence to exploit, than in closed-source binary code.

Instead of allowing programmers to introduce memory errors, programming languages can be designed to *enforce* the absence of memory errors, otherwise known as memory safety. One way to do this is to equip the language with a restricted memory-access interface. For a suitable set of such restrictions, it may be possible to prove that any program written in this language is free of typical memory access violations (such as using a memory cell after it was declared unused ["free"] in the program). This approach was implemented quite recently in the Rust programming language.

While memory safety will generally significantly improve program reliability, it is important to realize that it is not a panacea to program correctness:

- It does not guarantee the program to be free of logical errors, such as incorrect assumptions about inputs.
- It does not prevent problems elsewhere in the build stack, such as in the compiler or the hardware.
- Languages (like Rust) that implement memory safety via a restricted access interface typically have an "unsafe" escape mechanism, with significantly relaxed access rules. Programmers can use these

mechanisms to circumvent the safety restrictions, because they believe it will be easier or produce more efficient code, or to incorporate code from other languages.

(i) Supporting rewrites of critical open-source software components in memory safe languages. Embracing memory-safe programming is not just about writing new code. New projects may require incorporating open-source code available only in a traditional unsafe language. Moreover, the project may rely on existing internal legacy code bases. These circumstances motivate the contemplation of rewriting existing legacy code into a memory-safe language.

Recommendations. We recommend these questions be considered when contemplating a rewrite of legacy code into a memory-safe language:

(a) Which memory-safe language should be used as the rewriting target and in future development?

(b) Should the project *rewrite the code* or (more broadly) *reimplement the application* in the new language?

The answers to these questions depend on the application at hand and on particular circumstances. Instead of giving necessarily imprecise generic advice, we discuss the key considerations that should be taken into account when making individual decisions.

(a) Which memory-safe language should we use as the rewriting target and in future development? We recommend basing this decision on two main aspects:

Language usability: What is the learning curve associated with the new language? Is it similar in "look and feel" to the languages developers have traditionally used? Is there a supportive community?

Languages like Java and Rust are similar in syntax to `C/C++` and enjoy wide community support, but they differ in the required learning effort: Rust's restrictions on the memory access interface inevitably generate a learning curve for the programmer and may initially be perceived as a nuisance.

Code efficiency: Does memory safety come at the expense of decreased code performance? This is typically the case for languages whose memory-safety is achieved via automated Garbage Collection (GC), such as Java, C# and Go. A follow-up question, however, is whether any expected performance degradation is large enough to matter for the application at hand. Huge progress has been made over the years since the inception of automated GC. For high-performance scientific computing, gaming, and many systems applications, the degradation may still be unacceptable. Evidence (mostly anecdotal, at this time) suggests that well-designed code in the non-GC language Rust delivers performance similar to `C/C++`. Rust is currently one of the very few strong contenders in high-performance, memory-safe computing.

(b) Should we rewrite the code, or should we more broadly reimplement the application in the new language? The difference is that the former attempts to reuse the algorithms, data structures and code design from the source-language implementation and to convert them into the target language, staying close to the source where possible. The former considers the implementation in the new language mostly like a new implementation project; any existing legacy implementation only serves as guidance. Which technique to choose depends on the proximity of the two languages involved, but also on factors such as the age and condition of the code in the source language: that code may be in need of an overhaul anyway.

(iii) Developing tools to automate and accelerate the refactoring of open-source software components to memory safe languages, including code verification techniques. The process of migrating existing legacy software components, and especially open-source software components, into a chosen memory-safe target language is often too time- and work-intensive to be practical to do manually. Work has begun to automate this process and develop tool support. We summarize our recommendations as follows, and subsequently go into more detail about them.

Recommendations. We recommend a policy of rewriting components of code in a memory-safe target language, also known as *migrating* them, subject to the following constraints. This advice was developed during our work on the CRAM project [6]:

Complexity: Automatic translation between high-level programming languages is a computationally hard problem; a fully automated migration solution is likely unrealistic. For example, a partially-automatic process may involve a user by asking them to decide among different options for expressing a certain passage from the source language as a corresponding passage in the target language. Such involvement can help to alleviate user skepticism of an otherwise black-box migration tool. It is key, however, that users not be burdened with repeated assistance requests of the same kind. A learning mechanism can greatly increase the usability and acceptance of the migration tool.

Migration in stages: Specifically for migrating `C++` code into Rust, it proved useful to delay the conversion of the input code into Rust until that code was first refactored to (i) *harden* it against common code safety and security violations, as well as (ii) prepare the future migration into Rust. Such pre-migration code refactoring simplifies the path toward migration into the target language. It also creates a hardened version of the source program as a by-product, which can be used if the subsequent migration into the target language takes more time than expected or succeeds only partially.

Maintainability: The automatically generated code should be human-readable, well-formatted, and such that programmers in the new language can pick it up and work with it comfortably. This is important both for detecting any possible errors that may be introduced during migration, as well as for future code development.

Assurance: The migration should produce evidence that the migrated code has the same behavior as the original code. Test suites (and test harnesses) should be migrated alongside the code. Additional test cases that specifically scrutinize the correctness of the migration may be needed, for instance for language-specific patterns that cannot be migrated faithfully.

Mixed-language compilation: When a language migration is partial (containing portions of code in different languages), it cannot be compiled by traditional compilation tools. Without compilation, it is nearly impossible for users to experiment with the output code. A solution is to support compiling mixed-language code. For example, the CXX library [4] allows programmers to define interface functions in C++ and Rust that act as a bridge between code regions in the two languages. The compilers from the two languages and a specialized build environment produce an executable binary. As a result, a codebase partially migrated to Rust can be run and tested.

Both the considerations for the choice of target language, and the above requirements for software rewrites will impact adoption of the new language. We believe that language usability and maintainability of the migrated code are the most critical criteria to ensure adoption of the language and the migrated code in the long run. Transition efforts will struggle if using the new language is fundamentally disruptive for developers (especially if accompanied by poorly understood benefits). It will also suffer if the migration process produces obscure source code unsuitable for human consumption and hence impossible to share back with the open-source community. As discussed above, code performance efficiency is an application-dependent criterion and not universally the top priority. Migration assurance techniques can short-cut the path to correctly migrated, memory-safe code. In the absence of such techniques, we can treat the migrated code essentially as new and subject it to rigorous testing and verification regimes, just as we would for code written from scratch.

Further discussion. As an example of the *migration in stages* approach recommended above, consider *nests* of non-constant references to the same memory location in the `C++` language. While perfectly legal, such nests are a common source of hard-to-track memory errors. Moreover, they are forbidden in the target language, Rust. The CRAM project [6] therefore eliminates such nests, within the `C++` language, before migrating any code into Rust.

Pre-migration code refactoring has several advantages, including:

1. It simplifies the path toward migration into the target language, by separating the operation and purpose of each of the two steps: (i) eliminating memory-unsafe coding practices and (ii) migrating from one language to another. Existing tests can be reused without change, and correctness reasoning ("intra-language assurance analysis") is much easier than after crossing into a new language.

2. It creates a hardened version of the source program as a by-product, which can be used if the subsequent migration into the target language takes more time than expected, or succeeds only partially.

As mentioned in the introduction to Section 1.1, certain memory-safe languages offer an escape path to the programmer, where code labeled "unsafe" (or similar) has the freedom to access memory with a much weaker set of constraints and, consequently, no guarantees regarding the absence of memory errors. Use of such unsafe code is very common in open-source Rust software, since the community is still developing an understanding of the memory access rules in Rust, and when it is really necessary to circumvent them. We expect that "unsafe" is used in more situations than necessary.

As a remedy, open-source software offers the advantage that users can identify such fragments and subject them to extra scrutiny. In the long run, we suggest multiple strands of *research* to more sustainably improve this situation, such as investigating:

- the need for unsafe code: in what circumstances is it really necessary?
- the consequences of using it: are there cases when using unsafe code is safe?
- automated techniques to lift unsafe code to safe code.

1.2 Sub-Area: Reducing Entire Classes of Vulnerabilities at Scale

Background. One of the principal challenges posed by relying on open-source software is the inability to effectively judge its safety and security. Assessing safety and security of software implementation largely remains to be a painstaking manual effort, despite recent advances in software analysis and formal methods. As a result, stakeholders either opt for commercial solutions, where the trust is derived from software-vendor reputation, or choose to develop software in-house, where they have direct control over the development process.

In contrast to the software itself, software development processes are much more uniform, and thus, are easier to assess and measure. This also makes it easier to judge what best practices are and whether software projects follow those best practices. A good development process integrates target-specific mechanisms for assessing software quality, such as rigorous code reviews, systematic testing, and regular use of analysis tools for detecting software defects. Thus, a well-structured, rigorous development process is more likely to result in high-quality, secure, and resilient software.

Recommendation: We recommend policies to extend existing open-source development infrastructure to increase visibility into the development process. A concrete way to increase visibility is by incorporating quantitative, easy-to-grasp *metrics* that systematically measure the rigor of the development process. The metrics will reflect the qualify of the resulting software and will allow stakeholders to make informed decisions about selecting and funding open-source projects.

Making the development process more transparent, observable, and measurable provides many important benefits for the open-source community and enables a systematic adoption and use of open-source software for a wider range of applications. Metrics, which are measurable quantities that provide information about features of interest, are a tool for increasing visibility. Existing open-source infrastructures, such as GitHub and GitLab, are well-positioned for this. They already provide several easy-to-digest metrics about hosted open-source projects, including the numbers of downloads, forks, and contributors. These metrics, however, shed little light on software quality. Incorporating additional quantitative, easy-to-grasp metrics that systematically measure the rigor of the development process will impact the open-source community as follows:

- Stakeholders can use these metrics in selecting software that meets their expectations. For example, safety-critical applications may require rigorously developed, verified software.
- The government can rely on the metrics to offer incentives for open-source developers to adopt more rigorous development practices. The metrics will also enable measuring the progress of such adoption effectively.
- Funding agencies can rely on the metrics to allocate funds in accordance to their needs. For example, some may prefer to sponsor the extension of well-established, properly engineered software, while others may choose to support projects with more advanced features, but less software quality maturity. The metrics will help the agencies identify the relevant projects.

Achieving this vision of measurable software quality requires advances in fundamental research as well as extensions to the existing open-source development infrastructure:

Research. Advances in fundamental research are required to develop the methodology for formalizing and quantitatively assessing software development processes. Development processes bring together many heterogeneous factors: developer interaction, developer's discipline in adhering to coding guidelines, choice of tools employed for software verification and validation, selection of the toolchain for building the software, quality of the test suite, and many others. An approach is needed that can effectively merge these diverse factors into a handful of quantitative metrics that are clearly understood by all stakeholders: developers, managers, security analysts, government officials, and software acquisition personnel.

Infrastructure. Existing open-source development infrastructures already incorporate facilities for automating and streamlining various aspects of software development. For example, they provide user interfaces for reviewing code changes and tracking development issues, repositories for maintaining software documentation, and hooks for automatically building, testing, and packaging software. However, the underlying mechanisms themselves are typically implemented in ad-hoc ways that are specific to each project. To enable uniform reasoning and assessment of the development-process quality, additional visibility into the implementation of individual development-process steps is required. The existing infrastructures must be extended to expose the end-to-end flow of the software-development process, the semantics of its individual steps, as well as a uniform interface for inspecting each step's outcome.

1.3 Sub-Area: Strengthening the Software Supply Chain

Detection and mitigation of vulnerable and malicious software development operations and behaviors.

Background. Open-source software (OSS) aims to distribute the costs associated with developing, maintaining, and reusing software across the community of interested users. The public nature of OSS presents challenges in areas that historically relied on proprietary control and privacy—would-be attackers have full access to the underlying code, allowing for reverse-engineering and identification of vulnerabilities. Securing critical OSS supply chains requires novel approaches to mitigate the risks associated with the public nature of the underlying systems.

Current strategies for securing OSS supply chains include vulnerability scans, formal verification, and software bill of materials (SBOM) tracking. These techniques are labor intensive, do not always work at a large scale, and are often reactive in nature. Reactive approaches require prior knowledge of the underlying security risks, but this information is often incomplete. Public vulnerability databases are often used to aggregate information and propagate patches to identified problems, but this post-hoc approach means that systems often remain vulnerable for considerable periods of time. For example, the well-known major security vulnerability known as Heartbleed existed for years prior to its public recognition and the subsequent fix. To allow for rapid development and deployment, users of OSS often treat it in the same way as proprietary software (i.e., "generally considered safe") because of a lack of resources to fully vet every part of the supply chain. Furthermore, there is often a temptation to delegate the proper vetting of OSS to traditional reactive tools and accept the results as an implied guarantee of safety for the underlying systems, which does not account for the known limitations of the reactive approaches.

Recommendations. To address these issues, we recommend encouraging a shift in the OSS security philosophy from traditional reactive to emerging *proactive* approaches. To do so, we recommend funding and pursuing research in these proactive approaches and incentivizing projects to use these approaches. In practice, it is impossible to find and patch all unknown vulnerabilities in OSS. A proactive strategy emphasizes layered, defense-in-depth solutions to protect against yet-unknown attacks and treat OSS supply chain dependencies with zero trust. Such an approach can effectively stay ahead of attackers by future-proofing systems against both known and unknown attacks, by providing system-, deployment-, and environment-specific protections. These protections can and should take many forms (e.g., dynamic runtime protection, directed test-based vulnerability discovery, and preemptive static vulnerability identification). The different approaches have different strengths and weaknesses and can complement each other to allow for additive security, when combined. Critically, these solutions should also evolve with systems, as the software maintenance process is known to be costly. To not further burden the maintenance process, we require protections that are easily adaptable or can be automatically reconfigured as the target OSS changes.

GrammaTech has a strong basis of confidence in the defense-in-depth, proactive approach to securing OSS supply chains. One such strategy involves autonomic (or self-healing) systems that dynamically monitor program execution for errors and take immediate corrective action to mitigate faults. A study of our autonomic tooling found that developed security and behavioral policies are capable of covering 13 of the

16 vulnerability categories in the NIST taxonomy of flaws and vulnerabilities[1]. Critically, this tooling can also address vulnerabilities that reactive scans typically miss (e.g., DDoS, server probing, side-channel information leaks, etc.). These types of attacks rely on timing or working within the bounds of allowable behavior, characteristics to which traditional reactive tools are often blind. State-of-the-art proactive dynamic and static tools often provide detailed, explainable forensic evidence for identified vulnerabilities. This information facilitates both automated and manual patching as well as human understanding for the high volume of issues present in OSS supply chains. These tools lay the foundations for defense-in-depth proactive security strategies and have shown early successes against a wide range of critical vulnerabilities in real OSS (e.g., web servers, networking protocols, coreutils, etc.). We recommend further funding in support of developing and maturing these tools.

We expect the proposed paradigm shift to have a positive impact across the software development and cybersecurity landscape. These proactive protection mechanisms generalize to a broad range of domains, systems, and attack vectors, and do not rely on patching individual vulnerabilities only when the associated risk is detected and deemed critical. Potential positive impacts widely-deployed, mature proactive solutions include reduced intrusions, data breaches, ransomware attacks, and intentional sabotage of safety-critical systems. General, evolvable, and future-aware solutions to the current problems in OSS supply chains will ease the maintenance burden for the involved systems which allows for cost reduction and better allocation of resources within the overall security landscape.

2 Area: R&D/Innovation

2.1 Sub-Area: Application of Artificial Intelligence and Machine Learning Techniques to Enhance and Accelerate Cybersecurity Best Practices with Respect to Secure Software Development

Background. In the last few years we have experienced incredible advances in the area of Artificial Intelligence (AI) and in particular in generative AI, Large Language Models (LLMs) being the most prominent example. LLMs have shown impressive code understanding and code generation capabilities, which represent an enticing opportunity for increased automation and enhanced productivity in software development.

At the same time, LLMs are known to suffer from "hallucinations" (incorrect or otherwise improper output) and provide over-confident responses. LLMs have been trained on existing code, which contains a variety of weaknesses and vulnerabilities. The LLMs might inadvertently reproduce unsafe practices and generate vulnerable software. Thus, a naive application of LLMs alone in software development will likely result in *faster* but not *better* software development.

Recommendations. We believe LLMs present great opportunities for improving software security when applied to 1) problems whose solutions can be independently verified, and 2) in combination with tools for formal reasoning. We recommend the following specific research areas for increasing software security that deserve more attention and research funding:

Test-case generation. Testing is the most common and widely adopted approach for ensuring that programs behave as expected. However, developing good test suites is time consuming and often tedious; developers are often pressured to cut corners. LLMs can be used to generate test cases automatically [10]. However, a tighter integration with other testing tools is required, e.g., tools to measure code coverage. This will increase the effectiveness of LLMs' automatic test generation capabilities.

[1]https://www.nist.gov/itl/ssd/software-quality-group/taxonomy-software-flaws

Formal software verification. This family of methods, which uses techniques based in mathematical reasoning such as proofs, can provide strong safety and security guarantees. In contrast to testing, formal verification can prove the absence of entire categories of bugs and ensure that certain safety properties are maintained under all circumstances. However, formal verification is costly and requires great expertise, which limits its applicability to only the most demanding safety-critical environments. LLMs have shown promising code reasoning skills, which can be a significant asset in the formal verification process. For example, LLMs can potentially be tuned to generate proof strategies and produce natural-language explanations of the generated proofs that lower the required expertise level. A requirement of their use in this domain, however, is to treat such proof strategies (or any other LLM-generated components) as *suggestions*. The proof must be *executed* by sound, formal technology, such as automatic and interactive theorem provers, and perhaps static analyzers.

Vulnerability discovery and remediation. The current landscape of tools for identifying, triaging, and repairing vulnerabilities in complex software systems is fragmented, with limited automation or interoperability. This places a significant burden on analysts to select, configure, and process the output of these tools. LLMs' code generation and understanding capabilities can be applied to reduce this burden. For example, some vulnerability identification tools, such as taint analysis tools, require users to select which data in a given software is sensitive. This selection is currently done manually because it is very context and domain dependent, and it can only be done with an understanding of the goals of the software and the context in which it is executed. LLMs' extensive knowledge derived from large quantities of training data can potentially address this problem and perform this data selection automatically.

Vulnerability prioritization. A recent survey reports that two-thirds of IT organizations have a backlog of more than one-hundred thousand vulnerabilities [9]. Furthermore, the cost of prioritizing detected vulnerabilities was cited as the greatest bottleneck preventing remediation. Assessing a vulnerability's severity and exploitability requires understanding the context and the software's application domain. We believe relevant information can be automatically extracted from the source code using LLMs, which can leverage this information to make determinations about the criticality of vulnerabilities and propose fixes and mitigations.

Automatic code translation. Large language models have shown great capabilities at automatic natural language translation (e.g., from French to English), even when not explicitly trained to do so [12]. Research has begun to investigate translation between programming languages [11]. Automatic and semi-automatic translation between programming languages can be employed to improve cybersecurity properties, such as translating code from memory-unsafe languages to memory-safe languages. If successful, such translation can prevent large classes of vulnerabilities as well as increase the code's maintainability.

References

[1] GrammaTech, Inc. SEL: Software evolution library.
`https://github.com/GrammaTech/sel2018`

[2] GrammaTech, Inc. GTIRB: Intermediate representation for binaries.
`https://github.com/GrammaTech/gtirb`

[3] GrammaTech, Inc. DDisasm: Datalog disassembler.
`https://github.com/GrammaTech/ddisasm`

[4] `https://cxx.rs`.

[5] Galois, Inc., and Immunant, Inc. The C2Rust project.
`https://c2rust.com`

[6] GrammaTech, Inc. CRAM: C++ to Rust Assisted Migration.
`https://cpp-rust-assisted-migration.gitlab.io`

[7] Executive Office of the President, Office of the National Cyber Director. Request for Information on Open-Source Software Security: Areas of Long-Term Focus and Prioritization. RIN: 0301-AA01, Docket ID: ONCD-2023-0002. In: Federal Register, Vol. 88, No. 153, Thursday, August 10, 2023.

[8] National Institute of Standards and Technology, Acquisition and Assistance Division. The Economic Impacts of Inadequate Infrastructure for Software Testing. Final Report, RTI Project Number 7007.011, May 2002.

[9] Ponemon Institute. The State of Vulnerability Management.
`https://www.rezilion.com/wp-content/uploads/2022/09/Ponemon-Rezilion-Report-Final.pdf`, 2022.

[10] C. Lemieux, J. P. Inala, S. K. Lahiri and S. Sen. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models.
45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023.

[11] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. Understanding the effectiveness of large language models in code translation, 2023.

[12] W. Zhu, H. Liu, Q. Dong, J. Xu, S. Huang, L. Kong, J. Chen, and L. Li. Multilingual machine translation with large language models: Empirical results and analysis, 2023.

Appendix

Information about GrammaTech and the Authors

This document is being submitted by GrammaTech, Inc.[2], a leading developer of software assurance tools and advanced cybersecurity solutions. The company was founded in 1988 by two Cornell University faculty members and has since then been well-known in the software research community. GrammaTech also provides open-source technology, including the Datalog disassembler DDISASM based on the GTIRB binary representation [1, 2], and the SEL software evolution library for code analysis and refactoring [3]. Related to this Request for Information, GrammaTech is actively involved in open-source research on semi-automatically refactoring legacy `C++` code to more modern and memory-safe code, and on migrating such code into Rust, with the CRAM `C++`-to-Rust source-code migration tool [6]. GrammaTech has public repositories for the distribution of open-source code on GitHub and GitLab[3].

GrammaTech has ongoing fundamental research in several areas, including areas that affect open-source software (OSS) supply chain security. These technologies address the defense-in-depth, multi-layered protection strategy, falling into four main proactive categories: runtime monitoring and mitigation, directed dynamic exploitability analysis, fuzz testing for identifying corner case vulnerabilities, and static hardening transforms. GrammaTech offers "autonomic" (i.e., self-regulating) protections for checking program and firmware execution at runtime against policies of allowable behavior. The deployed tooling can automatically mitigate threats in real time, guarding against both known and unknown bugs and vulnerabilities.

To complement runtime monitoring and mitigation, GrammaTech offers Proteus, a dynamic tool for finding and patching vulnerabilities in binaries via exploitability analysis. Proteus can help identify sufficient conditions for exploits without them having to occur. This added information helps to understand and improve the full security landscape.

GrammaTech offers another dynamic layer of security via the fuzz testing tools Bindle and REAFFIRM to identify corner case execution paths that may lead to vulnerabilities. Targeting both software and firmware, these tools automatically craft test harnesses and inputs using local analysis during execution. The results pinpoint unsafe program states and conditions to provide input for patches and fixes.

In the static proactive domain, GrammaTech has a series of hardening and diversification transforms that can be used to automatically transform software binaries using the GTIRB representation. These transforms apply patches for known common problems that are vulnerable to malicious attacks.

The combination of static and dynamic proactive techniques offered by GrammaTech embodies a defense-in-depth, generalizable approach to securing zero trust OSS supply chains. These techniques are complimentary in nature and address many diverse security concerns, stressing automation and evolvability to allow for rapid deployment on arbitrary systems and their dependencies.

The authors of this response are Senior Scientists at GrammaTech, working in diverse fields of software research and development. Their combined expertise includes code security, software reliability, autonomous system design, and the use of Artificial Intelligence in support of such fields, as well as teaching many of these subjects in universities, at the undergraduate and graduate levels.

[2] `https://www.grammatech.com`, `https://grammatech.github.io`
[3] `https://github.com/GrammaTech`, `https://gitlab.com/GrammaTech`