Revolutionizing Cyber Resiliency for Military Systems

Safeguarding our military systems against current and future software vulnerabilities

"Software is everywhere… Fighting and winning on the next battlefield will depend on DoD's proficiency to rapidly and securely deliver resilient software capabilities." *-- 2022 DoD Software Modernization Strategy*

Our national security enterprise relies on an aging IT infrastructure supporting countless networked systems across the globe. The accepted security standards in software development over the last 30 years – from virus scanning and patching software to intrusion detection systems – have led to inherently vulnerable systems. Department of Defense IT systems are no exception, as vulnerabilities exist in its legacy architectures and most advanced fighter aircraft and weapons systems.

Our nation's adversaries have developed cadres of highly skilled cyber actors trained in advanced vulnerability exploitation to readily disrupt, degrade, and destroy our aging IT systems. For example, a group of Chinese-sponsored hackers, Volt Typhoon, has secured access to critical telecommunications and utility infrastructure.[1] Moreover, nation-state adversaries with highly skilled cyber actors have also demonstrated a willingness and ability to steal and re-engineer intellectual property -- including source code used to design and build our nation's most sensitive military systems and platforms.

Technology to minimize exploitable software vulnerabilities is available today.
Until recently, the complexity of implementing formally verified software has prevented these techniques from being widely adopted across the defense industrial base. However, advances[2] over the past decade have made them more accessible for mainstream practice, making way for the possibility of scaling these capabilities to apply to all existing and future DoD systems and platforms.

DARPA has produced a set of scalable tools that can secure and prove the absence of exploitable vulnerabilities across nearly all existing and future DoD systems. These tools employ software development practices based on formal mathematical methods (i.e., 'formal methods'). Nothing works better than formal methods when it comes to cyber-hardening. Like other engineering disciplines, using a mathematically based development process to create proof that software does what it is supposed to and does not do what it is not supposed to implies someone cannot hack it.

The revolution in formal methods practicality began with a demonstration in 2016 with DARPA's High-Assurance Cyber Military Systems (HACMS) program. This effort demonstrated how a broad collection of techniques for reasoning about software in a machine-checked way (collectively known as formal methods) could be applied to military systems, including quadcopters, helicopters, and automobiles, making software inherently more secure. Following the success of the HACMS program, DARPA has invested in follow-on programs that apply formal methods to legacy code, software certification, and standard software development tools. Many of these tools and techniques have transitioned to the services for further maturation and operational use. Additional information about DARPA's Resilient Software Systems portfolio can be found at https://www.darpa.mil/work-with-us/i2o-thrust-areas#rassystems.

The universal applicability of these software tools provides a channel for drastically improving the security of the DoD's massive catalog of deployed legacy code.

Rapid action to implement these tools in legacy and future systems can dramatically reduce the DoD's vulnerability ahead of future global conflicts.

---

[1] *During his remarks at an April 2024 Vanderbilt University event, FBI Director Christopher Wray described Volt Typhoon's persistent access to U.S. infrastructure using tactics known as "living-off-the-land." They're exploiting built-in tools that exist on victim networks to get their sinister job done, tools that network defenders expect to see in use and so don't raise suspicions—while they also operated botnets to further conceal their malicious activity and the fact that it was coming from China.*

[2] *Several factors enabled the formal methods revolution, including increased processor speed, better infrastructure like the Isabelle/HOL and Coq theorem provers, specialized logics for reasoning about low-level code, increasing levels of automation afforded by tactic languages and SAT/SMT solvers, and the decision to move away from trying to verify existing artifacts and instead focus on co-developing the code and the correctness proof. (Source: Fisher Group)*

Formal Methods Work – DARPA's HACMS Case Study

The High-Assurance Cyber Military Systems (HACMS) program proved that it's possible to deliver high-assurance software capable of withstanding cyber threats and operate effectively in military applications.

Traditionally, air gaps and obscurity provided security for embedded systems. A lack of network connectivity made these embedded systems an unattractive target for cyber threat actors. However, the proliferation of commoditized software in embedded systems coupled with advances in re-engineering and vulnerability exploitation tools have made embedded systems an attractive target for cyber-attacks.

HACMS employed formal methods to construct high-assurance software – i.e., functionally correct software that satisfies appropriate safety and security properties – and generate machine-checkable proofs that the code was safe and secure. These high-assurance software tools are publicly available today and have transitioned to parts of the U.S. government, and defense and commercial industries.

The Test

DARPA tested the software developed under HACMS on real military hardware and systems to verify its performance and compatibility in operational environments. First using a small quadcopter as a testbed, then graduating to a much larger helicopter, Boeing's Unmanned Little Bird, DARPA demonstrated the benefits of software written with formal methods. HACMS conducted simulated cyber-attacks (often called red team exercises) to test the resilience of its software against real-world threats. These exercises identified weaknesses and vulnerabilities that could be addressed before deployment. Throughout the development process, HACMS continuously monitored and evaluated the software to ensure that it met the desired security and reliability standards. This iterative approach enabled ongoing improvements and corrections.

When the project started, the Red Team was able to remotely take over the systems. At the end of the HACMS program, they repeated that experiment while Little Bird was in flight with two test pilots on board. DARPA's HACMS team had become so confident in the formally verified software that they were willing to risk the lives of those two pilots as the Red Team attempted to hack the helicopter – the Red Team failed, and the pilots remarked they couldn't even tell the difference in flying the high assurance version. **To this day, the system has yet to be successfully hacked. The following provides the abbreviated story of how.**

Proving What's Possible

The HACMS program achieved these results by first charging a "Blue Team" of formal methods researchers to improve the security of a hobbyist quadcopter. The team rewrote ~80,000 out of the 100,000 lines of code with formal methods and divided the code on the system's mission-control computer into partitions. This approach was required because the system contained off-the-shelf, monolithic software. This attribute, which is shared by many legacy code bases, meant that if an attacker broke into one piece of it, they had unfettered access to the entire system. Unlike most computer code, which is written informally and evaluated based mainly on whether it works, formal methods-based software can be analyzed like a mathematical proof: Each statement follows logically from the preceding one. An entire program can be assessed with the same certainty that mathematicians prove theorems.

With the formal method upgrades in place, Blue handed the quadcopter over to Red to see if they could hack into the system. The tools and methods worked as intended and the hackers were unsuccessful. This methodology has stood the test of time and subsequent hacking events, including a 'hack the drone' challenge at DEF CON, have proved unsuccessful.

Following the success with the quadcopter, the DARPA team applied the formal methods approach to the Boeing Little Bird. This platform provided a proxy for the version used for U.S. special operations missions. The Boeing engineers were confident the system was immune to cyber-attacks. They were wrong. Without formal methods integrated into the system, a Red Team of ethical hackers quickly gained access to the helicopter's computer system. From there, they easily gained access to Little Bird's onboard flight-control computer. The Red Team owned the entire system.

To secure the helicopter, the Blue Team applied the same approach developed on the quadcopter into the Little Bird. Following this system modification, the Red Team was granted six weeks to hack the system. **The formal method-generated code foiled all Red Team attempts to crack the Little Bird's defenses.**

After a first round of failed attempts, the Red Team was granted access to a partition on the helicopter that provided a non-critical function (i.e. the helicopter's camera system). Because of the formal methods design approach, the hackers were mathematically guaranteed to remain stuck in that partition...and that's exactly what happened. Even with this initial foothold, the Red Team could not expand access to additional Little Bird systems, nor disrupt operations beyond their initial partition. Once again, the formal methods approach demonstrated that the high assurance code was acting as intended.

The Blue Team repeated this test while the helicopter was in flight with two test pilots at the controls. Again, the Red Team was not able to break out of their partition. In addition to reporting no operational impacts due to the Red Team's access, the test pilots noted no performance consequences to the software functionality. The cyber-hardened Little Bird operated the same as the legacy system. This test confirmed that it is possible to significantly enhance security using formal methods without compromising system performance.

Since the Little Bird test, DARPA successfully applied the HACMS tools to other military systems, including satellites and self-driving convoy trucks. Each successful project has validated the advantages of applying a formal methods approach to software security.

The Role of Secure, Verified Parsers in HACMS

The HACMS program highlighted the critical role *parsers* play in ensuring the security and reliability of software systems. (Parsers are software routines that convert external input to in-memory representations; buggy parsers are responsible for a significant fraction of successful hacker attacks). Key aspects of the role parsers play include the following:

- **Input Validation and Sanitization**: Parsers are essential for validating and sanitizing input data received by software systems. They ingest incoming data, which could come from various sources such as user inputs, sensors, or communication channels. Proper parsing ensures that the data is correctly formatted and free from malicious content or potential exploits like buffer overflows or injection attacks.

- **Defense Against Cyber Attacks**: One of the primary concerns addressed by parsers in HACMS was their role in defending against cyber-attacks. By carefully ingesting and validating input data, parsers help prevent common attack vectors such as SQL injection, cross-site scripting (XSS), and command injection. High assurance parsers are the first-line of defense for maintaining the integrity and security of military systems that may be targeted by adversaries seeking to exploit vulnerabilities.

- **Interoperability and Compatibility**: Parsers also facilitate interoperability and compatibility by correctly interpreting data formats and protocols used in military systems. They ensure that data exchanged

between different components or systems is properly understood and processed, thereby supporting seamless communication and integration within complex military environments.

- **Formal Verification and Assurance**: In the context of HACMS, parsers were subjected to formal verification techniques, which involved using mathematical methods to rigorously analyze and prove the correctness of parser implementations. Formal verification ensured that parsers behaved as intended under all possible inputs, reducing the risk of errors or vulnerabilities.

- **Integration into Secure Software Architecture**: Parsers were integrated into the broader secure software architecture developed under HACMS. This architecture emphasized principles such as least privilege, defense-in-depth, and compartmentalization to enhance overall system security. Parsers played a role in enforcing these principles by securely handling input data and contributing to the overall resilience of the software systems.

Videos of the HACMS demonstrations available at:

- https://www.youtube.com/watch?v=OyqNpn6JpBk
- https://www.youtube.com/watch?v=6cllzGGxRfE

A Deeper Dive into Formal Methods Technology

- Formal Methods
- Parsers
- seL4
- Model-based systems engineering (MBSE) & Design Languages

Formal Methods

Formal methods are mathematically rigorous techniques that create mathematical proofs for developing software that eliminate virtually all exploitable vulnerabilities. These techniques achieve this end by specifying, developing, analyzing, and verifying software and hardware systems. Like other engineering disciplines, the use of formal methods in software development establishes a design's correctness and robustness, which are essential to a system's safety and overall security.

Employing formal methods to improve software security has a rich history spanning more than half a century. Even in the early days of computing, there were efforts directed at mathematical specifications and proof of properties of programs. Several U.S. agencies invested in research in formal methods, motivated by emerging uses of computing software and hardware in critical systems (e.g., space or aircraft flight control, communication security, and medical devices). For decades, however, formal methods tools and ecosystems could operate only on problems and systems of extremely modest scale.

The lack of broad application of formal methods tools was also due to computer science students lacking exposure to formal methods techniques and tools. The techniques were largely considered merely theoretical in nature and lacked tangible benefits that could be practically applied to real-world software engineering projects. However, recent advances in formal methods tools, practices, training, and ecosystems have facilitated the application of formal methods at larger scales. These tools can now be employed in a manner that is affordable and usable by software and hardware engineers.

Benefits

Successfully applying formal methods provides many benefits in the short and the long term, including the following.

- Discover software bugs early, eliminate them, and then verify their absence.
- Reduce the number of design iterations needed to achieve system quality goals, which reduces time-to-market and total cost of development.
- Enable reasoning about design choices early in the process.
- Provide a principled and efficient means to generate test plans.
- Aid in integration efforts, providing stronger and less subject-to-change interface designs.
- Ensure that all system elements are appropriately adapted when design changes are needed.
- Integrate seamlessly with zero trust architectural approaches.
- Generate correct and easily read text-based interpretations that allow stakeholders to quickly understand the underlying formal mathematical models and analyses.[3]
- Automate key elements of process to obtain authority to operate, including establishing high assurance of key properties of requirements, system architecture, design commitments, implementation choices, configuration settings, test coverage, and maintenance operations.

Drawbacks

- Applying some types of formal methods requires a high level of expertise, which can be challenging for software developers without a strong background in relevant areas.
- Training non-formal methods experts (e.g., software engineers and developers) can add time and resources to the development process due to a steep learning curve. However, DARPA's PROVERS program is developing new tools to guide non-experts through designing proof-friendly software systems and reduce the proof repair workload.
- Developers accustomed to traditional software development methodologies may find it difficult to adapt to the rigorous and mathematical nature of formal verification, which creates a deficit in trained users of formal methods.
- Available formal methods tools are less polished and require more significant upfront investment in time and effort compared to traditional software development approaches. However, initial investment is offset by long-term benefits, including enhanced security, reduced development time, and improved software quality.

Parsers

Parsers are software components that serve as the "trusted gatekeepers" of high-assurance software; computer programs rely on them to consume input from untrusted sources. They take input data (e.g. text or binary from sensors) and build a data structure. The data structure is often some kind of parse tree, abstract syntax tree, or other hierarchical structure that provides a structural representation of the input while checking for correct syntax.

Typical parsers are written by hand in languages like C, with inevitable errors leading to security vulnerabilities. DARPA's Safe Documents (SafeDocs) program focused on advancing the state of the art in verification of the security of parsers and eliminating the primary source of parsing vulnerabilities. Researchers developed a methodology and tools to capture de facto specifications of electronic data formats in machine-readable form. The process eliminated format ambiguity and reduced the syntactic complexity of document specifications to levels within reach of automated verification tools. The methodology also created unambiguous definitions that help computers reason about document formats and used automatically generated scanners (parsers that simply

- [3] Note that "formal" does not necessarily mean mathematical and impenetrable to non-expert engineers. Modern type systems illustrate that it is often possible to "hide the math" and create high levels of usability for formally powerful capabilities.

check to see if a document is well formed without creating a data structure) to reject maliciousness and avoid confusion caused by ambiguity.

The program's development of verified parser construction kits was a result of a collaborative effort between big data and machine learning experts, and data format reverse engineering and exploitation experts. This interdisciplinary collaboration was essential in creating verified memory-safe parsers for commonly used electronic document formats.

The successful prototype tools developed under the SafeDocs program have paved the way for capturing and describing the de facto syntax of data formats in human-intelligible, machine-readable form. The insights gained from the development approach, measured results, and accompanying analyses all point to a primary conclusion: SafeDocs solutions should be expanded beyond documents to other file formats. **It is difficult to overstate the impact of reduced systems vulnerability. If every data format was designed with SafeDocs tools, this change alone could eliminate a significant fraction of known software vulnerabilities (roughly 80% of the software vulnerabilities in the MITRE CVE are related to parsing).**

seL4 microkernel

A *kernel* is a piece of software that runs at the heart of any software system and controls all access to resources, including the communication between system components. The kernel is the most critical part of the software system and runs in a privileged mode.

A microkernel is compact compared to a typical kernel, performing only the basic functions universal to all computers. Designed to be integrated into different operating systems, it works with OS-specific servers that provide higher-level functions.

The secure embedded L4[4] (seL4) is a small and simple microkernel, at core managing a collection of *partitions* and the resources they require. It provides the highest assurance of *isolation* between applications running in its partitions, meaning that a compromise in an application running in one partition can be contained and prevented from harming other, potentially more critical applications running in other partitions.

As the first operating system (OS) kernel with some degree of formal verification, seL4 has maintained its position as the leading formally verified OS kernel after more than a decade of continued research and engineering. Its formal verification, combined with its status as the fastest OS kernel on the ARM architecture[5] makes seL4 the ideal basis for high-assurance systems. It demonstrates the level of fidelity and scale formal verification can achieve. It supports multiple architectures (Arm, x86-64, and RISC-V), provides deep security properties, such as integrity, confidentiality, and availability, and comes with formally verified user-level system initialization.

While an OS kernel is a crucial foundation for a high-assurance system, seL4 offers more than that for systems engineering. A simpler, real-time OS might be formally verified but would be insufficient for specific engineering tasks. But the true power of seL4 lies in its ability to scale formal analysis and verification to the much larger code bases that make up entire systems. It does so by providing strong isolation among user-level components. This isolation means that components can be analyzed separately from one another and be composed safely.

Ultimately, seL4 provides the foundation for the soundness of highly automated analysis tools. Because of seL4, the following are all possible:

- Untrusted virtual machines can be run while being securely monitored
- Filtering and monitoring code can be guaranteed not to have been tampered with; and
- The limited communication channels assessed by analysis tools can be guaranteed to be the only ones that are available to the system's components.

---

[4] *L4 is a family of second-generation microkernels, used to implement a variety of types of operating systems (OS).*

[5] *ARM stands for Advanced RISC Machine - a family of reduced instruction set computer (RISC) architectures for computer processors.*

Architectural Analysis & Design Languages (AADL)

Model-based system engineering (MBSE) tools and languages offer a common system design approach. Such MBSE methods have been extended with formal languages and automation, making the development process efficient, rigorous, and repeatable. One of the key technologies that enable the power of MBSE are the system architecture modeling languages. The Architecture Analysis and Design Language (AADL) is a prime example, capturing essential design concepts in complex systems. AADL presents the hardware and software architecture in a hierarchical format, offering a high degree of flexibility and supporting incremental development. This format means that an architecture can be refined to increase levels of detail, providing a clear path for system evolution. AADL's architecture model includes component interfaces, connections, and execution characteristics, but not component implementations. It describes the interactions between components and their arrangement in the system, while keeping the lowest-level components as black boxes. Their implementations can be described separately using traditional programming languages, which may be included by reference in the architecture model. AADL also provides rich semantics and precise run-time specifications, both of which are crucial for high-assurance systems.