Trustworthy Computing: Making Cyberspace Safe—Guest Editor's Introduction

Susan C. Lee

In the 21st century, cyberspace has emerged as the foundation for a new way of life and a new approach to warfighting. As a logical domain not subject to the laws of physics and with a complexity greater than its human creators can understand, this foundation is shaky at best. We are in the early stages of conquering this man-made space, as humanity has conquered the sea, air, and outer-space domains in previous centuries. Over the coming decades, researchers will discover the scientific underpinnings and engineering disciplines that will allow us to build confidently on the cyberspace foundation, with no more concern for its reliability and security than we have for brick-and-mortar structures. Researchers at APL are part of that bold endeavor. This issue of the Johns Hopkins APL Technical Digest *describes their use of formal methods, software analysis, new languages, new hardware, and new protocols to derive the principles and define the techniques that will allow even greater value to be derived from the use of cyberspace, without the current possibility of catastrophic loss.*

INTRODUCTION

An explosion of innovation and investment in the early 1980s and 1990s propelled the United States from a country where computers were arcane devices with no impact on most people to a country where computers are vital to our lifestyle, our critical infrastructure, and even our national security. Although the reliability of computing increased dramatically over this period, security actually decreased as functionality and connectivity grew. Today, the frequency and nature of cyber attacks prompt a well-founded concern that our way of life and national security are at risk.

Most cyber attacks are enabled by exploitable software flaws. Some believe that the risk is created by careless software developers who leave numerous, accidental flaws. Others are more concerned that offshore outsourcing of software development is leaving us vulnerable to malicious code implanted by foreign adversaries. The primary threat, however, is neither offshore development

nor careless software developers. The primary threat is our current lack of technical capability for building and understanding complex software systems. Today's technology cannot reliability pinpoint exploitable flaws or implanted malicious code hidden in millions of lines of functional code.

As the threat from cyber attacks grew, so did the demand for security products such as firewalls, intrusion-detection systems, virus checkers, and particularly at the Department of Defense, "cross-domain solutions" that allow networks at different security levels to be connected safely. Security products are add-ons to the products we use to supply functionality, such as web browsing, e-mail, and document creation. Security products are intended to enforce what the security community calls a "security policy"—that is, a definition of the actions that the computing system should and should not do. These actions range from downloading web pages from known malicious sites to allowing all comers to connect to the enterprise network.

All of these security products share a common flaw, however; they are all large, complex computing systems themselves, executing in the same space as potentially malicious software (malware). Moreover, many of them depend on services provided by other software (such as commercial operating systems) that are known to be exploitable. It is common for cyber attacks to disable security products without any visible, outward sign of tampering. Use of these vulnerable security products induces users to trust their vulnerable applications; however, security cannot be built out of a conglomeration of insecure systems.

Rather than add to the list of vulnerable security products, a small group of APL security researchers chose to focus on the fundamental, underlying issue: how can we build trustworthy computing systems? What combination of hardware, software, and firmware can we trust to enforce a system's security policy? This issue of the *Johns Hopkins APL Technical Digest* discusses the avenues of research and approaches to creating trustworthy computing that are being explored at APL.

TRUST

In the context of computer security, trust may be best described as blind faith; that is, we have an expectation of correct operation without having any evidence of correctness. A "trusted system" is a system whose failure allows a security policy to be broken, but there is no way of detecting its misbehavior. In this sense, most computing systems in use today are trusted systems. Clearly, a system can be trusted, yet untrustworthy. To make a system trustworthy, we need some basis for confidence, some assured evidence of correct operation.

The difficulty inherent in obtaining such evidence can be best understood through a thought experiment. Imagine that we have a system, A, whose operation is monitored and assessed by another system, B. If B does not flag any incorrect operation by A, can we trust that A is operating correctly? Yes, but only if we trust that B is operating correctly. To determine whether B is operating correctly, we need a third system, C, to monitor and assess B's operation. Then, if C flags no incorrect operation of B, we can trust that A is operating correctly—but only if we trust that C is operating correctly. Inevitably, there will be at least one trusted system in every trustworthy computing system. In logic, this would be called an axiom. In trustworthy computing, it is called "the root of trust." From it, we can construct what is called a "chain of trust." In our thought experiment, the chain of trust is C→B→A.

Building a chain of trust is not quite as simple as building a series of assessments. If B is to reliably assess the operation of A, then A can have no means of influencing the operation of B. If it does, then there exists a possibility that A can cause B to fail to detect A's misbehavior. This is the flaw in many of today's security products that was described previously—that is, these products can be influenced by, and sometimes depend on, the very systems they are monitoring. This allows them to be bypassed or subverted, defeating their purpose. The property of "separation," as it is called in trustworthy computing, is as important to building a trustworthy computing system as the correctness of the root of trust.

A chain of trust can operate at many levels. For example, A, B, and C could be executing programs, with A providing functionality and B and C providing security. Or A could be an executing program, B could be a suite of test vectors, and C could be the human who devised the test vectors. Here, the root of trust is a human who we trust to devise a complete and comprehensive test. Separation in the latter case may be provided by independent verification and validation, where the test vectors are not created by the same person who wrote the code.

The goal of trustworthy computing research is to create the principles, architectures, tools, and techniques required to build a trustworthy computing system.

STATIC ANALYSIS

Human constructions in the physical world are constrained by the laws of physics. Human constructions in cyberspace are not. Certainly, the hardware portions of a computing system are subject to the laws of electromagnetics and the electrical properties of materials, but the operation of computing hardware is driven by the arrangement of these materials—an arrangement that in today's chip-manufacturing world is controlled by software. Field-programmable gate arrays (FPGAs) and application-specific integrated circuits are designed by writing a program in some special programming lan-

guage called a hardware description language. This program is interpreted by a computer driving the machinery that creates the specified chips. The chips themselves can be thought of as the designer's program carved in silicon. There is little loss of generality then to claim that proving the correctness of a computing system is equivalent to proving the correctness of software, an entity that is not subject to the laws of physics.

Despite its independence from physical constraints, the operation of software is not without limits. In cyberspace, logic takes the place of physics. No matter how unexpected or odd the behavior of software seems, we can be sure that it is obeying the laws of logic. Although few programmers think they are composing elaborate theorems that can, in principle, be proven to describe some specified behavior, this is what programs are. Formal Methods (FM) are the collection of logical constructs and tools that allow us to prove the theorems we write as programs. Today, the state of the art in FM does not allow most programs to be proven correct. In fact, most commonly used programming languages are not amenable to applying logic to prove properties about the programs, and the programs are far too large and complex.

To address scalability, FM are applied to small, but critical, portions of a program that can be cast in a form amenable to transformation through the application of valid rules of logic. These potions of the program can be proven (or disproven) to function correctly for all possible inputs. Proving correctness for essential operations can greatly increase confidence in the entire program. The first article, by Pendergrass, "Verification of Stack Manipulation in the Scalable Configurable Instrument Processor," illustrates this approach. Here, FM are used to prove that the VHSIC Hardware Description Language specification for an FPGA implementation of the Scalable Configurable Instrument Processor (SCIP) correctly handles stack operation in all cases. The SCIP is designed to execute programs written in a specific programming language that performs the majority of logical and arithmetical operations using the stack; thus, correctly handling stack manipulation is essential to allow programs to perform correctly on the SCIP.

In addition to scaling limitations, FM can suffer from limitations in expressiveness. "Expressiveness" is the ability of a particular formal method to express, or describe, certain aspects of a program's operation. Until recently, existing FM did not express the properties of concurrency or physical–cyber interfaces well. The difficulties in applying FM to these two common aspects of modern programs can be easily appreciated. To prove correctness of concurrent operation, the formal method must include some means of expressing the temporal relationship between two (or more) programs operating simultaneously and independently. The rules of logic must be capable of proving that some correct relationship will hold for all inputs, for all times. In principle, no amount of testing can establish the validity of that claim, and in practice, even large amounts of testing have been found wanting. In "Applying Mathematical Logic to Create Zero-Defect Software," Kouskoulas and Kazanzides describe how they applied a new formal method to proving an algorithm that handled concurrency. In his first attempt, it could not be proven, thus identifying a logical flaw in the algorithm. The way his proof failed suggested an approach to fixing the algorithm, which was implemented and subsequently proven correct.

Hybrid cyber–physical systems also present challenges to FM. Specifically, a cyber system that operates at discrete time steps must correctly follow and control a physical system that operates continuously. The physical system obeys the laws of physics, and its behavior can often be described by a set of equations. A cyber system can only sense the system state and apply controls to change it at discrete intervals. To prove that the cyber system will correctly control the physical system, again for all inputs and all times, a formal method must be able to describe the continuous operation of the physical system and express the (desired) relationship between it and the cyber system. In "Formal Methods for Robotic System Control Software," Kouskoulas et al. describe the application of another recent development in FM to prove a crucial aspect of the control mechanism in a medical system where an error could literally be fatal.

Although either of the errors discovered in these two articles could cause the underlying program to malfunction, there is no reason to believe they are "exploitable," that is, that they can be caused to occur by the intervention of an adversary. At APL, researchers are experimenting with using FM to find exploitable errors, usually called vulnerabilities. The technique begins with the undesirable behavior (e.g., load and execute arbitrary malware) and analyzes the program to determine whether, and—if so—under what circumstances, its logic allows the undesirable behavior to occur. Like other FM, this technique suffers from scaling problems. Current research involves implementing it on a cloud in order to increase the size and complexity of code that can be addressed.

INFORMAL ANALYSIS

FM are the only approach to proof of correctness, but their application is severely limited by scale and complexity at the current time. On the other hand, many good analysis tools are available that will find and expose a significant number of errors in software. These are often far easier to use, and will handle much larger programs than FM, although at the expense of an absolute guarantee. Use of these tools increases confidence that the end product will operate correctly. They can uncover errors in software before testing and, if speci-

fications and designs are created in a formal language, before coding. It is well known that the earlier in the life cycle a flaw is detected, the easier and cheaper it is to fix.

Because APL delivers software—either on its own or embedded in systems—to our sponsors, the Asymmetric Operations Department (AOD) established the Software Analysis Research and Applications Laboratory (SARA Lab) in late FY2012. This facility is envisioned as a place where software and hardware developers across APL can bring requirements, designs, or code for analysis. By doing so, developers gain insights into their products that they would not have had otherwise, and AOD researchers experiment and learn the capabilities and limitations of commercial and research software analysis tools. In "Theory and Practice of Mechanized Software Analysis," Pendergrass et al. describe the future envisioned for the SARA Lab, the tools that are available for use in the SARA Lab today, and some early successes resulting from its use.

DYNAMIC ANALYSIS

So far, only static methods for establishing correctness have been discussed. Going back to the original thought experiment, B and C could be programs assessing the correct behavior of program A as it executes (that is, performing "dynamic analysis"). Some security products, such as intrusion-detection systems, continuously monitor the behavior of network traffic or host behavior in real time. Even when these security products are safely separated from the objects of their assessment, they have limited utility. They are based on precise descriptions of how known malware acts (so-called signatures), heuristic descriptions of how malware acts, or models of normal behavior built through observation of the executing software during a period when behavior is (presumably) good. Products using signatures fail to detect new malware; further, in practice, even known exploits can be slightly modified to escape detection. Because both heuristics and models are an imprecise description of behavior, they must increase their tolerance (threshold) for abnormal behavior, lowering their probability of detection to avoid generating a large number of false alarms.

In contrast, Pendergrass and McGill's article, "LKIM: The Linux Kernel Integrity Measurer," describes an approach that extracts a precise specification of possible behavior from a program's code. A program's code governs how the program structures and evolves memory as it executes. By examining memory periodically during execution, we can detect when its state is not consistent with the code that should be executing; these deviations always indicate that something is amiss—most often the presence of malware. Because the assessment is based on the logic of the desired code, it will reliably reveal the presence of malware, even previously unknown malware, with a very low rate of false alarm. The system implementing this approach is generally known as "LKIM," which stands for Linux Kernel Integrity Measurer, its first application; however, the approach is general and has been implemented for other operating systems.

Today, extracting the specification of possible behavior from code requires a considerable degree of manual analysis. Even so, the technique has been applied to several different operating systems and is currently in service protecting some APL servers. If the initial step of extracting the specification could be more highly automated, the number and variety of applications would grow. In their article, "Ensuring the Integrity of Running Java Programs," Thober et al. describe a programming language that supports that automation. Although the widespread adoption of new programming languages is not assured, a language like the one Thober et al. describe could be used by security-aware developers of highly sensitive software components.

The LKIM approach captures periodic snapshots of memory on which to perform assessment. It can reveal that something unwanted has occurred between snapshots, but it cannot say what occurred, or when it occurred, to a granularity finer than the snapshot interval. In their article, "Analysis of Virtual Machine Record and Replay for Trustworthy Computing," Grizzard and Gardner describe a framework for capturing the behavior of an executing program at the instruction level. This recording of behavior is faithful enough to be "played back" and precisely reproduce the execution that took place. Such a capability can be used to apply a wide variety of assessment techniques to the program's behavior. It could be used retrospectively in a traditional computer forensics application to determine when, and more importantly, exactly how a computer was exploited, revealing the vulnerability. The playback would also reveal exactly what damage the malware did. Although the ability to do such precise forensics is a step forward, the more exciting notion is to play back an execution stream in parallel with, and only slightly lagging, the original program. Time-consuming diagnostics that would introduce unacceptable latency and delay in the original program can be applied to the playback, with the playback catching up with the original during relative lulls in activity. This capability allows for the possibility of detecting and preventing exploits in near real time. Eventually, research could lead to a capability for diagnosing and repairing the vulnerability in near real time and then restarting the original program before the exploit.

Separation

The dynamic analysis systems that are described in this issue of the *Digest* are all subject to malicious interference unless they can be assured separation from the objects of their analysis (the vulnerable programs). Fortunately, commercial producers of computing hard-

ware and software recognize the need to give developers mechanisms that they can use to provide this hardware-assured separation for roots of trust. At the beginning of this century, major manufacturers, such as Intel and Microsoft, formed a consortium call the Trusted Computing Group (TCG) to create specifications for separation support that can be implemented by processor manufacturers and used by software developers of sensitive applications. Their foundational specification is of the Trusted Platform Module (TPM), a separate cryptographic processor and memory that allows establishment of a software root of trust. Today, a TPM has been implemented in nearly all commercial processors and shipped with new products. If your computer is less than 5 years old, it almost certainly contains a TPM.

APL participates on TCG working groups and has contributed significantly to the specification of the TPM. In "Trusted Platform Module Evolution," Osborn and Challener provide background on the TCG and the TPM and describe the newest TPM specification. McGill participates on a new working group for extending the TPM specification to mobile devices. Her article, "Trusted Mobile Devices: Requirements for a Mobile Trusted Platform Module," describes the impediments to implementing a TPM-like capability on mobile platforms and how they might be overcome to create a specification for a TPM for mobile devices.

CONCLUSION

The first steps toward taming cyberspace, and placing it under the control of its developers and users, are being made right now. Advances in software analysis, particularly FM, are needed to replicate the physics-based model used in physical space. Precise models of normal behavior, derived from the functional code itself, are needed to reliably distinguish between good and malicious behavior in executing code. Strong separation techniques are needed to ensure that security products are safe from malicious interference from the objects of their observation and control. APL researchers are making some of the initial steps toward meeting each of these goals. Although not widely known or supported across the community, it is research in these areas that will ultimately make cyberspace safe for further development and use.

The Author

As Chief Scientist of the Asymmetric Operations Department, **Susan C. Lee** is currently employed developing system concepts, technology roadmaps, and sponsor engagement strategies in a domain broadly described as Asymmetric Operations, comprising cyber operations and special operations for national security and homeland protection. Her domain knowledge includes intelligence, information operations, radar propagation, spacecraft development and operation, and biomedical devices. The foundation of her technical skills comes from interactive and real-time software development. She holds several patents, including those for a neural network-based intrusion detection system and a mission impact-based network defense system. Her e-mail address is sue.lee@jhuapl.edu.

The *Johns Hopkins APL Technical Digest* can be accessed electronically at **www.jhuapl.edu/techdigest**.