Self-Checking Software for Information Assurance

Tod Reinhart
AFRL-IFTA
2241 Avionics Cir.
WPAFB, OH
45433-7334

Carolyn Boettcher
Raytheon
PO Box 92426
Los Angeles, CA
90009-2426
MS RE/R7/P575

Roberta Gotfried
Raytheon
PO Box 92426
Los Angeles, CA
90009-2426
MS RE/R1/A521

Mark Kuckelman
Raytheon
PO Box 92426
Los Angeles, CA
90009-2426
MS RE/R1/A520

Abstract

The key to achieving information dominance by the US armed forces is the transmittal of accurate and timely information to the warfighter when and where it is needed. Information Assurance (IA) is the foundation for ensuring that critical information is both accurate and timely. However, innovative approaches are needed to achieve high levels of information assurance for military applications such as command and control (C2). An adaptation of theoretical checker results derived from the research of Dr. Manuel Blum at the University of California at Berkeley has been applied to several real-life applications at Raytheon with promising results. Under the USAF Self-Checking Embedded Information System Software (SCEISS) program, we have begun studying the feasibility of using checkers to enhance the information assurance of a system. In this paper, various problems in information assurance are presented, along with examples of how checkers might be applied as solutions. The feasibility of using checkers to solve these problems is analyzed and the benefits of using checkers instead of, or in conjunction with, more traditional methods of information assurance are assessed. We conclude with our near term plans to demonstrate and validate the use of checkers for information assurance in a realistic C2 application.

1. *Introduction*

The USAF vision to field a C2 capability to the Air & Space commander will be achieved by providing C2ISR capabilities to collaborate globally in support of the National Command Authorities (NCA), all Commander-In-Chiefs (CINCs), services, allies, and the Expeditionary Aerospace Force (EAF). The Dynamic Aerospace Command (DAC) is the overarching concept for all future United States Air Force (USAF) Command and Control (C2) systems to satisfy the C2 vision. This DAC concept is dependent upon the implementation of an architecture that contains technology-based enablers, namely Global Grid and Global Awareness.

Global Grid is the interoperable, virtual networks of defense, national, commercial, and international communications and data systems that will store and move information and provide the assured, high capacity connectivity for commanders and forces to dynamically interact. In addition, Global Grid, the fundamental enabler of the DAC, is identified in the Aerospace C2ISR Campaign Plan 2000 as a focus area. The C2ISR Campaign Plan 2000 Focus Areas establish the C2ISR strategic direction while concentrating on capability requirements and issues. This focus area, which addresses the need for information assurance within the Global Grid, states, “Provide a Global Grid infrastructure to obtain seamless, protected, reliable, worldwide connectivity to support all C2ISR mission needs. The infrastructure must process and exchange information regardless of transport medium, environment or information type and be sustained in a high state of readiness."[1] Part of the strategy identified to achieve this goal is: 1) Improve Global Grid information assurance to guarantee the protection, integrity, and availability of warfighter information, and 2) Implement emerging information assurance techniques to significantly improve levels of protection.

As the system of systems paradigm and interoperability become critical to aerospace dominance within the Global Grid, the assured performance of these systems becomes more paramount. Many of these embedded information systems are heavily dependent upon software that operates within the critical constraint of real-time deadlines in the pursuit of mission effectiveness. In addition, as these systems are designed to be more capable, the software becomes much more complex. As a result, assuring the performance of the mission-critical real-time embedded system software is a significant challenge to the Air Force.

The verification and validation of these systems is particularly difficult because they often produce inexact outputs based on computations from a succession of heuristic and approximate algorithms. Many undetected software errors involve a rare combination of circumstances and events to occur. Because of their rarity, a very large number of independent tests would be needed to create these special circumstances, if they can be reproduced at all, in the laboratory. As a result, these embedded information systems, key components of the Global Grid, often have software residual errors that are not detected and corrected before a system is deployed.

Several years ago, Raytheon surveyed problem reports of errors that were not detected in a production radar system until after radar subsystem integration. The survey classified the types of errors that often remain in embedded systems after subsystem and system integration. It was determined that many of the residual errors resulted from an unusual combination of external circumstances that were unlikely to be encountered in the integration laboratory and that might not even be encountered during operational testing. Therefore, these residual errors might never be encountered until the embedded system is performing as a core component within the wartime Global Grid.

In order to identify and eliminate as many errors from the embedded information systems software as possible before deployment, various methods of validation are performed. Testing is the most widely used method of validating that systems are performing as requirements dictate. However, it has been shown that it is not feasible to use testing alone to validate large systems to a high degree of reliability [Butler 1993]. Therefore, we also comment on alternative approaches for ensuring system correctness. For example, correctness-proving techniques are sometimes used to verify system correctness. However, because of the difficulty of these techniques, it is not practical to apply them to anything but small, well contained portions of larger systems, such as a trusted kernel that is able to guarantee certain properties of a larger system built on top of the kernel.

Another validation approach used to ensure reliable performance is functional redundancy. For example, three versions of software may be independently implemented to perform a given function. The results of each version are compared, with a matching majority declared as the winner. However, there are a number of practical problems with functional redundancy. Often the implementations whose results are compared are not statistically independent. This is because system designers and implementers who are working independently often make correlated errors. In addition, the amount of run-time resources required to execute the function is increased by a factor of three. In systems where run-time resources are tight, this multiplicative increase may not be acceptable.

How then can residual software errors, which significantly increase the system's cost of ownership and may contribute to mission failures, be significantly reduced or effectively eliminated? After consideration of the alternatives previously discussed, we concluded that an entirely new paradigm was needed to ensure that large, complex systems with limited run-time resources can be maintained in a cost effective manner so that they continue to operate correctly.

To improve the validation process and ensure the operational correctness and integrity of critical embedded systems, the Air Force has invested in a research program investigating self-checking software. The self-checking software, also known as results checking technology, continuously monitors itself to detect suspicious events. The technology is based on checker software that executes at critical points in the embedded software to check the correctness of the deeply embedded algorithms. Because the checkers are inserted during development and remain in the operational software, checkers can detect errors during all phases of testing and operational use. In addition, checkers can detect erroneous results that do not produce obvious symptoms at the system level and thus, might easily be overlooked by testers.

Embedded information systems operating as a component of the DAC and providing information to C2 nodes must perform as a publisher or subscriber of secure, reliable information. These systems require additional validation methods to ensure the operational correctness and integrity of the system to achieve the goals of the USAF C2 vision. The SCEISS program is applying checker technology to a range of embedded information system applications with the expectation that this technology will result in a dramatic improvement in the reliability of fielded software. Based on early results demonstrating that checkers are able to detect “suspicious” events as they occur in mission-critical systems, we have begun applying checkers to provide improved information assurance for software-intensive systems that are vulnerable to accidental and malicious attacks. Our goal is to use checkers to detect both types of attacks as they are occurring so that effective action can be taken to counter such threats. The results checking technology we will now discuss is a proven validation method to assure the performance of the embedded system and is a potentially viable method to verify the security of its information.

2. *Prior Research in Self-checking Systems*

Universities, Raytheon, and the Air Force have sponsored considerable prior checker research. The seminal research in checkers has been ongoing for more than 10 years, led by Dr. Manuel Blum at the University of California at Berkeley, who defined the results checking paradigm that is the foundation for the SCEISS effort [Blum and Kannan 1989]. Several years ago, Raytheon began a small checker demonstration in conjunction with a production radar upgrade program [Boettcher and Mellema 1995]. Two of the three checkers developed as part of the demonstration detected errors that were not discovered until flight test. This result further convinced us of the value of self-checking software in an overall system development process.

A checker is special software that is embedded in code to continually check results over a large number of executions. It must have a good probability of eventually detecting any suspicious events or conditions, while maintaining a very low probability of false alarm. To be called a simple checker, it must be much simpler and more reliable than the original software being checked and add only a small amount of execution and memory overhead. The fact that the checker is simpler than the original software is also evidence of its statistical independence**.**

2.1 Checking for the Pentium Division Bug

The division bug in Intel's original release of the Pentium processor provides an excellent example of the potential value of checkers and illustrates that checkers can be useful for detecting anomalies that may be caused by hardware errors, software errors, or even data tampering. The Pentium division bug corrupted results very rarely, less than 1 in every 8 billion inputs. Nevertheless, users of the Pentium processor soon discovered it and Intel was forced to recall the “buggy” processors and correct the error.

When news of the Pentium division bug was published, Blum and Wasserman saw it as an opportunity to illustrate the power of the checker paradigm. They invented a combined software checker and corrector that provides an “a priori” solution to the Pentium bug [Blum and Wasserman 1996], which both detects and corrects the erroneous division result. This Pentium division checker/corrector does not even need to know what caused the data to be erroneous or even that an error is present.

2.2 Performance Checking Demonstration

Early in the SCEISS program, we applied checkers to a production radar upgrade program that was adding a new radar mode to the existing software [Reinhart, et. al. 1999]. The software for the new mode was based on existing software from another radar program, but had to be substantially modified to adapt it to a different platform. The SCEISS team analyzed the software requirements for the modifications and identified candidate functions for checking, eventually selecting the constant false alarm rate (CFAR) control loop.

The CFAR control loop is an example of an algorithm where there are no definitive rules for determining if a result is right or wrong. Rather, the CFAR loop seeks to establish stable values for the CFAR threshold so that the system detects as many real targets as possible, while rejecting false targets and maintaining a low false alarm rate. As a result, our CFAR checker looks for "suspicious" results that probably indicate significant system performance degradation, rather than "wrong" results.

In general, CFAR control loops seek threshold values that keep their systems operating in an optimal manner, even while the environment, as measured by the sensor, is changing. Based on feedback from the sensor, the threshold must be continually adapted in a stable manner to changing environmental conditions. To help maintain stability, the requirements for the CFAR control loop specify a band of allowable values for the threshold. If the calculated value is outside of the specified range, it is reset to the maximum or minimum value as appropriate.

To get statistics on the expected distribution of threshold values and help quantify "suspicious" values, we first simulated the CFAR threshold calculation 100,000 times using randomly generated input data representative of that seen in a real system. Based on those statistics, we decided to set the checker to fire at constant values close to the minimum and maximum. In addition, we allowed the checker to fire only when the values were suspect over 90% of the time during a representative time interval.

Checking was performed in the system integration laboratory in parallel with the regular system integration effort, so the program's flight test schedule would not be negatively impacted by the checker experiment. When the checker was run in the integration laboratory prior to flight test, the threshold values were consistently low, causing the checker to fire almost constantly under several different test conditions. The software development team responsible for the production tape upgrade was informed about the problems with the CFAR loop uncovered by the checker and subsequently corrected the application code. The checker was run with the corrected code for the final time in the integration laboratory after the system had been in flight test for some time. In the final checker demonstration, the checker fired considerably less often than before, providing evidence that the CFAR loop performance had been considerably improved.

This experiment illustrates the value of even very simple checkers. With the relatively insensitive checker described here, we were able to detect suspicious events occurring in the system which were not being detected by any other means in the laboratory. When a checker was included in the software, it fired and immediately informed the tester that there was a problem with the system performance. As a result, the software was improved before expensive flight test hours were wasted.

3. *Research in Information Assurance Checkers*

Information assurance (IA) is defined as information operations that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and non-repudiation. This includes providing for restoration of information systems by incorporating protection, detection, and reaction capabilities. We define an *IA checker* as one that regularly performs some service in the area of confidentiality, integrity, availability, authentication or non-repudiation. For example, an availability checker would regularly verify that a system resource is not being dominated in an undesirable way by any process. An authentication checker would verify a user's identity on a continual basis, while an integrity checker would validate code and/or data blocks regularly. In the following sections, we present some ideas for candidate IA checkers, discuss the potential benefits of each, and assess their feasibility..

3.1 *Authentication Checker*

Authentication is defined as a security measure designed as a means to establish the origin of a transmission, message, or originator, or to verify an individual's identity. A typical authentication method is to use a password that cannot be easily guessed by an unauthorized user. The use of biometrics, or measurable biological characteristics, is a more sophisticated method for authentication. In computer security, biometrics refers to authentication techniques that rely on measurable physical characteristics that can be automatically checked. Examples include computer analysis of fingerprints or speech.

Although authentication techniques in use today vary in complexity and effectiveness, a common attribute is that the process occurs at initiation of a user or process requesting a resource. A typical authentication consists of a one-step process, specifying the value of a single parameter, usually a password. An obvious vulnerability of one-factor authentication is that an intruder only has to defeat one layer of defense to gain access to the system. Another weakness is that users often leave consoles unattended while still logged on, allowing an intruder with an easy opportunity for misuse. A less common problem occurs when a user's screen is visible to a sudden visitor. It would be beneficial to have an easy method for the user to blank the screen, possibly by just looking away from it.

An authentication checker would improve on the one-step authentication process by periodically re-checking that users are indeed who they claim to be. A checker would challenge the user on a *continuing* basis, increasing the chance of an impostor being detected. Two ideas for an authentication checker are described here: the first based on biometrics, the second using continual client profile challenges.

The basis of the first idea, biometrics, is increasing in use today as an additional means of authentication to supplement the traditional password method. A biometric authentication checker would continually check physical attributes of the user to ensure his or her identity. This approach adds assurance to the system by continuing the authentication process indefinitely. It would be especially valuable in high-security areas to verify that the present operator is the same one that was originally authenticated upon log-in. In this case, operator proximity would be checked at intervals to determine whether he has moved away from a console. Another feature might be blanking the screen whenever the operator looks away from it. Possible biometric methods include retinal, iris, or face verification.

The second idea, client profile authentication, consists of continual challenges to a user at random intervals, each querying for a (new) piece of private information. Similar to the biometric checker, the benefit of the client profiler is that the authentication process never ends. An additional benefit of this approach is that, although one piece of information might be compromised at some point, it is less likely that many pieces of information would have been obtained by the perpetrator. Examples of this additional challenge information are SSN, mother's maiden name, hometown, pet name, year of birth, and year of parent's birth.

Example. Consider the case of an information system in a hospital emergency room where medical personnel need rapid access to patient information in life or death situations. However, patient records contain sensitive information, including medical history, address and billing information that passersby might be able to view if it isn't carefully protected. This poses a difficult security challenge.

In one real-life situation, a teenager obtained the name and address of several patients from an unattended terminal in an emergency room. He later wrote letters to each of the patients, pretending that he represented the hospital, and informed them that their blood samples tested positive for HIV. Numerous lawsuits resulted.

A biometrics-based authentication checker might have prevented this regrettable situation through the use of a continual retinal scan of the user. Once the checker had detected that the nurse or doctor had left the terminal unattended (perhaps hurrying to another patient in distress) the screen would have been blanked. Another benefit of this type of authentication checker is that the user can blank the screen (or perhaps remove sensitive information from it) merely by looking away from it. This would be accomplished by the retinal (or iris) scanner failing to detect the valid user's biometric during the next check. The advantage of this feature is that it allows the user to protect information with a nearly passive action - glancing away from the screen - instead of having to pressing a button or a key combination. Sometimes the effectiveness of a security feature is how easy it is to use it.

Benefit/Feasibility. Checkers based on biometric authentication push the state of the art to such a degree that we recommend postponing further research and development at this time. Although the field of biometrics is advancing rapidly, it is still in its infancy. In addition**,** there are still significant spoofing problems with iris and face recognition. However, this is an interesting area for checker research when the field of biometrics has matured.

3.2 *Security Configuration Checker*

Although security capabilities on workstations have improved recently, ensuring the ability and motivation of users to properly configure these capabilities has not been successful to date. People neglect to select appropriate settings, or set them temporarily to one state, then forget to set them back. An improper security configuration represents a vulnerability. A checker that continually verifies that these settings are consistent with the system security policy would be useful and add to the overall IA of the system.

A security configuration checker would continually check the settings on a workstation or group of workstations within a domain to ensure that the settings conform to the organization's security policy. The checker would fire when inappropriate configurations were detected. This is challenging since security configurations vary widely depending on system type (NT, Unix, Sun Solaris, HP-UX) and mission. A refined checker of this kind could be extended to firewalls, routers and other configurable devices. To illustrate, we describe the application of a security configuration checker to web browsers.

Example. Most organizations have greatly increased their usage of the internet in the last few years. This fact, coupled with the vulnerabilities introduced with downloading untrusted software (including Java applets), suggests that it would be beneficial to use the checkers approach to continually verify that workstations are in their safest configuration. In a related example, automatic virus checkers have become widely used in the last few years to significantly lessen losses incurred from downloaded viruses. It is possible that a similar benefit would be achieved through use of a *browser security configuration checker,* as outlined below.

Checkers can be used to monitor the security settings on an Internet browser, including the Java security settings, which are often left with the default value. This is a security vulnerability, since default settings are often "no security" or "medium security" (in the case of Internet Explorer). Browser configuration files are accessible to an administrator, since personal computers store their browser settings in a shared file. The system administrator's console could use a checking mechanism to periodically verify that security settings are consistent with the system security policy. In the case of Netscape, a typical path to the preferences file is *c:/program files/Netscape/users/(user's name)/prefs.js.*

Access to the file is a security policy issue that is dependent on the operating environment. If the system is a Windows NT system running on a FAT partition, then all files are accessible to the outside world. If in NTFS, the file may not be accessible, but can be made so by the user. If a user's settings aren't available to the checker, this would also be flagged as an illegal situation.

Benefit/Feasibility. A security configuration checker would be quite useful, primarily for its convenience to users. Again, simplicity of security features seems to make them more effective, since they are used more often. Additional benefits of this checker are that it runs with low overhead, can easily be changed to reflect a modification to the security policy, and can regularly ensure that browser configuration settings are maintained at an appropriate level. However, a number of security scanning tools are available today that perform similar functions, although they are sometimes difficult to use and are error-prone. Performing this function within an OS or embedded in an application seems to provide only a marginal increase in utility over these security scanning tools.

3.3 *Code Integrity Checker*

Integrity is the quality of an information system reflecting the correctness and reliability of the system software; the logical completeness of the hardware and software implementing the protection mechanisms; and the consistency of the data structures and occurrence of the stored data. Note that, in a formal security definition, *integrity* is applied more narrowly to mean protection against unauthorized modification or destruction of information. Code integrity is the property that software has not been altered or destroyed in an unauthorized manner.

Integrity is one of the cornerstones of information assurance. In any system, it is critical to assure that data is valid, is not harmful, and has not been changed in an unauthorized fashion. A checker that could provide any of these assurances would be useful. Code modification is often the means whereby an unauthorized person gains access to a system. A checker would continually verify that important code segments remain unmodified.

A code integrity checker would execute inside the OS or security kernel on a regular basis to ensure that no unauthorized modification has occurred on critical blocks of code. Such code modifications can occur by malicious intent or as the result of a fault or error. The checker verifies the integrity of a larger unprotected security kernel, an entire OS, a subset of critical modules, or a combination of these.

A simple example of a checker verification method is the computation of a simple cryptochecksum upon initialization, which is then regularly re-checked to ensure there has been no intentional or unintentional corruption of the code block. It could even be a weak checker that occasionally checks a very small portion of the OS or trusted code block. The point is that it executes so often that it eventually will detect the existence of rogue or corrupted code.

Example. During the mission load of an avionics Operational Flight Program (OFP), a baseline cryptochecksum is calculated for critical subsets of code, including the security kernel. At regular intervals or after key events during the mission, the code segments are checked to ensure that they have not changed. If a code segment's hash is ever found to mismatch the original, the checker fires. In this way, critical portions of the OFP have increased assurance against modification.

Due to time constraints of real-time systems, it might not be possible to use full cryptochecksum algorithms, such as MD-5 (Message Digest-5) or SHA (Secure Hash Algorithm). A simpler checksumming method can be constructed that is quite effective and requires much less execution time. Of course, the trade-off here is that the more elegant hashing algorithms provide better protection against tampering, but require more time to execute. Each implementation must evaluate this trade-off.

Benefit/Feasibility. The area of integrity checkers appears very promising because cryptochecksumming technology is a mature field. The idea of regularly verifying the validity of a system's security kernel and important programs is both appealing and feasible. In addition, an integrity checker is portable, since the fundamental idea of hashing is well-understood and standardized.

3.4 *Data Integrity Checker*

Data integrity is the property that data has not been altered or destroyed in an unauthorized manner. The integrity of data is an important IA issue because data modification is often the means whereby an unauthorized person undermines the functioning of a system. A data integrity checker would continually verify that important data segments remain unmodified throughout their useful life. Its basic principles are similar to the code integrity checker.

A Data Integrity Checker would verify that selected blocks of data remain unaltered subsequent to reception, or during any period where modification would not be authorized. The protected data block might have been modified in an unauthorized fashion through malicious intent or as the result of a failure. The data integrity checker can verify: 1) selected data blocks; 2) an entire database; 3) a subset of critical data blocks that are meant to remain unchanged; or 4) a combination of the prior choices

The choice of verification method for this checker would be driven by execution time as a primary qualifier. Standard algorithms for SHA-1 or MD-5 are available for the common platforms of today, but these might be too time-consuming to be of use in a checker. A data integrity checker might implement a simplified cryptochecksum algorithm, which gives an effective hash with very low computational overhead.

Example. As an example, consider the operation of a distributed simulation exercise, consisting of a number of military vehicles and weapon systems. Current simulation technology permits the individual vehicle simulation units taking part in an exercise to be remotely located. In order to maintain proper vehicle orientation and status, a packet of information is submitted by each member of the simulation at regularly-spaced intervals. For simulation purposes, a certain data update rate is required, typically on the order of two seconds, in order to synchronize vehicle movement and operator actions.

During initialization of the exercise, a common terrain database is selected by the instructor, and the appropriate vehicles are initialized. The exercise parameters are established and the simulated exercise begins. The exercise includes the movement of tanks and vehicles in a simulated tactical environment, with communications existing between participants and a commanding officer, all coordinated by the simulation instructor. At any time the instructor can choose to stop the simulation by pressing the FREEZE button, but can continue to communicate with any participant in the exercise. The simulation progresses with each simulation system accessing the common terrain database for cues and information. Each individual trainee station is a separate computational system linked to appropriate hardware that resembles the actual tactical vehicle or weapon.

Now consider the case of an error occurring during the simulation. At some point, an anomalous visual effect appears on the screen of one of the vehicles, distracting the operator. After a few confusing moments, he reports the situation to the simulation instructor, and the simulation exercise is halted. They then spend a bit of time trying to duplicate the anomaly in the database, but are unable to. This problem occurs infrequently over a period of weeks, and finally the situation is analyzed and duplicated. The cause is a process in the simulation has a software bug that corrupts the terrain database in some minute way. Only one memory location is corrupted, but it is enough for a negative training effect, due to the unrealistic terrain characteristic that it causes. After several months of trying to duplicate the problem and research, the erroneous data is located and corrected.

Now consider the above situation if a data integrity checker had been in use. At the outset of the simulation, the terrain database appropriate for the exercise would have been loaded into memory of each simulator's computer. The checker would have scanned the entire terrain database, and determined a cryptochecksum (hash) for given data blocks. At this point, the hashes would have been stored in a protected part of memory for later retrieval. Other data items are similarly checked, hashed and stored. During the simulation, then, the checker verifies that the local terrain database load is uncorrupted by performing a hash on it, and comparing it to the hash in memory. Upon mismatch, the checker fires and the simulation is halted, allowing forensics to study the origin of the error.

In the normal case, the checker will not detect a problem. As the simulated vehicle moves through new portions of the database, new chunks of digital terrain are loaded in from disk. The disk transfer is not complete until the checker has computed a new hash for the data block, and has stored it safely away. Each new disk access activates the front end of the checker, which then computes the hash. The memory block is re-verified regular intervals (for example, every 2 seconds), and the simulation continues. Although the checker process is quite computation-intensive, and the computational system is already busy trying to keep up with the real-time demands of the tactical simulation, one might wonder how a checker can find the processing time it needs to perform its duties. The answer lies in the increase in computational speed that's been continually appearing in industry. While better simulations (visual fidelity, realism of vehicle movements) will be possible with higher processing speeds, it is also key that the reliability and error logging of systems will improve also. A checker is an ideal approach to verifying the correct system functionality.

Benefit/Feasibility. Integrity checkers are promising because cryptochecksumming technology is a mature field, complete with efficient algorithms. The idea of regularly verifying the validity of a system's important data structures is both appealing and feasible. In addition, an integrity checker design is fundamentally portable.

3.5 *Data Validity Checker*

In many systems, data is delivered in frames that are either related to previous frames of data or have inherent data correlation qualities. Such data frames are vulnerable to a perpetrator that might intercept data and modify it for his own motives. In many cases, frames can be validated either by comparison to previous values or by comparison of qualities of the current frame. For example, a frame of video data can be determined to be valid when compared to certain aspects of a previous frame. Similarly, radar data can be checked to determine various degrees of validity.

A data validity checker would verify a data block (for example, input data) in a cursory check to see if it's within an allowable range, obeys the laws of physics, or is consistent with correlatable portions of other data available.

Example. As an example of how the data validity checker might operate, consider the networked simulation example discussed in the previous section. During the simulation exercise, packets of data are exchanged amongst all vehicles participating in the exercise. This packet delineates the position, orientation, firing status and appearance of the vehicle, in addition to other qualities. As the packet is routed to each of the other simulators taking place in the simulation, the information is extracted and used to update relative positions, depending on the ownship orientation and viewing capabilities, as well as details on such items as weaponry. Currently, if anything is out of order, that is, if another vehicle has moved too quickly, or jumped several yards in position, or somehow violated the laws of physics, the simulation continues. This is a prime place for a checker to be inserted that would check that each vehicle in the simulation operates according to the laws of physics, instead of blindly processing the input received from the packets sent from each node. In this example the frame of data is compared to data from a previous frame of data to verify its validity.

Benefit/Feasibility. Although it would be valuable to validate data packets, this type of checker has limited portability due to its application-specific nature. Each application has its own classes of valid data, and varies widely from platform to platform. At this time, our preference is to focus on an area with greater portability, to enhance its usefulness both to the US military and to industry. Data validity checkers are currently being studied in other parts of the SCEISS program with some success.

4.0 *Summary and Future Plans*

Information assurance is critical to achieving the USAF C2 vision of providing C2ISR capabilities for global collaboration. In this paper we have described how self-checking software can be applied to help provide information assurance for critical embedded information systems operating as a component of the DAC. Several types of IA checkers were described, examples were given, and their benfits and feasibility were assessed. Code and data integrity checking were shown to be the most promising focus areas for the current IA checker effort due to their perceived benefits, their portability and general applicability across application domains, and the maturity of the underlying cryptochecksumming technologies used to implement them.

Work on IA checkers is proceeding to identify detailed requirements for an integrity checker. These requirements will be used as evaluation criteria during subsequent proof of concept experiments. We are also currently in the process of selecting an application in which to demonstrate the benefits of the integrity checker. Once the integrity checker requirements are documented and a target application is chosen, the proof of concept experiments will be conducted. These experiments will demonstrate the behavior of the checker in a controlled testbed environment in preparation for integrating the checker into a real-life C2 application.

5.0 *References*

[Butler and Finelli 1993] "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software", IEEE Transactions on Software Engineering, Vol. 19, No. 1, January, 1993.

[Blum and Kannan 1989] "Designing Programs that Check their Work", Proceedings of the ACM 21st Annual Symposium on the Theory of Computing, May, 1989.

[Blum and Wasserman 1996] "Reflections on the Pentium Division Bug", IEEE Transactions on Computers, Vol. 45, No. 4, April, 1996.

[Boettcher and Mellema 1995] "Program Checkers: Practical Applications to Real-Time Software", Test Facility Working Group Conference, 1995.

[Raytheon 1998] Self-Checking Embedded Information System Software Interim Report, December 1997 - November 1998, Los Angeles, CA.

[Raytheon 1999] Self-Checking Embedded Information System Software Interim Report, December 1998 - November 1999, Los Angeles, CA.

[Reinhart et. al. 1998] "An Automated Testing Methodology Based on Self-Checking Software", NAECON Proceedings, IEEE, July, 1998.

[Reinhart et. al. 1999] "Self-Checking: Improving the Reliability of Mission Critical Systems", IEEE/AIAA Digital Avionics Systems Conference Proceedings, Oct. 25-29, 1999.

[Shreve et. al. 1997] "Real-time Checkers: Built-in Test for Mission Critical Software", IEEE/AIAA Digital Avionics Systems Conference Proceedings, Vol. I, October 26-30, 1997.

[Wasserman and Blum 1997] "Software Reliability via Run-Time Result-Checking", Journal of the ACM, Vol. 44, No. 6, November, 1997.

6.0 *Definitions*

Although many definitions are included herein, a good source for the definition of infosec terms can be found at website **http://www.cultural.com/web/security/infosec.glossary.html**.