Towards a Trustworthy Foundation for Assured Autonomous Systems

Mr. Thomas Macklin

Dr. Paul West

US Naval Research Laboratory
Charleston Southern University
USA

thomas.macklin@nrl.navy.mil; pwest@csuniv.edu

ABSTRACT

Many modern information systems mitigate cyber security risks by making use of commercial software security technologies that continually monitor behaviour and apply updates. Unmanned and/or Autonomous (UA) system implementations prove substantial challenges for both of these mitigation strategies, as network connectivity is limited and unreliable, and system complexity renders patching impractical outside of a routine update process. The task of performing security assessments on UA system implementations is a challenging proposition as UA systems often lack the interfaces by which traditional software testing is applied (e.g., a network server or a human-machine interface.) Further, many control decisions must be made automatically in the UA system instead of being made by a human. In this paper we provide an in-depth characterization of an alternative, proven approach to UA system design and implementation that reduces risks and costs associated with these challenges. These benefits are realized by making use of technologies and techniques that promote desirable properties such as correct-by-construction, least privilege, and least functionality while doing so in a manner that more easily facilitates security assessments.

1.0 INTRODUCTION AND CONTEXT

Unmanned and/or Autonomous (UA) systems frequently operate in environments where defensive countermeasures are scarce, both with respect to kinetic threats and with respect to cyber security threats. In order to ensure that a UA system can successfully execute its mission, developers must provision these systems with adequately robust cyber defenses such that they can prevent adversaries from compromising the execution control flow of programs within the platform. Unfortunately, cyber defense tactics that are common in typical information systems are difficult to implement in UA systems. Due to the limited bandwidth available during a mission, continuous patching and updates are often impractical, and intrusion detection systems are difficult to monitor in real time. Because the software systems used within a given UA system are typically highly customized, they are frequently not supported by traditional security scanning tools (e.g., NESSUS, STIG) (1). Further, UA systems tend to require higher levels of determinism, and are also frequently constrained by CPU and memory limitations. Even if one overlooks the limited benefits associated with near real-time security scanning (e.g., endpoint protection systems), limited system resources (e.g. memory, CPU) render implementing such security controls implausible. Lacking access to the software security solutions that are used in enterprise IT systems, UA system developers must consider constructing a secure, modular, and testable runtime environment in order to achieve robust cyber security within a UA system. We propose the use of low-level technologies such as bare metal hypervisors, separation kernels, unikernels, real time operating systems (RTOS), and carefully constructed Linux

implementations to run software according to the needs of each particular application. These runtimes can then be provisioned to make use of message passing via shared memory regions in order to facilitate partitioned information flow control amongst different software functions that are governed by different safety, security, and reliability policies.

In order for a secure runtime to be practical, it must facilitate the implementation of software that effectively implements the functionality desired by the system developers. Similarly, a secure runtime must allow developers to leverage existing Commercial Off The Shelf (COTS), Government Off The Shelf (GOTS), and open source software components when developing and integrating solutions, as frequently existing software has been rigorously assessed and ultimately approved to serve a specific purpose within a system. We propose a development model that makes use of message passing as the sole means for software components to invoke behaviour in other software components (e.g., libraries, processes.) This approach enables integrators to compose systems using heterogeneous technologies and development approaches, while simultaneously enabling developers to use tools and languages that are most suitable to their needs.

The decision control system that directs the actions of the UA system present new security concerns. This system must be pre-programmed in the UA system before deployment to recognize the environment from sensory inputs. For example, a UAV may have multiple sensors (e.g., infrared, heat) that it uses to recognize troops and ground vehicles to provide useful reconnaissance. In this case, both the sensors and the machine learning recognition software that processes the inputs both provide adversaries with new attack vectors.

Finally, we consider the problem of system assurance. For any given system, claims of robust security must be backed up by measurable evidence in order for stakeholders to have confidence in the presence of that security. In order to generate such evidence of security, a system must be subjected to comprehensive testing and analysis, which typically must include independent 3rd party testing and analysis. Considerations for how UA systems can be designed and provisioned for these assessments is briefly described.

2.0 PROBLEM STATEMENT

Unmanned and/or Autonomous (UA) Systems perform increasingly critical roles within today's military operations. As operations become dependent on UA systems for success, it is essential that self-protection countermeasures implemented in these systems evolve at a more rapid pace than countermeasures that are meant to degrade UA system effectiveness. With respect to cyber security, current best practices for self-protection often are insufficient to counter advanced threats. Some general threats and vulnerabilities that frequently apply to UA systems will be briefly described.

1. Communications- UA systems often operate in environments that are denied, disconnected, intermittent, and limited (DDIL) (2). UA systems are typically designed with these limitations in mind and have countermeasures to handle planned degradations. In cases where the countermeasures are insufficient, attackers can adversely impact mission effectiveness.
2. Cyber Security Self-Protection- UA systems typically use a variety of communications systems with external interfaces. Attackers can use a variety of tactics such as (3) to attempt to exploit software vulnerabilities and eventually gain access to sensitive data and software within the UA system.
3. Implementation Defects- UA systems often make use of complex software components that are difficult to rigorously assess. If attackers can get the UA system's software to process maliciously formed inputs such as (4), then the attackers could take control of the software via well understood software exploitation techniques (5).

4. Machine Learning Subversion – Since some or all recognition decisions are tasked to subsystems within the UA system instead of to a human operator, these software components must be programmed to 'recognize' phenomena relevant to the system's purpose (e.g., surveillance.) With the advent of recent machine learning research, real-time target recognition has become feasible. To protect this recognition software, we must address how to secure the different components of a modern machine learning system.
5. Ineffective Assessments- Security assessments (e.g., penetration tests, Common Criteria) have traditionally focused on traditional information systems such as personal computer local area networks. As UA systems currently are heavily reliant on sensors and networks, techniques for assessing the risks associated with these dependencies must also evolve. If the military cannot adequately assess cyber risks associated with UA systems, then they risk overall mission failures due to an undetected cyber security failure within a mission critical UA system.

In order for a UA system to maintain effectiveness throughout its life cycle, each of these vulnerabilities must be adequately addressed. In this paper, we briefly describe techniques that have a demonstrated level of effectiveness in countering these threats. Each technique is enhanced with a practical example. While these examples are informed by real-world implementations, they are not specifically related to any particular program known to the authors.

2.1 Example: A Notional UA System

Throughout this paper, we will refer to a notional UA system in order to illustrate key concepts. This system will be an Unmanned Aerial Vehicle (UAV) with the primary mission of surveillance for high value targets using a camera. This vehicle will make use of a variety of passive sensors for input, a classifier program[1] that analyses input from these sensors, radio frequency (RF) communications equipment for network connectivity between the UAV and the remote operators, and software that controls a camera based on inputs from the classifier[2]. Further, we will assume that a program office responsible for developing the UAV is planning this system as a major upgrade to an existing system that is deployed on a manned aircraft. While major components of the system will be new (e.g., the classifier, mission control), it must reuse major hardware and software components of the legacy system (e.g., passive sensor control, camera control) . This system must be designed and implemented in a manner that meets notional mission capabilities, but must also maintain system security despite a robust threat profile. Logically, the information flow for this system is depicted in Figure 2-1 Notional UAV Architecture.

[1] For this paper we define a classifier as an algorithm with a data control map that classifies (recognizes) incoming signals/data. See section 2.2 (Classifiers) for details.

[2] Any similarities between this notional UA system and any real-world platforms is coincidental. This example is designed solely for illustrative purposes.

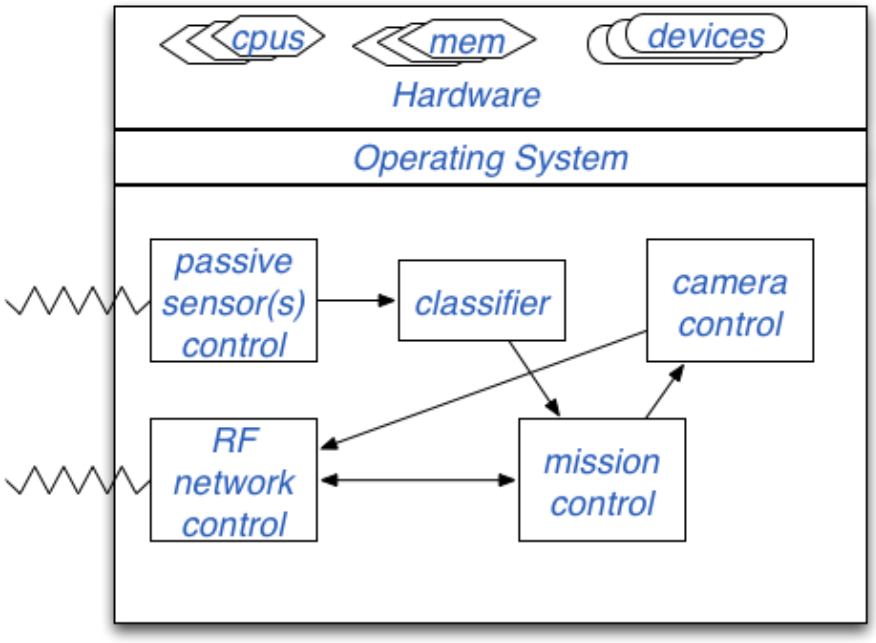

Figure 2-1 Notional UAV Architecture

2.2 Classifiers

Prior to describing this system implementation approach, we will briefly characterize the central component of the autonomous subsystem: the classifier. Subsequently, the classifier will be referenced throughout the paper as appropriate. UA systems must make decisions based on their sensory inputs. What was once handled exclusively by a human controller (e.g., a pilot) has transitioned partially or in some cases completely to a computer system. The crux of this recognition software is developing a system to classify signals to provide valid input for control software. Recent developments in machine learning have proven both accurate and real-time recognitions systems can be developed for certain domains (6). With the addition of a practical 'deep' machine learning system (7), a general-purpose tool has been created for generating recognition systems for a variety of purposes. The core of both conventional and 'deep' machine learning are the classifiers used to recognize inputs. Modern machine learning classifiers are composed of their algorithm and algorithm parameters. Because the parameters are critical to the effectiveness of the classifier, its security must be assessed. For example, (8) trained a neural network to detect Diabetic Retinopathy in a human eye, and then sent a training set into a machine learning algorithm that trained a neural network to identify the need of treatment with > 99% agreement with ophthalmologists. During training, the machine learning algorithm produces weights (i.e., the parameters) that will be fed to the final detection neural network algorithm. In this case, one way for an attacker to manipulate the classifier would be to modify the detection algorithm or the parameters (the weights in this example.)

In our example UAV, the classifier program will use inputs from the passive sensor controllers to classify different phenomena. It then provides inputs into the mission control subsystem. These inputs would be then used to automatically control the camera, to provide enhanced situational awareness to a human ground operator, or perform a combination of the two actions.

3.0 CYBER SECURITY: TECHNICAL APPROACH

Layering security assurance throughout a system has proven an effective technique in developing secure systems, as is described in (9). In keeping with this approach, we propose implementing four key layers of system security within the system[3]. At each layer within the system, in order to maximize system security, it is necessary to minimize attack surfaces (10). The four layers are:

1. Domain Isolation
2. Functional Implementation
3. Behavioral Integration
4. System Evaluation

3.1 Domain Isolation

UA systems have different domains of criticality with respect to confidentiality, integrity, and availability. For example, a steering control system may be critical with respect to integrity and availability, while an image classifier may be sensitive with respect to confidentiality and integrity. In initial system design, each domain in the system should be isolated from all other domains within the system. This isolation enables system designers to tailor security policies according to the requirements for each domain and in turn to implement robust partitioned information flow control policies between the disparate domains by explicitly defined shared resources as is described in (11).

One mechanism which this isolation achieves is a separation kernel (SK) such as (12). In the context of an SK, each domain can be governed by a security policy that is to be tailored to its requirements and may make use of a runtime that is best suited to implement that policy (e.g., RTOS, Linux, unikernel, FPGA.) The overall system security policy is a composite of the disparate security policies and the partitioned information flow control policy that governs the interconnections between these domains.

From a top level, our example UAV has three separate domains: passive sensor and RF network control, classifier software, and mission control. Each one of these domains has different functional requirements, different security policies, and different runtime requirements. Initially, each domain is isolated from the others in the system. Each domain is tailored to meet its specific requirements. In our example, the classifier and the mission control domains require high levels of confidentiality, while the communications and passive sensor domain require high levels of integrity[4].

Software systems are more effectively secured when incrementally migrating from one paradigm to another in a manner that “keeps it working” throughout each phase of the migration (13) (14). To that end, domain isolation facilitates migrating existing system components from a lower assurance monolithic system architectures into a higher assurance architecture via discrete steps. As described previously, our notional UAV system must make use of substantial components of the legacy system that does not have a robust security architecture. In order to make use of this software while not undermining our goal of robust security, the legacy application is virtualized and then integrated into the system. For our example, we will assume a separation kernel is used for virtualization. With a means for domain separation in place, critical components can be incrementally extracted and partitioned

[3] Protecting communications (i.e., COMSEC) is not considered in this paper, although it is an important consideration in overall system security.

[4] Admittedly, this is an overly simplistic summary of the security policies, but it nonetheless serves to characterize the policies for the purpose of this example.

off into isolated processing domains. Once partitioned, these components can then be integrated back into the application through properly controlled message passing channels. By iteratively applying this technique, developers can incrementally migrate a non-robust system implementation into a (potentially) provably correct implementation throughout the course of the system's lifecycle. In the case of our example, we would first extract the security critical classifier software into a separate security domain from the rest of the system. Thus isolated, the classifier would be protected from vulnerabilities such as (15). Figure 3-2 Notional UAV Architecture shows how a separation kernel could be used to separate the classifier from the rest of the system.

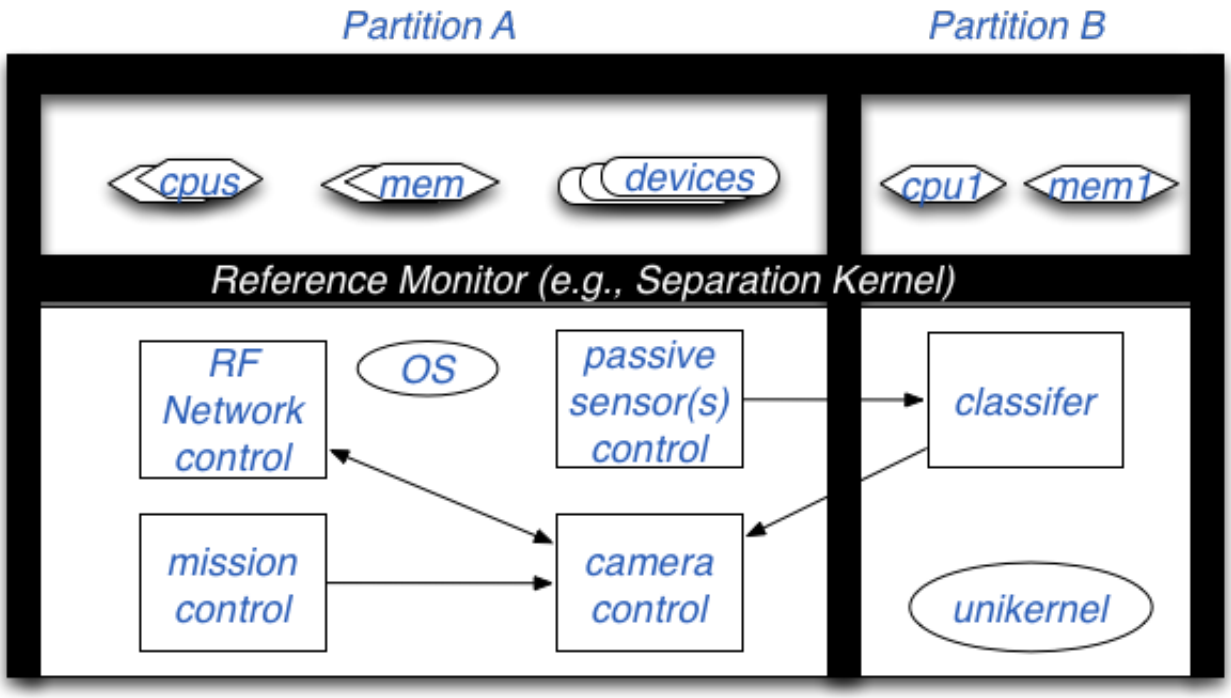

Figure 3-2 Notional UAV Architecture

3.2 Functional Implementation

While architectural and environmental controls can provide substantial system assurance, system implementation is the developmental phase where critical functionality is implemented, and ultimately design and implementation defects introduced in this phase lower the upper bound of the system's overall robustness. In general, secure software development practices are well documented, even if not ubiquitously applied. In this paper, we will focus our attention on message passing as a technique that drives developers toward more secure concurrent UA systems for reasons that will be described throughout the paper.

The models describing defect probabilities for software are well understood. Typically, enterprise information systems use techniques such as continuous patching cycles and end-point protections to mitigate the risks of a security defect being exploited by an adversary. Given the high need for determinism in UA systems, the DDIL environment in which UA systems operate, and the minimal opportunities that users have to interact with these systems (e.g., answering a security prompt), these techniques are as limited in their effectiveness in defending

against emerging cyber threats as they are costly and risky for UA system developers to support during operations. Formal system design and verification techniques offer an alternative means to gain security assurance when these traditional mechanisms are impractical, and such techniques are gaining some traction in cloud and IoT systems (16). In distributed systems, rigorously defining and analyzing shared resources enables system developers to predict and measure defect rates (17). By using message channels as the exclusive means for integrating system components, these shared resources can be easily identified, quantified, and potentially modelled formally such as described in (18). Additionally, rather than being defined only in terms of an API, message passing and message processing routines can be implemented as traits, which enables both re-use within a distributed system and more formal security analysis.

In order for a system using a distributed message passing architecture to be correct, data record formats and semantics must be explicitly defined and assessed prior to any implementation of non-declarative software components (19). Without pre-defined message schemes, imperative code is more likely to drive tighter coupling between the logical data structures, the code that processes them, and the native format used to represent the data structures. This tight coupling frequently results in code defects that allow attackers to exploit logic bugs in complex and/or hard to test data processing code (i.e., the classifier.) This is because large applications (typically implemented imperatively) process data in the same memory space that contains the program's executable code. An attacker could then craft maliciously formatted messages that could in turn be executed by defective software components (e.g., stack corruption.)

Well-defined and structured records can be processed in a functional manner that clearly separates data from code. System developers can exploit this benefit and use functional programming techniques to develop data guards to inject into critical message channels throughout their system. These guarding processes may be driven by simple validation routines, more complex security policies, or even processes that use machine learning to detect threats in real time. In each case, because the sole purpose of these data guard[5] processes is to validate message contents (and not process them), developers can easily code the program to store all data records in non-executable memory spaces. This minimizes the risks that any malicious content could end up executed as code. Figure 3-3 Notional UAV Architecture logically depicts how a data flow guard can be injected into the message channel in order to minimize security risks associated with malformed inputs being introduce from the lower security domains (i.e., communications and passive sensors) into the higher security domains (i.e., the camera and the classifier.)

[5] The term *data guard* is being used as a generic term for a process that provides security validation services for data flowing between security domains with different policies. Industry terms such as *controlled interface* or *cross domain solution* are intentionally avoided in this case.

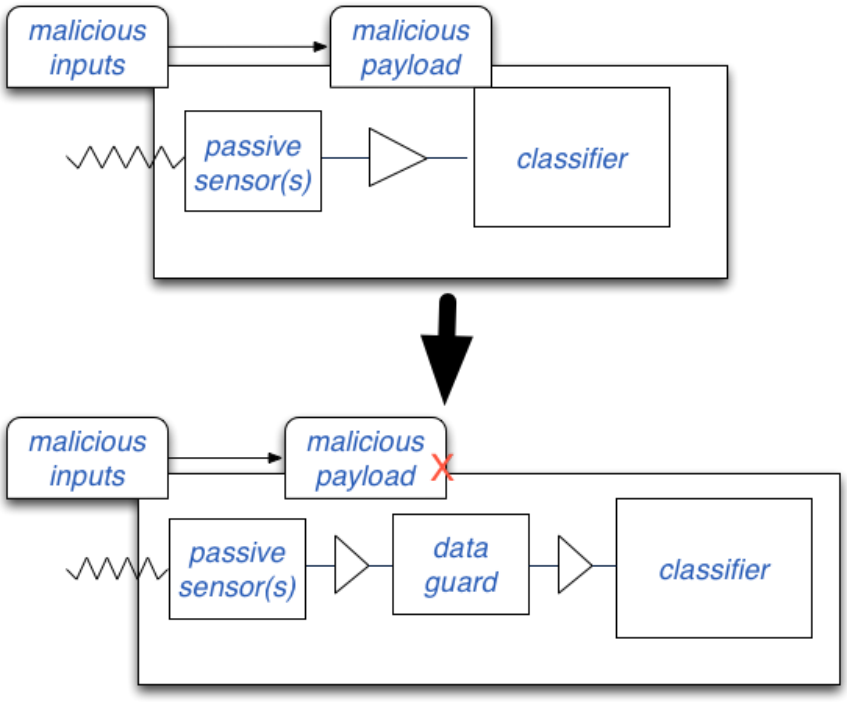

Figure 3-3 Notional UAV Architecture

Message passing promotes a loosely coupled architecture for both time and implementation (20). This architecture allows developers to choose the best language, software tools, and implementation for their particular problem. For example, low level operations such as the passive sensor controllers may use C, while complex classifier logic may be implemented in Haskell or OCaml. With functions decoupled and integrated via message passing channels, applications are immunized against the propensity towards an excessively complex code base (13). Further, developers can readily demonstrate the modularity required of higher assurance systems such as those described in (9). Lastly, with an addition of message queues, message passing can allow developers to scale the system up to meet increasing workload demands.

One difficulty that arises in implementing a distributed system with isolated security domains is that any given process typically depends on data it receives and/or sends to other security domains. With each security domain isolated, data produced in one security domain cannot be accessed by processes in other domains that may need that data in order to properly function. However, system developers can make use of testing doubles (21) to implement the needed behaviours for implementation and testing. These test doubles are often available for complex devices and applications via commercial and/or open source software, and when they are not they can typically be implemented early in the development process. Doubles also enable developers to more easily introduce faults and edge cases, which will drive higher test coverage throughout the development process. Further, doubles can be used by system security assessors to discern what the developers have tested and to perform additional tests. Figure 3-4 Notional UAV Architecture depicts how doubles can be provisioned for both feature implementation and enhanced testing.

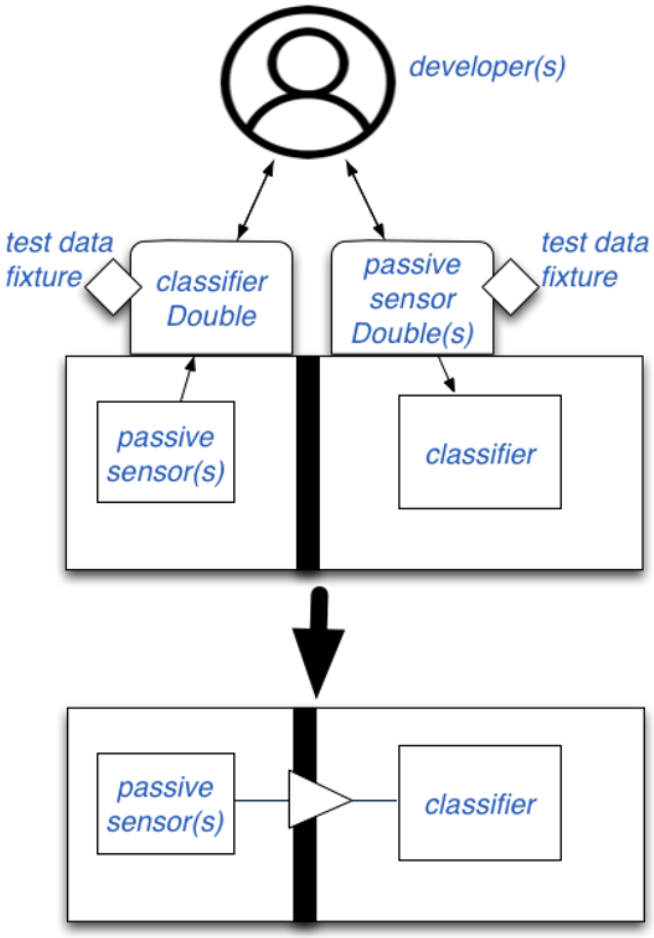

Figure 3-4 Notional UAV Architecture

3.3 Training Set Secure Initial State

One of the first attacks developers of classifiers must consider is an attacker's ability to manipulate the training set. If an attacker is able to comprise the integrity of the training set, then the final classifier's parameters would be trained to identify objects in a way the attacker desires. (22) Demonstrates that injecting carefully crafted inputs to the training set forces the classifier's final accuracy to 0%. Therefore, the integrity of the training set is imperative.

3.4 Behavioral Integration

UA Systems frequently require flexibility to support the integration of a wide variety of additional subsystems to support various missions (e.g. communications relay, reconnaissance, strike.) Subtle differences between these systems can often cause defects to emerge over time as updates are made to system components. Even if test doubles are methodically updated and used for each component during development, system integration inevitably uncovers a variety of logic errors, such as race conditions, deadlocks, and time of check/time of use (TOCTOU) vulnerabilities such as described in (23). These defects create additional functional and security problems that were less common in static, single-purpose systems. At present, there are few tools that exist today that helps

automatically detect and/or remediate these sorts of vulnerabilities (24), such as (25). Long-run load testing and code review by security experts, while expensive and inconsistent, are the mostly practical ways currently to detect these classes of defects. It is advisable that program managers and system integrators plan to iteratively perform functional integration with rigorous integration testing performed at the end of each iteration[6] in order to reduce the risks of defects being discovered after deployment.

Figure 3-5 Notional UAV Architecture shows the system's security architecture in a production ready state, and Figure 3-6 Notional UAV Architecture shows the logical information flows implemented within the system. At any of the shared memory (shm) boundaries in the security architecture, test fixtures as depicted in Figure 3-4 Notional UAV Architecture could be injected into the message passing channel for use by system assessors.

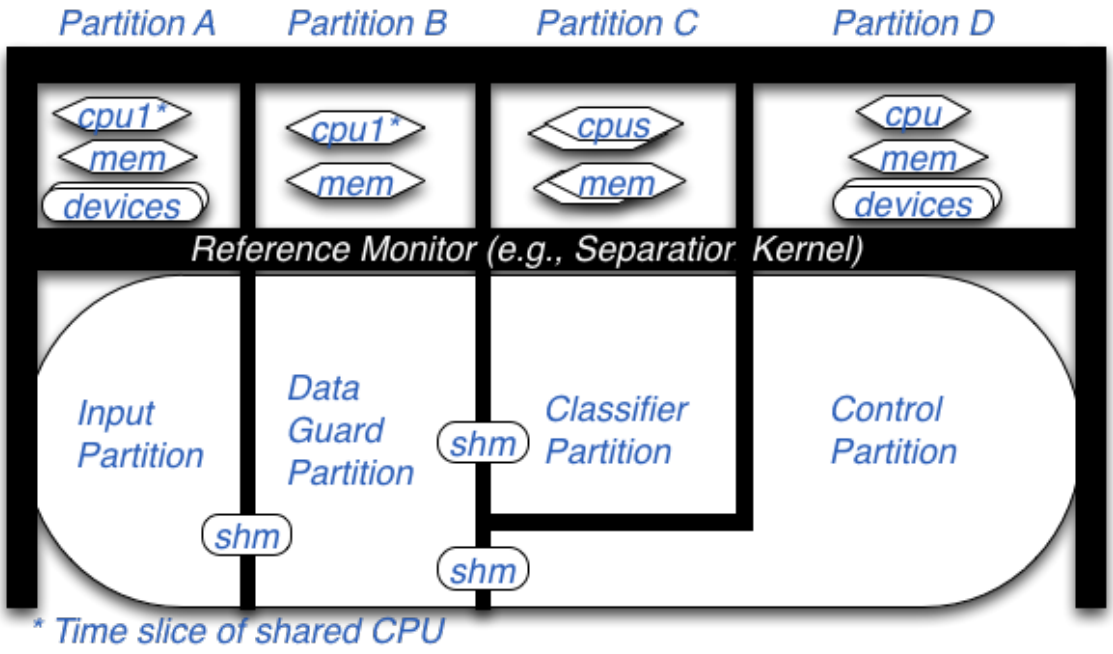

Figure 3-5 Notional UAV Architecture

[6] Continuous integration testing is always advisable throughout development, but is distinct from long-run testing and code reviews that are costlier.

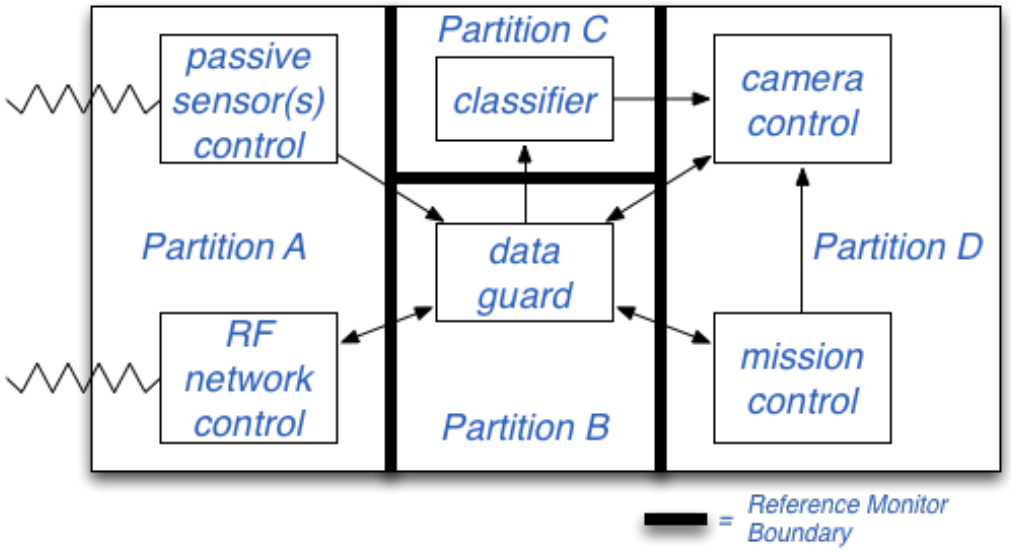

Figure 3-6 Notional UAV Architecture

Another attack that presents risks to systems using a classifier is manipulation of the sensory inputs to the classifier. This sort of attack would happen when the UAV was in theatre attempting to detect objects. If the sensor inputs were manipulated in subtle ways, the classifier could misidentify objects. For example, an attacker may create a simple film over objects to fool image detection. Another example is radio/thermal noise or dispersion to misdirect a classifier. Therefore, knowledge of the classifier and/or the types of inputs that it processes helps the attacker identify and develop effective counter-measures. Further, with access to the training set and the machine learning algorithm, the attacker can generate their own classifier and then evaluate weaknesses. Two methods: Projected Gradient Descent (24) and Gradient Sign Method (26) both demonstrate that generating a second classifier allows an attacker to create counter-measures with > 93% misdirection. Hence, the confidentially must be maintained for both the training set and the machine learning algorithm.

3.5 System Evaluation

Security is undoubtedly an important property for all UA systems to exhibit, but system stakeholders frequently require explicitly demonstrated security assurances by way of an assessment in order to authorize use of the system. As is documented throughout this paper, the development model proposed provides artifacts demonstrating security assurance at all levels of the system. These artifacts can be used by security assessors, penetration testers, code reviewers, and even static analysis tools to build an assurance case such as described in (27). At the top level, information flow control implementation can be documented in a top-level specification, such as described in (11). Data processing logic can be assessed at each data flow guard implemented throughout the system. Implementation correctness can be assessed at the trait and/or application level, using whatever software security quality processes that the organization requires (e.g., static analysis, symbolic execution.) Any implementation decisions that developers make can be exploited by attackers to gain unauthorized access in ways the developer did not consider. Therefore, regardless of the quality of security assessment, security errors can exist in provably correct systems. As such, independent penetration testers should assess levels of the system for design and/or data model errors. In order to maximize test coverage, developers should provide the penetration testers with a special version of the system that is provisioned specifically for assessment, as described in the previous section. These test fixtures can

be used by the penetration testers (and other assessors) in order to perform rigorous white box testing. This rigorous approach enables assessors to uncover defects that were introduced during behavioural integration, and were not present during functional implementation.

4.0 CLASSIFER ATTACK COUNTERMEASURES

To combat these two classes of attack against classifiers we recommend the following:

- Create a secure high-assurance initial training set. The initial training set should be constructed and not pulled from an open source repository. It is recommended that signal inputs (e.g. real objects) are constructed in a controlled environment to generate the initial training set.
- Each training datum should have a checksum (stored on a separate machine) to confirm the integrity. The checksum confirms the integrity of the datum. If at any point the checksum fails, the datum should not be used. Furthermore, with a corrupted datum, a security investigation of the incident should be conducted to confirm the confidentiality of the training set.
- The training set should reside in a separate system than the machine learning system. This measure provides both a defense in depth measure protects the algorithm and classifier from each other, but also provides a decoupled system to aid in development.
- A system should exist that sends signals of misrecognized objects for evaluation and eventually injects into the training set. When a UA system returns (like our UAV for reconnaissance), the recorded sensory data must be evaluated for correctly identified and misidentified data. Any confirmed misidentified objects should be injected into the training set and then the machine learning algorithm rerun to create the new parameters for the classifier.

Even the best security of the classifiers will eventually fail. For example, the attacker may acquire access to the training set, access to a classifier, or continuously attempt different detection counter-measures until discovering one that works. Therefore, a system must both quickly respond to issues (react) or prevent attacks (proact) from being effective in the first place.

A reactive defense must protect the confidentiality and integrity of the system, and be monitored so that developers can respond quickly to misidentifications. As shown in (28) adding adversarial data correctly to the training set will generate a final classifier with < 2% error rate. Therefore with knowledge of the attacker's counter-measure, a security assessment team can rerun the machine learning algorithm to effectively produce a new classifier for recognition[7].

Attempting to prevent these attacks before they occur (i.e., a proactive approach) requires a deep understanding of the training set. In a study of guarding a recognizer for language letters, (29) demonstrated that evaluating common distortion techniques into a trainings set and feeding it back to the machine learning algorithm can catch most of the adversarial example (81% - 99%.) If this is not an acceptable level of accuracy, the original machine learning algorithm would need to be revisited. Nonetheless, generating an adversarial training set (fuzz testing/mutant testing) improves the robustness of the classifier's security posture when faced with such input-driven attacks.

[7] The message passing architecture described and recommended in section 3.0 would also facilitate more straightforward update process.

5.0 FUTURE WORK

Machine learning is an evolving field. More research is necessary to improve efficacy among military applications. First, most or all current research assumes the same set of training data. While it is the authors' belief these techniques will carry over to the military world, the authors did not discover any paper supporting this idea. The first evaluation will be to determine the accuracy on the recognition of different sensory inputs. Second, these learning systems need to be evaluated for integration into existing platforms for deployment feasibility. Finally, the classifier's resilience to adversarial attacks (like PGD) will need to be continually evaluated against the performance of human recognizers. While training a specific adversarial training set can be very effective, there is research to train a generic neural network to expect adversaries. As of this writing, current research has achieved 81% confidence in many cases (30), which shows promise but needs further investment to achieve acceptable levels of confidence for widespread use in mission critical systems.

6.0 CONCLUSIONS

UA systems have great potential to positively contribute to a wide variety of military, civilian, and commercial applications and are likely to be adopted at an increasing rate in the coming years. However, given the newness of the field and the broad attack surface that these systems often expose, UA systems face a daunting set of security problems. While many improvements to embedded cyber security are emerging from the IoT space, these approaches typically do not account for the DDIL environments in which military UA systems will operate. In this work, we propose a general approach that uses a variety of techniques that have been demonstrably effective in protecting UA systems such as domain separation and the use of data guards. We also describe a means by which this approach can be enhanced to address threats specific to UA systems. In general, developing UA systems is a process that crosses many disciplines that are interdependent for mission success. Cyber security and machine learning are two rapidly developing fields that must cause UA system developers to re-think many classical approaches to systems engineering and acquisition processes. Specifically, with respect to machine learning security, there are a growing number of relevant developments on offense and defense that should be monitored. However, there are some limitations to current research that should be considered.

REFERENCES

[1] **US NIST.** Considerations for Managing IOT Cybersecurity and Privacy Risks. *US NIST.* [Online] 10 08 2018. [Cited: 01 09 2018.] https://www.nist.gov/sites/default/files/documents/2018/08/10/considerations_for_managing_iot_cybersecurity_and_privacy_risks_workshop_summary.pd.

[2] **US DARPA STO.** Strategic Technology Office Outlines Vision for "Mosaic Warfare". *https://www.darpa.mil/news-events/2017-08-04.* [Online] 04 08 2017.

[3] **Goodspeed, Travis.** Poc || GTFFO 0x03. *Poc || GTFFO 0x03.* [Online] 22 03 2014. [Cited: 01 09 21018.] https://archive.org/stream/pocorgtfo03#page/n2/mode/1up.

[4] **Ma, Hyperchem.** Badbarcode: How to Steal a Starship With a Piece of Paper. *Tencent's Xuanwu Lab.* [Online] 2015. [Cited: 01 09 2018.] https://www.slideshare.net/PacSecJP/hyperchem-ma-badbarcode-en1109nocommentfinal.

[5] **Perla, Enrico and Oldani, Massimiliano.** *A Guide to Kernel Exploitation: Attacking the Core.* s.l. : Syngress Publishing, 2010.

[6] *Robust Real-Time Face Detection.* **Viola, Paul and Jones, Michael.** 2, s.l. : Kluwer Academic Publishers, 2004, Vol. 57.

[7] *Large Scale Distributed Deep Networks.* **Jeffrey Dean, Greg S. et al.** p. 11.

[8] *Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs.* **Gulshan et al.** s.l. : Journal of the American Medical Association, 2016.

[9] *A Multi-layered Approach to Security in High Assurance Systems.* **Alves-Foss, Jim et al. .** Honolulu, HI : Proceedings o fthe 37th Hawaii International Conference on System Sciences, 2004.

[10] **Zhang, Zhi.** arxiv. *KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels.* [Online] 20 02 2018. [Cited: 01 09 2018.] https://arxiv.org/pdf/1802.07062.pdf.

[11] *Security Policies and Security Models.* **Goguen, J. and Mesenguer, J.** s.l. : IEEE Symposium on Security and Privacy, 1982.

[12] **Klein, Gerwin et al.** seL4: Formal Verification of an OS Kernel. *SigOps- ACM.* [Online] 2009. [Cited: 01 09 2018.] https://www.sigops.org/sosp/sosp09/papers/klein-sosp09.pdf.

[13] **Foote, Brian and Yoder, Joseph.** Big Ball of Mud. *Researchgate.* [Online] 09 2003. [Cited: 01 09 2018.] https://www.researchgate.net/publication/2938621_Big_Ball_of_Mud.

[14] **Feathers, Michael.** *Working Effectively with Legacy Code.* Upper Saddle River, NJ : Prentise Hall PTR, 2004.

[15] *Meltdown: Reading Kernel Memory from User Space.* **Lipp, Moritz et al.** Baltimore, MD, USA : 27th USENIX Security Symposium, 2018. Vol. 27.

[16] **Nazario, Jose.** The Problem with Patching in Addressing IoT Vulnerabilities. *fastly.* [Online] 27 09 2017. [Cited: 01 09 2018.] https://www.fastly.com/blog/problem-patching-addressing-iot-vulnerabilities.

[17] **Yu, Tingting et al.** ConPredictor: Concurrency Defect Prediction in Real-World Applications. *arxiv.org.* [Online] 27 06 2017. [Cited: 01 09 2018.]

[18] **Allwein, Gerald and Harrison, William Laurence.** Distributed Modal Logic. [book auth.] Michael Dunn. *Information Based Logics.* West Lafayette, IN, USA : Springer Publishing, 2016.

[19] **Van Roy, Peter and Haridi, Seif.** *Concepts, Techniques, and Models of Computer Programming (Chapt. 4).* Cambridge, MA, USA : MIT Press, 2004.

[20] **Eugster, Patrick Th. et al. .** Chapter 8: Loosely Coupled Components. s.l. : https://pdfs.semanticscholar.org/fc43/d603fbbf5d9ed3fed90270ed011fc5002173.pdf.

[21] *Mock Roles, Not Objects.* **Freeman, Steve et al.** Vancouver, British Columbia, Canada : OOPSLA '04 (ACM), 2004.

[22] *Adversarial Examples for Adverse Models.* **Kos, J. et al.** p. 25.

[23] **Gu, Rui et al.** Understanding and Detecting Concurrency Attacks. *Columbia University.* [Online] 30 03 2018. [Cited: 01 09 2018.] https://www.cs.columbia.edu/~ruigu/papers/conattack.pdf.

[24] **Madry, Aleksander et al.** Towards Deep Learning Models Resistant to Adversarial Attacks. *arxiv.org.* [Online] 9 11 2017. [Cited: 01 09 2018.] https://arxiv.org/abs/1706.06083v3.

[25] *Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features.* **Schwarz, Michael et al.** s.l. : Asia CCS, 2018.

[26] **Kannan, Harini et al.** Adversarial Logit Pairing. *Arxiv.org.* [Online] 16 03 2018. [Cited: 01 09 2018.] https://arxiv.org/abs/1803.06373v1.

[27] **Weinstock, Charles B. et al.** Arguing Security- Creating Security Assurance Cases. *CMU SEI.* [Online] 01 2007. [Cited: 01 09 2018.]

[28] **Goodfellow, Ian J. et al.** Explaining and Harnessing Adversarial Examples. *Arxiv.org.* [Online] 05 03 2015. [Cited: 03 09 2018.] 1412.6572v3.

[29] *Assessing Threat of Adversarial Examples on Deep Neural Networks.* **Graese, Abigail et al.** 2016, p. 6.

[30] *Adversarial Attacks Against Medical Deep Learning Systems.* **Finlayson, Samuel et al.** 2018, p. 15.

[31] **US Department of Defense.** *DoD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria.* s.l. : US Department of Defense, 1985.