**Security of Cyber-Physical Systems through Dynamic Component Management**

Anton D. Hristozov

Purdue University

ahristoz[@purdue.edu](mailto:first.author@myemail.edu)

Dr. Eric T. Matson

Purdue University

ematson[@purdue.edu](mailto:second.author@myemail.edu)

Dr. Eric Dietz

Purdue University

jedietz[@myemail.edu](mailto:second.author@myemail.edu)

Dr. Roger Marcus

Purdue University

rogersmk[@myemail.edu](mailto:second.author@myemail.edu)

**Abstract**

The environment in which Cyber-Physical Systems (CPS) operate can vary in complexity and experiences changes constantly. These systems work mostly autonomously, without supervision, and with little or no maintenance. They continuously interact with the environment and with other systems. The expectation is for them to last for long periods, sometimes years or even decades. Their operation is under constant evolving threats, which presents new challenges to designers.

The ability to replace software components on the fly can be an effective way to improve software security. Techniques such as redundancy and diversification become feasible if a general mechanism for software components update is available during run-time. These goals can become possible if issues such as component interfaces, state, authentication, and dynamic management can be handled practically. This study focuses on the feasibility, mechanisms, and limitations of dynamic management of software components to improve security. The focus of our experiments is on software architectures such as the PX4 autopilot. Still, they can be applied to various CPS, such as edge devices, smart sensors, and even autonomous robots. The use case explores the approach's feasibility for Unmanned Aerial Vehicles (UAV)s. The study presents an analysis of the effects of such techniques on the overall mission of the system.

**Introduction**

**CPS Operation in the Presence of Attacks**

Like any other cyber system, CPSs are prone to different attacks. Some of these can be from their network connections, and some from attacks on sensors, controllers, and actuators (Wang et al., 2019). Another source of disruption can come from hardware or software failures due to reliability issues or design and implementation mistakes. The software complexity in today's CPS is so great that it is impossible to guarantee that a system will be implemented with no defects (Henkel et al., 2011). The reality is that a certain number of defects per lines of code (LOC) will most likely continue to persist in the future. These sources of disruption need to be addressed independently of their reason. A genuinely resilient system should adapt and continue operating in the presence of attacks, failures, or both. Truly resilient systems can go through the following phases when experiencing disruptions (Fitzgerald, 2016):

* Avoiding the disruption
* Surviving the disruption
* Recovering from the disruption

Any system that can handle resilience must be designed to cope with these three phases. Metrics such as how long a system can avoid disruption and survive it and how long it takes to recover can further quantify the readiness for operation in harsh conditions. Another factor to consider is what kind of attacks can be handled. It is hard to claim the ability to detect and neutralize any possible attack. A different approach is needed to make security possible in practical scenarios.

The attacker is always trying to use weaknesses in existing architectures, which becomes more achievable if they use well-known models and operate in a predefined fashion. One way to make things way more complex for the attacker from a security point of view is to introduce uncertainty in time and space. Diversity in time can be achieved by running certain portions of the software at different points instead of constantly. Diversity in space can be achieved by running different software types instead of one application or portions of an application. This approach can make the attacker's job more difficult or downright impossible. With the increased sophistication of attacks and the knowledge that attackers have, this is a promising direction for protecting systems left to work in different and complex environments. Such environments can be in urban and war zones where potential sophisticated attackers can be abundant.

**Class of Systems**

The CPS is a reasonably large classification of very diverse types of systems. Our study will narrow the scope to systems run on Unmanned Aerial Vehicles (UAV) or Unmanned Ground Vehicles (UGV). A further narrowing of the scope is done to include autopilot systems such as Ardupilot and PX4. The techniques that are explored, though, apply to any other autopilot system and even to the Robotic Operating System (ROS) (Lauer et al., 2016). These systems have common subsystems and components that are prone to the same vulnerabilities. The vulnerabilities come from the components that have interfaces to the outside world and are essential for the control of robotic vehicles. Our selection to use the PX4 autopilot software for our experiments was driven by several crucial characteristics that this system has. They are the following:

* PX4 has a modular architecture that can be extended to illustrate approaches needed in this study
* There is a convenient infrastructure to run experiments through simulators such as JMavsim and Gazebo. The benefits are that the same code that is run on the UAV is run in simulation, and disturbances in the flight performance can be seen with different types of UAV types used in the simulator.
* There is a possible integration with MAVSDK, which allows for developing mission applications that can control drones through the Mavlink protocol. This allows for complex missions to be run in order to test the autopilot software without the need for real UAV equipment and a flight site.
* PX4 is a prevalent platform, and any studies that reveal vulnerabilities and countermeasures are of interest to the community of researchers and practitioners in the industry
* Since this platform has high complexity, any techniques that can be deployed successfully here can be used in other platforms with similar or better success

PX4 has modules that are essentially software components, and each one runs in a separate thread or in a work queue (Meier et al., 2015). Work queues are mechanisms that use a common thread for several modules to share by sequentially running their Run() methods periodically. The work queues present further challenges from the perspective of security because of the increased complexity and difficulty in manipulating individual modules. Work queues make timing guarantees and predictability even harder and introduce dependencies of modules and their main thread functions. Our approach to software components and their dynamic management is irrelevant to whether they are run in a separate thread, work queue, or even in separate processes, as is often the case in ROS systems (Malavolta et al., 2020).

The test setup used in this study is shown in Figure 1. It uses a high-level mission control application that sends the mission commands through the Mavlink protocol. The autopilot software is connected to a simulator, which can be Gazebo or Jmavsim. In addition to the autopilot software, a new system is added that takes care of the dynamic management of software components. The logs are used for analysis of the flight characteristics to do a comprehensive analysis of the results.

Dynamic Framework

Autopilot System

Data Collection

(Simulator)

Real System

Mission Controller

Figure 1: Test Setup

**Literature Review**

There are some studies that discuss dynamic system reconfiguration and run-time techniques in general. Many of these studies focus on some of the aspects of implementing dynamic changes, but they often do not comprehensively look at all problems, especially the state, authentication, and timing. In addition, studies do not typically focus on dynamic behavior with respect to improving security as our paper does. Dynamic reconfiguration may be done for purposes of maintenance, repair, upgrade, and, most frequently, functional changes. Dynamic reconfiguration for the main goal of improving security and safety is the main contribution of this paper. Nevertheless, our study builds on what is already available in the literature and proposes a different angle of the techniques used and the use cases.

There is a significant number of papers that focus on reconfigurable manufacturing systems (Bortolini et al., 2018). These papers focus on the challenges related to Industry 4.0 and the ability to quickly reconfigure a manufacturing or any other large-scale system in order to provide new functionality. These studies consider concepts such as availability and reliability, but security and safety are normally not explored in depth. Some of the studies discuss self-healing and self-management systems (Shin et al., 2018) and the use of machine learning to optimize their operation. The proposed techniques focus on how systems adapt to abnormal situations using self-adaptation techniques.

A general approach to dynamic reconfiguration is discussed by (Saadi et al., 2018), where the notion of a reconfiguration manager is introduced. This component relies on events that trigger actions, and based on a reconfiguration policy, a new configuration is chosen and implemented. The architecture is represented in the form of a graph. The authors point to the challenges of verifying and validating the possible configurations. This opinion is also shared in many other papers as the possible combinations can make guarantees for architecture stability difficult to provide.

An important technique described in the literature is the Moving Target Defense (MTD) technique. It creates diversity in the memory space and in the instruction set used in order to create obstacles for the attacker (Potteiger et al., 2020). MTD changes properties in the system as it runs so that it can minimize the chances of success of an adversary who tries to succeed in devising ways to reverse engineer a working system. This is a form of a proactive approach toward a large number of attacks. The most frequent attacks that the techniques help against are in the category of memory attacks, which can include code injection and code reuse.

Some works propose a dynamic reconfiguration scheme based on representing the system through graphs. Each node represents a module or component, and each edge is a connection between two components (Pavlenko et al., 2019). The authors define routes through the graph, representing a cyber-physical system, and choose different reconfigurations based on information about which node is being compromised by an attack. Choosing a reconfiguration dynamically can maintain the stability of the system. This assumes that there is a mechanism for detecting attacks on any of the nodes.

One of the approaches in systems that can do self-adaptation and reconfiguration is that such systems can provide redundancy of important components. This approach improves fault tolerance and reliability in general. The author suggests that certifying s dynamically reconfigurable system is very hard (Isakovic, 2022). The idea of such reinforced components is used in our paper too, where the scope is narrowed to individual components to make it easier to certify and implement a solution.

The majority of the encountered sources dealt with high-level techniques and did not have a case study that implements a working model that proves that a certain technique can work in practice. As a consequence, they do not focus on a particular class of systems and rarely provide concrete test results. Our study focuses on a specific class of systems, an attack model, and a practical approach to improving security against a large class of attacks. A real-world example of a UAV autopilot is also proposed as well as test data for analysis to help derive relevant conclusions. The following sections introduce the approach and how it complements the existing state of the art.

**Attack Model**

This paper will detail the mechanisms of launching two types of attacks on a running instance of a PX4 autopilot. These attacks can be targeted and can affect specific components of the autopilot software. Attacks can be pretty different, and their classification is not the purpose of this paper (Yaacoub et al., 2020). The assumptions for the attacks that are made are:

* The attacker can get into the OS where PX4 runs and can gain privileged access
* The attacker is familiar with the architecture of the running autopilot software, for example, PX4 or Ardupilot
* The attack will rely on well-established techniques that work in a Linux environment
* The attack will take advantage of the fact that the software is written in C++, and therefore, it inherits its vulnerabilities, such as buffer overflow, for example

These assumptions are relevant to a large percentage of drones used today. Many of these drones are running some version of Linux and autopilot software. In addition, security is not always a deep concern for many UAVs because they are still considered experimental, and standards are still being established. Therefore widespread attacks such as buffer overflow and /proc file system memory attacks are a real possibility (Nayak et al., 2017).

Attacks can be launched through many different methods. The actual type of attack may use a way to corrupt the memory of the running autopilot process. Our method of defense aims to create a moving target for the attacker so that they have a very dynamic and much harder environment to attack. Most attackers expect a well-known static environment with a defined memory layout. Figure 2 shows the mechanism of a buffer overflow attack as a popular method to get to the memory of a running process. The attacker plants a custom socket server that can be contacted later through a client process. It can exploit a buffer overflow attack to get to a component in the autopilot software and compromise it.

Malicious Socket Client

Custom Socket Server

PX4 Autopilot ORB

Component Manager

Compromised Component

New Componment

Buffer

Overflow

Receives a bad message or memory overwritten

Executes a malicious function that overwrites memory

Figure 2: Attack Setup

**Dynamic Component Management Approaches**

In order to create protection, the choice was to replace a working component with another component that has the same functionality. By running each component for a short period of time, the functionality of the autopilot is maintained. The focus is on one of the software components that is central to the autopilot system in order to prove that the approach works. The technique can be applied to more than one component with some penalty of increased CPU and memory usage. The goal of our experiment is to be able to preserve the functionality and timing as expected for the normal operation of the UAV.

**Dynamic Component Replacement**

Component replacement while the system is running with a new component that can continue functioning at least, as well as the replaced component, is a complex undertaking that requires careful analysis. This technique is an effective countermeasure against the types of attacks described in the previous section. The operation is depicted in Figure 3. This can happen by designing a new component manager module. Component A, as shown in the figure, is a running component that needs to be replaced with component B during run-time. By assuming components A and B have compatible interfaces, they can be switched by unloading one of them and loading the other.

Component Manager

Component A

Component B

Component Repository

State Manipulation

Diagnostics

Interfaces

Figure 3:Dynamic Component Replacement

The component manager needs to take care of the following concerns:

* Assures that the interfaces of the new component are compatible with the interfaces of the replaced component
* Provides authentication so that there is some assurance that the new component is not bogus and is coming from a trustworthy source
* Makes sure that the new component is brought up in a state that matches the state of the replaced component
* The operation does not have a significant effect on the dynamics of the system

The sequence of how a replacement can happen is illustrated in Figure 4. There is a time when the new component is loaded in memory, but it does not run right away. This is useful so that the new component can have its state updated, then the old component can be stopped and unloaded, and the new component starts running in its place. If the system dynamics permit such an operation, the system will not experience any significant effects. This also depends on the component complexity and its state and timing characteristics.



Figure 4:Sequence Diagram of Component Replacement

Dynamic component replacement can be helpful in situations where an upgrade is needed, repair of a component, or a switch in the operation of components due to environmental changes or security requirements. One such situation is related to an ongoing attack on the component, which makes its operation incorrect and dangerous. Another situation is when a component is updated with a newer and improved component that fixes specific vulnerabilities and capabilities of the existing one. In both situations, replacing an existing component has to be done according to an algorithm described in detail later.

**Alternating Component Execution at Run-Time**

When the number of attacks is large, and the system is executing in real-time, using a proactive approach against potential attacks is better than detecting and remediation in real-time. One such technique assumes that there can be two components with the same functionality but not necessarily the same implementation. If these components are run in separate threads or other execution units, such as processes or tasks, they can be restarted periodically, and their execution can alternate. This guarantees that there is always a running component that can handle the expected functionality.

The definitive advantage of the approach followed is that it creates a moving target for the attacker by constantly swapping threads that have different thread identifiers (ID)s and memory footprints since threads are created and destroyed periodically. The approach takes advantage of the principle of code diversity and time and space randomization. To further improve security, refinements such as randomly changing the names of the threads and the interval of swapping increase the difficulty of a feasible attack. Ultimately, the alternated threads can have completely different implementations but can perform the same tasks. An example is a traditional PID controller and an LQR controller being swapped repeatedly. This is particularly true for components that have no state, for example, a PID controller.

This operation is at the expense of some overhead in time and CPU resources, although the approach can be applied only for the most vulnerable threads of the autopilot software. The timing of the exchange happens according to the following diagram - Figure 5, where the following time periods are detailed:

Tload

Tready

Tswap

Tunload

Trun Old

Trun

New

Figure 5: Timeline for Component Replacement

Tload - time to load a new component

Tready - time to reach a running state

Tswap - time to swap the components

Tunload - time to unload a component

Trun - time to run

**Challenges with Dynamic Component Management**

There are numerous challenges while implementing a dynamic component replacement. These challenges are based on complexity, security, timing, and dependencies on the specific software architecture. Each component can be represented through the interfaces it supports. Components with the same interfaces can be replaced independently of their implementation if they comply with the architecture and require certain services to be present. Some of the issues discussed in this section can be implemented during design time. This includes the analysis of component interfaces and the preparation of the components for their run-time authentication. State handling and effects on the architecture are concerns that need to be handled during run-time.

**Component Interfaces**

The representation of interfaces that can be shared among components throughout the architecture cannot happen without a unique approach. A promising way to address this issue is to use or design a Domain Specific Language (DSL) that can capture the interfaces, which includes the contracts that the component abides by (Holthusen et al., 2016). Having a DSL allows the use of automatic tools for integrity and run-time monitoring and performing component rejuvenation and state restoration. The DSL can capture three crucial things that can help in many different directions when designing software architectures with smart software components:

* The interfaces
* The contracts that are represented by the interfaces
* The state of the component
* The parameters that are used for configuration

Creating a DSL can happen through tools such as Antlr (Stockmann et al., 2019) or Xtext, or any other tool for the generation of custom parsers. This is going to produce a custom parser of language grammar in general that can also be used to do code generation (Parr & Quong, 1995). A DSL can have high-level constructs that unambiguously define the interface of components. Parsing all component descriptions can produce C++ code that can be used at run-time along with the existing code that components already have. This approach allows for retrofitting existing deployments of software components and giving new properties to them. A possible prototype can be developed using the PX4 autopilot software as it has all the characteristics of a multi-component system with well-defined components (Meier et al., 2015).

A simpler approach to using a DSL is to use a component description language in a structured format such as JSON format. This can arguably be regarded as a form of a DSL, just without the specifics in grammar. The goal of either approach is to express the component's interfaces in a very descriptive and unambiguous way. This could allow comparisons of components and operations such as loading, reloading, destroying, and switching between components to happen easily at run-time. The interface definition of a component associated with the code that uses it can be considered an adapter attached to an existing component for the purposes described so far.

**Authentication**

Loading software components dynamically introduces the risk of loading a malicious or tampered component that can jeopardize the regular work of the systems. The first attack based on buffer overflow described earlier uses this vulnerability. Since there is no authentication on what kind of component can be loaded with the existing dyn command in PX4, anyone who has access to the PX4 shell can load an arbitrary module. A module can be loaded even without access to the shell by using the available API that PX4 provides, leading to even more subversive attacks.

These issues require a mechanism to be developed that can assure that components can be trusted, and approaches such as component authentication and attestation come into play. Architectures that use attestation servers and key management have been proposed (Van, 2017) to handle this. A simple approach is to use a shared key between the component manager and the build systems that create the components and attach a calculated hash, which is a sequence of bytes to the component that can be verified only by somebody who owns the secret key. This approach is simple, and it works, provided the secret key is guarded.

An HMAC approach is discussed in (Beri & Mishra, 2019), where a keyed hash algorithm is used. It allows for data integrity and proof of origin, which is what is needed in this case. Guarding the shared key is essential to ensure that there is a secure solution. The technique described in this paper is practical and may be used as a basis for the implementation as part of this study. The overhead of checking the HMAC needs to be considered, but since it is something that is done only one time while the component is loaded, it is practical. An adequately long HMAC like SHA512 can provide better security at the expense of some computational overhead. This may not be a big hurdle, though, since the HMAC will need to be calculated once the component is compiled and then recalculated again just before a decision is made if the component is authentic and can be loaded and run.

**Timing Considerations**

Effects on timing are another concern when replacing components (Knight & Strunk, 2004) during run-time. Some components can take a long time to reach a steady state since they need to collect data based on sensor measurements. One such component is the Extended Kalman Filter (EKF), used in various autopilots and other control systems. Such components can affect the overall stability of the mission, and their timing and inertia need to be considered. Another valuable consideration that needs to be mentioned is how long it takes for a component to be restarted and the effect on the system's overall stability. This includes the time for loading, initializing a component, and updating its state. If a new component is loaded that replaces a compromised or vulnerable, and obsolete component, there is a need to consider the time the new component will be ready to take over. These timing dependencies are shown in Figure 5.

The components that are used in this study are the attitude controller and the position controller. Another interesting part of the system is the GPS coordinate handling. This is important since many attacks are launched through GPS spoofing, and their effects can lead to controlling the vehicle's mission. The EKF is also of interest as its importance in the system is very high. It presents a real challenge for dynamic component management because of the complexity described above and its six instances in a running PX4 autopilot process. Therefore, the focus would be on the attitude and position controller components and leave the EKF observer for future explorations.

**State Handling**

Handling the state in a system is a complex task since the system state is general, but the local state is specific to individual components. The state can be modified at system initialization and during run-time (Kapova et al., 2010). There is a connection between the internal state of each component and the overall state and behavior of the system. Expressing the internal state of a component can help make this connection. Many existing systems use static methods for state evaluation, although the state changes and can be different at different times. This happens because many components change their internal state as the system continues to operate (Lauer et al., 2018). Some have very complex states, and some have no state or a very minimal state. Recovering this internal state of a component that has been replaced or refreshed is a major challenge because the effects this state can have on the system can be detrimental. Representing the state in a universal way and providing facilities, part of the component so that it can be recovered is possible, but it is far from trivial. The difficulty comes from the fact that the state of a component can be constantly changing, and the change can be relatively fast or gradual. In addition, components can have multiple parallel threads of execution, making things even more complicated. This can mean that the state can be distributed among threads that are part of one component.

Software components retain states in variables with different types; in many cases, they are complex objects and structures. In order to be able to save the state of each variable independently of its type, a technique called serialization needs to be used (Grochowski et al., 2019). This technique takes the variables of an object and converts them to bits that can be stored on a disk. The deserialization is the opposite and converts the bits from the disk to values that can populate the variable of an object. Some languages support serialization, but C++ does not appear to do so in the standard libraries, and serialization can be accomplished through additional libraries. This is relevant since the typical autopilot software is written in C or C++. Thankfully some useful libraries are appropriate for tackling the task of state management of objects that belong to components.

**Effects on the System**

Our approach will compare the performance of the original system and a modified system that experiences dynamic component updates while the system is running through a complex mission. Dynamic component updates can affect the system's operation, so a feasibility study needs to be performed. Our use case section attempts to determine if this approach is practical for a modern software architecture like the one used for the PX4 autopilot. For this assessment, the setup shown in Figure 1 is used.

Dynamic component replacement would be more accessible when a system is executing a simple mission or, in the case of a UAV, is simply hovering. Many works consider simple cases, such as hovering operations that are not practical scenarios for real UAV systems (Hidenori et al., 2019). Some factors worth mentioning determine the feasibility of the dynamic management approach. Some of them are:

* The complexity of the component, which includes state and timing dependencies
* The moment in the mission when the dynamic operation occurs
* The speed of the CPU, memory, and overall hardware of the UAV
* The software architecture of the autopilot

**The Concept of Root of Trust and Component Management**

The vision is that there is a specialized module called component manager, as shown in Figure 3, is the central component that allows for any dynamic scheme to be put in place. The most straightforward approach is to implement it in user space, as was done for our prototyping. Though, a more thoughtful approach is needed. The component manager needs to be protected from potential attacks, and therefore it needs to be in a different place where the root of trust is. Candidates for this are the kernel space of the OS that is used or, even better, in a hypervisor that can control the software components from a secure implementation. Both approaches are discussed next as a potential for future experiments.

Figure 6 shows a general approach to the software application running in user space, as is the usual case for most systems. The kernel space implementation of the component manager can be as simple as dynamically swapping components to maintaining a graph of component interactions and refreshing different components when they do not operate. This allows for the smarts about component dependencies and their state to be kept in the kernel space. This guarantees survivable components in user space because of the dynamic control from the kernel space.

Component Manager

SC1

SC2

SC3

User Space

Kernel Space

SC1 – Sw Component 1

SC2 – Sw Component 2

SC3 – Sw Component 3

Figure 6:Kernel and User Space Implementation

**Kernel Space Implementation**

The component manager can be implemented as a kernel module that controls the user space components from kernel space. This approach protects the component manager and the dynamic component management scheme from attackers who have access to user space only. Access to kernel space requires superuser privileges and is harder to penetrate for most attackers. There is some complexity in implementing the component management in kernel space and the interaction with the user space, but it is still possible in a rich environment such as Linux. In addition, it is uncommon to have a kernel module that controls user-space threads, and this approach is somewhat unfamiliar to traditional architectures known to attackers.

**Hypervisor Implementation**

Another robust approach is to use an implementation in a hypervisor so that the execution of the component manager is at an even higher level of protection (Vasudevan et al., 2016). Hypervisors can run at the highest privileged level, and their code cannot be accessed from the OS, even from its kernel space. This approach also has the advantage that a solution can be part of a well-designed hypervisor's verified and minimal codebase. It is a more complicated way to implement the solution, but it protects against attacks since the component manager cannot be compromised. Thus, it can continue to manage the user space and keep it resilient through a dynamic refresh of components in the presence of persistent attacks.

**Use Case**: **PX4 Autopilot and Dynamic Component Management**

Our experiments are based on running PX4 in a simulation environment with Jmavsim or Gazebo. Jmavsim is included with the distribution of PX4 and is written in Java. It is a simpler simulator that allows modifications, although it is not a physical simulator. Gazebo is a physical simulator that can be extended through plugins and provides a flexible architecture, including the possibility to create simulation worlds. Both are adequate for our experiments. MAVSDK is used as a tool that can generate complex missions for the simulated UAV to execute. MAVSDK is flexible and communicates through the Mavlink protocol in order to send and receive commands from the vehicle. During the flight dynamic component replacement and alternative component execution are performed. In the first case, the process is to go through the replacement of a running component, as depicted in Figure 4. Our objective is not to disturb the mission too much and to be able to complete it while implementing the dynamic scheme. A quantitative analysis of how much the flight parameters have been affected is done based on analyzing the flight logs.

**Authentication Implementation**

Authentication is a two-step process where the component that needs to be authenticated is prepared during compilation and verified at the time before it is run. A strong algorithm is needed to ensure that it cannot be easily broken. The technique that is both practical and secure is HMAC. The build system uses a shared key, and the same key is used during run-time. Key generation and storage are other critical concerns. Key length and the HMAC algorithm itself determine the strength of the solution. SHA512 can be used to generate an HMAC on the contents of the module and use the same key to compare the generated and calculated HMAC before the module is loaded. The computational overhead is not a significant concern since this is done only once at run-time and once during compilation.

For our testing, cryptocpp library was used, which has a pretty good arsenal of algorithms for cryptography. Cryptocpp was selected since it is open source and has different algorithms that can be called from the C++ code in the PX4 autopilot. The selected algorithm is SHA512 based on its safety. It is pretty straightforward to add a post-processing task during compilation and create an HMAC. When the module is loaded, the software follows the same procedure of calculating an HMAC on the contents of the module and loading it only if the HMAC is the same. This is constantly emphasized with a shared key approach, although the techniques for accomplishing this are well outside this study's scope. Using a shared key with 32 bytes length was an approach that was selected, although the handling and storage of the shared key were not part of the study.

**State Recovery**

Each software component in PX4 typically has a main class with class variables that are used to preserve the state during the operation of the component. For state saving and recovery of a running component, the bitsery C++ library was used, which allows for serialization of variables with different and complex types. Bitsery allows for variables with arbitrary types to be effectively serialized to disk or RAM and de-serialized when needed. This technique allows for the state of one component to be saved to disk or memory and then transferred to a newly loaded component. Our tests concluded that this is feasible for the attitude and position controller components. This proves that the approach works both from a timing and complexity perspective in a real-world scenario.

**Dynamic Component Management**

The component manager is the central piece of the experiments. It allows for loading and unloading components at different rates. The pxh module in PX4 allows for easier control through a programmatic interface. An improvement of the existing dynamic loading mechanism was made in order to add authentication capabilities and interface checking before the component can be used. The component manager was implemented as a PX4 application thread running in user space, but as discussed previously, future implementation can move it to kernel space.

**Test Results**

Several tests were performed to estimate the feasibility of such a dynamic scheme with continuous component updates. There were two major classes of experiments. The first class of experiments was to replace an existing component dynamically just once. This included handling authentication and state recovery before the new component was run. The purpose of our test setup was to prove that a component can successfully be reloaded while the system operates and still provide smooth operation. Our experiments showed that this is possible, and the effects of authentication and state handling did not perturb the mission, as well as swapping components in mid-flight. The illustrated scenario is to have a component under attack and, after replacing it with a new component in order to erase the effects of the attack and resume normal operation. The previously described attack methods were used to launch an attack and affect the operation of the original component.

The choice was to perform an attack on the original attitude controller that is part of PX4. This controller subscribes to several topics and calculates the vehicle rates and the attitude setpoint based on the data it receives from the messages. Our attack used the /proc file system method to overwrite these variables constantly as the attitude controller works. The assumption is that the attacker is familiar with the offsets for these variables in the component's memory. Once the new instance of the attitude controller is loaded, the compromised component is unloaded from memory, and the effects of the attack are erased.

Initially, both experiments were done for a hovering drone to prove that the solution was working. Finally, a full mission application was run while alternating modules and continuously kept replacing a component dynamically. The mission was successful. The tested dynamic schemes were able to effectively counter an isolated attack against the chosen vulnerable and crucial attitude controller component. The same approach can be implemented for the rate and position controller, the EKF, and any other component that can be under attack and is crucial to the safety of the mission. The UAV mission was preserved in all cases with normal, alternating, and dynamic component replacement. The most significant deviation in the mission was seen during the continuous alternation of the position controller, as shown in Figure 7 compared to Figure 8.

Chart, line chart

Description automatically generated

Figure 7:Original Mission Plot

Chart, line chart

Description automatically generated

Figure 8: Alternating Mission Plot

The authentication of a dynamic component is the most CPU-intensive and slow operation. The time needed to calculate and HMAC was measured with the SHA512 algorithm having a 32 byte long shared key and a compiled component with a length of a little more than 6MB. The time was equal to 26.85 milliseconds on a modern laptop. Since this is done only once, the time is adequate for our experiments and for most CPS in use today. *key:546573744861636B31323334353637384861636B546573743837363534333231*

*plain text length 6066896*

*hmac:7DCC4B68C65478C60CF19C31B61F535B2D9EB26384539FC5172866F0121831096D828AFFF28560A9656BEBE2953D1279E77FEEFED155EC3C34352263985B54CF*

*Time difference = 25986[µs] + 25986585[ns]*

This experiment did not require a state update since the components were given enough time to come up to speed before switching, and the attitude controller does not have a very complex state. The time to save the state for the mc\_att\_control component was an experiment for completeness so that one can have an idea in case the method for state recovery needed to be used in future experiments. The time to save the state was much longer than the time to restore it, but still, at 97 microseconds, this is negligible and allows for implementations even in restricted CPU environments.

*Time to save state = 97[µs] and 97346[ns]*

*Time to restore state = 8[µs] and 8966[ns]*

The second class of experiments was more ambitious and included having two components with identical or similar functionality, alternatively being in control. A second version of the position controller was created during this scenario and had the original and the alternative position controller change as frequently as two times a second. Each component's instance runs for that period, and then it gets destroyed, and a new instance is spawned in its place, ready to run. During this time, the new controller runs, and then it gets unloaded, and the cycle continues, as shown in Figure 5. This scenario did not perform authentication since the two controllers were loaded at the initialization of the software. There was state recovery since the position controller component has a PID loop and other flight-related state variables. While changing the instances in control were changed, the mission was completed successfully.

This test explores the elements of software diversity and redundancy as methods for improving reliability and security. The goal is to prevent an attack in the first place. Alternating two threads, performing the same task when possible while their memory footprint changes dynamically, is one strategy to increase the difficulty of devising a successful attack. Since it is not hard to introduce randomness in the time that each thread runs, this can further make the solution more resistant. The tests with the alternate implementation of the attitude controller showed no change or degradation in the flight quality since there is no significant state that the attitude controller contains.

Alternating the controller is possible, although some effects are due to timing and state recovery. Overall, the mission was successful; it was noticed that the flight time had increased from 2 minutes and 20 seconds to 3 minutes and 20 seconds. The mission trajectory showed some deviations, although not very significant ones. All flight parameters were affected compared to the pristine case, where there were no controller changes during the flight, but the approach was feasible for a relatively complex and dynamic mission which was completed successfully during our simulation.

The results were analyzed through the flight review tool as part of the PX4 ecosystem. They showed that the local position deviates slightly from the estimated position during the test, where our position controller is swapped continuously. This is shown in Figure 9 and Figure 10. The actuator outputs also show larger swings during alternative execution, as shown in Figure 11 and Figure 12. There was also some increase in the vibrations of the UAV. Finally, it can be seen that there is a slight increase in CPU and memory usage as can be seen from Figure 13 and Figure 14. These results are expected as an aggressive restart of alternate versions of the position controller was performed, which is one of the most important controllers in PX4. Overall, the results are very promising because the mission was completed with some precision and time penalties, but the important result was that it finished successfully. This result is very pertinent to safety-critical missions where safety is more important than efficiency. This is particularly true for military operations where the loss of life and equipment is more important than time to complete the mission or some other mission criteria.

Chart, line chart

Description automatically generated

Figure : Local X Position Normal

Chart, line chart

Description automatically generated

Figure 10: Local X Position Alternate

Graphical user interface, chart

Description automatically generated

Figure :Actuator Outputs Normal

Chart, bar chart

Description automatically generated

Figure 12: Actuator Outputs AlternateGraphical user interface, application

Description automatically generated

Figure 13: CPU and RAM Normal Run

Graphical user interface, application

Description automatically generated

Figure 14: CPU and RAM Alternate Run

**Conclusions**

This study aims to explore the possibility of improving the security of existing systems by retrofitting them with schemes allowing dynamic component management. For the study, modern software architectures such as PX4 were explored as an example, although the results are applicable to other similar systems. The running behavior of the modified autopilot with dynamic component management proved that such an approach is not only possible but holds much promise for the ever-increasing threats in the quickly changing world of deployed UAVs. The results from our experiments showed minimal effects on the mission execution and remarkable resilience against a specific class of widespread attacks. This happened independently of the fact that alternate versions of the position controller were continuously reloaded. Further studies will be needed to continue exploring the possibilities in this study. Hopefully, these results and recommendations will affect architectural decisions of current and future systems regarding their resistance to malicious attacks and failures.

**References:**

Beri, P. S., & Mishra, A. (2019, 23-25 April 2019). Dynamic Software Component Authentication for Autonomous Systems using Slack space. 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI),

Bortolini, M., Galizia, F. G., & Mora, C. (2018). Reconfigurable manufacturing systems: Literature review and research trend. *Journal of Manufacturing Systems*, *49*, 93-106. <https://doi.org/https://doi.org/10.1016/j.jmsy.2018.09.005>

Fitzgerald, J. S. (2016). Title : Resilience Profiling in the Model-Based Design of Cyber-Physical Systems.

Grochowski, K., Breiter, M., & Nowak, R. M. (2019). Serialization in Object-Oriented Programming Languages.

Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., . . . Wunderlich, H. J. (2011, 9-14 Oct. 2011). Design and architectures for dependable embedded systems. 2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS),

Hidenori, Kobayashi, Yang, J., & Sethumadhavan, S. (2019). YOLO Frequently Resetting CPS for Security.

Holthusen, S., Quinton, S., Schaefer, I., Schlatow, J., & Wegner, M. (2016). Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates. PrePost@IFM,

Isakovic, H. (2022). *Towards Dependable CPS/IoT Ecosystem* Wien].

Kapova, L., Buhnova, B., Martens, A., Happe, J., & Reussner, R. (2010). *State dependence in performance evaluation of component-based software systems* Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering, San Jose, California, USA. <https://doi.org/10.1145/1712605.1712613>

Knight, J. C., & Strunk, E. A. (2004, 2004//). Achieving Critical System Survivability Through Software Architectures. Architecting Dependable Systems II, Berlin, Heidelberg.

Lauer, M., Amy, M., Fabre, J., Roy, M., Excoffon, W., & Stoicescu, M. (2016, 7-9 Jan. 2016). Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS. 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE),

Lauer, M., Amy, M., Fabre, J. C., Roy, M., Excoffon, W., & Stoicescu, M. (2018). Resilient computing on ROS using adaptive fault tolerance. *Journal of Software: Evolution and Process*, *30*(3), e1917.

Malavolta, I., Lewis, G., Schmerl, B., Lago, P., & Garlan, D. (2020, 5-11 Oct. 2020). How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems. 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP),

Meier, L., Honegger, D., & Pollefeys, M. (2015, 26-30 May 2015). PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. 2015 IEEE International Conference on Robotics and Automation (ICRA),

Nayak, P., Hibler, M., Johnson, D., & Eide, E. (2017, 2017//). A Wingman for Virtual Appliances. Run-time Verification, Cham.

Parr, T. J., & Quong, R. W. (1995). ANTLR: a predicated-<i>LL(k)</i> parser generator. *Softw. Pract. Exper.*, *25*(7), 789–810. <https://doi.org/10.1002/spe.4380250705>

Pavlenko, E., Zegzhda, D., & Poltavtseva, M. (2019, 6-9 May 2019). Ensuring the sustainability of cyberphysical systems based on dynamic reconfiguration. 2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS),

Potteiger, B., Zhang, Z., & Koutsoukos, X. (2020). Integrated moving target defense and control reconfiguration for securing Cyber-Physical systems. *Microprocessors and Microsystems*, *73*, 102954. <https://doi.org/https://doi.org/10.1016/j.micpro.2019.102954>

Saadi, A., Oussalah, M. C., Hammal, Y., & Henni, A. (2018, 24-25 Nov. 2018). An approach for the dynamic reconfiguration of software architecture. 2018 International Conference on Applied Smart Systems (ICASS),

Shin, H.-J., Cho, K.-W., & Oh, C.-H. (2018). SVM-Based Dynamic Reconfiguration CPS for Manufacturing System in Industry 4.0. *Wireless Communications and Mobile Computing*, *2018*, 5795037. <https://doi.org/10.1155/2018/5795037>

Stockmann, L., Laux, S., & Bodden, E. (2019, 25-26 March 2019). Architectural Runtime Verification. 2019 IEEE International Conference on Software Architecture Companion (ICSA-C),

Van, J. (2017). VulCAN : Efficient Component Authentication and Software Isolation for Automotive Control Networks.

Vasudevan, A., Chaki, S., Maniatis, P., Jia, L., & Datta, A. (2016). {überSpark}: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. 25th USENIX Security Symposium (USENIX Security 16),

Wang, R., Song, H., Jing, Y., Yang, K., Guan, Y., & Sun, J. (2019, 11-12 Nov. 2019). A Sensor Attack Detection Method in Intelligent Vehicle with Multiple Sensors. 2019 IEEE International Conference on Industrial Internet (ICII),

Yaacoub, J.-P., Noura, H., Salman, O., & Chehab, A. (2020). Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things*, *11*, 100218. <https://doi.org/https://doi.org/10.1016/j.iot.2020.100218>

**Biographies**

**ANTON D. HRISTOZOV** graduated from the University of Pittsburgh with a Masters in Telecommunications and Information Science. He is currently pursuing a doctoral degree in Technology at Purdue University while working as a research engineer. His research interests involve embedded systems, Internet of things, smart sensors and real time systems. He is a Linux enthusiast and enjoys working with cyber physical systems which use sensors and convert energy in different forms. Smart energy and the use of artificial intelligence is a new and interesting direction, given the recent advances in use of machine learning in cyber physical systems. Anton may be reached at [ahristoz@purdue.edu](mailto:ahristoz@purdue.edu)

**DR. ERIC T. MATSON** Ph.D. is a Professor and University Faculty Scholar in the Department of Computer and Information Technology at Purdue University, West Lafayette.  He is the Director of the of the Korean Software Square at Purdue and the co-founder of the M2M Lab at Purdue University, which performs research at the areas of multiagent systems, cooperative robotics and wireless communication.  The application areas are focused on safety and security robotics systems.  His research has generated high levels of external funding, approximately 190 published, refereed articles, numerous MS and PhD students, and practical, demonstrable results for a number of sponsors.  Prof. Matson has a Ph.D. in Computer Science and Engineering from the University of Cincinnati, M.B.A in Operations Management from Ohio State University and B.S. and M.S.E. degrees in Computer Science from Kansas State University. Dr. Matson can be reached at [ematson@purdue.edu](mailto:ematson@purdue.edu)

**J. ERIC DIETZ**, PhD, PE is the Director of the Purdue Homeland Security Institute and Professor of Computer and Information Technology.  Eric also directs the Purdue Military Research Institute, which seeks to support military officers research who will maintain a technology superiority in national defense and security programs.   Dr. Dietz research includes measurement and modeling of cyberphysical security systems, energy security and heath response to disaster.  Eric earned a BS and MS from Rose-Hulman Institute of Technology and a PhD in Chemical Engineering from Purdue University. He can be reached at [jedietz@purdue.edu](mailto:jedietz@purdue.edu)

**DR. MARCUS ROGERS** is a Professor and Assistant Dean for Cybersecurity Initiatives at Purdue Polytechnic. He earned his BA, MA and PhD in Forensic Psychology from The University of Manitoba. Dr. Rogers interests include digital forensic analysis, applied behavioral analysis and digital evidence. Dr. Rogers can be reached at [rogersmk@purdue.edu](mailto:rogersmk@purdue.edu)