# DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems

Tigist Abera*, Raad Bahmani*, Ferdinand Brasser*, Ahmad Ibrahim*, Ahmad-Reza Sadeghi*, and Matthias Schunter†
*Technische Universität Darmstadt, Germany
†Intel Labs, Portland, OR, U.S.A.
{tigist.abera, raad.bahmani, ferdinand.brasser, ahmad.ibrahim, ahmad.sadeghi}@trust.tu-darmstadt.de,
matthias.schunter@intel.com

***Abstract*—Networks of autonomous collaborative embedded systems are emerging in many application domains such as vehicular ad-hoc networks, robotic factory workers, search/rescue robots, delivery and search drones. To perform their collaborative tasks the involved devices exchange various types of information such as sensor data, status information, and commands. For the correct operation of these complex systems each device must be able to verify that the data coming from other devices is correct and has not been maliciously altered.**

**In this paper, we present** DIAT **– a novel approach that allows to verify the correctness of data by attesting the correct generation as well as processing of data using control-flow attestation.** DIAT **enables devices in autonomous collaborative networks to securely and efficiently interact, relying on a minimal TCB. It ensures that the data sent from one device to another device is not maliciously changed, neither during transport nor during generation or processing on the originating device. Data exchanged between devices in the network is therefore authenticated along with a proof of integrity of all software involved in its generation and processing. To enable this, the embedded devices' software is decomposed into simple interacting modules reducing the amount and complexity of software that needs to be attested, i.e., only those modules that process the data are relevant. As a proof of concept we implemented and evaluated our scheme** DIAT **on a state-of-the-art flight controller for drones. Furthermore, we evaluated our scheme in a simulation environment to demonstrate its scalability for large-scale systems.**

## I. Introduction

Embedded systems have been omnipresent for many years mostly performing simple tasks in isolation. However, emerging applications such as IoT (e.g., smart cities/homes/factories) and autonomous systems (e.g., cars, drones) require embedded systems to be highly connected and carry out *autonomous* as well as *collaborative* tasks. In autonomous collaborative system no central entity coordinates actions of the individual (autonomous) devices. The involved devices interact with each other and coordinate their actions by exchanging information, such as sensor data, status information, and commands. The correct and secure functioning of an autonomous collaborative system strongly relies on the integrity of the devices involved in its operation. In particular, it must be ensured that the data (sensor, commands, status) on a device has not been *maliciously* modified before it is exchanged with any other device.

*Remote attestation* is a powerful security service for verifying the integrity of the software state of remote devices. It enables a remote entity, called *verifier*, to verify the integrity of the software running on an untrusted device, called the *prover*. However, designing attestation schemes for autonomous collaborative systems poses a number of challenges that cannot be met by the existing attestation solutions. Conventional remote attestation solutions face two problems. They are static and not scalable.

Static attestation approaches provide a proof that the software initially loaded by the prover is unmodified [33], [25], [16], but cannot detect run-time attacks that exploit, for instance, code-reuse techniques [34], [9], [17]. Recently progress has been made in tackling the static character of conventional attestation: The work in [2], [14], [36], [15] proposes *control-flow* attestation of the code running on the prover device. This enables the verifier to detect run-time attacks based on code reuse. Enforcement techniques for control-flow correctness, such as control-flow integrity (CFI) [1], do not provide information about the executed control-flow path on collaborating devices. Control-flow attestation, however, gives the verifier information about the executed control-flow, which enables the verifier to detect not only attacks that do not conform to a software program's control-flow graph, like return-oriented programming (ROP) [34], but also a subset of attacks that lead to a valid but unintended program execution, i.e., non-control-data attacks [11], [20]. Moreover, control-flow attestation enables the verifier to determine the appropriate reaction, in case of an attack. With CFI, a violating device would be stopped and possibly crashed, which is particularly dangerous in safety critical applications. With control-flow attestation, in contrast, contextual reactions can be implemented, e.g., excluding a compromised device from the collaboration and the entire communication.

Unfortunately, control-flow attestation solutions assume a powerful verifier that can maintain and search in a very

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23420
www.ndss-symposium.org

large database of execution paths which is very expensive for autonomous systems where each embedded device must be able to act as both verifier and prover.

The other limitation of conventional attestation solutions lies in their scalability, i.e., they only allow for attesting individual devices [33], [25], [16]. This problem was addressed by *collective attestation* of networks of connected devices [6], [5], [21]. However, these schemes mainly assume a central verifier which is not available in autonomous networks where the verifiers are distributed.

**Goal and Contributions.** We aim at tackling the above-mentioned problems and present DIAT – the first efficient and secure collective *run-time attestation* scheme for autonomous collaborative systems of embedded devices. In particular, DIAT ensures on-device data integrity against malicious manipulation. In the context of Data-Flow Integrity (DFI) the term data integrity is used to describe that *all* operations on data and variables obey a program's Data-Flow Graph (DFG). In contrast, in this work we focus on data that has been explicitly selected to be controlled. This data inherits its integrity from the integrity of the software modules that processed it. We provide a detailed comparison to related work on data integrity in Section IX.

We achieve efficiency by decomposing the underlying embedded software into small interacting software *modules* and attest the control-flow of those modules that are relevant for data exchanged in a given interaction. For example, consider two drones that are interacting by exchanging their location information (e.g., GPS coordinates). The software modules to be attested are those responsible for determining and possibly processing (altering) a drone's GPS coordinates. The control-flow attestation guarantees that the data is only processed in a benign execution path. We implemented and evaluated DIAT on collaborating autonomous drones, demonstrating its effectiveness and efficiency.

In summary, DIAT provides end-to-end data integrity protection for collaborative autonomous networks. This requires tackling multiple challenges that constitute our individual technical contributions:

- *Data Integrity Attestation.* We secure devices' interaction by ensuring the integrity of data shared between devices (e.g., sensor readings). This is done by linking the data with an attestation report reflecting the correct generation and processing of the data, given our adversary model (see Section II).
- *Modular Attestation.* We present the design and implementation of modular attestation – The software on prover devices is decomposed into simple interacting modules and only those modules that process a particular data of interest are attested. Modular decomposition reduces the attestation overhead on both the prover and the verifier devices.
- *Novel Execution-Path Representation.* We propose a novel representation of execution paths, which allows control-flow attestation of complex software programs. Our solution has linear overhead on the verifier, compared to the exponential overhead imposed by previously proposed solutions [2], [14].
- *Implementation and Evaluation.* We present a proof-of-concept implementation of our scheme based on Pixhawk PX4 flight controller software.[1] Our simulation results for autonomous networks show the scalability of DIAT.

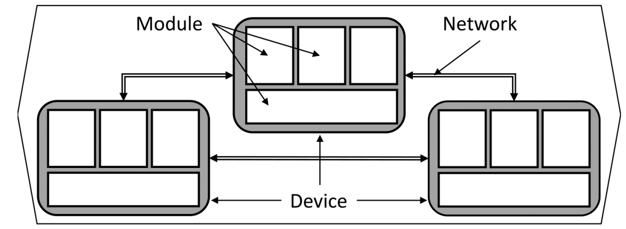

Figure 1: Abstract view of a collaborative autonomous system.

## II. System Model and Assumptions

We consider *collaborative autonomous systems*, i.e., networks of connected entities (devices) which interact with each other to perform one or more tasks. In such a network, typically no central entity is needed to coordinate actions of individual devices. However, a central entity may be used for maintenance reasons. The devices within the network are mutually distrusting. Redundancy in the network allows the overall system to tolerate the misbehavior of individual devices that can be caused due to faults or by attacks on devices. Redundancy is achieved, for instance, when the network is formed of homogeneous devices, so that one device can easily be replaced by another device in the network. In this work, for brevity, we assume a *homogeneous* network. We stress that the focus of this paper is on how to detect the misbehavior of a device in a collaborative autonomous system. How to react to an infected device is a complementary problem and depends on the underlying security policy.

**Devices.** Each individual device is self-contained and autonomous, and can perform basic tasks by itself. Further, in order to coordinate their actions while performing more complex tasks, the devices exchange information, such as sensor data, status information, and commands. The software stack of embedded devices is less complex than on typical desktop or server systems. However, this does not preclude devices with embedded operating systems that run multiple tasks in parallel (e.g., real-time operating systems). Figure 1 shows the software model we assume in this work. Individual tasks or *software modules* of a device are strongly isolated from all other software components, including operating system and other privileged software, based on lightweight embedded hardware security architecture (see Section VIII). The communication between software modules takes place over a well-defined interface which allows tracking of data-flow between the modules. Our implementation on a popular flight controller for drones shows that our assumptions are realistic for the class of devices for which DIAT was designed.

[1] https://dev.px4.io/en/concept/flight_stack

**Communication.** Devices are connected through wireless network technology, like WiFi, Bluetooth or some custom solution, where each device can be uniquely addressed. Communication does not need to be direct, i.e., devices could, for example, form a meshed network to forward messages to each other.

## *A. Adversary Model*

We assume the adversary has compromised and gained control over a subset of devices in a collaborative system. The adversary's goal is to manipulate collaborative tasks by sending manipulated data to other, un-compromised autonomous devices.

We assume a stealthy adversary that has the goal to undermine the correct behavior of autonomous devices while evade detection. Therefore, denial-of-service (DoS) attacks, e.g, jamming the network communication between devices or trying to destroy devices (e.g., one device physically attacking/crashing into another device in case of drones or vehicles), are out of the scope of this work.

As common in remote attestation literature, we consider software-only attacks. However, unlike conventional attestation schemes we assume that the adversary can manipulate code, e.g., while stored on persistent storage, as well as launch run-time attacks. Run-time attacks can be divided into: (1) *control-data* attacks, that introduce non-existing edges to the execution path, which are not part of the program's Control-Flow Graph (CFG), e.g., Return-Oriented Programming (ROP) attacks [34]; and (2) *non-control data* attacks. Non-control data attacks can be split into two sub-classes. (2a) Attacks that do not add new edges but have an observable effect on the control flow of execution, like unexpected number of loop iterations; and (2b) attacks that do not change the executed control flow at all, e.g., changing of variables used in the generation of data. We excluded attacks that do not change the control flow as they are subject to active research and no general detection policy is known at the time of writing. If an appropriate policy is developed in the future DIAT can be adapted to utilize it to also cover such attacks.

Each device of the network is equipped with a lightweight hardware security architecture, which protects its TCB. All other software, including the untrusted OS, is assumed to be potentially compromised. The hardware security architecture further ensures isolation between a device's software modules. Hardware attacks are considered out of scope in this work. Embedded security architectures are highly integrated into the system-on-a-chip (SoC) designs and cannot be easily attacked [36].

We assume that sensors and actuators of a device are trusted and report benign readings and perform actions as instructed.[2] This precludes false data injection attacks like spoofed GPS signals. However, the software and drivers controlling the sensors and actuators might be controlled by the adversary.

[2]Misreading sensors and misbehavior of actuators due to faults are an orthogonal problem and can be handled by fault tolerant designs.

## *B. Requirements*

The main goal of DIAT is to enable efficient and secure interaction/collaboration of embedded devices in an autonomous system. This concerns several objectives as follows:

- *Code integrity on devices*: Unintentional/malicious alternation of the code running on a device can be detected.
- *Data integrity on devices*: Unintentional/malicious alternation of the data on a device (before being sent out to other devices) can be detected. This means data can only be modified in a non-malicious way. This is necessary as devices do not only exchange raw data, like sensor readings, but mostly processed data. For instance, when sending position information the receiver expects coordinates instead of a set of timestamps sent out by the GPS satellites (as received by the GPS driver module).
- *Data integrity and authenticity during transportation*: Malicious alternation of the data when traversing from one device to another must be detected.

Attestation schemes are used to ensure the code and data integrity on devices, i.e., any manipulation can be detected by the verifying device. To capture the run-time behavior of the code DIAT adopts the idea of control-flow attestation [2]. However, in the context of autonomous embedded systems the attestation scheme must have the following properties:

- *Attestation efficiency*: Attestation is applied only to the critical code, i.e., those modules that process a particular data of interest. Attesting the entire software on a device would allow making a statement about the correctness of the data being processed. However, this induces a huge overhead as the code processing a specific piece of data is usually only a small subset of the entire software stack.
- *Attestation latency*: The attestation should not cause delays beyond the bounds required by autonomous systems. More concretely, the generation of the attestation report on the prover side must not delay the sending of data, while the verification of the attestation report on the verifier side must not delay the processing/usage of the received data.

To meet the above mentioned objectives, each device must fulfill the following requirements:

- *Isolation architecture*: The software modules/components are isolated from each other. Additionally, function modules are isolated from privileged software components such as operating system. Therefore, the OS is not part of DIAT's TCB.
- *Data-flow monitoring*: Software components/modules that access the data of interest can be identified.
- *Control-flow monitoring*: The control-flow of individual modules must be captured when required.
- *Device key*: Each device is equipped with a platform key-pair. The device's secret key and other cryptographic parameters are protected by the hardware security architecture. It is exclusively accessible from the device's TCB. Hence, the adversary cannot forge the attestation reports generated by the device's TCB or extract device's

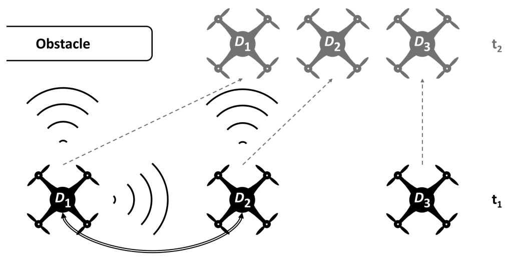

Figure 2: Example of collaborative drones.

secret keys.

- *Attestor*: In a remote attestation protocol, the component(s) involved in measuring, attesting, and verifying a system's state are protected as part of the TCB.

Examples of lightweight security architecture providing run-time isolation, secure storage, and secure boot (integrity) in order to protect DIAT's TCB have been developed by academia (e.g., TrustLite [24] and TyTAN [7]) as well as industry (e.g., ARM TrustZone-M[3]). We discuss how these solutions can be adopted by DIAT in Section VIII.

On the communication between the individual devices the following requirements are imposed:

- *Secure channel*: The integrity and authenticity of data sent over the network between the devices must be ensured. To enable the establishment of a secure channel each device must be equipped with platform key-pair.
- *Infrastructure*: Public key infrastructure (PKI) must be in place.

**Use-case Example.** We will now introduce an example use case of autonomous systems to illustrate our concept. The example is simplified for the sake of clear illustration, however, the principle remains the same even for larger and more complex systems.

Consider a set of autonomously flying drones that collaborate to perform a distributed ground search, e.g., as part of a search and rescue mission. Each drone has a set of sensors, like cameras, which allow it to monitor a certain area at a time. In order to cover more ground in a short time frame, multiple drones fly in formation. When the drones operate in close proximity they have to coordinate with each other to avoid collisions. Figure 2 shows a scenario where a drone has to evade an obstacle.

The three drones ($D_1$, $D_2$, and $D_3$) are flying in line abreast. Each drone is an autonomous entity that can operate on its own. In particular, each drone is equipped with sensors which enable it to position itself (e.g., GPS), measure its movement (accelerometer, gyroscope, etc.), and sense its environment (e.g., via ultrasonic, infrared, Lidar). Note that, those sensors are common in commercial off-the-shelf (COTS) systems and even customer grade drones are equipped with such sensors.[4] Finally, the drones are wirelessly connected to each other and can coordinate their actions.

[3] https://community.arm.com/processors/trustzone-for-armv8-m/b/blog
[4] http://www.dji.com/phantom-4-pro

During the drones' mission, $D_1$ detects an obstacle through a front-facing distance sensor. $D_1$ has three options for reacting to this situation: (1) it could abort its mission, i.e., stop or fly back; or it could circumnavigate by either (2) moving to the left, or (3) to the right. $D_1$ by itself does not have sufficient information to be able to decide in which direction to move, as the obstacle could expand in both directions. By exchanging information with other drones, however, $D_1$ can learn that no obstacle has been detected in front of $D_2$. As a consequence, $D_1$ decides to evade the obstacle by moving to the right. Albeit this would prevent $D_1$ from colliding with the obstacle it would lead to a collision with $D_2$. Note that, $D_1$ is also aware of $D_2$'s position through position sharing among the drones. Therefore, in order to avoid another collision, $D_1$ has to coordinate with $D_2$. In particular, $D_1$ requests $D_2$ to also move to the right to make space.

The result of the interaction between $D_1$ and $D_2$ is shown in Figure 2. The gray drones show their positions after they have coordinated their actions, where all three drones can pass by the obstacle safely.

## III. DIAT Design

The core idea of DIAT is to enable autonomous devices to trust the exchanged information needed to perform a collaborative task within a network. This is done by means of a remote run-time attestation scheme that allows devices to provide an authentic integrity proof of the data they are exchanging. Whenever information is exchanged, the sender augments the data with a proof that this data has been generated and processed correctly. The receiver can then verify this integrity proof and gain trust in the correctness of the received data.

Static (binary) attestation allows the verifier to detect manipulations of the static code and data, e.g., due to a malware infection. However, static attestation cannot capture the run-time behavior of the code and hence cannot detect run-time attacks that leverage state-of-the-art code-reuse techniques, such as return-oriented programming (ROP) [34]. To detect this class of attacks, researchers have recently taken the first steps towards run-time attestation schemes by means of control-flow attestation that records the execution path of the code running on the prover [2], [14]. Unfortunately, control-flow attestation schemes pose a significant overhead on both the prover and the verifier making it impractical for resource-constrained embedded systems. DIAT significantly reduces the overhead of control-flow attestation and makes it applicable to collaborative systems of embedded devices.

### A. Challenges

The main challenge of DIAT is to enable secure and efficient data integrity and control-flow attestation. Efficiency plays an important role given that collaborating devices are resource constrained embedded devices that must act simultaneously as both verifier and prover. As mentioned before, run-time attestation incurs significant overhead on the involved entities since: (1) as provers they need to continuously monitor the execution of their software, and (2) as verifiers, they

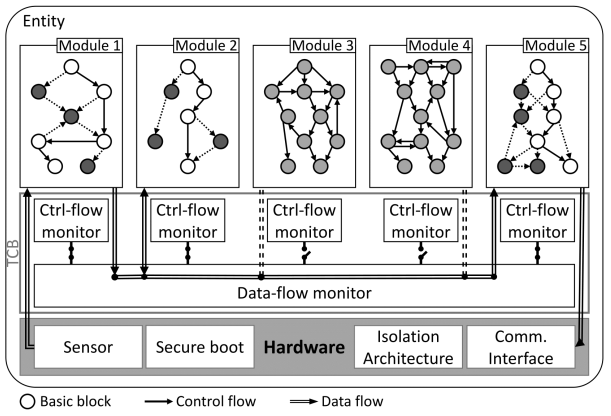

Figure 3: DIAT system architecture. Open switches symbolize activation of control-flow monitoring.

need to know the benign execution paths and verify them at run-time. The main contribution of DIAT is to tackle these challenges by (1) minimizing the size of the code to be attested, (2) minimizing the attestation time, and (3) reducing the size of attestation reports, allowing for secure and efficient attestation. In particular, code size is minimized using *modular attestation*, attestation time is reduced based on *data-flow monitoring*, and attestation reports are compressed by using multisets for *execution path representation*.

## B. Architecture

Figure 3 shows the abstract view of a device using DIAT. To enable data integrity attestation all modules that process the data of interest are attested while they are processing this data. This reduces (1) the code to be attested to only those modules that process the data; and (2) the attestation time, i.e., modules are only attested during the time frame in which they are processing the data. As shown in Figure 3, the data-flow monitor DFMonitor traces the flow of data within the device and activates attestation for each module that processes the data of interest. In Figure 3, data flows first from the sensor into `Module 1`, further into `Module 2` and finally into `Module 5` from where it is sent out. The DFMonitor traces this flow of data and activates the control-flow monitor CFMonitor for the corresponding modules (shown by the closed switches between CFMonitor and DFMonitor for the involved modules). We refer to data of interest as *sensitive data* and to software modules involved in the processing of sensitive data as *critical tasks* or *critical modules*.

Please note that sensitive data is application-dependent. For instance, in the scenario described in the use case example, the location information is considered sensitive data (Section II). However, in other scenarios (or applications) different data may be considered sensitive. Consequently, critical modules must be determined dynamically at run-time.

The control-flow information of critical modules is recorded using a multiset that indicates the number of executions for all branches taken at run-time. Details of control-flow recording are presented in Section IV. Using this execution path representation the verifier learns for instance how many times a loop has been iterated through. This enables the detection of some classes of data-only attacks, e.g., those which increase the number of iterations of a loop (see Section VII). However, unlike previous control-flow attestation schemes [2], the verification overhead is significantly reduced, as the verifier does not need to maintain and search a database which includes all valid execution paths of software modules. It is sufficient for the verifier to know the Control-Flow Graph (CFG) of the attested modules. In the following we elaborate on the three main building blocks of our design (1) modular attestation, (2) data-flow monitoring, and (3) execution path representation.

**Modular Attestation.** The software on each device is decomposed into a number of small interacting software modules or tasks. Modular attestation is enabled by the fact that modules are isolated, i.e., the control-flow path recorded for a module does indeed represent the behavior of that module. The underlying lightweight hardware security architecture [7] enables isolation of modules such that the memory and the execution of a module cannot be influenced by other modules or even by privileged software like the operating system. We discuss lightweight hardware security architectures for isolation in Section VIII. Modules are defined/identified using static analysis based on data and control-flow dependencies. DIAT does not require programmer's assistance, e.g., in form of code annotation. Further, modules can be overly large, however, this would only influence performance but not the security of DIAT. Note that, many avionic/automotive systems (including flight controllers for drones) already have modular design [28]. This modular design is due to safety aspects mainly in autonomous systems in vehicles. Modular software design is a well-investigated topic in software engineering [4], [19], [3]. The cost of modular transformation is highly application dependent and may not be formulated in general statements. DIAT targets complex software that is designed in a modular fashion. Small embedded software, e.g., controller for a sensor, may indeed be monolithic. However, such software is not the focus of DIAT.

**Data-Flow Monitoring.** Identifying the modules involved in generating a particular data is accomplished via coarse-grained data-flow monitoring. As modules are isolated, they cannot directly communicate with each other (e.g., by memory access). Communication happens through a well-defined interface controlled and monitored by a component of DIAT referred to as DFMonitor. This enables DIAT to trace the data-flow between modules and dynamically identify the critical modules for particular sensitive data. Services for communication between modules or tasks are common in most system architectures. Therefore, DIAT does not incur a system redesign. The communication service is only slightly extended to trace the flow of data. DFMonitor is described in more detail in Section V.

**Execution Path Representation.** Existing control-flow attestation schemes [2], [14] induce a high overhead on the verifier, who is expected to store and repeatedly search a very large database of *all* possible execution paths. The size of this database grows exponentially with the number of control-flow events in the code. To tackle this problem, we designed a novel execution path representation that is based on Multiset Hash (MSH) functions [12]. Our representation provides an under-approximation of executed paths. However, the amount of information provided by this representation is sufficient to detect all attacks that cause deviation from the benign control-flow, e.g., ROP attacks, as well as general Data-Oriented Programming (DOP) attacks. We elaborate on this in Section VII. In particular, our MSH-based representation preserves the overall execution path and the number of times loops are executed. However, for each loop it does not preserve the order of different paths that are executed within the loop. This leads to a verification overhead which is linear in the number of control-flow events. The verification policy, which enables the detection of different kinds of attacks can be freely defined by the verifier and updated after the system deployment.

**Summary.** DIAT identifies at run time the software components, which process a particular sensitive data (e.g., sensor inputs that we consider benign). If all of these components are benign, then the integrity of the data is preserved. In order to detect malicious data modifications, we link exchanged data with a run-time attestation report, which attests the execution path of all components involved in the modification of this data. Hence, data integrity is provided by a proof of the correct processing of data based on control-flow attestation. Other unauthorized data modifications are prevented by module isolation and secure channels.

## IV. Protocol Description

In this section we describe DIAT's attestation protocol in more details. For brevity we explain the protocol between two devices and later extend it to connected networks. DIAT augments the devices' messages and execution with the necessary control-flow attestation and authentication in order to secure devices' interactions, hereby providing a secure and resilient autonomous network.

### A. Multiset Hash Function

Before going into protocol details we first explain the Multiset Hash (MSH) function that is used to record the control-flow of modules. A multiset is a set that allows its members to occur multiple times. The number of occurrences of an element in a multiset is called the multiplicity of that element. A multiset hash function is a hash function that generates a fixed-length digest, denoted by MSH-value, for a given multiset of arbitrary number of elements. Unlike traditional hash functions, MSH allows updating the hash digest by incrementally adding an element to the multiset. Consequently, two hash values for the same multiset can be compared regardless of the order in which the elements were added to the hash digest. As an example, consider the multiset $M = \{<A,1>, <B,1>, <C,3>\}$, which is formed of the elements $A$ with multiplicity 1, $B$ with multiplicity 1, and $C$ with multiplicity 3. Consider two MSH-values $val_1$ and $val_2$ that are generated over $M$. $val_1$ is generated by adding the elements $B$, $C$, $C$, and $C$ in that order to the hash digest of $A$, while $val_2$ is generated by adding the elements $C$, $B$, $C$, and $A$ to the hash value of $C$. A MSH scheme allows verifying that $val_1$ and $val_2$ are equivalent, i.e., they are indeed hash values of the same multiset $M$. Finally, A MSH is *multiset-collision resistant* if it is difficult to find two different multisets for which the MSH generates equivalent MSH-values.

### B. DIAT *for Two Devices $D_i$ and $D_j$*

Figure 4 shows our protocol (referred to as interact) for interaction between two devices $D_i$ and $D_j$. interact starts when $D_i$ sends $D_j$ a request for sensitive data which must be generated and processed securely. This request is accompanied with a fresh random nonce $N$. After receiving the request $D_j$ generates the requested data $\mathbf{data}_j$ (e.g., GPS coordinates) by executing the necessary software exec(**code**).

The data-flow monitor (DFMonitor) monitors continuously the communication between different modules running on $D_j$ and returns the IDs $x_1, \ldots, x_d$ of critical tasks for $\mathbf{data}_j$ (i.e., the IDs of all software modules which influence the final value of the sensitive data $\mathbf{data}_j$).

Concurrently, the control-flow monitor (CFMonitor) starts monitoring the execution and recording the control-flow within critical modules whose IDs $x_1, \ldots, x_d$ are determined by DFMonitor. The control-flow of each critical module is recorded by CFMonitor as a MSH of executed control-flow events, i.e., edges in the Control-Flow Graph (CFG) of the module. For example, for a critical module `Module 1`, the MSH-value $val_1$ of multiset $M_1 = \{<e1,2>, <e2,1>\}$ shows that during execution of `Module 1`, the edge $e1$ was executed twice and the edge $e2$ was executed once. The output of CFMonitor is the set of control-flow attestation results $\mathbf{h}_{x_1}, \ldots, \mathbf{h}_{x_d}$ for the critical modules $x_1, \ldots, x_d$, where $\mathbf{h}_{x_1} = \{val_{x_1}, M_{x_1}\}$.

The data $\mathbf{data}_j$ generated at $D_j$ and the attestation results $\mathbf{h}_{x_1}, \ldots, \mathbf{h}_{x_d}$ are authenticated with a digital signature $\sigma_j$ generated by $D_j$'s secret key $sk_j$. $D_j$ then sends the authenticated data back to $D_i$. Finally, $D_i$ verifies $\sigma_j$. It then verifies the integrity of $\mathbf{data}_j$, by checking the control-flow integrity of all critical tasks on $\mathbf{data}_j$ (verifyCFP). verifyCFP checks whether the executed edges comply to the CFG in order to detect code-reuse attacks, and to a set of predefined policies, e.g., whether the execution of loops is within a certain range or executed privileged function was intended. If all checks succeed, $D_i$ concludes that $\mathbf{data}_j$ is generated correctly (**finalCheck** $\leftarrow 1$).

### C. DIAT *for Autonomous Networks*

Collaboration in an autonomous network may involve interaction between two or more devices. Collaborations involving more than two devices can be secured by running interact (introduced under Section IV-B) recursively between interacting devices. This recursive version is referred to as interact+. An

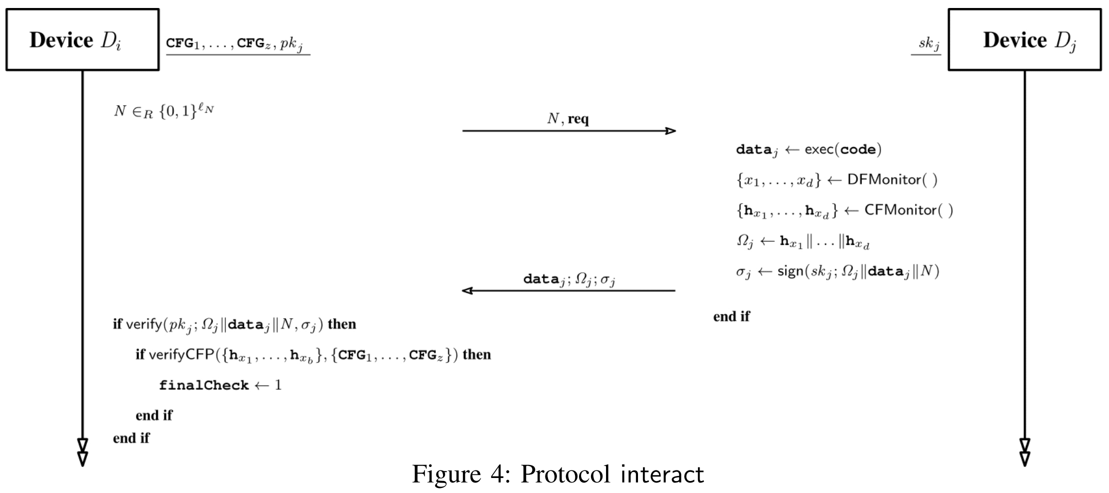
Device $D_i$
$\texttt{CFG}_1, \ldots, \texttt{CFG}_z, pk_j$
$sk_j$
Device $D_j$
$N \in_R \{0,1\}^{\ell_N}$
$N$, **req**
$\texttt{data}_j \leftarrow \text{exec}(\texttt{code})$
$\{x_1, \ldots, x_d\} \leftarrow \text{DFMonitor}(\ )$
$\{\mathbf{h}_{x_1}, \ldots, \mathbf{h}_{x_d}\} \leftarrow \text{CFMonitor}(\ )$
$\Omega_j \leftarrow \mathbf{h}_{x_1} \| \ldots \| \mathbf{h}_{x_d}$
$\sigma_j \leftarrow \text{sign}(sk_j; \Omega_j \| \texttt{data}_j \| N)$
$\texttt{data}_j; \Omega_j; \sigma_j$
end if
if verify$(pk_j; \Omega_j \| \texttt{data}_j \| N, \sigma_j)$ then
if verifyCFP$(\{\mathbf{h}_{x_1}, \ldots, \mathbf{h}_{x_b}\}, \{\texttt{CFG}_1, \ldots, \texttt{CFG}_z\})$ then
$\texttt{finalCheck} \leftarrow 1$
end if
end if

Figure 4: Protocol interact

example of collaboration involving multiple devices is truck platooning, where a device may need to recursively verify the position data of all trucks ahead of it. The main difference to interact is that in interact+ exchanged data is augmented with control-flow attestation results of all critical modules on *every* device that was involved in that collaboration. In more details:

- When a device $D_j$ receives data request from $D_i$, it sends a *new request* to other devices which are supposed to participate in the interaction.
- $D_j$ collects the attestation results of other devices and send them together with its own attestation result to $D_i$.
- By verifying all the attestation results, $D_i$ is ensured that all the data from *all participating devices* were generated correctly.

## V. Implementation

In this section, we detail our implementation of DIAT on the PX4 – an open-source flight controller used for many commercial drones. We first present our implementation platform in Section V-A. Then, in Section V-B we provide details of our implementation, i.e., we elaborate on the implementation of DFMonitor and CFMonitor and their integration within PX4.

### A. Implementation Platform

**Flight Controller (PX4).** We implemented DIAT based on PX4, an autopilot software[5] designed for resource-constrained autonomous aircrafts. PX4 is a modern and well-designed software which is widely used in academic and industrial projects. As depicted in Figure 5, PX4 consists of two software layers: *flight stack* and *middleware*. The flight stack provides the functionalities that are necessary for controlling an aircraft, such as navigation, position estimation, and servo motor control. These functionalities are divided into *software modules* communicating with each other through the middleware using a message distribution system. The middleware consists of two main components:[6] a lightweight *Object Request Broker* (uORB) and *Micro Air Vehicle Link* (MAVLink). We will elaborate on uORB and MAVLink in Section V-B1.

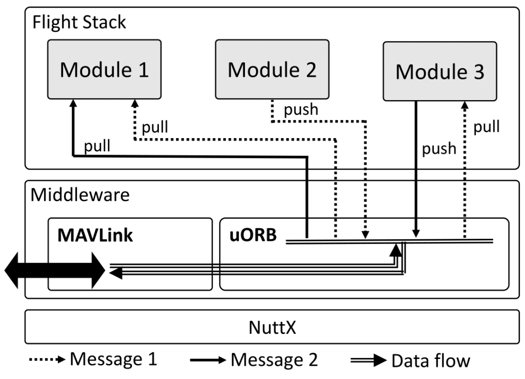

Figure 5: PX4 Architecture

**RTOS (NuttX).** Embedded real-time systems are constrained in processing power. Further, they are expected to fulfill strict timing requirements. These impose high demands on scheduling and resource management tasks. The common approach to fulfill these requirements is to deploy a real-time operating system (RTOS) ensuring the compliance with these demands. The RTOS used primarily together with PX4 is NuttX. The RTOS is not part of DIAT's TCB. Since modules are isolated by the underlying security architecture, the RTOS cannot modify a module's state and hence does not need to be trusted.

**Device Security Architecture.** DIAT can be implemented adopting different underlying security architecture, for instance, TyTAN [7] or TrustZone-M. The processor of our prototype platform does not provide a lightweight hardware security platform. We elaborate on how these security architectures can be used with DIAT in Section VIII.

[5]An autopilot is a software that is responsible for controlling an aircraft and keeping it stable.

[6]Middleware includes further components, e.g. a *simulation layer*, which are not relevant in this work.

### *B. Implementation Details*

DIAT enables the verification of the state and the control-flow of modules involved in the computation of exchanged data. As described in Section III-A, our design includes two main component. (1) DFMonitor that traces the data-flow in the system and activates attestation for relevant modules, and (2) CFMonitor that records the control-flow of these modules. In this section, we describe both components in detail and outline how we integrated them into PX4 autopilot software.

DIAT is a generic solution applicable to embedded systems that satisfy the requirements introduced in Section II. Our prototype implementation is based on the popular open source flight controller for drones – PX4. Our implementation of CFMonitor is platform-agnostic; it can be used for all ARM-based systems. The implementation of DFMonitor can be adjusted and integrated into other systems with minimal effort.

For the remainder of this section, we use *attestation report* to refer to the proof of paths executed by critical modules. The data for which an attestation report is generated is referred to as *attested data* and the process of generating an attestation report is called *attestation*. In order to acquire attested data, the verifier sends an *attestation request* to a device that generates this data. An attestation report is verified by confirming that it reflects legitimate execution paths for all modules that processed the attested data. We refer to this process as *verification*.

*1)* DFMonitor*:* In DIAT the data-flow monitor (DFMonitor) is a trusted software component, which observes the data exchanged between software modules and, based on this information, identifies the critical modules for different computations (see Section III). To achieve this goal, the main demand on DFMonitor is to maintain a comprehensive record of the data exchanged in the system. In PX4 the messaging system (uORB) is the only communication channel between software modules, and hence, it is able to create and continuously update the model of the system's entire information exchange. In DIAT, we have extended uORB with new functionalities. We refer to the extended version as DFMonitor. In the following, we first introduce the internal (uORB) and external (MAVLink) communication components of PX4. Afterward, we describe the functionalities added to uORB. Specifically the adaptations required to enable (1) receiving attestation requests, (2) identifying module dependencies to determine the critical modules for sensitive data, (3) activating CFMonitor for critical modules, and (4) discarding buffered data.

PX4 **Communication Components.** To realize efficient communication among modules and to external entities while providing a flexible and well-designed software architecture PX4 leverages two software components: uORB and MAVLink. uORB is a software component offering a messaging API used by software modules for communication *on* the device. This component provides the functionality for sending/receiving predefined messages using a publish/subscribe scheme. In order to send a message, a software module is required to *register* itself with uORB as a *publisher* of that message type and use the corresponding functionality to send messages. Similarly, to receive a specific message type a software module needs to register itself with uORB as a *subscriber*. When a message is sent, uORB notifies all subscribers of that message type. MAVLink (Micro Air Vehicle Link) is the implementation of a same-named and highly efficient communication protocol for unmanned vehicles. In PX4, MAVLink is used for communicating with the external entities (e.g., other drones or the ground control system). Further, MAVLink integrates with uORB to provide the possibility to forward messages from the flight stack to external entities and vice versa.

**Attestation Requests.** To enable DIAT to receive attestation requests we extended the MAVLink message format with a flag, which indicates if requested data is considered sensitive and therefore, the process of generating this data is to be attested. DIAT handles an attestation request by including the corresponding attestation report in the response. For example, an attestation request for GPS data is handled by including the GPS data together with the corresponding attestation report in the response.

**Identification of Module Dependencies.** In DIAT, `Module 1` depends on module `Module 2`, if (1) `Module 1` receives a message from `Module 2`, or (2) there is a transitive dependency between these modules (`Module 1` depends on `Module 3` which depends on `Module 2`). DIAT maintains the module-dependency model dynamically. This is enabled by the fact that in DIAT the software modules communicate over a well-defined channel. This requirement enables updating the dependency model even if new software modules are added to the system at run-time.

**Buffered Data.** PX4's messaging system is asynchronous. This means that messages are not sent on request by a receiver but a sender sends messages whenever data is available. Sent messages are buffered in queues and delivered to receivers at a later point in time. This behavior leads to a problem in attesting data when buffered data are used which have been generated before the attestation process is started. This would mean that the control-flow of involved modules has not been monitored and therefore the correct run-time behavior of those modules is not verifiable. To avoid this problem, we have added the functionality to flush existing values from the uORB. At the beginning of an attestation process, this functionality is used to discard the buffered values. Therefore, the required data have to be (re)generated when the attestation is active and the control-flow of involved modules is recorded. This enables the verification of their run-time behavior.

*2)* CFMonitor*:* CFMonitor is a trusted component which records the control-flow within software modules. The common approach, which enables the control-flow recording is *code instrumentation*. In this work, instrumentation means modifying a software module to enforce the recording of its execution path, i.e., the module is extended such that it uses the functionality of CFMonitor to record its own execution path.

**Instrumentation.** In DIAT, instrumentation enables the recording of the execution path by tracing *integrity-relevant control-flow events* which are executed at run-time. An integrity-relevant control-flow event refers to an instruction which transfers the flow of the program's execution from the current address to an address which is determined at run-time. The reason for this distinction is that only these events can be exploited by run-time attacks. This approach enables us to precisely and efficiently represent the behavior of a software module at run-time. Our instrumentation is performed in the build process of PX4 where the following modifications are applied to the code in assembly form:

- A unique ID is assigned to each software module. This ID is used to identify software modules at run-time.
- A dispatch instruction is inserted before each integrity-relevant control-flow event. This instruction redirects the program flow to the CFMonitor. Further, the dispatch instruction passes the following arguments to the CFMonitor: (1) the ID of the software module; (2) the source address of the control-flow event; and (3) the destination address of the control-flow event.

**Recording execution path.** The main part of CFMonitor is a logic which records control-flow events and adds them to the execution path of the corresponding module. The execution path of every critical module is stored in a separate data structure. Since parallel execution of processes is a common feature in embedded systems, it is required that CFMonitor records the control-flow events atomically to avoid race conditions and prevent the untrusted OS from interfering with the process of recording the control-flow path. To achieve this, we disable the interrupts for a very short time period upon entering this procedure and enable them before the control-flow is transferred back to the software module. To ensure the compliance with the real-time requirements special care is taken so that the code which is executed as the interrupts are disabled (1) is minimal and (2) has a bounded execution time (i.e., the time required for storing a control-flow event in an internal buffer).

Figure 6 depicts how CFMonitor works. CFMonitor includes three data structures: (1) *Critical modules*: the list of software modules involved in the current attestation; (2) *MSHV table*: the table which stores the MSH-values of the critical modules; and (3) *Path table* which contains multisets of control-flow events. Each multiset represents the execution path of a critical module. For instance, the execution path of `Module 1` is represented by a multiset including four elements. This multiset shows that the control-flow event *e1* is executed once in the `Module 1` (indicated by *<e1, 1>*). As described in Section V-B1, the critical modules are determined by DFMonitor. As these modules run, the dispatch instructions within each module forward the module ID and for each control-flow event the source and the destination address to CFMonitor. If the module ID belongs to a critical module, CFMonitor uses the received information to update

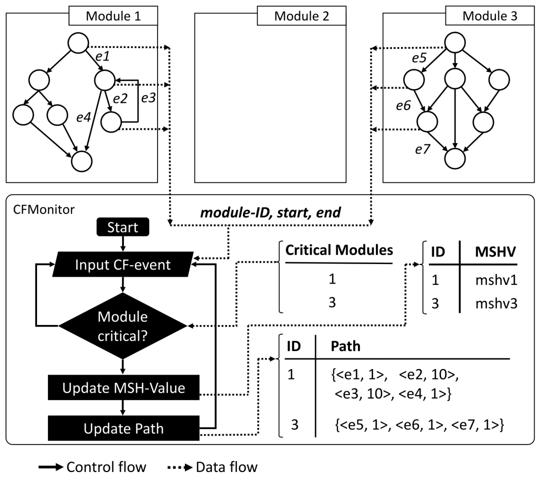

Figure 6: CFMonitor Logic

the MSH-value and the execution path of the corresponding module. After the system has generated the requested data, the multiset hashes and the execution paths are used to generate the corresponding attestation report.

Our representation of execution path allows us to: (1) reduce the size of attestation report as we only count control-flow events rather than storing the whole execution path. In particular, for each control-flow event, we only need to specify how often it is executed instead of including it in the path as many times as it is executed. and (2) significantly reduce the amount of required secure memory, as we only need to store the MSH-value in the secure memory.

**Integration into PX4 Autopilot Software.** Figure 7 shows the concept of DIAT and how it is integrated into the PX4 autopilot software. Software modules communicate with each other using the functionality offered by uORB. CFMonitor records the information (*CF Data*) about control-flow events executed in critical software modules, which are identified by DFMonitor. To allow this integration, the following to components are required:

- Filter is the part of DFMonitor that authorizes the incoming messages/requests. This functionality is required, since MAVLink allows sending unauthorized control messages to a device that modify the device's internal state (e.g. the current GPS coordinates). In our system, Filter rejects messages of this kind.
- Quoter is a component with exclusive access to the device's secret key and is responsible for authenticating the generated attestation reports.

## VI. Performance Evaluation

We now present our evaluation results for DIAT. First we provide microbenchmarks for the Multiset Hash (MSH) function. Afterwards, we evaluate attestation and verification costs for individual software modules. Finally, we provide simulation results demonstrating DIAT's scalability for large networks. Our evaluation is based on a popular flight controller

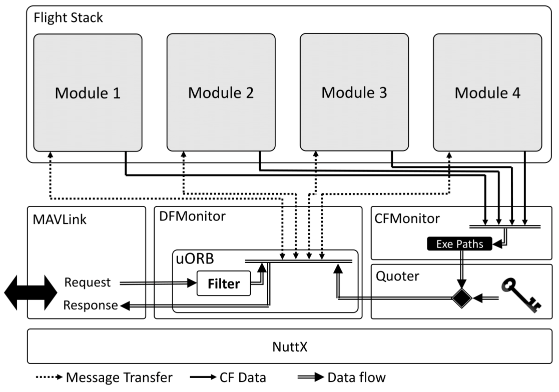

Figure 7: DIAT Implementation

for drones – Pixhawk.[7] Pixhawk is an open source hardware project providing a Cortex M4F CPU (up to 168 MHz), 256 KB of RAM, and 2 MB of Flash Memory. We use the Pixhawk hardware board in conjunction with PX4 software stack on top of the NuttX Real-time OS (RTOS). We discuss the performance impact of lightweight hardware security architectures in Section VIII.

### A. *Multiset Hash-Function*

Before evaluating the performance of DIAT, we first show the evaluation results of our implementation of the additive MSH in comparison to the *Blake2* hash function, which is used in existing control-flow attestation [2].[8] The goal is to understand the impact of using MSH on the overall performance of DIAT. Table I shows the execution times for the individual phases of both hash functions.

Table I: Performance of Hash functions

| Function | MSH ($\mu s$) | Blake2 Hash ($\mu s$) |
|---|---|---|
| Initialize | 46 | 8 |
| Update | 37 | 1 |
| Finalize | 2 | 22 |

When considering the run-time of the hash function independent of the overall attestation scheme, it is easy to see that a conventional hash function outperforms MSH. However, the use of MSH provides DIAT with several advantages that makes it outperform existing control-flow attestation protocols [2], [14].

When a conventional hash is used, the execution path has to be hashed either (1) in parallel to the computation or (2) when the computation is finished. In the first case, which is the approach used by existing control-flow attestation [2], the verification overhead is extremely high. In this scenario, the verifier is required to have access to a database, which contains a hash value for every valid execution path of the prover and the verification process involves a search over this database. This is not feasible for embedded systems.

[7] https://pixhawk.org
[8] https://blake2.net

One variation of this approach is to reduce the verification overhead by sending the authenticated hash value together with the entire execution path to the verifier. In this approach, the verification process would involve the recalculation of the hash value and comparing it with the reference value received from the prover. However, this approach causes an unacceptable communication overhead. For example, the execution path for the critical modules for GPS data verification includes 22249 edges, as we will elaborate in the subsequent section. If each edge is represented by two addresses (source and destination), a total amount of ca. 178 KB needs to be transferred, i.e., total size = path length $(22249) \cdot 8$ Bytes (4 Bytes for source address and 4 Bytes for destination address). Using MSH this execution path can be represented as a multiset of edges that includes the number of appearance of each edge in the path. Since the execution path includes 173 different edges, this mutliset can be represented by 2.8 KB, i.e., total size = number of edges $(173) \cdot 16$ Bytes (8 Bytes for the edge + 8 Bytes for the count). This represents an improvement of ca. 98% in communication overhead.

Calculating the hash after the computation makes it possible to compress the execution path. However, in this approach (1) a large amount of secure memory is required to store the complete path, and (2) the calculation of hashes cannot be performed in parallel (e.g. using a dedicated coprocessor) [14].

### B. *Modular Attestation*

*1) Attestation and verification:* In this section, we show the performance of DIAT in terms of the overhead for generating and verifying attestation reports for a specific sensitive data, i.e., GPS coordinates. The results of our evaluations are shown in Table II.

Table II shows the 13 software modules that are typically running on the Pixhawk in the time frame that is needed to generate new GPS coordinates on a drone. For each of these modules, the table also shows (1) the CFG size, (2) the length of the execution path that was actually taken, (3) the required time for generating the attestation report, and (4) the verification time of the attestation report. Out of these 13 modules only the *GPS* module is involved in generating GPS coordinates and is therefore the only module that needs to be included in the attestation report for GPS coordinates. Hence, the time required for attesting and verifying GPS coordinates with DIAT is equal to the attestation and verification time of the GPS module shown in the first row of Table II. Using existing attestation schemes [2], [14], attesting GPS coordinates requires generating and verifying an attestation report for all 13 executed modules. Additionally, the NuttX RTOS must also be attested in existing schemes, which would add significant overhead. A simple comparison of attestation overhead with and without DIAT's modular attestation (i.e., only *GPS* vs. all 13 components) shows that DIAT's modular attestation entails an improvement of at least 95%. A numerical comparison between DIAT and existing schemes, like C-FLAT [2] and LO-FAT [14], is not possible due to the lack of verifier performance evaluation provided by these works. These schemes consider the problem of verification to be out of scope and assume a very powerful verifier that has comprehensive information about all possible execution paths. Their verification overhead is exponential in the size of the CFG while the verification overhead in DIAT is linear (see the last column in Table II).

Table II: Performance of DIAT on Pixhawk

| Module | CFG | Exe Path | Attestation (ms) | Verification(ms) |
|---|---|---|---|---|
| *GPS* | 2922 | 22249 | 835 | 849 |
| *Gyroscope* | 912 | 20004 | 748 | 760 |
| *e-Compass* | 1468 | 18907 | 716 | 718 |
| *IMU Sensor* | 1905 | 158671 | 6341 | 6357 |
| *Pressure Sensor* | 1051 | 1150 | 46 | 46 |
| *FMU* | 1828 | 38132 | 1510 | 1511 |
| *PX4IO* | 3661 | 12723 | 484 | 489 |
| *LED Driver* | 532 | 32 | 1 | 1 |
| *STM32 ADC* | 251 | 21274 | 805 | 808 |
| *Commander* | 7852 | 9418 | 354 | 365 |
| *Load Monitor* | 135 | 8 | 0.3 | 0.4 |
| *Sensors* | 2032 | 40410 | 1618 | 1623 |
| *Systemlib* | 2555 | 662142 | 26341 | 26365 |
| **Total** | 27014 | 1005120 | 39799, 3 | 39892, 4 |

*2) Number of Attested Modules:* GPS data represents an extreme case where only a single module generates and processes the data. In order to investigate the effect of modular attestation on the overall performance gain, we evaluated DIAT with data that is generated and processed by multiple modules. Table III shows the savings of DIAT for different data. It compares the modules that were in general executing on the device while the data was generated and processed (Executed Modules) against the subset of modules that did actually process the data (Critical Modules). It lists for both sets the (1) number of these modules (Count), (2) total CFG size (sum of CFGs of all modules – $\Sigma$ CFGs), and (3) total length of execution paths ($\Sigma$ Executed Paths).

The results show that for generating, for instance, *sensor_gyro* data eight modules are executed, two of which are critical (25%). Further, the total length of execution paths of critical modules is 2817 which represents 20% of the total length of execution paths in all executed modules (13873). The percentages shown in this table reflect the positive impact of modular attestation.

### C. Network Simulations

We used network simulation to further investigate the impact of DIAT on collaborations involving a large number of devices. We simulated DIAT using the OMNeT++ [31] network simulator. DIAT was implemented at the application layer and the simulation parameters (e.g. different delays) were set based on real measurements we made on the Pixhawk board. The communications rate for links between devices was set to 250 Kbps, which is the actual data rate provided by NodeMCU[9] – an ultra-low power WiFi module commonly used for communication between drones.

[9] http://nodemcu.com/index_en.html

We simulated three possible collaboration scenarios: (1) Serial collaboration, where devices sequentially execute their roles in a collaboration; (2) Parallel collaboration, where all devices execute their roles in parallel; and (3) Hybrid collaboration, where devices collaborate both serially and in parallel. Serial and parallel collaboration represent two extreme cases, while hybrid collaboration falls in the middle. We varied the number of devices involved in the collaboration from ten to $10,000$ devices.

The simulated collaboration involved multiple devices generating and exchanging GPS positions. For each collaboration scenario (serial, parallel and hybrid) we simulated three cases: the first case provides the baseline for our evaluation with no security measures deployed; the second case deploys the basic integrity protection for data during transmission, i.e., authenticating messages based on ECDSA; and in the third case DIAT was used to protect the integrity of data by attesting their generation and processing.

We make the following observations regarding the run-time of different collaboration scenarios:

**Serial Collaboration.** The run-time is *linear* in the number of devices involved in all three collaboration scenarios. This is caused by the sequential exchange and processing of data (and consequently sequential data exchange, authentication and attestation).

**Parallel Collaboration.** The run-time is *constant* in the number of devices involved when no security is deployed, since all communication and processing is done in parallel. The run-time is *linear* in the number of participating devices in the two other cases where messages are authenticated, and data integrity is provided by DIAT, respectively.

**Hybrid Collaboration.** The run-time is *logarithmic* in the number of devices involved in all three cases. This is due to the distribution of the communication, processing, attestation, and verification burden across the network.

**Summary.** DIAT incurs around 400% overhead over regular authentication in both sequential and hybrid collaboration scenarios. However, DIAT does provide much stronger security guarantees compared to simple authentication of data.

In comparison to existing control-flow attestation schemes [2], [14] DIAT has higher run-time overhead on the prover side. This is due to the use of the more expensive hash function (i.e., MSH [12]). However, due to our modular decomposition, DIAT is applicable to autonomous devices with complex software (e.g., drones) without impairing the functionality of the devices or incurring noticeable delays. Also, the performance on the prover side can be significantly improved using hardware acceleration engines, as demonstrated before [14].

On the other hand, the use of MSH allows efficient verification of control-flow paths, which is paramount for collaborative autonomous systems where embedded devices have to act as both prover and verifier. This was not possible using previously proposed control-flow attestation schemes [2], [14].

Table III: Critical modules vs. executed modules

| Data | | *cmd_state* | *battery_status* | *sensor_accel* | *sensor_gyro* |
|---|---|---|---|---|---|
| **Count** | **Critical Modules** | 12 | 12 | 2 | 2 |
| | **Executed Modules** | 12 | 13 | 7 | 8 |
| | **Percentage** | 100% | 92% | 28% | 25% |
| **$\Sigma$ CFGs** | **Critical Modules** | 197823 | 46860778 | 194 | 250 |
| | **Executed Modules** | 197823 | 46862156 | 1590 | 1328 |
| | **Percentage** | 100% | 99% | 12% | 18% |
| **$\Sigma$ Executed Paths** | **Critical Modules** | 26572 | 26572 | 3373 | 2817 |
| | **Executed Modules** | 26572 | 27104 | 13622 | 13873 |
| | **Percentage** | 100% | 98% | 24% | 20% |

### *D. Drones Demonstrator*

To demonstrate the feasibility of control-flow attestation for collaborating autonomous networks we implemented a demonstration showing multiple drones flying in formation. The coordination between drones is based on the exchange of GPS data. Each drone attests the control-flow of the code executed on other drones. The demo shows the applicability of our control-flow attestation to systems with strict real-time requirements. Our autonomous drones systems is flying and collaborating without any problems or noticeable delays. The video of the demonstration is available online.[10]

## VII. Security Consideration

DIAT ensures the integrity of data exchanged in a collaborative autonomous system. In this section, we will show that DIAT fulfills all security requirements identified in Section II-B. An adversary aiming to violate DIAT's security guarantees can manipulate sensitive data in three different phases: (1) while the data is generated or initially sensed by the platform's hardware, (2) while the data is processed on the platform, and (3) when the data is transferred to the verifier.

### *A. Off-Device Security*

Spurious sensor data constitute an orthogonal problem, DIAT relies on the correctness of the initial data. While the data is transferred to the verifier the data is protected by cryptographic means, i.e., data is digitally signed by a key only accessible to the prover's trusted computing based (TCB). Hence, the adversary cannot manipulate the data without being detected as he cannot generate a valid signature for altered data. This means that DIAT fulfills the security requirement for *data integrity and authentication during transportation*, as identified in Section II-B.

### *B. On-Device Security*

To ensure data integrity on the platform itself DIAT has to counter a number of attack vectors and strategies. The attacker can target (1) DFMonitor or CFMonitor, (2) a module that is *not* processing sensitive data, and (3) a module that is processing sensitive data.[11] (4) The attacker can aim to exploit dynamic data dependencies and try to inject "untrusted" data into the processing of sensitive data.

*1) Control-flow and data-flow monitor:* constitute the trusted computing base (TCB) of DIAT. Our TCB is assumed to be immune to attacks, as described in Section II-A. DIAT's platform security architecture (Section VIII) ensures the isolation of CFMonitor and DFMonitor as well as their initial integrity by means of secure boot.

*2) Non-critical modules:* are by definition not relevant for the integrity of sensitive data, therefore any compromise of a non-critical module does not give the attacker any advantage towards the goal of compromising the integrity of sensitive data.

*3) Critical modules:* can be subject to a number of attacks:

**Code integrity.** The attacker can aim at manipulating the code of a module either before the module is loaded or at runtime. Manipulation of a module before load-time is detected by the static attestation that is performed for every module when it is loaded. At runtime, the module's code is protected by the isolation provided by the security architecture. Therefore, the attack can only manipulate the code from *within* the module, i.e., by techniques like code injection. However, these attacks are prevented by data execution prevention (DEP), which eliminates the possibility to insert new code as well as the option to alter or overwrite existing code. Hence, the code integrity of DIAT's modules is ensured and the requirement for *code integrity on device* is fulfilled.

**Module data integrity.** The integrity of a module's data is crucial for the correct operation of the module, i.e., that the module is processing sensitive input data correctly.

Different types of run-time attacks modify different data and have different effects on the control-flow as described in Section II-A. Control-data attacks, like Return-Oriented Programming (ROP) [34], directly influence the control flow of module's code and introduce new control-flow edges in the executed control-flow path. DIAT's CFMonitor captures *all* control-flow transitions executed in a module while processing sensitive data. This information is provided to the verifier who can easily check whether all executed transitions – independent of the order in which they were executed – are legitimate, by checking if they are contained in the module's Control-Flow Graph (CFG).

Non-control data attacks do not introduce control-flow edges outside of the CFG in a module's execution path. The class of

[10] https://youtu.be/tI7dkWdQ3jA

[11] For sake of lucidity we concentrate on an attacker targeting a single module, extending the security arguments to multiple modules is straight forward.

non-control data attacks can be divided into two sub-classes, (1) attacks that do influence a module's execution path [11], [20], and (2) attacks that leave the execution path completely unchanged.

Existing DOP attacks, which fall into the first sub-class, are often target-specific and aim to achieve the execution of program functionality that is restricted, i.e., should not be available in the given execution context [11]. DIAT can detect these types of attacks, given sufficient context information on the verifier side. In particular, DIAT's Multiset Hash (MSH) representation does reveal to the verifier whether a restricted control-flow path was executed in a module. More general are DOP attacks that provide the attacker full control over a target program's operation, i.e., DOP attacks achieving Turing-completeness [20]. These attacks combine multiple instruction sequences, called data-oriented gadgets, which are chained together using a dispatcher gadget. The dispatcher gadget is a loop (or loop-like construction) that has to be iterated through repetitively. This unexpected count of iterations can be easily detected by the verifier in DIAT's control-flow report, despite the loss of order information due to its constant size MSH representation.

DOP attacks from the second sub-class neither execute an unexpected path in a module's CFG nor execute transitions (e.g., of loops) atypically often. These attacks are currently not detectable with DIAT. For instance, attacks that modify the values of variables that are used in calculations. In general, non-control data attacks are subject to active research and no generic detection policy for these attacks existed at the time of writing. However, DIAT can be adapted and extended with new detection policies on the verifier side if they are developed in the future.

*4) Data dependencies:* The integrity of sensitive data is dependent on the operation applied to it, i.e., the correct operation of the modules processing it, as well as other data involved in its processing. An attacker could aim at manipulating data that is eventually integrated into sensitive data. In general, DIAT uses dynamic control-flow tracing to determine which modules provide input to the processing of sensitive data and considers those input data sensitive as well.

In particular, if some module processes sensitive data and receives or requests input data, its input is considered sensitive. Therefore, `Module 1` providing this input data has to be monitored. If `Module 1` is producing the input data "on-demand" its monitoring can be activated before the sensitive data is produced, i.e., the integrity of the input data can be verified. However, for data that has been produced at an earlier point in time no information is available about the correct creation of the data. DIAT addresses this problem by ensuring that all inputs for generating a sensitive data are generated on-demand, i.e., DIAT provides transitive verifiable data generation and processing.

In our prototype implementation (Section V) modules that produce data on-demand would publish their results for use by other modules. In case that a critical module `Module 2` requires such data, the buffer of already published results is flushed and the monitoring of `Module 1` is started while it is generating new results. `Module 2` can then use the new results which were generated by `Module 1` while being monitored, i.e., the correctness of `Module 1`'s results is verifiable.

DIAT's transitive verifiability of data integrity fulfills the requirement for *data integrity on device* as identified in Section II-B.

## VIII. Discussion

In order to fulfill the requirements listed under Section II-B and achieve its security goals, DIAT places the following four requirements on the underlying security architecture: (1) secure storage, (2) support for secure boot, (3) isolation of software modules, and (4) secure Inter-Process Communication (IPC). In this section, we introduce two security architectures, one from academia and one from industry, and show that the requirements of DIAT can be fulfilled by either of these architecture.

### A. TyTAN

Tiny Trust Anchor for Tiny Devices (TyTAN)[7] is a hardware/software co-design which enables strong hardware-based memory isolation for embedded systems. TyTAN has low complexity and is highly flexible. It is designed to be deployed in low-end embedded systems with real-time requirements. In order to isolate the software components from each other, TyTAN deploys an execution-aware memory protection unit (EA-MPU) [24]. EA-MPU validates memory accesses based on (1) the memory address being accessed and (2) the address of the instruction currently being executed. In TyTAN, the access control rules can be updated dynamically. This feature enables the security critical software modules to be loaded and unloaded at run-time. Further, TyTAN enables the interruption of (isolated) software modules without information leakage. In particular, to securely process an interrupt, the module's state is stored in a protected region and the CPU registers are cleared before the execution is passed to the untrusted interrupt handler. TyTAN has a minimal Trusted Computing Base (TCB). In particular, the operating system is not included in the TCB. Finally, the integrity of TyTAN's TCB is ensured through secure boot.

TyTAN fulfills DIAT's requirements on the security architecture as follows:

- *Secure storage:* TyTAN enables the derivation of storage encryption keys from the protected platform key, aka sealing.
- *Secure boot:* TyTAN provides secure boot as part of its architecture.
- *Software module isolation:* Providing isolation of software modules is the main goal of TyTAN and is implemented based on the EA-MPU.
- *Secure IPC:* TyTAN provides a mechanism for secure IPC.

**TyTAN performance.** The overhead caused by TyTAN on the system performance is minimal, as reported by the authors [7].

TyTAN was evaluated with regard to different system tasks, in particular, creating a secure task, measuring a task, and saving and restoring the context of a task.[12] TyTAN incurs overhead when creating a new task, however, when the system is running the only overhead of TyTAN stems from the secure context switch between isolated modules. The evaluation shows that the overhead for secure context switches is minimal, it is basically one additional jump instruction plus wiping out CPU registers.

### B. *TrustZone-M*

The term TrustZone-M is widely used to refer to the *Armv8-M Security Extension*. Armv8-M specifies the 8-th generation of ARM processors designed for deeply embedded systems – postfix *M* for *Microcontroller*. The security extension presents a set of optional instructions with the purpose of providing a strong hardware based isolation mechanism. In a system with TrustZone-M, the processor is either in *secure state* or *non-secure state*. Further, system resources, e.g. memory areas, are assigned to one of these security states. If a resource is assigned to the secure state, it can only be accessed if the processor is in the secure state. In each security state, there are two execution modes: (1) thread mode and (2) handler mode. While user applications are executed in thread mode, tasks such as exception handling and resource management are performed in handler mode. A system with TrustZone feature starts up in the secure state. The secure state software performs optional security checks, e.g., integrity check of the non-secure state software, before the execution is transferred to the non-secure state. Changing the processor's security state takes place according to strict rules, which – together with techniques such as register/exception banking – prevent information leakage from the secure state to the non-secure state.

TrustZone-M fulfills DIAT's requirements on the security architecture as follows:

- *Secure storage:* Using the platform key, which is exclusively accessible by trusted secure state software, storage encryption keys for each module can be derived enabling secure storage.
- *Secure boot:* TrustZone-M starts in the secure state and provides secure boot for the secure state software.
- *Software module isolation:* Software modules can be isolated by updating the memory access control policy (stored in and enforced by the memory protection unit MPU) on every context switch leveraging a small trusted management software, which it protected in the secure state.
- *Secure IPC:* Shared secure memory between software modules and DFMonitor can be used for IPC. Similar to module isolation, shared memory between modules can be managed by a small software component, which is protected in the secure state.

[12]For details of the evaluation please refer to the original paper [7].

## IX. Related Work

**Static Device Attestation.** Attestation schemes fall into three main categories: (1) Software-based attestation [33], [18], [26] which – under some strict assumptions – enables attestation for legacy and low-end embedded devices, as it requires no secure hardware and does not rely on cryptographic secrets; (2) Coprocessor based attestation schemes [25], [27], which are based on complex and/or expensive security hardware (e.g., TPM); low-end embedded devices; and (3) Hybrid schemes [16], [24], which aim at minimizing the hardware security features required for enabling secure remote attestation. Such security features can be as simple as a read only memory (ROM), and a simple memory protection unit (MPU). DIAT can be built on top of the same minimal hardware support providing security against run-time attacks and applicability to emerging autonomous systems.

Noorman et. al. describe a system architecture and programming model that enforces data-flow between protected modules on distributed embedded devices [30]. Their approach used attestation to provision the modules with secret data which is used to later authenticate the interaction of the modules. Unlike our work they do not consider run-time attacks on the software modules. Also, their approach statically defines legitimate data-flows at compile-time, while DIAT dynamically tracks data-flows.

**Collective Attestation.** Collective attestation [5], [6], [21], [22] enables scalable static attestation of large groups of interconnected devices. It was first proposed by SEDA [6]. SEDA extends the software-only attacker assumed by most *single-prover* attestation schemes to, so-called device swarms. It exploits minimal security hardware to enable neighbors' verification and secure hop-by-hop aggregation, thus, achieving scalability through the distribution of the attestation burden across the whole network. SANA [5] extends SEDA with a novel aggregate signature scheme, which enables low verification overhead while requiring minimal trust anchor only for the attested devices. Finally, DARPA [21] builds on top of SEDA to enable security under a stronger adversary model based on absence detection and periodic heartbeats. Unlike existing techniques, DIAT allows efficient *control-flow* attestation (see below) in autonomous collaborative systems. It enables every collaborating devices to verify the integrity of all exchanged data.

**Control-flow Attestation.** Several schemes for control-flow attestation have been recently proposed [2], [14], [36], [15]: C-FLAT [2] enables a prover to attest the exact control-flow path of an executed program to a remote verifier. However, C-FLAT is not scalable and poses high verification overhead on the verifier. Hence, it cannot be simply combined with existing collective attestation techniques. DIAT is an efficiently verifiable control-flow attestation protocol which enables collective attestation of autonomous systems. To improve the performance of C-FLAT on the prover, LO-FAT [14] was developed. LO-FAT leverages hardware assistance to track control-flow events and performs hash calculations parallel to program execution.

Moreover, LO-FAT supports control-flow attestation of legacy code since binary instrumentation is not required. However, LO-FAT also induces a high overhead on the verifier and is not applicable to autonomous systems. DIAT can leverage hardware assistance described by LO-FAT to further reduce the overhead on the prover in an autonomous system. Finally, control-flow attestation is capable of detecting run-time attacks that are not detectable by control-flow integrity (CFI). It also allows safe reactions to attacks, which is particularly important in autonomous systems.

**Data Integrity.** Data-Flow Integrity (DFI) [8] aims at preventing run-time attacks by preserving the integrity of the data of a vulnerable program, e.g., buffers and strings. DFI is based on generating a Data-flow Graph (DFG) based on static analysis and ensuring that data flow of the program complies with its DFG at run-time. Much of research has been conducted aiming to ensure the integrity of a program's data at run-time by enforcing memory safety for programming languages [23], [29], exploiting dynamic tainting [13], [10] or applying bound checking [32], [35]. The common goal of existing approaches is to detect run-time attacks such as control-flow attacks and data-only attacks. Unfortunately, these solutions have large overhead, require porting entire programs to different languages, incur a large false positives rate, or are not capable of detecting all kinds of run-time attacks. More importantly, such enforcement techniques are not applicable to safety-critical real-time systems. DIAT preserves the integrity of data generated by a program/device by ensuring the detection of a large number of run-time attacks based on means of control-flow attestation. It has a low overhead and it follows the attestation paradigm which allows safe reaction to attacks' detection. To summarize, the purpose of DIAT is not to enforce DFI, but rather to ensure integrity of input data, which seems more realistic in practice.

## X. Conclusion

In this work, we present DIAT – a control-flow attestation scheme for autonomous collaborative systems. DIAT combines three novel building blocks: data integrity attestation, modular attestation, and novel representation of execution paths. We assemble these building blocks to enable efficient run-time attestation in a setting where embedded systems must act as both, prover and verifier. We demonstrated DIAT's applicability to real embedded systems with strict real-time constraints by implementing and evaluating it on a state-of-the-art flight controller for drones. In future work we aim at further improving the performance of DIAT in order to allow its application to more restricted real-time settings. We are also working on exploring advanced verification policies that will allow verifier devices to detect sophisticated data-only attacks.

## Acknowledgements

This research was co-funded by the German Science Foundation, as part of project S2 within CRC 1119 CROSSING, HWSec, and Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

## References

[1] M. Abadi et al., "Control-flow integrity: Principles, implementations, and applications," *ACM TISSEC '09*, 2009.

[2] T. Abera et al., "C-flat: Control-flow attestation for embedded systems software," in *CCS '16*, 2016.

[3] T. N. Al-Otaiby et al., "Toward software requirements modularization using hierarchical clustering techniques," in *ACM-SE '05*, 2005.

[4] E. Almentero et al., "Towards software modularization from requirements," in *SAC '14*, 2014.

[5] M. Ambrosin et al., "SANA: Secure and Scalable Aggregate Network Attestation," in *CCS*, 2016.

[6] N. Asokan et al., "Seda: Scalable embedded device attestation," in *ACM CCS*, 2015.

[7] F. Brasser et al., "Tytan: Tiny trust anchor for tiny devices," in *DAC*, 2015.

[8] M. Castro et al., "Securing software by enforcing data-flow integrity," in *OSDI '06*, 2006.

[9] S. Checkoway et al., "Return-oriented Programming Without Returns," in *CCS '10*, 2010.

[10] S. Chen et al., "Defeating memory corruption attacks via pointer taintedness detection," in *DSN '05*, 2005.

[11] ——, "Non-control-data attacks are realistic threats," in *USENIX '05*, 2005.

[12] D. Clarke et al., *Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking*, 2003.

[13] M. Costa et al., "Can we contain internet worms," in *HOTNETS '04*, 2004.

[14] G. Dessouky et al., "Lo-fat: Low-overhead control flow attestation in hardware," in *DAC'17*, 2017.

[15] ——, "Litehax: Lightweight hardware-assisted attestation of program execution," in *ICCAD '18*, 2018.

[16] K. Eldefrawy et al., "SMART: Secure and minimal architecture for (establishing a dynamic) root of trust," in *NDSS*, 2012.

[17] A. Francillon et al., "Code injection attacks on harvard-architecture devices," in *CCS '08*, 2008.

[18] R. Gardner et al., "Detecting code alteration by creating a temporary memory bottleneck," *IEEE TIFS*, 2009.

[19] W. G. Griswold et al., "Modular software design with crosscutting interfaces," *IEEE Software*, 2006.

[20] H. Hu et al., "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE S&P '16*, 2016.

[21] A. Ibrahim et al., "DARPA: Device Attestation Resilient against Physical Attacks," in *WiSec*, 2016.

[22] ——, "Us-aid: Unattended scalable attestation of iot devices," in *SRDS '18*, 2018.

[23] R. W. M. Jones et al., "Backwards-compatible bounds checking for arrays and pointers in c programs," in *HP Labs Tech Report*, 1997.

[24] P. Koeberl et al., "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.

[25] X. Kovah et al., "New results for timing-based attestation," in *IEEE S&P '12*, 2012.

[26] Y. Li et al., "VIPER: Verifying the integrity of peripherals' firmware," in *ACM CCS*, 2011.

[27] J. McCune et al., "TrustVisor: Efficient TCB reduction and attestation," in *IEEE S&P '10*, 2010.

[28] L. Meier et al., "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *ICRA '15*, 2015.

[29] G. C. Necula et al., "Ccured: Type-safe retrofitting of legacy software," *TOPLAS '05*, 2005.

[30] J. Noorman et al., "Authentic Execution of Distributed Event-Driven Applications with a Small TCB," in *STM '17*, 2017.

[31] OpenSim Ltd., "OMNeT++ discrete event simulator," http://omnetpp.org/.

[32] O. Ruwase et al., "A practical dynamic buffer overflow detector," in *NDSS '04*, 2004.

[33] A. Seshadri et al., "SAKE: Software attestation for key establishment in sensor networks," in *DCOSS*, 2008.

[34] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07*, 2007.

[35] J. Yao, "Bcc: run-time checking for c programs," in *USENIX STC '83*, 1983.

[36] S. Zeitouni et al., "Atrium: Runtime attestation resilient under memory attacks," in *ICCAD '17*, 2017.