2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

ADLP: Accountable Data Logging Protocol for Publish-Subscribe Communication Systems

Man-Ki Yoon and Zhong Shao
Department of Computer Science
Yale University

***Abstract*—Reasoning about the decision-making process of modern autonomous systems becomes increasingly challenging as their software systems become more inexplicable due to complex data-driven processes. Yet, logs of data production and consumption among the software components can provide useful run-time evidence to analyze and diagnose faulty operations. Particularly when the system is run by a number of software components that were individually developed by different parties (e.g., open source, third-party vendor), it is imperative to find out where the problems originated and thus who should be responsible for the problems. However, software components may act unfaithfully or non-cooperatively to make the run-time evidence refutable or unusable. Hence, this paper presents Accountable Data Logging Protocol (ADLP), a mechanism to build accountability into data distribution among software components that are not necessarily cooperative or faithful in reporting the logs of their data production and consumption. We demonstrate an application of ADLP to a miniaturized self-driving car and show that it can be used in practice with at a moderate performance cost.**

***Keywords*-Accountability; data logging; publish-subscribe communication; data distribution service;**

I. Introduction

One of the greatest stumbling blocks for modern autonomous systems is the lack of ability to reason about their complex decision-making process. Unlike classical control systems that are governed by analytic processes, modern autonomous systems are operated by a plethora of software components that interact with one another, making intelligent decisions based on *data* processed by other components in addition to sensory information from the surrounding environment. Due to the high-complexity of their interactions and the non-determinism of data-driven algorithms, it is often impossible to analytically find out the cause of every fault.

Therefore, the importance of collecting *run-time evidence* becomes more crucial as this can help reconstruct the system's behavior, which leads to improved ability to diagnose faults and thus to enhanced overall safety. For instance, a number of car makers have opted to equip their cars with an 'automotive black box' called Event Data Recorder (EDR) [1] which, however, records only mechanical status (e.g., speed, brake application, etc.). As self-driving cars are increasingly deployed on the road and involved in accidents, the need for a black box mechanism for the software system is pressing. The *software black box* will keep track of the status of software components, their decisions, and most importantly, the record of data communicated among them. Self-driving car manufacturers already collect a massive amount of data from software as well as hardware components for debugging and post-incident analysis. Such records helped reveal the causes of recent incidents [2], [3]. However, the manufacturers often use proprietary devices and formats for recording the data, making it difficult for a third-party investigator (such as the National Transportation Safety Board) to examine and analyze them independently. Furthermore, such records can hardly serve as legally-binding evidence due to the non-transparency in log collection, and more importantly, the lack of *accountability* in data production and consumption. Since the software components are often developed by different vendors, it becomes critical to assign *responsibility* to each individual component for its association with data that they produce, process, and consume, especially when a faulty incident happens.

Hence, we propose *Accountable Data Logging Protocol* (`ADLP`), a mechanism to build the accountability into data communication among software components in data-driven autonomous systems. The key element of `ADLP` is that software components in a publisher-subscriber communication model are enforced to prove that their records of actions (i.e., log of data production and consumption) conform to what they actually performed. A well-constructed log of data flow among software components can help detect the origin of a faulty operation by keeping track of dependencies between data production (output) and consumption (input). `ADLP`'s goal is to make the log *provable*, serving it as *irrefutable evidence* that can resolve a dispute between potentially responsible components.

The main challenge, however, is that components may act freely and thus can be *unfaithful* and *non-cooperative* in the logging process. For instance, a component may falsify or even hide the logs that indicate its association with particular data production/consumption to avoid any potential liability. The problem becomes more difficult when components communicate in a decentralized (thus non-observable) way.

The challenges described above make the data logging problem unique and thus make relevant techniques ineffective. For instance, in the traditional non-repudiation services [4], [5], tamper-evident logging [6], [7], and data provenance problems [8], [9], the participants' goal is to *prove their correctness* to the counterparts or *protect the integrity of their own log records*. That is, every participant is assumed to be faithful. On the other hand, in the problem discussed in this paper, the participants (i.e., data publisher and subscribers) are motivated to act unfaithfully (i.e., forge, hide, or alter

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00116

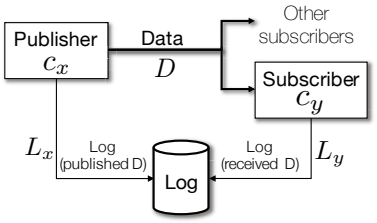


Figure 1. Data publication/subscription and logging model.

log and/or data) in order to make their potentially wrongful operations *unprovable* or to accuse others of any *incorrect* or *inconsistent* logs.

We show that novel applications of primitive cryptographic functions in data transmission and logging processes can enable an accountable data logging. We implemented an instance of ADLP as part of ROS (Robot Operating System) [10] that is transparent to the application layer. Hence, no modification at the application level is required, facilitating the support of legacy codebase. Using an example self-driving application, as well as synthetic data, we demonstrate that ADLP incurs a reasonable amount of overhead that is low enough for a practical use.

II. System Model

We consider a system of N software components, $\mathbf{C} = \{c_i | i = 1, \ldots, N\}$, that communicate with each other via a *publish-subscribe* communication model (e.g., Robot Operating System [10], OpenDDS [11]). Figure 11(b) in Section V shows the data flows among components that comprise an example autonomous navigation application running on a prototype self-driving car (more detail is provided in Section V). Each component c_i takes a set of inputs $\mathbf{I_i} = \{I_{i,p} | p = 1, \ldots, |\mathbf{I_i}|\}$ by subscribing to relevant data (e.g., camera image, LIDAR scan). After processing, each component publishes outputs $\mathbf{O_i} = \{O_{i,q} | q = 1, \ldots, |\mathbf{O_i}|\}$ (e.g., class of traffic sign, steering command). The input or output sets can be empty (e.g., sensors and actuators). Each data is accompanied with a sequence number (either as part of the data or in the form of message header). An end-to-end data flow (e.g., from Camera to Steering) can be formed by a sequence of alternating publication and subscription of data. We use I_i, O_i, or D_i to denote an arbitrary input, output, or any data item of component c_i, respectively, when no ambiguity arises or even drop the subscripts when necessary. We assume that data is eventually delivered unless connection is permanently lost.

There can be no two components who publish the same data type (e.g., no two lane detectors). We use $\texttt{type}(D)$ to denote the type of data D. If any ambiguity arises (e.g., due to redundancy), the types are uniquely labeled. Hence, given a correct type label, the corresponding publisher of the data is uniquely identified.

A data transmission from c_x to c_y for data of type D is denoted by $D_{x \to y}$. Upon receiving data, the subscriber c_y creates a *log entry*, $L_y(D)$, stating the receipt of D and then enters it to a local log file or a remote server, as shown in Figure 1. Similarly, the publisher c_x creates a new log entry, $L_x(D)$, stating the production of D. When no ambiguity arises, we denote the above log entries as L_x and L_y.

The collected logs provide provenance records of data flow among the components, which can be used later for finding out faulty components or erroneous intermediate data. In this paper, we do not consider the problem of verifying data processing algorithms (e.g., correctness of vision algorithm or machine learning model); an incorrect behavior (e.g., due to logic error) would likely be reflected on erroneous outputs which will be logged. A component may opt for maintaining a separate, detailed provenance record of its internal data processing for its own benefit (e.g., debugging trace) using any data provenance technique [12], which is orthogonal to our data logging problem.

A. Trust Model

Unfaithfulness: We consider *non-cooperative* components that cannot be assumed to faithfully enter *every* log entries about their data production/consumption *correctly*. A component may forge or hide log entries in accordance with its interests to, for instance, disturb future forensic analysis. However, we do not need to know which components are faithful or not. The system may contain as many unfaithful components as possible.

An unfaithful component may not necessarily act unfaithfully in relation with every component that it communicates with. For instance, in Figure 2, B may forge logs for $D_{C \to B}$ while it always correctly enters logs for $D_{B \to A}$.

Collusion: A group of components may *collude* among each other if, for example, they are supplied by a same non-compliant vendor. Data transmissions within a colluding group can be arbitrarily forged and even be hidden due to the non-observability of transmission; the simplest scenario is when both publisher and subscriber (e.g., B and C in Figure 2) do not write any log entries even when there was a data transmission between them, in which case there is no way to tell if they were hiding or no actions have actually taken place. We can define a *collusion group* as follows:

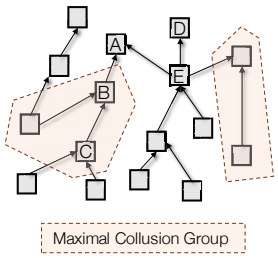


Figure 2. (Non-singleton) Maximal collusion groups.

Definition 1 (Collusion Group)**.** *A* collusion group, $\mathbf{C_{cg}} \subseteq \mathbf{C}$*, is a set of components in which for any* $c_i \in \mathbf{C_{cg}}$*, either (i) there exists* $c_k \in \mathbf{C_{cg}}$ *(where* $i \neq k$*) that colludes with* c_i *or (ii)* $\mathbf{C_{cg}}$ *is a singleton (i.e.,* $|\mathbf{C_{cg}}| = 1$*). A* $\mathbf{C_{cg}}$ *is* maximal*, denoted by* $\mathbf{C_{mcg}}$*, if it cannot be extended by including one more* $c_j \in \mathbf{C} \backslash \mathbf{C_{mcg}}$ *that colludes with some* $c_i \in \mathbf{C_{mcg}}$*.*

Hence, an outside component $c_j \in \mathbf{C} \backslash \mathbf{C_{mcg}}$ for a particular $\mathbf{C_{mcg}}$ (e.g, A in Figure 2) does not collude with any $c_i \in \mathbf{C_{mcg}}$ (e.g., B and C) as $\mathbf{C_{mcg}}$ is maximal. Also, by the first definition, a single component itself forms a colluding group (e.g., D), and it is maximal if it does not collude

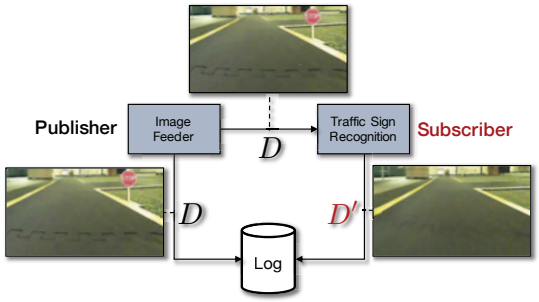


Figure 3. Example scenario when the subscriber received data D while reporting that it received $D' \neq D$.

with any other. It is unknown if collusion groups exist in the system.

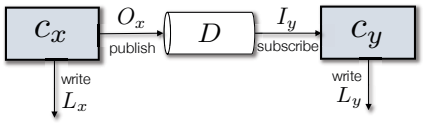
A system is *collusion-free* if every $\mathbf{C_{mcg}}$ in the system is a singleton. That is, every publisher-subscriber pair in the system is a *non-colluding pair*. This paper does not make the collusion-free assumption. In a later section we prove that no unfaithful component can avoid a detection by ADLP when the system is collusion-free.

Trusted Logger: Logs are entered to a trusted logger that is not necessarily part of the underlying data distribution system. It could be a remote log server, a local file, or even a trusted hardware device/execution environment [13], [14], depending on the need for on-line analysis, the level of integrity protection, and the resource availability (CPU/storage). In this paper, we do not address the security of the loggers and we assume that a tamper-resistant or tamper-evident logging mechanism is in place [7], [15] for the protection of log integrity. We also assume that (i) each component is capable of generating a pair of public key and private key [16], (ii) its public key is securely transferred to the logger, and (iii) a standard security mechanism is in place to protect the private key in each component.

III. Accountability in Data Logging

A. Motivational Example

Let us consider a two-components scenario depicted in Figure 3. The image feeder component periodically publishes a camera image which is subscribed by the traffic sign recognition module. Suppose the image feeder captured an image D (with stop sign) from a camera and reports to the logger correctly. However, the subscriber writes to the logger that it received $D' \neq D$. In fact, it may even always log the receipt of D' (that does not include stop sign) no matter what data it actually receives, in case it fails to recognize stop signs correctly and thus wants to avoid potential liability. The component may even not enter any log at all, as if the publisher did not send any data, to simply attribute any faulty consequence to the publisher. Similar situations can happen in the other direction. These unfaithful actions would unlikely stem from a malicious intention to actually disturb the system operation (e.g., faking image to cause an accident). Instead, components could more likely be motivated to act unfaithfully in order to make the logs *unusable*. In such situations, log entries may conflict with one another or even be missing, leaving it difficult to decide whose log is correct and thus which component is responsible for a faulty consequence (e.g., accident due to running over the stop sign).



$$L_x : \Big(id_x, \texttt{type}(O'_x), \texttt{out}, t_x, O'_x\Big) \quad L_y : \Big(id_y, \texttt{type}(I'_y), \texttt{in}, t_y, I'_y\Big)$$

Figure 4. Two components writing log entries about data transmission of type D under the naive logging scheme.

B. Problems with Naive Logging Protocol

We first describe problematic situations in which components' faithfulness are unknown and discuss the challenges that have not been addressed by relevant techniques. Let us first consider a naive logging scheme as follows:

Definition 2 (Naive Logging Protocol)**.** *Component c_i that publishes or receives data D of $\texttt{type}(D)$ at time t_k enters the following log entry:*

$$L_i : \Big(id_i, \texttt{type}(D), \texttt{direction}, t_k, D\Big),$$

where id_i is a unique identifier of c_i and $\texttt{direction} \in \{\texttt{out}, \texttt{in}\}$ for data publication or subscription, respectively.

Now, let us consider data transmission $D_{x \to y}$ as shown in Figure 4. Here O_x, O'_x are the actual data c_x sent out and the one it reported in the corresponding log entry L_x, respectively. Similarly, I_y, I'_y are the actual data c_y received and the one it reported in the corresponding log entry L_y, respectively. Note that $O_x = I_y$ always holds. Then, the following actions can be taken by an *unfaithful* publisher c_x (or similarly by an unfaithful subscriber) for its own interest:

- Hiding: c_x publishes O_x but does not report a log entry about it. That is, $L_x(O_x)$ cannot be found.
- Falsification: c_x publishes O_x but enters a log entry $L_x(O'_x)$ stating that it sent out O'_x where $O'_x \neq O_x$.
- Fabrication: c_x did not publish any data but enters a log entry $L_x(O'_x)$ stating that it has sent out O'_x.
- Impersonation: c_x publishes O_x but enters a log entry $L_z(O'_z)$ as if another component c_z sent out O'_z where O'_z may or may not equal O_x.
- Timing Disruption: c_x sets the timestamp t_x incorrectly so that causal relation among data transmissions become unverifiable. A concrete scenario and analysis are presented in Section IV-B2.

For instance, a falsification by c_x will result in

$$L_x{:}\Big(id_x,\texttt{type}(O'_x),\texttt{out},t_x,O'_x\Big), L_y{:}\Big(id_y,\texttt{type}(O_x),\texttt{in},t_y,O_x\Big).$$

However, it is equally possible that c_y was being unfaithful (as in Figure 3) and thus falsified L_y while c_x indeed published O'_x.

The problematic situations described above arise mainly due to the lack of ability to observe actual data transmis-

sion between the components. In fact, data transmission in publish-subscribe communication system is often implemented by a point-to-point transportation (e.g., TCP/IP for ROS) due to its ability to reliably transfer critical data such as sensor data and actuation commands [17]. Hence, O_x and I_y in Figure 4 are *not observable* by others. Therefore, a component may make up any log entries (even invalid ones) or even stay stealthy in an attempt to make the logs unusable, without ever being detected. Although having a central entity that routes all data could solve most of the problems described above, such a centralized approach is prone to a single-point failure (due to software error or security attack) that would be fatal especially to safety-critical systems. Hence, the challenge is how to enforce components to report complete and correct logs of their data production/consumption in such a decentralized communication environment.

C. Accountability

Consider a transmission $D_{x\to y}$ from publisher c_x to subscriber c_y for data D. The ideal ADLP system requires the following properties to hold:

Definition 3 (Unforgeability)**.** *Neither L_x nor L_y must exist in the log if $D_{x\to y}$ did not happen.*

Definition 4 (Completeness)**.** *Both c_x and c_y must report the publication and consumption of D in L_x and L_y, respectively.*

Definition 5 (Correctness)**.** *Both L_x and L_y must correctly reflect the publication and consumption of D, respectively.*

Classification: Let $\mathbf{L}^*_\mathbf{C}$ be the set of all log entries that the system of components $\mathbf{C}$ are supposed to enter faithfully to the log (i.e., the ideal system). Now, let us consider a non-ideal system, and let $\mathbf{L_C}$ be the set of log entries that are entered by the components and thus observable by ADLP. $\mathbf{L_C}$ consists of sets of valid and invalid log entries ($\mathbf{L_{V,\bullet}}$ and $\mathbf{L_{I,\bullet}}$, respectively), as shown in Figure 5. $\mathbf{L_H} = \mathbf{L^*_C} \backslash \mathbf{L_C}$ is the set of log entries that are *not entered*. $\mathbf{L_H}$ consists of those that are result of collusion, $\mathbf{L_{H,c}}$, and those that are hidden by non-colluding components (i.e., purely unfaithful), $\mathbf{L_{H,u}}$. Note that $\mathbf{L_{H,f}} = \emptyset$ as faithful components do not hide log entries. $\mathbf{L_C} \backslash \mathbf{L^*_C}$ is the set of log entries that are *forged*. Colluding components can arbitrarily forge entries that look valid, $\mathbf{L_{V,c}}$, or (although unlikely) invalid, $\mathbf{L_{I,c}}$. Non-colluding but unfaithful components can also forge entries ($\mathbf{L_{V,u}}$ and $\mathbf{L_{I,u}}$). If the system is ideal, $\mathbf{L^*_C} = \mathbf{L_C} = \mathbf{L_{V,f}}$ hold.

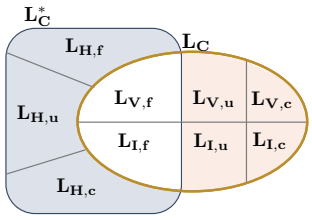


Figure 5. Classification of log entries – **H**: hidden, **V**: valid, **I**: invalid, **f**: faithful, **u**: unfaithful, **c**: collusion.

Goal: The goal of ADLP is to identify the class of each log entry correctly. ADLP classifies each $L_i \in \mathbf{L_C}$ into either $\widehat{\mathbf{L_V}}$ or $\widehat{\mathbf{L_I}}$ (i.e., valid or invalid, respectively) and finds hidden entries, $\widehat{\mathbf{L_H}}$. However, the possibility of collusion may limit ADLP's ability to identify them correctly and completely. Nevertheless, we show that ADLP can prove that the log entries entered by any faithful components are always identified as valid (i.e., $L_i \in \mathbf{L_{V,f}} \Rightarrow L_i \in \widehat{\mathbf{L_V}}$). This means that log entries that are results of data transmission between a non-colluding pair cannot be hidden or misclassified. Hence, if there is a dispute between a non-colluding pair, ADLP can verify whose log entry conforms the reality. Due to this property, in a collusion-free system, ADLP can *unambiguously identify* any unfaithful component and thus correctly classify every log entries. We will prove these in Section IV-B1.

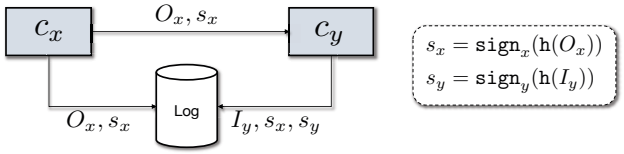


Figure 6. Signed hash of data is used for non-repudiation of log entries.

IV. Accountable Data Logging Protocol

A. Design Principles of ADLP

The main limitation of the naive logging scheme presented in Section III-B is lack of a mechanism to assert an *interdependence* between the log entries. For instance, the publisher's entry does not include any information about the subscriber's 'view' on the received data. Furthermore, the naive protocol cannot force the participants to enter logs. Hence, certain meta-information that can indicate the counterpart's faithfulness in log entry and a method to penalize non-cooperating components are key ingredients of ADLP. In what follows, we present the design principles of ADLP. In Section IV-B, we formally prove the accountability property of the protocol.

A partial, but incomplete, solution to the problems described in Section III-B is the use of cryptographic signature[1] to assert the origin of message (in our case, log entry) [4], [5]. By forcing the software components to include a signed hash of the data in the log entries, ADLP achieves the non-repudiation property of the log entries. ADLP extends this by having publisher send its signed hash, s_x, along with the data it intends to publish, O_x, as illustrated in Figure 6. The subscriber is required to report the signed hash s_x along with the data it received and its own signed hash of the data in its log entry. The key function of the publisher's signature s_x is that it leads the publisher to act more faithfully (i.e., report what is actually published) because the subscriber, whose faithfulness is unknown to c_x, would likely report what it receives from c_x as is. From s_x reported by the subscriber, an auditor can verify the correctness of c_x's log entry. Therefore, under an assumption of non-colluding pair, the publisher cannot (i) pretend to have sent data O'_x when the actual data was O_x (i.e., falsification) and (ii) make log

[1]We use the following notations – $\mathtt{h}(\cdot)$: a cryptographic hash function that is preimage resistant and collision resistant. $\mathtt{sign}_i(\cdot)$: a cryptographic signature with c_i's private key. $\mathtt{verify}_i(\cdot, s)$: a signature verification function of signature s and a given digest, using c_i's public key.

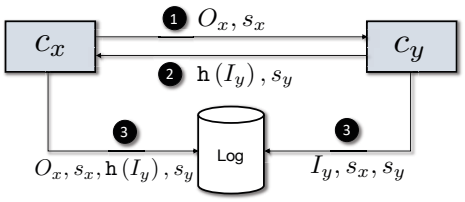
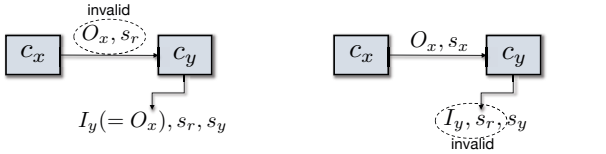


Figure 7. Extended logging policy for incorporation of subscriber's acknowledgement $\mathtt{h}(I_y)$ and s_y.



(a) c_x sends an invalid pair (O_x, s_r) (b) c_y fabricates an arbitrary pair (I_y, s_r)

Figure 8. Publisher c_x can falsely accuse subscriber c_y of fabricating log entry by sending an invalid pair of data O_x and signature s_r. The two cases, (a) and (b), are not differentiable from an auditor's point of view.

entries look as if they were entered by other components (i.e., impersonation).

However, c_y may act unfaithfully by accusing the publisher c_x of sending wrong data $I_y \neq O_x$. This is possible because c_y can freely choose any I_y and s_y (as long as $\mathtt{verify}_y(\mathtt{h}(I_y), s_y) = \mathtt{True}$) for the false accusation. Furthermore, components can be completely non-cooperative by not reporting any log entries (i.e., hiding). Even more worse, a component can still fabricate a log entry as if it has received data when it actually did not (i.e., fabrication). Signing alone cannot prevent these situations. Let us consider Figure 6 again. The subscriber c_y may not write any log entry upon receiving data from c_x as if c_x did not send any data to c_y. In this case, only the following entry would be found:

$$L_x : \Big(id_x, \mathtt{type}(O_x), \mathtt{out}, O_x, s_x\Big).$$

This entry alone cannot prove c_x's publication of O_x, because the log would look exactly same with what would be made when c_x indeed did *not* publish any data but fabricated it.

ADLP's solution to such stealthy, non-cooperative components is to force publisher and subscriber to *exchange* their signed hashes. Hence the policy in Figure 6 is extended to having the subscriber c_y send its signed hash s_y, along with the hashed data $\mathtt{h}(I_y)$, to the publisher c_x which in turn reports them in its log entry L_x as shown in Figure 7. The subscriber's return message plays two important roles. First, similar to the publisher's signed hash, it makes the subscriber less likely make up arbitrary data I_y and signature s_y. More importantly, it serves as an *acknowledgement* that exposes c_y's receipt of the data, preventing c_y from claiming that it has not received any data from c_x when it did. If the subscriber does not commit itself on the receipt of data, the publisher may penalize it by, for example, not sending data.

A further complication is that components may try to reuse the counterpart's signature. For instance, an unfaithful subscriber c_y may reuse O_x and s_x received previously and create a new log entry based on them, claiming that c_x has sent the same data when it has not. ADLP's solution is to incorporate *freshness* information (i.e., sequence number) into signatures, log entries, and messages. Hence, for the seq^{th} publication of data D by publisher c_x, s_x now becomes $s_x = \mathtt{sign}_x(\mathtt{h}(seq||D))$, where $||$ is the concatenation operator, and the sequence number seq is sent along with data D and also included in the log entry. The subscriber also includes the sequence number in $s_y = \mathtt{sign}_y(\mathtt{h}(seq||D))$. In the rest of the paper, unless otherwise necessary, we do not include sequence numbers and timestamps in the descriptions for brevity and assume that they are embedded in message digest (which is the case in ROS message).

For the logging protocol presented above to provide truly provable logs of data flow among components, one key requirement must be satisfied; publisher and subscriber cannot send *invalid signatures*. For illustration, let us consider a case when unfaithful publisher c_x sends out data O_x while reporting to the logger that it sent $O'_x \neq O_x$. As will be discussed in Section IV-B, it is in c_x's best interest to choose an *invalid* pair of O_x and s_x in order to avoid the detection of sending $O_x \neq O'_x$. This is because a valid pair of O_x and s_x can only be given by c_x. Now, suppose c_x has sent an invalid pair of O_x and s_r (e.g., by generating a random signature s_r) as shown in Figure 8(a). Then, the subscriber's log entry will include them, i.e., $I_y = O_x$ and s_r. When an auditor tries to verify its validity, it will fail because s_r is not a valid signature for I_y. But, the auditor cannot accuse c_x of sending the invalid signature because c_x can refute the accusation by claiming that subscriber c_y has arbitrarily fabricated I_y and s_r in the log entry as illustrated in Figure 8(b). Moreover, c_y cannot provide any proof that it did not fabricate them. A similar situation can happen in the opposite direction.

Hence, it is important to enforce that the signatures exchanged between the components to be valid with regard to the data that are sent along with. Among a number of possible approaches, in our ROS-based prototype implementation (presented in Section V), we make the signing process transparent to the logging process. That is, given data to be published, a signature is computed and included in ROS message together with the data at the ROS transport layer.

$\mathtt{h}(I_y)$ vs I_y: As shown in Figure 7, the subscriber's log entry contains the data it received from the publisher. However, this is unnecessary because (i) the log verification is done using the hashed version and (ii) the data is already reported in the publisher's log entry. Hence, subscriber can instead store the hash of the received data, $\mathtt{h}(I_y)$. This could save storage for the logs especially for large-size data (e.g., images). Similarly, the subscriber can return data I_y instead of $\mathtt{h}(I_y)$ to the publisher in the return message, especially when the data is small. In Section VI, we evaluate the storage overhead of ADLP with respect to data size and the use of $\mathtt{h}(I_y)$.

B. Protocol Analysis

In this section, we prove the accountability property of ADLP. For this, let us consider a generalized diagram of data transmission from publisher c_x to subscriber c_y, shown in Figure 9. We use simplified notations for brevity. When

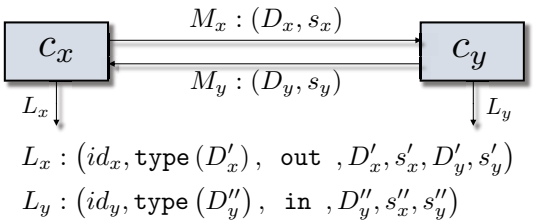


Figure 9. Generalized diagram of data flow and logging protocol under ADLP for the analysis in Section IV-B. Recall that sequence number and timestamp are omitted for brevity.

c_x publishes data D_x, it is sent along with its signed hash s_x. We call it message M_x. It is what the underlying data distribution system delivers to the subscriber c_y (e.g., an ROS message). Then, c_y returns an acknowledgement message M_y that consists of D_y and s_y.

If c_x is faithful, it reports what it sent and received as is:

$$D'_x = D_x \text{ and } s'_x = s_x \text{ and } D'_y = D_y \text{ and } s'_y = s_y \quad (1)$$

hold, where $s_x = \texttt{sign}_x(\texttt{h}(D_x))$. Similarly, if c_y is faithful,

$$D''_y = D_x = D_y \text{ and } s''_x = s_x \text{ and } s''_y = s_y \quad (2)$$

hold, where $s_y = \texttt{sign}_y(\texttt{h}(D_y))$. Recall that D_x, s_x, D_y, and s_y are what the components are supposed to report in the logs, but these are unobservable to the logger because of the point-to-point communication. As explained in Section IV-A, the subscriber may store the hashed data, $\texttt{h}(D''_y)$ in its log instead of the data D''_y itself (e.g., for space saving), and also in the return message M_y. However, for brevity and consistency, we simply use data symbols. Also recall that sequence number and timestamp are omitted except where necessary and that the former is a part of digest for signing.

Obvious Detection: Components would not likely risk obvious detection. For example, since the components' public keys are known, the authenticities of L_x and L_y are easily verifiable. Hence, at a minimum, s'_x (resp. s''_y) must be valid with regard to D'_x (resp. D''_y) in L_x (resp. L_y):

$$\texttt{verify}_x(\texttt{h}(D'_x), s'_x) = \texttt{verify}_y(\texttt{h}(D''_y), s''_y) = \texttt{True}, \quad (3)$$

Therefore, it is of little relevance to consider the cases when these do not hold. This also implies that no component can write a log entry as if it was created by someone else.

Also, we can assume that $\texttt{type}(D_x)$ = $\texttt{type}(D'_x)$ = $\texttt{type}(D''_y)$ = $\texttt{type}(D_y)$ always hold because otherwise it is obviously detectable. In addition, as explained in Section IV-A, the signatures exchanged between the components (i.e., s_x in M_x and s_y in M_y) are valid with regard to data:

$$\texttt{verify}_x(\texttt{h}(D_x), s_x) = \texttt{verify}_y(\texttt{h}(D_y), s_y) = \texttt{True}, \quad (4)$$

1) Proofs of the Accountability Property: We now prove the accountability property (listed in Section III-C) of ADLP. We first prove that components cannot pretend to have sent or received data when there was no actual data transmission.

Lemma 1 (Unforgeability)**.** *Publisher (resp. subscriber) cannot fabricate a log entry for data that it did not send (resp. receive).*

Proof: Suppose publisher c_x tries to fabricate a log entry L_x without publishing any data. This means c_x should fabricate c_y's signature s'_y to include it in L_x (recall Figure 9). c_x may reuse a previously received message M_y which has c_y's valid signature. However, remind that we use *sequence numbers* to attach freshness information to log entries. Hence, for c_x to make up a log entry about the seq^{th} publication, say D'_x, without sending it out, c_x should be able to successfully forge $s'_y = \texttt{sign}_y(\texttt{h}(seq||D'_x))$, which is not possible unless c_y's private key is known to c_x. Then, c_x may claim that (i) it indeed has sent D'_x but (ii) the subscriber c_y returned a wrong pair of D_y and s_y, and (iii) c_y did not write a log entry about D'_x. However, (4) eliminates such a possibility.

Subscriber cannot fabricate a log entry either because of the impossibility of creating a valid signature for publisher (i.e., s''_x in L_y). The subscriber then may try reusing an old data-signature pair M_x that it received previously, but the sequence numbers deter such an attempt. Subscriber c_y may create a random signature instead, as in Figure 8, in an attempt to accuse the publisher of giving an invalid M_x. However, this is impossible due to (4). ■

Therefore, the publisher's log entry L_x alone cannot prove its publication unless it can forge the subscriber's signature s'_y that is valid with regard to the sequence-numbered data. A similar argument applies to the subscriber's case.

The next lemma proves that components cannot hide their data production and consumption.

Lemma 2 (Completeness)**.** *Publisher (resp. subscriber) cannot hide its publication (resp. receipt) of data if subscriber (resp. publisher) is faithful.*

Proof: Suppose publisher c_x sent out data in message M_x. Subscriber c_y will receive the valid pair of data and signature in M_x due to (4) and will write log entry L_y based on them. Hence, c_y's log entry will prove that c_x has published data that agrees L_y. c_x may then claim that c_y replayed an old M_x, which is not possible due to the freshness asserted by the sequence number as explained in the proof of Lemma 1.

Now, in order for c_y to keep receiving data from c_x, the protocol requires the subscriber to return an acknowledgment message M_y to c_x. Hence, even if c_y does not write any log, it should send a return message M_y to c_x. Since M_y cannot be an invalid one due to (4), the pair $D'_y = D_y$ and $s'_y = s_y$ in the publisher's log entry L_x reveals that the subscriber acknowledged the receipt of c_x's data. ■

The following lemma proves that components cannot pretend to have sent/received data that differs from what is actually sent/received.

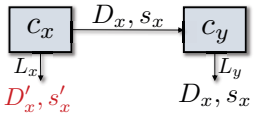
Lemma 3 (Correctness)**.** *(i) If subscriber c_y is faithful, publisher c_x who sent D_x cannot write a log entry L_x such that $D'_x \neq D_x$. Simiarly, (ii) if publisher c_x is faithful, subscriber c_y who received D_x cannot write a log entry L_y such that $D''_y \neq D_x$.*

Proof: (i) Suppose c_x published $D_x \neq D'_x$. Then,

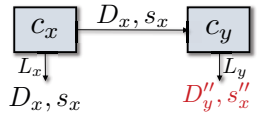
subscriber c_y writes the following log entry:

$$L_y : \Big(id_y, \texttt{type}(D_x), \texttt{in}, D_x, s_x, s_y\Big),$$

because $D''_y = D_x$ and $s''_x = s_x$ by (2). From the data type information, which is unique, the auditor can use c_x's public key to verify (4) from s_x in L_y, and conclude that the data and signature must have been given by c_x. However, the auditor finds D'_x from the publisher's log entry L_x. Since the log entries do not agree with each other (i.e., $D'_x \neq D''_y = D_x$), it must be that L_x has been falsified. Therefore, c_x cannot send D_x that differs from D'_x without being detected.



(i) c_y is correct



(ii) c_x is correct

(ii) The subscriber c_y cannot prove that c_x has sent D''_y, not D_x, because the verification of s''_x for D''_y in L_y will fail (because c_y cannot forge a correct signature for c_x). c_y may refute by claiming that c_x has sent a wrong data-signature pair in M_x, which cannot be true because of (4). Therefore, c_y cannot accuse the publisher c_x of sending wrong data as long as c_x is faithful, i.e., (1) holds. ■

Combining the results above, we now prove that ADLP can attest the faithfulness of any faithful component that operates under ADLP.

Theorem 1. *If c_i is faithful in logging under ADLP, $L_i(D)$ for every $D_{i\to j}$ (i.e., publication) and $D_{j\to i}$ (i.e., subscription) must be found and identified to be valid by ADLP. That is, $L_i \in \mathbf{L_{V,f}} \Rightarrow L_i \in \widehat{\mathbf{L_V}}$ (see Figure 5 in Section III-C).*

Proof: A direct application of Lemmas 1–3 proves that for any non-colluding pair, ADLP can unambiguously identify faithful and unfaithful components. That is, (i) by the completeness, $\mathbf{L_{H,f}} = \emptyset$, and thus L_i must be found, and (ii) by the unforgeability and the correctness, $\mathbf{L_{I,f}} = \emptyset$. Therefore, $L_i \in \mathbf{L_{V,f}}$ cannot be classified into $\widehat{\mathbf{L_H}}$ or $\widehat{\mathbf{L_I}}$. ■

The theorem states that $\mathbf{L_{V,f}} \subseteq \widehat{\mathbf{L_V}}$ holds. That is, no valid entries can be incorrectly classified as invalid even when collusion exist. However, due to the possibility of collusion (thus $\mathbf{L_{V,c}}$ may not be non-empty), $\widehat{\mathbf{L_V}} \subseteq \mathbf{L_{V,f}}$ does not necessarily hold. Nevertheless, Theorem 1 implies that any 'edge' component of a collusion group (e.g., B in Figure 2) that communicates with an outside component cannot enter invalid log entries or hide them for the transmission without being detected. Extending this property, if the system is collusion-free, any unfaithful act is always detectable.

Theorem 2. *If the system is collusion-free, any unfaithful component can be detected by ADLP.*

Proof: In addition to $\mathbf{L_{I,f}} = \emptyset$ and $\mathbf{L_{H,f}} = \emptyset$, the correctness property ensures $\mathbf{L_{V,u}} = \emptyset$. Also, due to the non-collusion property, $\mathbf{L_{V,c}} = \emptyset$. Hence, $\widehat{\mathbf{L_V}} = \mathbf{L_{V,f}}$. Since for each publisher-subscriber pair, at least one is faithful and identifiable (due to Theorem 1), we can always verify the validity of the counterpart with respect to the faithful component. Therefore, $\widehat{\mathbf{L_I}} = \mathbf{L_{I,u}}$ and $\widehat{\mathbf{L_H}} = \mathbf{L_{H,u}}$, i.e., log entries that are invalid or hidden by unfaithful components are identifiable. ■

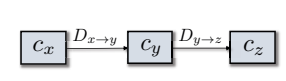
2) Temporal Causality: Timestamps in log entries, together with sequence numbers embedded in data or message header, can help establish temporal causal relations among data transmission. For illustration, let us consider three components, c_x, c_y, and c_z, and data transmissions $D_{x\to y}$ and $D_{y\to z}$ shown in Figure 10(a). Here, the data flow $D_{x\to y}$ precedes $D_{y\to z}$. The middle component, c_y, enters two log entries; upon receiving (resp. sending) data from c_x (resp. to c_z). Let us denote them by $L_{y,in}$ and $L_{y,out}$, respectively. c_x and c_z also write logs entries. Let us denote them by $L_{x,out}$ and $L_{z,in}$, respectively. Now, let $t_{x,out}$, $t_{y,in}$, $t_{y,out}$, and $t_{z,in}$ denote the timestamps in the log entries. If all of the components are faithful, $t_{x,out} < t_{y,in} < t_{y,out} < t_{z,in}$ must hold as shown in Figure 10(b), assuming a proper time synchronization mechanism is in place. The following lemma proves that an unfaithful component cannot change the precedence relation between $D_{x\to y}$ and $D_{y\to z}$ without being detected.

Lemma 4 (Temporal Causality)**.** *If $D_{x\to y}$ happened before $D_{y\to z}$, the precedence relation between them cannot be changed by any unfaithful component while avoiding a detection unless all of the components collude together.*

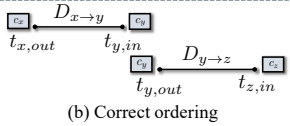
Proof: It is easy to show that c_x alone cannot change the precedence relation between $D_{x\to y}$ and $D_{y\to z}$ because this would require c_y to change the timestamps in $L_{y,in}$ and $L_{y,out}$ such that $t_{y,out} < t_{y,in}$. However, this is not possible because c_x and c_y is not a colluding-pair. For a similar argument, c_z cannot change the precedence relation.

Now suppose c_y is unfaithful. It can alter its timestamps, $t_{y,out}$ and $t_{y,in}$, such that $t_{y,out} < t_{y,in}$. As can be seen from Figure 10(c), c_y alone cannot break the precedence relation between $D_{x\to y}$ and $D_{y\to z}$. For c_y to avoid the detection, it needs to collude with both c_x and c_z, as shown in Figure 10(d) so that $t_{y,out} < t_{z,in} < t_{x,out} < t_{y,in}$ would hold. However, unless all of the them collude together, this is not possible. ■

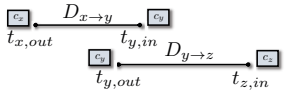
It should be noted that timestamps are only useful to establish a precedence relation between data transmissions. The timestamp value itself does not prove the exact timing



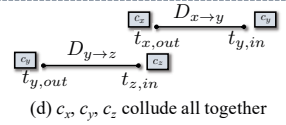
(a) Series of data flow



(b) Correct ordering



(c) c_y forges timestamps so that $t_{y,out} < t_{y,in}$

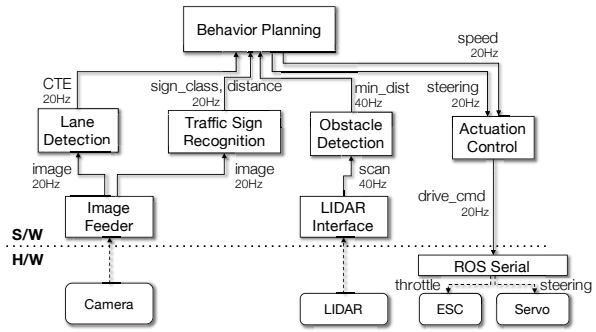


(d) c_x, c_y, c_z collude all together

Figure 10. Temporal causal relation between two data transmissions generated by a chain of components.

Data	Size (bytes)
image	921641
CTE	20
sign_class, distance	28
min_dist	20
scan	8705
speed	20
steering	20
drive_cmd	24

(a) Self-driving car platform and data used for autonomous navigation



(b) ROS nodes and data topics

Figure 11. 1/10-scale self-driving car navigating an indoor tracking using a camera and a LIDAR sensor.

of data production or consumption because it can be set at an arbitrary point of the communication process by the log entry owner.

V. Prototype Implementation

In this section, we present the implementation details of ADLP on a miniaturized self-driving car platform.

A. Self-Driving Car Platform

We built a miniaturized self-driving car, shown in Figure 11(a), that can autonomously navigate an indoor track using a camera and a LIDAR (Light Detection and Ranging) sensor. The computing module is Intel NUC mini PC [18] which has a dual-core Intel Core i5-7260U processor operating at 2.20 GHz, and a main memory of 8 GB. The actuation commands are sent from this main computer to a microcontroller board that outputs control signals to the car's actuation modules.

We run unmodified Ubuntu 16.04 on which ROS (Robot Operating System) Kinetic [10] is installed. The self-driving application is comprised of a set of ROS *nodes* (equivalent to components in this paper). Most of the them are written in Python and thus use rospy [19], the Python client library for ROS. Figure 11(b) shows the ROS nodes that comprise the self-driving application, their publisher-subscriber channels, and data (called as *topic* in ROS) transferred via the channels. These ROS nodes run concurrently on the quad-core processor as individual Linux processes.

B. ADLP Implementation

We implemented ADLP as part of rospy. ROS uses TCP/IP socket for data transmission from publisher to subscriber (whether or not they are on the same machine). We modified the ROS transport layer in rospy to instantiate ADLP's extended messaging scheme. For logging operations,

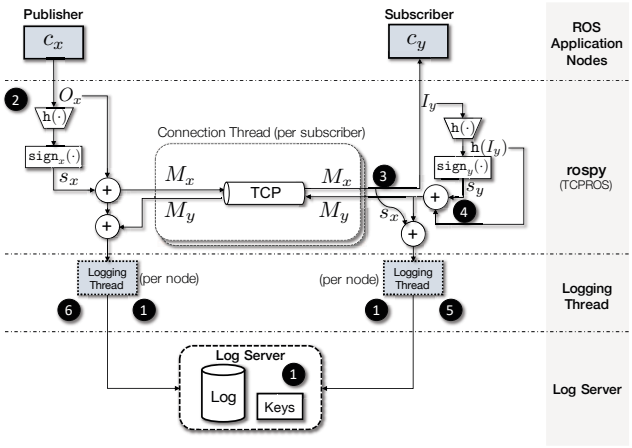


Figure 12. ADLP implementation in rospy and the messaging and logging processes. $M_x = (O_x, s_x), M_y = (\mathtt{h}(I_y), s_y)$

we created a *Logging Thread* that runs in parallel with each node's main thread. One logging thread is created per ROS node, no matter how many topics the node publishes and subscribes. Figure 12 shows the ADLP implementation in rospy as well as the messaging and logging process under ADLP, which we detail in what follows. This process is transparent to the ROS application layer. The key benefit is that applications are not aware of ADLP's operation and even its presence, hence no modification is needed at the application layer. This allows system developers to reuse legacy applications while utilizing ADLP.

① **Key registration:** When a logging thread starts up, it first generates a pair of public and private keys. In our implementation, we use RSA-1024. Each node generates its keys using PyCrypto library [20]. The log server stores the public keys so that they can be used for verifying log entries later. As aforementioned, each ROS node runs as a *standalone* Linux process. Hence each node's private key is protected from others as long as proper Linux-process security mechanisms are in place.

② **Signing and publication:** Publisher c_x publishes data. The hash of the data is computed and then signed by using SHA-256 and PKCS#1 v1.5 [21], respectively. Using the data and the signature, a message M_x is composed and sent to each of the subscribers separately (ROS runs a connection thread per subscriber, not per topic). Note that the hash and the signature are computed just *once* for a single publication. If the acknowledgement to the previously published message has not been received from a particular subscriber, the new message is not sent to the subscriber. The message M_x is also stored at the logging thread for a future use in step ⑥.

③ **Receiving message:** Subscriber c_y's ROS transport layer receives the message M_x. It is decomposed into the data, $I_y = O_x$, and the publisher's signature, s_x. s_x is stored at the subscriber's logging thread for a future use in step ⑤.

④ **Sending ACK message:** The hash of the received data is computed and signed with the subscriber's private key that is stored in its logging thread. The acknowledgement message

M_y is sent to the publisher while the data I_y is sent up to the subscriber's application layer. This acknowledgement message has a fixed size of 160 bytes – 32 bytes for $\mathtt{h}(I_y)$ and 128 bytes for s_y due to `SHA-256` and `RSA-1024`, respectively.

⑤ **Entering subscriber's log entry:** The message M_y is given to the logging thread. It then creates a log entry based on M_y and s_x as well as the basic information (i.e., unique node name, seq, time, data type, and data flow direction). s_x was obtained from the publisher in step ③. We use Google protocol buffers [22] for serializing the log entries on the network and the disk at the log server. Note that the same log entry structure (using only the required fields) is used for the naive logging scheme (Definition 2).

⑥ **Entering publisher's log entry:** Upon receiving the subscriber's acknowledgement message M_y, the logging thread writes a log entry based on M_y and M_x, the latter of which was obtained in ②, as well as the basic meta-information.

Each ROS message includes a header field which we use to set the sequence number. That is, the sequence number is a part of the ROS message digest which is hashed and signed. In addition, notice from Figure 12 that, there is no dependence of the ROS side on the log server; log entries are simply pushed into the server. Hence, `ADLP` is free from a single-point failure – any failure at the log server does not interrupt a normal operation of the ROS nodes.

VI. Evaluation

In this section, we evaluate `ADLP` in terms of the overhead due to the introduction of the cryptographic functions and the extended messaging scheme under `ADLP`.

A. Message Latency

The use of cryptographic functions in `ADLP` incurs indispensable computational overhead. Hence, we first measured the times to perform hashing and signing on representative data types that have different sizes. Table I shows the average times to perform hashing-only and hashing + signing on different types of data for the sample size of 3000 respectively. In order to see the impact of these overheads on the *message latency*, we also measured the end-to-end message times from publisher to subscriber for synthetic data of varying size as shown in Figure 13. Here we compare `ADLP`, which attaches signed hash of data in each message, against the naive scheme ('baseline' defined in Definition 2), which sends only data in each message. In the latter, no crypto functions are used. We can see from the results that the increases in latency under `ADLP` is approximately the twice of hashing+signing time. This is because the subscriber also

Table I
Hashing and signing time for different data types.

Type	Size (bytes)	Hashing only Avg. time (stdev)	Hashing + Signing Avg. time (stdev)
Steering	20	0.109 ms (0.025 ms)	3.042 ms (0.684 ms)
Scan	8705	0.201 ms (0.049 ms)	3.129 ms (0.671 ms)
Image	921641	2.638 ms (0.353 ms)	3.457 ms (0.460 ms)

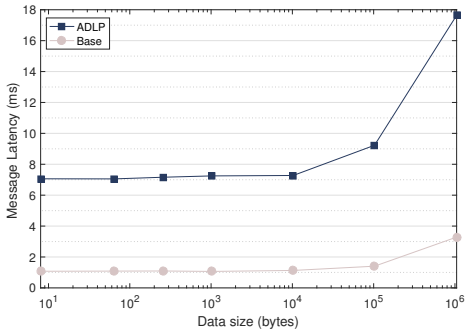


Figure 13. Average message latency from publisher to subscriber.

performs the crypto functions when receiving a message from the publisher. Recall that the subscriber is required to return an acknowledgment message that contains its own version of hash, $\mathtt{h}(I_y)$, and the signed hash, $\mathtt{sign}_y(\mathtt{h}(I_y))$ (see Figure 7 in Section III or Figure 12 in Section V), as soon as it receives data I_y. In the current implementation, this is performed in the middle of message deserialization step before passing the data to the subscriber's application layer. Hence, the latency at the subscriber's side can be improved by separating the acknowledgment step from the main execution path and running them in parallel, which will hide the added latency due to the crypto functions. Nevertheless, the current implementation is practical enough to publish messages at higher rates than that required. For instance, images can be published as frequently as at 60 Hz without backlog which is much faster than 20 Hz in our self-driving application.

B. CPU Utilization

The use of the crypto functions also results in increased CPU utilization. For this experiment, we used one Image publisher and a set of Image subscribers. The choice of Image data is to see the impact of large-size data on the CPU overheads. Figure 14 presents the average CPU utilization of the publisher measured for 5 minutes and the standard deviations (the error bars). We compare three methods – (i) no logging, (ii) base (i.e., naive) logging, and (iii) `ADLP`. Note that no crypto functions are used in (i) and (ii). Hence, by comparing (i) and (ii), we can estimate the bare minimum

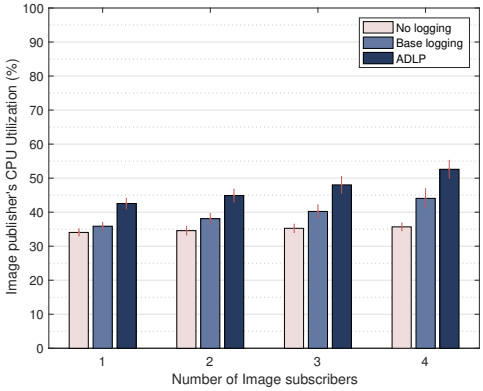


Figure 14. Average CPU utilization of Image publisher for different number of Image subscribers.

Table II
AVERAGE SYSTEM-WIDE CPU UTILIZATION WHEN RUNNING THE SELF-DRIVING APPLICATION ON THE PROTOTYPE PLATFORM.

	Idle	No Logging	Base Logging	ADLP
Average	26.03%	77.21%	83.24%	88.69%
Stdev	1.92%	2.63%	3.58%	3.36%

overhead due to the basic logging operations (i.e., sending log with data to the log server). For one publisher-subscriber pair, this overhead is about 1.8%, which accounts for the additional operations to create and enter log entries (including the data). The additional CPU utilization for the handling of more subscribers approximately linearly increases.

Now, from (ii) and (iii) we can estimate the overhead due to the crypto functions as well as the handling of subscriber's acknowledgement. For the case of single subscriber, this overhead is about 6.7%. However, it does not linearly increase with the number of subscribers; for the case of four subscribers, the overhead is still about 8.5%. This is because the publisher needs to execute the crypto functions *only once* for each publication, regardless of the number of subscribers. Hence, we can consider the cost of ADLP to be relatively fixed when compared to the indispensable overhead of base logging activities. Furthermore, the relative cost, i.e., $\frac{(iii)-(ii)}{(ii)-(i)}$, becomes smaller as more subscribers are served.

Table II shows the average system-wide CPU utilization when running the self-driving car application. As above, we compare the three cases as well as the case when the application is not running (labeled 'Idle'). We can first see that the application without any logging uses nearly 50% of the CPU utilization (equivalent to 2 logical cores out of 4). Additional 6% is used to enable the base logging operations. Consistent with what was observed above, the CPU resource needed for ADLP, which accounts for about 5.45%, is relatively smaller than that for the basic logging operation.

C. Message and Log Entry Sizes

We also evaluate the overheads in message and log entry sizes due to ADLP's extended messaging and logging schemes. Table III shows message sizes for different types of data under ADLP scheme. For a data of size $|D|$, the actual message size is $|D| + 4 + 128$ due to (i) a 4-byte length preamble attached by the ROS transport layer (to indicate the message digest size) and (ii) a 128-byte length signed-hash due to RSA-1024. Hence, the overhead is calculated by $\frac{128}{|D|+4}$. While the overhead is merely 0.014% for Image data, it can be a significant increase for small-size data such as Steering angle.

Table III
MESSAGE AND LOG ENTRY SIZES (ALL IN BYTES).

Type	**Message size**	**Log entry size**		
		Scheme	Publisher's	Subscriber's
Steering	152	Base	69	84
		ADLP	359	337
Scan	8837	Base	8752	8767
		ADLP	9042	350
Image	921773	Base	921687	921702
		ADLP	921977	350

Table III also shows the log entry sizes for the different data types. With the base logging scheme, both publisher and subscriber simply store data as is along with the node's name, data type, transmission direction, and time as defined in Definition 2. ADLP generally increases the log entry sizes due to the addition of signed hashes. However, the overhead becomes smaller as the data size increases. For instance, each log entry entered by Image publisher needs only 0.03% more extra bytes under ADLP. More importantly, subscribers do not need to store data as is. By storing the hashed version instead, the logger can save significant space; the entry size for an Image subscription is only 350 bytes, which is 0.08% of what it would have been when storing the data as is. For small-size data, however, the space can be saved by storing data itself instead of the hash.

D. Log Generation Rate

Data-driven applications generate a massive amount of data at high frequency. For instance, even a single Image Feeder node in our prototype generates about 18 MB/s of image data (about 900 KB/image at 20 Hz). The storage requirement becomes more significant when considering (possibly a number of) subscribers that also need to create log entries for their own accountability. In order to evaluate the storage overhead, we measured the log generation rate.

Figure 15 shows the log generation rates for (i) Steering angle and (ii) Image. For each data type, we compare three cases - (a) base logging scheme with the subscriber storing data as is (labeled 'Base'), (b)-(c) ADLP with the subscriber storing hashed data (labeled $\mathtt{h}(D''_y)$) or data as is (labeled D''_y), respectively. As discussed in Section VI-C, for large-size data such as Image, storing the hashed data in the subscriber's log entry can reduce a signification amount of log size. Also, the extra storage required by ADLP compared to the base logging is small.

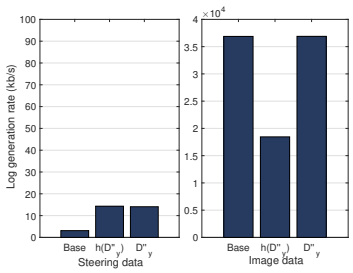


Figure 15. Log generation rates for Steering data (Left) and Image data (Right). Notice the Y-axis scales.

Table IV
THE LOG GENERATION RATES.

	Average (Stdev)
Base	36.893 (0.462) Mb/s
ADLP	37.297 (0.474) Mb/s

We also measured the system-wide log generation rates. Table IV shows the results of the basic logging and ADLP, in both of which the subscribers store hashed data. As we can see, ADLP generates about 1.1% more log than the base logging. In other words, ADLP uses 400 KB more per second, which is equivalent to a half of single image used in our self-driving application. Therefore, we can conclude that ADLP incurs a small overhead in storage space.

E. Limitations and Possible Improvements

Support for `roscpp`: The current version of ADLP is implemented in Python as part of rospy library. Cer-

tain legacy ROS codes are written in C++ using roscpp, the C++ implementation of ROS library. By extending to roscpp, the coverage of ADLP in ROS-based systems can be increased. Also, ADLP's computational overhead, mostly from the crypto and socket functions, can be greatly reduced.

Aggregated Logging: The storage and CPU overheads can be reduced by aggregating log entries: for instance, a publisher creates a single log entry per publication, regardless of the number of subscribers, containing all of the subscribers' hashes and signatures. This will significantly save the log storage especially for large-size data such as images. This kind of optimization can also be done at the log server-side.

Scalability: Full-scale autonomous systems use a number of high-throughput devices that generate a significantly large amount of data. Hence, a further validation is required to evaluate the scalability of ADLP. We envision that high-performance computing platforms (such NVIDIA Drive [23]) on such systems can facilitate the ADLP operations. We intend to investigate engineering solutions (e.g., lightweight crypto functions, log structure optimization, hardware accelerator) and their impact on the ADLP's scalability, as well as extension to other data distribution frameworks such as OpenDDS [11].

VII. Related Work

Logging is a useful technique as a forensic tool to collect system events, detect intrusions, and diagnose faults [6], [14], [24], [25]. Many work have focused on securing log from tampering by adversaries. Logs can be made *tamper-evident* by employing cryptographic solutions such as message authentication codes (MAC) [26], hash chains [7], [15], and Merkle tree [27]. The key idea is to frequently compute and update a MAC or hash over the current log in a way that an attacker's modification on the prior log cannot be hidden from detection. As aforementioned, tamper-evident logging is orthogonal to the problem discussed in this paper because it addresses how to ensure the integrity of the logs created by participants whose goal is to provide correct logs and/or keep their log intact. Whereas, ADLP concerns potentially unfaithful participants who are not guaranteed to log correctly. Nevertheless, a tamper-evident logging can provide added security and robustness to data logging.

Provenance has been used in scientific workflow to keep track of ownership and processing of scientific data when multiple parties are involved [8], [12]. Data provenance can provide useful information for forensic, auditing, test/debugging processes [9], [25], [28]–[30] and thus have been applied to distributed systems [31], networking [32], storage systems [33], etc. Data provenance shares similar challenges with tamper-evident logging – namely, protecting provenance record from adversaries who are motivated to alter the provenance history. Hence, similar measures, such as hash chain and signing, are employed for integrity protection or tamper detection [34]. Data provenance has become more important in cloud computing as data resides in shared environments [35]. However, the openness of cloud computing also brings in *confidentiality* issue [36] which often conflicts with accountability requirement.

Data privacy is becoming important also in robotics and autonomous systems because many sensory data (e.g., location, camera image) contain sensitive information [37]. This trend has elicited increasing interest in the security of robotics software system often powered by ROS. In particular, the publish-subscribe model poses a broad range of security challenges including unauthorized data subscription or data injection, denial of service, integrity protection, node authentication, and so on [38], [39]. A number of solutions based on secure communication using TLS (Transport Layer Security) [40] or application layer encryption [41] have been proposed, and these can help improve the security and thus safety of autonomous systems by being employed together with an accountable data logging mechanism.

VIII. Conclusion

Autonomous system developers wish to analyze systems behavior which are often non-deterministic due to the data-driven nature of modern intelligent computing. The technology for assigning responsibility for data production and use to software components in autonomous systems will have a profound impact on the industry since there is a pressing need for safety-critical autonomous systems to accommodate such advanced applications. This work explored the opportunity to instantiate an accountable data logging mechanism through a novel application of primitive cryptographic functions in data production, consumption, and recording. ADLP will drastically reduce the efforts to verify the validity of the runtime evidence, making the complex software system easier to understand and thus increasing the safety of systems.

Acknowledgment

This work is supported in part by NSF grants 1521523, 1715154, and 1763399. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] National Highway Traffic Safety Administration, "Event data recorder," https://www.nhtsa.gov/research-data/event-data-recorder.

[2] "Report: Software bug led to death in ubers self-driving crash," *ARS TECHNICA*, May 2018, https://arstechnica.com/tech-policy/2018/05/report-software-bug-led-to-death-in-ubers-self-driving-crash.

[3] "Self-driving car companies should not be allowed to investigate their own crashes," *The Guardian*, April 2018, https://www.theguardian.com/science/political-science/2018/apr/13/self-driving-car-companies-should-not-be-allowed-to-investigate-their-own-crashes.

[4] J. Zhou and D. Gollmann, "Evidence and non-repudiation," *Journal of Network and Computer Applications*, vol. 20, no. 3, July 1997.

[5] T. Coffey and P. Saidha, "Non-repudiation with mandatory proof of receipt," *Computer Communication Review*, vol. 26, 1996.

[6] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 2, May 1999.

[7] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[8] S. B. Davidson and J. Freire, "Provenance and scientific workflows: Challenges and opportunities," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[9] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2016.

[10] Robot Operating System (ROS). http://www.ros.org.

[11] OpenDDS. http://opendds.org.

[12] L. Moreau, J. Freire, J. Futrelle, R. E. McGrath, J. Myers, and P. Paulson, "The open provenance model: An overview," in *International Provenance and Annotation Workshop*, Springer, Berlin, Heidelberg, 2008, pp. 323–326.

[13] C. N. Chong, Z. Peng, and P. H. Hartel, "Secure audit logging with tamper-resistant hardware," in *IFIP International Information Security Conference*, 2003.

[14] V. Karande, E. Bauman, Z. Lin, and L. Khan, "Sgx-log: Securing system logs with sgx," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, 2017.

[15] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," in *Proceedings of the 7th Conference on USENIX Security Symposium*, 1998.

[16] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, Feb. 1978.

[17] Y. Liu and B. Plale, "Survey of publish subscribe event systems," Indiana University, Bloomington, Tech. Rep., May 2013, technical Report TR574.

[18] Intel NUC Kit NUC7i5BNK. https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc7i5bnk.html.

[19] rospy: Python client library for ROS. http://wiki.ros.org/rospy.

[20] Python cryptography toolkit (pycrypto). https://pypi.org/project/pycrypto.

[21] Rfc 8017 - pkcs #1: Rsa cryptography specifications version 2.2. https://tools.ietf.org/html/rfc8017

[22] Google protocol buffers. https://developers.google.com/protocol-buffers/.

[23] NVIDIA Drive. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/.

[24] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[25] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation," in *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[26] M. Bellare and B. S. Yee, "Forward integrity for secure audit logs," University of California, San Diego, Tech. Rep., Nov. 1997.

[27] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *Proceedings of the 18th USENIX Conference on Security Symposium*, 2009.

[28] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[29] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *Proceedings of USENIX Annual Technical Conference*, 2018

[30] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure network provenance," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2011.

[31] Y. S. Tan, R. K. Ko, and G. Holmes, "Security and data accountability in distributed systems: A provenance survey," in *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications*, 2013.

[32] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," in *Proceedings of the ACM Conference on SIGCOMM*, 2014.

[33] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2006.

[34] R. Hasan, R. Sion, and M. Winslett, "The case of the fake picasso: Preventing history forgery with secure provenance," in *Proccedings of the 7th Conference on File and Storage Technologies*, 2009.

[35] H. Takabi, J. B. D. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security and Privacy*, vol. 8, no. 6, Nov. 2010.

[36] R. Lu, X. Lin, X. Liang, and X. S. Shen, "Secure provenance: The essential of bread and butter of data forensics in cloud computing," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[37] C. Bloom, J. Tan, J. Ramjon, and L. Bauer, "Self-driving cars and data collection: Privacy perceptions of networked autonomous vehicles," in *Proceedings of the 13th Symposium on Usable Privacy and Security*, July 2017.

[38] R. Dóczi, F. Kis, B. Sütő, V. Póser, G. Kronreif, E. Jósvai, and M. Kozlovszky, "Increasing ROS 1.x communication security for medical surgery robot," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 2016.

[39] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner, "Security for the robot operating system," *Robot. Auton. Syst.*, vol. 98, no. C, Dec. 2017.

[40] B. Breiling, B. Dieber, and P. Schartner, "Secure communication for the robot operating system," in *Proceedings of the 11th Annual IEEE International Systems Conference*, 2017.

[41] F. J. R. Lera, J. Balsa, F. Casado, C. Fernández, F. M. Rico, and V. Matellán, "Cybersecurity in autonomous systems: Evaluating the performance of hardening ROS," in *Workshop on Physical Agents*, 2016.