LOW COST MULTICAST NETWORK AUTHENTICATION FOR EMBEDDED CONTROL SYSTEMS

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

for the degree of

DOCTOR OF PHILOSOPHY

in the department of

ELECTRICAL AND COMPUTER ENGINEERING

Christopher Johnathan Szilagyi

B.S., Electrical and Computer Engineering, Johns Hopkins University

M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University

Pittsburgh, Pennsylvania

May, 2012

Abstract

Security for wired embedded control networks is becoming a greater concern as manufacturers add increasing connectivity from these internal wired networks to the outside world. In the event that an attacker gains access to an embedded control network, the attacker might manipulate potentially safety-critical message traffic to induce a system failure. Protocols used in these networks omit support for multicast authentication to prevent masquerade and replay attacks. While many approaches for multicast authentication exist, the unique constraints of embedded control networks make incorporating these schemes impractical. Resource limited nodes must authenticate short periodic messages to multiple receivers within tight real-time deadlines while tolerating potentially high packet loss rates.

This work presents time-triggered authentication: a new multicast authentication technique to prevent masquerade and replay attacks in wired embedded control networks. This approach takes advantage of the existing temporal redundancy of many embedded control networks by verifying messages across multiple samples using one message authentication code (MAC) per receiver (OMPR), each being just a few bits in size. This approach can be applied to both state transition commands and reactive control messages, and allows a tradeoff among authentication bits per packet, application level latency, tolerance to invalid MACs, and probability of induced failure, while satisfying typical embedded system constraints.

This work also presents validity voting: a method to improve overall bandwidth efficiency and reduce authentication latency of OMPR in time-triggered authentication by using unanimous voting on message values and validity amongst a group of nodes. This technique decreases the probability of successful per-packet forgery by using one extra bit per additional vote, regardless of the number of total receivers.

We also show how to use two existing multicast authentication techniques (TESLA and a master-slave approach using hash tree broadcast authentication) in conjunction with time-triggered authentication in an embedded control network. We compared all four techniques in terms of scalability with respect to per-packet assurance (probability of successful per-packet forgery) and number of receivers. We also compared the techniques in terms of sensitivity to packet loss, node failure, and node compromise.

Finally, we demonstrated the applicability of time-triggered authentication using each of the four techniques in two case studies. First, we implemented each technique in a simulated elevator control network. Second, we examined the impacts of authentication on bandwidth for an automotive network workload.

Our comparisons and case studies show that OMPR and validity voting with few votes are the most bandwidth efficient approaches for embedded control networks characterized by few receivers and weak per-packet assurance. TESLA and validity voting using many votes are the most bandwidth efficient approaches for very large numbers of receivers or strong per-packet assurance levels. A master-slave approach can be one of the most bandwidth efficient of all approaches, assuming a trusted master node is available and no passive nodes (non-transmitting) are present in the network. Also, we show OMPR and TESLA are least sensitive to networks where packet loss, node failure, and node compromise. Thus, these two approaches are better suited to applications with requirements to tolerate these types of faults and failures than validity voting or master-slave.

Acknowledgements

This thesis is dedicated to my family. Thank you all for your support and encouragement. To my Mom and Dad: thank you for giving me all the tools I needed to succeed and teaching me how to see the important things through to the end. To Frank and Emily: thank you for keeping me humble like great siblings should. To Dave: thank you for joining me on this long journey and making me take the time to have fun.

I thank my academic advisor Professor Philip Koopman, for teaching me that if an idea is only obvious in hindsight, then it's probably a cool research result. Thank you for all your guidance over these past few years. Also, I thank my committee members Professor Adrian Perrig, Professor Bruno Sinopoli, and Dr. Charles Weinstock for their support and feedback.

I thank Erik Rennenkampf for championing this whole shebang and always having my back, even in my absence. I thank George Reynolds for showing me this opportunity. I thank Kevin Endlich for supporting me and handling so many details, both big and small. I thank George Kalb for sparking my interest in security and showing me how easy reverse engineering can be. Also, I thank Pat Tooman and Chris Meawad for continuing to support me while I finished.

I would also like to thank Justin Ray for taking the time to answer so many of my questions over the past few years. Thanks to Elaine Lawrence for keeping an eye out for me. Also, thanks to Teddy Martin, Nik White, and Andrew Jameson for allowing me to use their elevator design.

Finally, I thank my sponsors for funding this endeavor. This research was funded in part by General Motors through the GM-Carnegie Mellon Vehicular Information Technology Collaborative Research Lab.

Table of Contents

List of Figures

List of Tables

1 Introduction

While embedded control networks have traditionally been physically isolated, manufacturers are increasingly adding connectivity amongst internal networks, to external networks (e.g., wireless and Internet), and to multimedia devices [Koopman05]. This connectivity enables new features, but also introduces new avenues for attacks on a system. In the event that an attacker accesses the internal embedded control network, whether through physical manipulation or via a compromised network connection, they can trivially inject messages to disrupt system operation and subsequently violate safety requirements.

Such attacks have already been demonstrated on automotive control networks. Koscher et al. [Koscher10] have demonstrated that an attacker able to connect to an automotive control network (e.g., via a wireless connection through an attached MP3 player or laptop, or via physical access) can inject messages to control safety-critical actuators. An attacker might access the embedded control network through such a connection to engage an emergency brake in a car while it is traveling on a highway, unlock doors and start the engine, or shut off headlights while traveling at night.

Embedded control networks commonly use protocols such as Controller Area Network (CAN) [Bosch91], FlexRay [FlexRay05], and Time-Triggered Protocol (TTP) [TTTech03] for multicast communication over a shared broadcast bus. In *multicast* communication, a transmitting node broadcasts a single copy of a message to multiple receivers in the network (as opposed to unicast communication where a node transmits a distinct copy of the same message for each receiver). Applications include distributed automotive, aviation, robotics, and industrial control systems. Safety, reliability, and cost have traditionally been the primary concerns in these sys-

tems, with security a minor concern. Most embedded control networks do not have any built in security to support authenticating nodes, encrypting data, restricting message types a node can send, or preventing Denial of Service (DoS) attacks.

1.1 Problem statement

This thesis addresses the problem of masquerade and replay attacks on embedded control networks. Masquerade attacks [Schneier95] occur when a node sends a message in which it claims to be a node other than itself. This attack can be performed by broadcasting during another node's Time Division Multiple Access (TDMA) slot or by changing a message identifier value. Replay attacks [Schneier95] occur when a previously sent message is recorded and retransmitted by an attacker. Authentication allows a receiver to confirm the identity of a sender, typically via cryptographic mechanisms such as a Message Authentication Code (MAC) or a Digital Signature [Menezes96]. While wired embedded network protocols use error detection codes to verify message integrity, these codes can readily be forged, and are no substitute for strong cryptographic mechanisms.

As a practical matter, a successful masquerade attack in current embedded systems typically gives an attacker the ability to make a system unsafe in limitless ways. Multicast authentication is needed to prevent such attacks in systems implementing a wired broadcast network.

Thesis statement: Integrating multicast authentication into embedded control network protocols (e.g., CAN, FlexRay, or TTP) is challenging due to the limitations and requirements of these networks. Resource limited nodes must authenticate short periodic messages to multiple receivers within tight real-time deadlines while tolerating potentially high packet loss rates. Furthermore, authentication must consume a relatively small proportion of bandwidth compared to the

data being authenticated. A reasonable size for authenticators may be up to a few bytes of a data payload, similar in size as existing error detection codes. However, most existing multicast authentication techniques require hundreds or thousands of bits to authenticate each message. We propose new techniques to provide multicast authentication while enabling tradeoffs to meet embedded network constraints.

1.2 Time-triggered authentication

One simple method of reducing authentication bandwidth costs could be to use a single multicast authenticator to authenticate an entire batch of samples of the same message type, but this has several undesirable properties. While this approach could reduce bandwidth consumed by authentication to an arbitrarily small fraction, it also effectively reduces the sampling for that message type; a receiver cannot verify any of the messages in the batch until all are received. This is a problem for real-time control. This could reduce system performance and responsiveness to inputs. Further, it also reduces loss tolerance. If any of the samples in the batch suffer a transmission error, a receiver cannot verify any of them.

This thesis proposes an authentication technique called time-triggered authentication. This technique allows nodes in an embedded control network to verify periodic messages which drive state-changes and actuations over multiple message samples, using authenticators only a few bits in size. It allows verification of data integrity and authenticity on a per-packet basis and enables perfect loss tolerance. Time-triggered authentication takes advantage of the existing temporal redundancy in the system to amortize authentication bandwidth overhead across multiple periodic message samples. Transmitters truncate MAC tags to a number of bits based on the degree of temporal redundancy and criticality of each sample (i.e. the effect of an individual message sample on actuator outputs).

Time-triggered authentication can be combined with any multicast authentication technique based on symmetric authentication functions whose outputs can be truncated (e.g., hash based MAC functions). This work evaluates the use of four multicast authentication techniques in conjunction with time-triggered authentication: one MAC per receiver, validity voting, TESLA [Perrig00], and a master-slave approach based on hash tree broadcast authentication [Chan08].

Our approach enables design tradeoffs among per-packet authentication cost, application level latency, tolerance to invalid MACs, and probability of induced failure, while satisfying typical embedded system constraints. Further tradeoffs can be performed based on the multicast authentication technique used with time-triggered authentication.

1.3 Thesis contributions

This thesis makes four main contributions:

- Time-triggered authentication: an efficient technique for authentication of periodic messages in a wired embedded network that enables a tradeoff amongst authentication bandwidth overhead, application level latency, probability of maliciously induced failure, and tolerance to occasional invalid authenticators. Time-triggered authentication is first applied to one MAC per receiver.
- Validity voting: a technique that uses voting to allow a group of nodes to cross-check the validity of messages amongst themselves to improve the bandwidth efficiency of one MAC per receiver. This technique expands the trade space to include number of votes and sensitivity to packet loss.
- A comparison of one MAC per receiver and validity voting to two existing multicast authentication techniques: TESLA and hash tree broadcast authentication using a trusted master.

These comparisons illustrate tradeoffs amongst techniques which can be integrated with time-triggered authentication.

- Two case studies in which we applied time-triggered authentication in conjunction with each of the four techniques to representative embedded control network applications and observed impacts on system resources and performance. Both applications use the CAN protocol.

Our techniques are intended to enable authentication in common embedded network protocols (e.g., CAN, FlexRay, or TTP), without the need for any modifications of the protocol. However all implementations in this work use the CAN protocol.

1.4 Thesis outline

This document is organized as follows: Chapter 2 covers background material such as design constraints and work related to embedded network authentication. Chapter 3 introduces time-triggered authentication using one MAC per receiver as a baseline multicast authentication technique. Chapter 4 builds on time-triggered authentication with validity voting to improve bandwidth efficiency of one MAC per receiver using voting. Chapter 5 compares one MAC per receiver and validity voting with two existing multicast authentication techniques: TESLA and hash tree broadcast authentication using a trusted master. Chapter 6 describes a case study in which we implement all four techniques using time-triggered authentication in a distributed embedded elevator simulation. Chapter 7 describes an application of these same techniques to an industry automotive network workload. Chapter 8 discusses some variations on techniques. Finally, Chapter 9 discusses conclusions and future work.

2 Background and related work

This chapter discusses background material describing design characteristics of embedded control networks, our attacker model, and related work in securing embedded networks.

2.1 Design constraints

This section describes the typical embedded control network constraints and characteristics that impact the design of multicast authentication mechanisms in those networks.

Distributed embedded networks connect a number of hardware Electronic Control Units (ECUs). These ECUs broadcast periodic samples of system state variables and sensor inputs via a network using a protocol such as CAN, FlexRay, or TTP. These protocols are among the most capable of those currently in use in wired embedded system networks. Many other protocols are even more resource constrained, but have generally similar requirements. We assume that embedded networks exhibit the following characteristics:

Time-triggered (periodic) communication - This work focuses on authenticating periodic messages that drive state changes and actuations. Real-time embedded control systems are often designed to be time-triggered [Kopetz97]. A real-time system is time-triggered if all communications and processing activities are initiated at predetermined points in time from an a priori designated clock tick [Kopetz97]. Each node periodically broadcasts current values of state variables and sensor inputs to the rest of the network. Safety-critical messages are often broadcast with periods on the order of milliseconds to tens of milliseconds. Non-critical messages are broadcast less often. ECUs running control loops act on the most recent input data and update their outputs accordingly, requiring per-message authentication.

We assume each node periodically broadcasts current values for a set of predefined message types according to a predefined static schedule and all nodes know this schedule. Our time-triggered authentication approach relies on a few specific characteristics of such static schedules:

- Each sample of a message type is broadcast at predefined points in time, or within a short time span around that point in time (e.g., within one message period). Receivers know when a message sample should be received by nodes in the network.
- Message types have a well defined broadcast period. Senders broadcast only one sample of a message type during each period. For our approach, receivers must be able to easily identify which period a particular message sample belongs to (i.e., a message sample should not arrive on the "edge" between two broadcast periods). Extra samples of a message type within a period indicate an error has occurred.
- Receivers can identify that a transmission error has occurred, either because a message has not been received within the predefined time or the packet was malformed (e.g., error detection code is incorrect).

Protocols such as TTCAN [Führer00], FlexRay and TTP provide these properties using a static TDMA schedule. However, TDMA is not absolutely necessary. The CAN protocol can also be used as long as the application supports the three above properties. Our analyses on two representative network workloads in Chapters 6 and 7 are implemented using the CAN protocol.

Our time-triggered authentication technique may also be applied to periodic systems which are not strictly time-triggered. System-wide time synchronization is not required either. However, the system must support the three above properties to use time-triggered authentication.

Embedded control networks might also include some event-triggered message traffic. These communications are initiated as consequences of events (significant state changes in the system).

Event-triggered messages are typically sent once (possibly with a small number of retries), often relying on acknowledgements to ensure message delivery. Time-triggered authentication is not intended to provide message authenticity for event-triggered messages. Authentication of both message types in one control network may require more than one technique. Chapter 5 discusses techniques which are better suited to authenticating event-triggered messages.

Multicast communications over broadcast bus - Most distributed embedded networks are inherently multicast. This work assumes a single-hop network in which a set of ECUs communicate over a shared communications bus. All nodes connected to the bus can receive every packet. (In CAN, hardware performs message filtering at the receiver based on content.) Each packet includes the sender's identity, often implicitly through a message identifier (CAN; FlexRay) or time slot (TTP), but usually no explicit destination information. Multi-hop networks (e.g., networks with multiple routers or gateways) are outside the scope of this work.

Static network configuration - We assume the network configuration is fixed at design time, with no runtime reconfiguration. While embedded networks typically have few nodes attached (commonly 32 or fewer), there may be cases where more are attached. In our two case studies for the elevator and automotive networks, the maximum number of receivers is 7 and 12. Some messages are consumed by a single receiver. We examine how four multicast authentication techniques scale to larger numbers of receivers in Chapter 5.

Limited authentication bandwidth - Packet sizes are small in embedded network protocols when compared to those in enterprise networks. Packets have maximum data payload sizes as small as eight bytes in the case of CAN, with the larger payloads for FlexRay and TTP being 254 bytes and 236 bytes respectively. Cost, signal integrity, and network node synchronization concerns limit data rates to 1 Mbit/sec for CAN and 10 Mbit/sec for TTP and FlexRay. Low-cost

embedded networks can be orders of magnitude slower than that. Networks are often run at nearly 100% bandwidth to minimize cost. Authentication should incur minimal bandwidth overhead regardless of the protocol used.

Our goal is to produce very small authenticators that consume just a few bytes of the data payload of each packet. This size is similar to current error detection codes used in embedded network protocols. Even though more advanced protocols such as TTP and FlexRay can send larger packets, message workloads will likely be based upon or integrated with legacy implementations on more constrained protocols. For example, one of the target applications of the FlexRay protocol is automotive control networks. These networks have historically been implemented using one or more CAN busses, which use packets with data payloads of eight bytes or less. All time slots for time-triggered messages in FlexRay must be the same length [FlexRay05], so time-triggered message slots in FlexRay will likely be sized for eight byte data payloads (or slightly more) for bandwidth efficiency.

In Chapter 5, we show how authentication bandwidth overhead scales for each of four techniques based desired per-packet forgery probability and number of receivers.

Resource limited nodes - Processing and storage capabilities of nodes are often limited due to cost considerations. For example, the S12XD series, produced by Freescale [Freescale12], is a family of 16-bit microcontrollers designed for use in general automotive body applications. These microcontrollers provide up to 32 kilobytes of RAM, 512 kilobytes of flash memory, and four kilobytes of EEPROM, with a core operating frequency of 80 MHz. Flash memory is generally not written except for software updates, so EEPROM holds non-volatile application data. Buffering and storage for authentication consume space in RAM, which is far more expensive and scarce than flash memory in such systems. Authentication mechanisms which require large

amounts of processing power or storage in RAM may not be feasible. More powerful ECUs are impractical for most nodes in the system, and many nodes are 8-bit ECUs with significantly smaller memories due to cost and power considerations.

In this work, we do not perform a detailed analysis of processing and memory requirements for techniques we use (this work instead focuses on bandwidth consumption and impacts to loss tolerance). We limit techniques to those using symmetric cryptography (which execute an order of magnitude faster than those using asymmetric cryptography). We assume that nodes have sufficient processing and memory available to compute MAC functions for each packet received or transmitted. Groza and Murvay [Groza11] provide an analysis of processing time required for an S12X derivative microcontroller (with XGATE coprocessor) to perform MD5, SHA-1, and SHA-256 hash functions. Table 2.1 shows the processing time required for these three functions for the microcontroller operating frequency of 80 MHz. In this work, we use the HMAC algorithm which requires two executions of a hash function.

Table 2.1. Hash function processing time over 8 byte payload on S12X microcontroller [Groza11].

	MD5	**SHA-1**	**SHA-256**
Execution time	373 μs	1.146 ms	2.755 ms

Tolerance to packet loss - Distributed embedded systems are subject to message blackouts from environmental disturbances such as interference from large electric motors. High quality cable shielding is often impractical due to cost, size, and weight limits. As such, authentication schemes must tolerate packet losses as part of normal system operation.

Real-time deadlines - In real-time systems, processes must complete within specified deadlines. Authentication of nodes must occur within a known time bound, with that bound being fast

enough to match the physical time constants of the system being controlled (as fast as tens of milliseconds).

2.2 Attacker model

This thesis focuses on masquerade and replay attacks. Masquerade attacks occur when a node sends a message in which it claims to be a node other than itself. This attack can be performed by broadcasting during another node's Time Division Multiple Access (TDMA) slot or by changing a message identifier value. Replay attacks occur when a previously sent message is recorded and retransmitted by an attacker.

This work uses a Dolev Yao attacker model [Dolev81] that controls the network (i.e. an attacker may modify, inject, drop, or eavesdrop upon network traffic). This model assumes authenticators are unforgeable unless an attacker has access to the appropriate key. However, because we use small MAC tags in this work, there is a non-negligible probability of a single forged packet being accepted as valid. Thus, we slightly modify this model by assuming an attacker can also "guess" an authenticator; any message and MAC tag pair has a chance of randomly verifying as correct based on the number of MAC bits used.

We do not address how an attacker gains access to a network, but rather how to prevent masquerade and replay attacks from succeeding in the event that they do gain access. For example, an attacker may gain access to the internal network through a compromised gateway connection to an external network, malicious insider code, physically attaching a new node to the network, or tampering with nodes. Attackers accessing the network through compromised nodes will have access to the key material in those nodes and can send messages from those nodes. An attacker must not be able to masquerade as any critical node they do not already control to perform a suc-

cessful attack, except with some acceptably low probability.

Embedded networks may include a mixture of critical and non-critical nodes. Critical nodes contain software "whose failure could have an impact on safety, or could cause large financial or social loss." [IEEE610.12] This work assumes limited compromise of critical nodes. Once an attacker has compromised more than one or two critical nodes, they can likely cause a successful attack without having to resort to spoofed messages. Due to the potential for an attacker accessing the network through compromised nodes regardless of criticality, authentication approaches which tolerate some level of node compromise for both critical and non-critical nodes are desirable.

Successful masquerade and replay attacks on embedded control networks can be viewed as induced system failures, because they may cause unintended release of energy or violation of safety or operational requirements of the system. The techniques proposed in this work will prevent malicious failures due to masquerade and replay attacks from occurring no more often than non-malicious failures. We use failure rates based on Safety Integrity Levels (SILs) [IEC61508] to define acceptable rates of successful masquerade and replay attacks.

This work assumes an attacker is aware of existing error detection mechanisms along with the message schedule, and is capable of injecting well-formed packets at valid times. The message schedule constrains an attacker to one forgery attempt per message period. For example, an attacker is limited to injecting a message during valid time slot in a TDMA network such as TTP or FlexRay, since transmitters are only permitted to transmit a single packet per time slot in such protocols. Using CAN, receivers can identify if an attacker is "spamming" many samples of the same message type based on the message schedule.

2.3 Authentication

Preventing masquerade attacks requires some method that provides data integrity and data origin authenticity. All methods described in this work use hash based message authentication codes to provide these properties. Further, to prevent replay attacks, all methods include the current time or message round (agreed upon by both parties) .

Data integrity is the property by which data has not been changed, destroyed, or lost in an unauthorized or accidental manner [Shirey00]. Embedded network protocols often support data integrity using error detection codes, such as cyclic redundancy checks (CRCs) computed over the header and data payload of a message. However, data integrity alone cannot prevent a masquerade attack. We assume an attacker is well aware of the widely known functions used in published protocol standards and is capable of computing a correct error detection code for any packet they modify or inject in the network.

Data origin authenticity is the corroboration that the source of received data is as claimed [Shirey00]. With this property, a receiver is able to identify the source of a message. Receivers can confirm that messages have been transmitted only by the node assigned to send that message type in the message schedule. Data origin authenticity also implicitly provides data integrity (if a message is modified, the source has changed) [Menezes96].

To provide these two properties, we use the keyed-hash based message authentication code algorithm (HMAC): a message authentication code that uses a cryptographic key in conjunction with a hash function [Krawczyk97]. A *hash function* is a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called hash-values [Menezes96]. A cryptographic hash function *h* has the following properties:

- Preimage resistance - Given the output hash-value $h(x)$, it is computationally infeasible to find input x.
- 2^{nd}-preimage resistance - Given input x, it is computationally infeasible to find a second input x' ($x \neq x'$), such that $h(x) = h(x')$.
- Collision resistance - It is computationally infeasible to find two inputs x and x', such that $h(x) = h(x')$. The attacker may freely choose both x and x', so long as $x \neq x'$.

A *message authentication code* algorithm is a family of functions h_k parameterized by secret key k, with the following properties [Menezes96]:

- Ease of computation - For a known function h_k, given a value k and an input x, output $h_k(x)$ is easy to compute.
- Compression - h_k maps an input x of arbitrary finite bitlength to an output $h_k(x)$ of fixed bit-length b.

Furthermore, given a description of the function family h, for every fixed allowable value of k (unknown to an adversary), the following property holds:

- Computation-resistance - Given zero or more text-MAC pairs $(x_i, h_k(x_i))$, it is computationally infeasible to compute any text-MAC pair $(x, h_k(x))$ for any new input $x \neq x_i$ (including the possibility for $h_k(x) = h_k(x_i)$ for some i).

Without knowledge of the secret key k (shared only between sender and receiver), an arbitrary MAC tag of b bits on an arbitrary plaintext message may be successfully verified with an expected probability 2^{-b} [FIPS 198-1]. This property remains true even if the output of the MAC function is truncated to an arbitrarily small number of bits. Truncating the output of a MAC

function does not reduce the security of the key or underlying cryptographic functions. In general, if a MAC is truncated, then its output length *b* should be as large as practical (e.g., 32 bits or more). Fewer bits can be used so long as repeated trials to are not allowed for an attacker to present a non-authentic message for verification [FIPS 198-1]. In our approach, we use MAC tags of just a few bits in length. However, our required properties for static message schedules (Section 2.1) do not allow for multiple attempts to forge a message sample.

As a final note on authentication functions: a tempting option might be to use a secret key, initialization vector, or final XOR as part of a CRC or other error detection code computation. This approach has been proposed for safety-critical systems, which assume faults are random and independent [Morris03]. Thus, it would be difficult for a fault to accidentally produce a correct error detection code. Unfortunately, this approach is not cryptographically secure for a fault model which includes a malicious attacker. Even a proprietary protocol can be fully reverse engineered from its inputs and outputs [Ewing10].

2.4 Multicast authentication

To provide data origin authenticity and data integrity in a broadcast bus, multicast authentication is needed. Many methods for multicast authentication already exist. However, none of these approaches is ideally suited for the constraints of embedded networks.

The multicast nature of embedded network protocols makes authentication particularly challenging. Cryptographic mechanisms for point-to-point communications, such as appending a single MAC to a message using a shared secret key between nodes, do not provide adequate security in a multicast setting. If more than two nodes share the same key, a receiver cannot determine which of the other nodes created the MAC. Multicast authentication requires some form of key

asymmetry, so that no receiver can masquerade as a sender. Sending one full-size MAC per receiver can provide multicast authentication, using one unique symmetric key per pair of communicating nodes. Unfortunately, bandwidth and processing overhead scales linearly with the number of receivers. This can require authenticators that are tens to hundreds of times larger than data payloads. For this reason, one MAC per receiver is often avoided for enterprise networks broadcasting to hundreds or thousands of receivers. However, by taking advantage of the temporal redundancy and small numbers of receivers in most embedded control networks, we modify this technique to produce authenticators just a few bits in size.

Another asymmetric approach is to use digital signatures. This approach provides strong source authentication using public and private keys, but the processing overhead makes it impractical for a resource constrained device to compute digital signatures for each message for real time control. For example, pagers and Palm Pilots can take several seconds to compute a 512 bit RSA signature in resource constrained nodes [Brown00]. Some approaches suggest amortizing the cost of the digital signature over a set of packets [Miner01][Park02][Perrig00][Wong98]. But, a node would have to amortize the cost over several hundred messages for this to be effective, making it unsuitable for real-time control operations.

Schemes using one-time digital signatures [Even89][Gennero97][Perrig01] allow senders to sign messages much faster than with traditional digital signatures by using one-way hash functions, at the expense of increased message sizes. Unfortunately, one-time digital signatures can incur several kilobytes of authentication data per message. This makes them impractical for embedded networks with small packet sizes and time-triggered communication, even if amortized over many packets.

Canetti et al. [Canetti99] suggest a multi-MAC scheme which appends *k* one-bit MACs to each message, computed using *k* different keys. The keys are distributed amongst receivers such that at least *w* receivers must conspire to forge a message. While this is more efficient than using one MAC per receiver, it is vulnerable to collusion by multiple nodes that together can masquerade as some other node. Mitigating collusion can require hundreds or thousands of authentication bits per message.

TESLA [Perrig00] uses time-delayed release of keys to provide asymmetry. By releasing keys at a pre-specified interval after a MAC is released, receivers can confirm the authenticity of the data from a sender. The released keys are computed using one-way hash chains. The cost of storing the entire chain of keys is prohibitive, so techniques are used to reduce memory overhead at the expense of a small recomputation cost [Jakobsson02]. While TESLA sends a single MAC per interval, it also requires the sender to include a key for each interval of messages to be authenticated. In Chapter 5, we describe a slight modification of TESLA in which the sender truncates the MAC tag to just a few bits, but we do not propose to truncate the key. Truncating the key exponentially reduces the security of this approach. Hu et. al. propose a variation on one-way hash chains, called sandwich chains, which allows smaller keys to be released per message by regularly initializing new key chains [Hu03]. However, this technique assumes the attacker does not have the computational resources to break the current key before the next is released. Chapter 5 discusses further details and tradeoffs related to using TESLA within an embedded control network. Bergadano also proposes a similar protocol using time-delayed release of keys [Bergadano00].

Chan and Perrig [Chan08] propose a multi-MAC technique called hash tree broadcast authentication in their work on secure aggregation. This technique requires the transmitter to send only

a single hash value to receivers (computed over a MAC for each receiver). Subsequently, all receivers exchange MAC tags to for verification of the sender's hash value. Chan and Perrig examine tree [Chan08], linear, and connected topologies [Chan10]. We examine the use of this technique in a broadcast bus topology using a trusted master node.

Luk et al. identify a set of seven cardinal properties of broadcast authentication in sensor networks [Luk06]. In their work, they show that viable broadcast authentication protocols exist that satisfy any six of the seven properties, but not all seven simultaneously. In this work, most of our design constraints (Section 2.1) are a subset of the seven desired properties. However, we also consider embedded network applications which allow for weak per-packet assurance (Section 3.1).

2.5 Authentication in resource constrained wireless networks

Other approaches such as SPINS [Perrig02] and TinySec [Karlof04] apply security to resource constrained wireless sensor networks. However, those approaches are specifically designed for use in wireless networks, where energy (battery life) is typically the scarcest resource. Methods for reducing overhead related to security often focuses on reducing energy consumption. These network typically do not have real-time deadlines for safety-critical applications.

For example, μTESLA [Perrig02], a version of TESLA and part of the SPINS security suite, limits the number of authenticated senders and utilizes a base station for communications to reduce overhead. A base station is often cost-prohibitive for distributed embedded real-time control systems, which use peer-to-peer wired networks. An existing node, such as an embedded gateway, might act as a base station, but would be an undesirable single point of failure and obvious attack target for the entire network. A fully distributed approach is best for the types of

systems we are concerned with, though we do consider the use of master node in this work and illustrate some of the benefits and issues with such an approach.

2.6 Embedded control network security

Morris and Koopman [Morris03] identify the potential for masquerade failures to cause accidental or malicious failures, via non-critical nodes masquerading as higher criticality nodes. They propose the use of counter-measures of varying strengths to prevent masquerading failures between nodes of varying criticality. Their approach assumes non-malicious software faults or attacks from a cryptologically unsophisticated attacker. Fault tolerance mechanisms are not necessarily secure against malicious masquerade or replay attacks. Masquerade prevention for safety-based systems typically uses bus guardians or a symmetric key shared among all trusted nodes. Compromise of a single node would permit an attacker to masquerade as any system node.

Wolf et al. [Wolf04] provide an overview of the security vulnerabilities of various in-vehicle network protocols including Local Interconnect Network (LIN), Media Oriented System Transport (MOST), CAN, and FlexRay. These vulnerabilities primarily focus upon DoS attacks intended to disable networks. Additionally, they state the need for confidentiality and authentication. Wolf et al. suggest the use of digital signatures or the asymmetric MAC scheme proposed in [Cannetti99] for authenticating sent packets along with gateways between individual in-vehicle networks. These authentication schemes may not be suitable for some distributed embedded networks, as discussed in Section 2.4.

There have been several publications demonstrating attacks on the integrity and authenticity of messages and nodes in embedded networks. Nilsson and Larson [Nilsson08] detail the actions which an attacker might take, and demonstrate masquerade attacks on CAN using simulation.

Hoppe et al. [Hoppe07] and Lang et al. [Lang07] demonstrate a combination of eavesdropping and replay attacks on CAN. Koscher et. al. [Koscher10] demonstrated the ease with which spoofed messages allow an attacker to control safety critical actuators in a live automobile. With access to the on board diagnostics port, they demonstrated that they could disable the braking system in an automobile while driving.

Nilsson and Larson [Nilsson08_2] propose a unicast authentication scheme using a 64-bit MAC computed over four consecutive message samples. A transmitter divides the tag into four parts, and places each part in the CRC field of the each of the four packets. This introduces a four message period delay before the samples can be verified as a batch. It also requires a change in the CAN protocol to support this approach (replacing the CRC). This approach uses a similar idea to our approach, amortizing authentication costs over multiple samples, but batch authenticates multiple samples to only a single receiver. This effectively reduces the sampling rate of the system if receivers must act on the most recent system state variable and sensor data. This approach also reduces loss tolerance; if any of the four samples suffers a transmission error, all four are lost. This approach also does not provide a specific means to prevent replay attacks, though the work does discuss the need for ensuring fresh messages.

Herrewege et al. [Herrewege11] propose an authentication approach called CANAuth, which provides unicast authentication of individual messages samples by taking advantage of extra bandwidth of an out-of-band channel provided by the CAN+ protocol. CANAuth transmits a 32-bit nonce and 80 bit MAC tag for each message sample to be verified. Only a single MAC tag is computed using HMAC, providing only unicast authentication and requiring an out-of-band channel. Our analysis in Chapters 6 and 7 show that all four techniques we examine (including TESLA) can provide multicast authentication to typical numbers of receivers in an embedded

network using fewer bits per packet for most levels of per-packet assurance.

Two works by Groza and Murvay each propose the use of TESLA [Groza11] and BiBa [Groza11_2] respectively and examine tradeoffs associated with each in a embedded control network using CAN. For TESLA, they examine a trade space including number of key chains, key lengths, memory requirements, and processing requirements. They also examine processing overhead on a Freescale S12 microcontroller (commonly used in automotive applications). Our work differs in that we focus bandwidth requirements, while varying MAC tag size. For the one-time digital signature scheme BiBa, they examine the bandwidth consumed using this scheme for authenticating critical message traffic. With a bus speed of 128 KBps, this scheme allowed for authenticating 286 bits per second at a cost of a bus load of 16%. To authenticate 1000 bits per second required a bus load of 100%. This analysis illustrates that one-time signatures can require extremely high overhead for verifying even a few messages.

Lastly, Chávez et al. [Chavez05] propose using RC4 encryption to provide confidentiality on CAN buses. They dismiss authentication and non-repudiation as unnecessary in these networks, under the assumption that message identifiers and error detection provide sufficient confirmation of the sender's identity. Our work relaxes this assumption by assuming that sender identity can be forged, for example as illustrated in publications that demonstrate such attacks.

2.7 Fault tolerance

Our proposed method for validity voting in Chapter 4 also shares some similarity with approaches for voting and detecting disagreement among nodes.

Voting techniques and redundancy are a classic approach to improve system reliability [Neuman56]. These techniques enable fault detection and handling to prevent fault propagation in a

system. Typically system designers assume each input to a voter or comparator mechanism fails randomly and independently of others. In our approach, nodes detect differing views of message authenticity by voting on the validity of MAC tags from other nodes. We assume the outputs of each MAC function can only be successfully forged randomly and independently of other MAC functions.

Our voting approach also has similarities to the TTP group membership service [TTTech03]. This service provides agreement on current operating mode and set of nodes believed to be correct and alive. In TTP, nodes encode membership information into packet error detection codes. Disagreeing error codes indicate either the sender or receiver failed, and nodes take appropriate action to segregate out the failed node. We use a similar technique, computing a MAC function over a previous set of values seen from the network and a bit vector indicating each value's validity. In our approach, disagreeing authenticators indicate that an attacker may have fooled one or more receivers. Nodes then reject potential forgeries.

3 Time Triggered Authentication

This section introduces *time-triggered authentication,* a new method for authenticating periodic messages in wired embedded control networks.

Time-triggered authentication uses the temporal redundancy present in most time-triggered system designs to amortize authentication bandwidth overhead across multiple time-triggered packets, while verifying each packet individually using truncated MAC tags.

In time-triggered applications, nodes periodically broadcast current values of state variables and sensor inputs to the rest of the network. Receivers then update outputs and actuators based on the most current system state. This information is typically sampled faster than the time constraints of control stability requirements. As a rule of thumb, ten or more samples are sent within the rise time of a control system or prior to a system deadline [Kopetz97][Franklin02]. Choosing such a sample rate reduces the delay between a command and the system response, smoothes outputs to steps in control input, and tolerates lost messages.

System inertia often limits the effects of an individual message sample on the output of an actuator in the system. Typically, an actuator does not instantly reach a new output position commanded by an input to its controller. For example, typical passenger cars often have a maximum acceleration of 3 to 4 m/s^2 [SuperCoupe12], whereas throttle inputs are sampled on the order of milliseconds. Suppose a passenger car has a maximum acceleration of 3 m/s^2, and a throttle input sampling period of 10 milliseconds. In the time it takes for such a car to increase its speed by 3 m/s (about 6.7 miles per hour), the throttle will have been sampled 100 times (all sustaining maximum acceleration). Greater changes in speed require even more samples sustaining a maximum

acceleration. Thus, a single message sample commanding maximum acceleration will produce a very small observable increase in vehicle speed.

This existing periodic sampling already grants resilience to transient faults. An undetected fault affecting a single message sample is unlikely to cause a system failure (in the example vehicle, a single fault affecting a sample of the throttle input cannot cause the vehicle to drastically change speed). It may cause some vibration, slight delay in updating control outputs, or less smooth control.

From a fault tolerance point of view, if many input samples suffer undetected errors within a short period of time, then an unsafe event might occur. Thus, a system design must ensure that so many undetected errors have an very low probability of occurring together. Embedded network protocols include an error detection code in each packet to prevent this. Similarly, from a security point of view, if an attacker might cause a system to enter an unsafe state by forging a number of message samples within a short period of time, then the design must ensure that enough forgeries cannot occur without very high probability of being detected. Our approach includes MAC tags in each packet to detect such masquerade attacks.

Because of this over-sampling, senders can authenticate state changes and actuations over multiple packets using truncated MAC tags. All multicast authentication techniques in this work use hash based MACs. The sender computes MAC tags for the packet as defined by the selected multicast authentication mechanism. Then, the sender truncates each MAC tag to just a few bits before appending tags to the data payload.

Time-triggered authentication requires that the outputs of the MAC functions can be truncated down to an arbitrarily few number of bits without compromising the security of the function or the key. Only MAC functions that meet this requirement (e.g., hash based MAC functions) can be used for time-triggered authentication. MAC functions that do not meet this criteria should not be used.

To reduce the rate at which masquerade attacks induce system failures, nodes verify state changes and actuations over multiple time-triggered packets, each containing a truncated authenticator (Figure 3.1). Nodes execute state-changes after receiving a sufficient number of packets containing consistent values, each of which would trigger the same state change. Reactive control inputs are applied to actuators as they are received, relying on system inertia to force an attacker to forge multiple packets within a short period of time to place the system in an unsafe state.

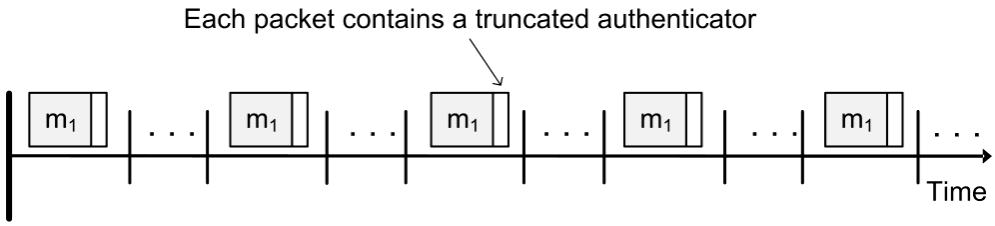

Figure 3.1. Time-triggered authentication. This approach verifies state changes and actuations across multiple truncated authenticators.

The primary advantage of time-triggered authentication is that the system designer can perform a tradeoff among authentication bits per packet, application level latency for state changes and physical actuations, and the acceptable probability of induced system failure for each message type. Typical requirements for acceptable failure rates in systems containing wired embedded networks might be defined at 10^{-3}/hr, 10^{-6}/hr, or 10^{-9}/hr of undetected message errors de-

pending on the severity of the resulting failure. The system designer can determine the number of authentication bits required per packet such that a successful masquerade attack can induce a malicious system failure no more often than the failure requirements for sources of other non-malicious failures.

A secondary advantage of time-triggered authentication is that it can be combined with many multicast authentication techniques that use MACs to validate packets during run-time. The MAC tags in such approaches can be truncated to an arbitrarily small number of bits without compromising the security of the underlying functions or keys. Further, being able to combine time-triggered authentication with other multicast authentication techniques enables additional tradeoffs amongst multicast authentication techniques. This work discusses tradeoffs amongst different techniques in Section 5.

3.1 Per-packet assurance

This work shows that by verifying state changes and physical actuations over multiple truncated authenticators, time-triggered authentication enables strong system-level assurance (very low probability of maliciously induced failures) that those state changes and actuation commands are correct and from a valid sender despite only having weak assurance that an individual packet contains a valid message sample value.

In time-triggered authentication, the degree to which an individual authenticator can be truncated depends on the required level of per-packet assurance. We define *per-packet assurance level* as the acceptable probability of successful forgery per packet. A *weak* per-packet assurance level gives an attacker a high probability of successfully forging each packet. Using a *strong* per-

packet assurance level creates a low probability of successful packet forgery. Achieving a stronger assurance level requires more authentication bits.

A system with sampling rates faster than the physical dynamics of the system (e.g., a typical time-triggered embedded control network) generally tolerates weaker per-packet assurance levels than a system that sends infrequent periodic samples or a single sample for some change in system state (e.g., an event-triggered system). In systems with high sampling rates, each packet has less net effect on the overall system state, requiring many successful packet forgeries to induce a system failure. However, in systems with low sampling rates or event-triggered systems, an attacker might induce a system failure with a single (or very few) packets. A more severe failure induced by a successful masquerade attack against a particular message type requires a receiver to authenticate across more samples or have stronger assurance of each sample.

Figure 3.2 shows the required per-packet assurance probability as we vary the number of samples an attacker must successfully forge consecutively to induce a system failure with probability no higher than 10^{-9} per message round.

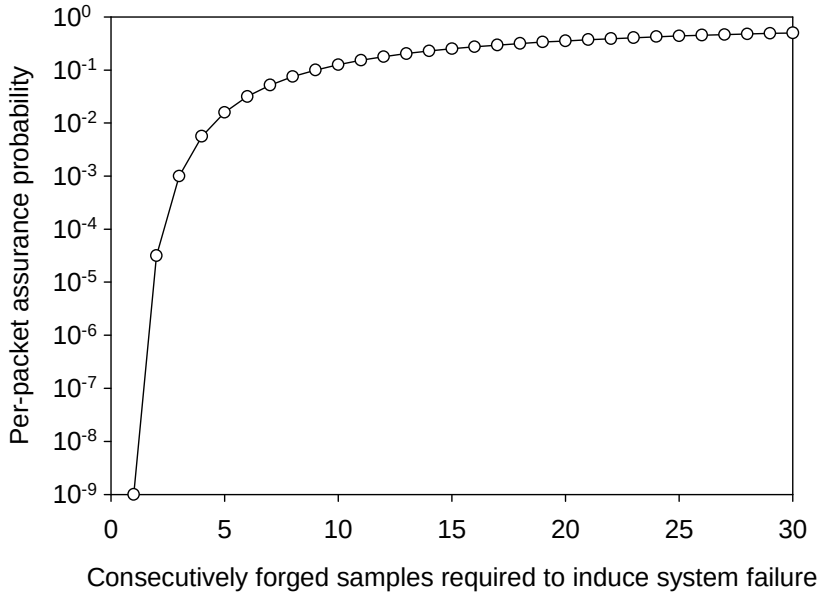

Figure 3.2. Per-packet assurance defined by forged samples required to induce system failure. Per-packet assurance probability required to prevent system failure with probability no higher than 10^{-9} per message round, varying the number of successfully forged samples required to induce the system failure.

We emphasize that the failure probability in Figure 3.2 is *per message round*. To achieve failure rates on a per-hour basis, a system designer must determine how many authentication bits are needed to achieve a sufficiently low expected failure rate per message round, taking into consideration the period of a particular message type.

3.2 Time-triggered authentication assumptions

Time-triggered authentication relies on multiple assumptions. This work assumes the following:

- The sampling rates of message types are sufficiently faster than the physical dynamics of the system, such that an individual message sample only requires a weak level of per-packet assurance. Packets are transmitted at a rate fast enough for a receiver to authenticate multiple consistent values for a message type within a system deadline or rise time of a system. In

Section 5, we examine which multicast authentication techniques scale best when individual packets require stronger per-packet assurance.

- A certification authority exists to assign key material to components when they are manufactured.
- Nodes use existing cryptographic one-way hash functions (e.g., SHA-1 [FIPS 180-3], MD5 [Rivest92], or SHA-256 [FIPS 180-3]) and MAC functions to implement authentication (e.g., HMAC [Krawczyk97]). We assume the underlying cryptographic primitives are secure. We do not rely on specific MAC or one-way hash functions to implement our scheme.
- The outputs of selected MAC functions can be truncated to an arbitrarily small number of bits without compromising the security of the MAC function, underlying hash function, or any key material.
- The output lengths of MAC functions and sizes of keys are fixed at design time and cannot change at run-time.
- The network configuration is fixed at design time; nodes are not installed or uninstalled on the fly. A message schedule exists so all nodes are aware of the set of message types broadcast by each node. The set of receivers for each message type is also known by all nodes.
- Nodes remain synchronized to the nearest message round.

We list other assumptions necessary for individual multicast authentication techniques in their respective sections below.

3.3 Using one MAC per receiver (OMPR) for time-triggered authentication

This section describes how to combine One MAC per Receiver (OMPR) into time-triggered authentication. OMPR is one of the most straightforward methods for multicast authentication; the

transmitter simply computes and sends one full-size MAC tag for each receiver along with the message. Time-triggered authentication allows us to scale the tag size based on the required per-packet assurance on a per-receiver and per-message type basis. First, this section states our assumptions. Then it discusses key initialization and replay protection. It shows how to sign and verify individual message values at run-time. Finally, it describes how to verify a series of individual message values, each with weak per-packet assurance, to provide strong assurance for state changes and actuations.

When using OMPR, a sender computes one MAC tag for each receiver and truncates each to a few bits. By using tags only a few bits in size, the sender can place one tag per receiver in the data payload of a packet. This approach allows authentication on a per-packet basis (batch authenticating multiple payloads is not required), has perfect loss tolerance, and perfect tolerance to compromised nodes. However, bandwidth requirements scale linearly as per-packet assurance and number of receivers increase.

3.3.1 OMPR Assumptions

When using OMPR, this work uses two assumptions in addition to those for time-triggered authentication in Section 3.2:

- Each sender has sufficient computational resources to compute one MAC per receiver per message value that is sent. The required computational resources depend on the cryptographic function used.
- The number of available bits in a packet's data payload is greater than the number of receivers of a packet. This allows authenticators for each receiver in the packet, leaving room for the message value.

3.3.2 Initialization

Key establishment - To prevent one node from masquerading as another, the set of nodes attached to the network must first established pair-wise shared secret keys. Keys are set up at initial installation or node replacement for maintenance. Any secure method of key establishment can be used. Maintenance or factory personnel can program each node with the respective shared keys when the node is installed into the system. This method might not be ideal since it requires additional work by personnel to establish the keys, and places a large amount of trust in these personnel. Alternately, another approach is to provide each node with a public and private Diffie-Hellman [Diffie76] key pair, which has been digitally signed by the manufacturer's secret key. Each node also has the manufacturer's public key. At time of installation, the nodes exchange their Diffie-Hellman public keys and certificates. Each pair of nodes then authenticates the certificates and uses the Diffie-Hellman key exchange protocol to compute a shared secret key for authentication. In a typical embedded system, all nodes wired to the network are known at design time. It is reasonable to assume a node will know the standard configuration and what nodes comprise the group it is communicating with. This is in contrast to enterprise networks, where network nodes are expected to change continually.

For a system with n nodes, this scheme might require establishing $O(n^2)$ keys. While this overhead is high, it is incurred only once at time of installation, while the system is inactive. Embedded networks have very stable hardware configurations, which often last for months or years. Thus, a one-time key distribution cost is a minor concern in most situations. Keys are stored as part of configuration data and do not change at run time. We assume system designs use an appropriately secure key length (e.g., 80 bits) [Lenstra01].

Time synchronization - Time-triggered authentication uses time synchronization to prevent replay attacks. At system startup, each pair of communicating nodes securely synchronizes to a common time base. Nodes agree upon the current time or TDMA round number using a protocol such as Secure Pair-wise Synchronization [Ganeriwal05]. This can provide synchronization on the order of microseconds to ensure freshness of messages for each message round, which can be tens to hundreds of milliseconds. Global synchronization is not needed, since only pairs of nodes share each secret key. For each packet to be broadcast, the sender includes the current time or TDMA round number as an input to any cryptographically secure MAC function used (depending on the multicast authentication technique being used). Synchronized time values must not roll over for some acceptably long period of time. This prevents the attacker from predicting the MACs over this period of time even for identical data values via playing back previously recorded messages. Because the MAC function compresses data, there is no limit on the size of the time value.

3.3.3 Producing a per-packet authenticator

When transmitting a message, the sender generates one MAC tag for each distinct receiver of the packet. The sender computes each MAC function over the packet header, message value, and synchronized time, using the appropriate pair-wise shared key for the corresponding receiver. The outputs of these MAC tags are then truncated to just a few bits each, and the sender appends the truncated MACs to the message value (Figure 3.3). Depending on the required per-packet assurance for the message type, the size of each truncated MAC tag can be as little as a single bit. By truncating tags to just a few bits, one MAC per receiver can be placed into each packet. All authentication data can be self-contained in each packet, given that at least one bit is availa-

ble per receiver. This allows each packet to be verified independently and ensures that lost packets do not affect the verification of any other packet.

Since the network configuration is fixed at design time, the location of each receiver's MAC tag within a data payload can be assigned at design time. Receivers are preprogrammed with the size and location of their respective MAC tags.

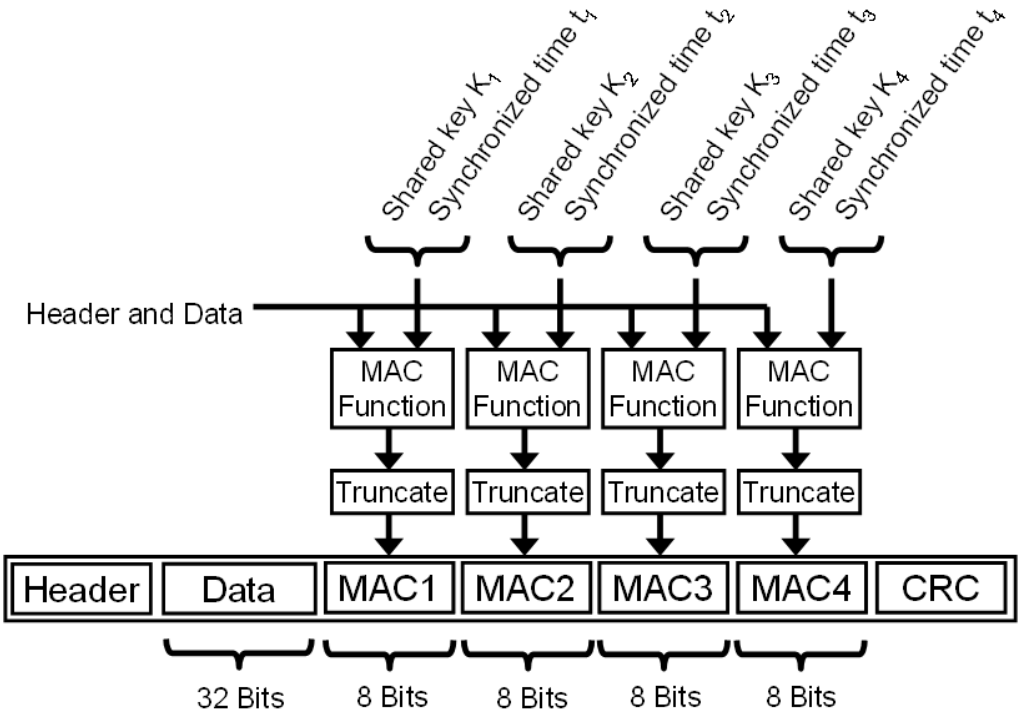

Figure 3.3. OMPR multicast authenticator generation. Example packet containing 32 bits of data and four 8-bit MACs, for four receivers. Each receiver *n* shares a secret key K_n and synchronized time t_n with the sender. These values are included as inputs to the MAC function along with the header and data. The outputs of the MAC function are truncated and appended to the data payload.

System designers select the number of MAC tag bits to use for each receiver at design time. The size of these outputs do not change during run-time. MAC tags do not necessarily need to be

truncated to the same number of bits in length. Different receivers may also have higher or lower priority for message assurances. For example, some receivers might need stronger assurances within shorter deadlines than other receivers. A sender can devote more MAC tag bits in the payload for those receivers with more strenuous requirements for security.

In the case that a message type's required per-packet assurance does not allow one truncated MAC tag per receiver to be placed in a single packet's payload (i.e., the size of the data value and truncated MAC tags exceed the size of a packet's payload), these MAC tags can be placed in a subsequent packet. However, this increases the delay for receivers to verify a message sample and decreases the loss tolerance of this approach.

3.3.4 Verifying a packet

Upon receiving any packet (or packets) containing a message value, a receiver first checks that the transmitted packet is well formed according the embedded network protocol and checks the error detection code. Then, if the packet is not malformed and the error detection code is correct, the receiver verifies its designated MAC tag. The receiver recomputes a MAC function over the same values the sender: packet header, message value, pair-wise and synchronized time, using the appropriate pair-wise shared key. The receiver then compares the output tag of the MAC function to the receiver's designated tag within the packet's payload.

Receiving a packet and verifying its message value has one of three results:

Lost - A message value is considered to be "lost" if the error detection code of the packet is incorrect, the packet is malformed according to the embedded network protocol, or if no packet is transmitted during a particular message period. This indicates that some error occurred during transmission. This result encompasses most non-malicious transmission errors. Dropping packets

does not grant an attacker any extra benefit while attempting to forge messages. This work does not address how to deal with malicious denial of service attacks, and assumes receivers take appropriate action in the event of observing a significant number of dropped messages.

Valid - If a message value is not lost and the recomputed tag matches the receiver's designated tag in the packet's payload, the receiver accepts the message value in that packet as "valid." The receiver trusts that the message value is indeed from the correct sender and the value has not been tampered with during transmission.

Invalid - If a message value is not lost, and the tag does not verify as valid, then a message value is designated as "invalid." This indicates that the message value might be a forgery attempt, or might have a transmission error undetectable by the error detection mechanisms in place.

By tampering with network traffic to inject or modify a message value, the attacker might occasionally succeed in forging a MAC tag. If the packet contains *b* MAC tag bits for a receiver using OMPR, the probability that any single MAC tag can be successfully forged is 2^{-b}. If an attacker correctly guesses the tag for the corresponding message value, then the receiver will observe a valid MAC tag.

3.3.5 Delayed or out of order messages

Timing delays may cause a message to be designated as invalid if a receiver uses a different time input when verifying a MAC tag than the sender used in computing the MAC tag in the payload. We assume nodes remain time synchronized to the nearest message round. However, in some cases, a message broadcast may be delayed (e.g., in CAN a low priority message may be delayed by a higher priority message).

If there are well-defined time boundaries for message rounds, two techniques can prevent a message from being accidentally designated as invalid. One possibility is for a receiver to try multiple synchronized time values (e.g., current and previous message round numbers). However, this increases the probability that an attacker could correctly guess a MAC tag (requiring more MAC tag bits). Instead, a sender can include the least significant bit of the message round number in the data payload. Thus, a receiver can identify a delayed message from a previous message round.

3.4 Verifying state changing messages

Time-triggered authentication provides strong assurance for state-changes by authenticating over a set of message values, each of which have weak per-packet assurance. A receiving node keeps an *explicit history buffer* for the authentication results of each message type used in its internal state machines. A history buffer acts like a First In First Out (FIFO) buffer in which receivers store the *n* most recent message values and the verification results for each sample ("valid" or "invalid"). At startup, nodes initialize history buffers so that all elements are set to a default value and stored as invalid.

Receivers verify each message value individually using the process described in Section 3.3.4. Lost message values are discarded and are not included in the buffer. Once verified as valid or invalid, a receiver discards the oldest value in the history buffer, shifts all values by one index position, and stores the newest value and its validity.

Upon checking and storing the verification results of a newly received message value, a receiving node checks if the contents of the history buffer satisfy the conditions to commit to a state change, as defined by its internal state machine. A node commits to a state change if a his-

tory buffer contains a sufficient number of valid message value samples that are all consistent. A set of values for a message type is *consistent* if all valid values would trigger the same state transition (the values do not necessarily need to be equal). In the case that a transition depends on multiple message types, the receiver would wait until all history buffers for those message types satisfy the condition for the state transition.

All values within the history buffer must be consistent for a state transition to be taken. If one of the values is not consistent, a state transition cannot occur.

Once a node commits a state change, the node clears its history buffers and resets them to default values stored as invalid.

For example, a node that controls a door lock in an automotive network might monitor the wheel speed message type, and automatically lock the door if the car is moving sufficiently fast. If the speed threshold is set at fifteen miles per hour, the door lock node would record each received message sample value and its validity in the history buffer. Once a sufficient number of wheel speed message values in the history buffer are valid and are all at least fifteen miles per hour, the node would commit to the transition and locks the door.

No tolerance for invalid MAC tags - Depending on the application, the system designer decides how many samples in the history buffer must be valid before committing to a state transition. In most applications, a receiver waits for *n* out of *n* consecutive values in the history buffer to be consistent and then commits to this transition. Committing to state changes after *n* of *n* consecutive valid message values assumes the application does not require any tolerance to invalid message values or that any single invalid message value indicates a malicious masquerade attack. If any of the *n* values were invalid, the state transition does not occur. Thus, in the event of a single invalid message sample, a state transition cannot occur until another *n* subse-

quent valid samples have arrived.

While it is likely that an attacker will be able to forge a single packet since we use just a few authentication bits per MAC, it is unlikely that they will be able to forge so many within the history of the buffer as to cause a successful masquerade attack, subsequently maliciously inducing a state change. Thus, this approach allows receivers to verify many message samples using weak per-packet assurance to achieve strong system-level assurance. If each message value is transmitted along with *b* authentication bits per receiver, the probability of per-packet forgery P_p is 2^{-b}. The probability of forging *n* consecutive message values in a history buffer is:

$$P_A = (P_P)^n \qquad (1)$$

Tolerating invalid MAC tags - Optionally, it may also be useful for some applications to have some level of tolerance to invalid message values. Allowing state changes to occur after validating a subset of MAC tags in the history buffer grants this approach a degree of tolerance to interspersed invalid MAC tags. Without this tolerance, an attacker might increase message latency or prevent authentication altogether while remaining undetected by occasionally injecting invalid packets. Packets with a correct CRC but invalid MAC might also be caused by non-malicious faults. For example, if the sender's and receiver's notions of time differ due to a temporary internal fault, the receiver would see an invalid MAC. Additionally, some message corruptions might be missed by error detection mechanisms, so occasional invalid MAC tags might result from transmission errors.

When tolerating interspersed invalid MAC tags, a state change occurs when at least *k* out of the past *n* time-triggered message values in the history buffer are consistent and valid. This allows a receiver to tolerate *n* - *k* invalid MAC tags interspersed within a series of *n* message val-

ues. State changes occur as soon as *k* message values out of the most *n* most recent have consistent values and are valid. An attacker can successfully forge at least *k* of a set *n* values in a history buffer with a binomial probability of:

$$P_A = \sum_{i=k}^{n} \binom{n}{i} (P_p)^i (1 - P_p)^{n-i} \qquad (2)$$

We emphasize that all message values in the history buffer (including the invalid ones to be tolerated) are all consistent.

Tradeoffs for state changing message verification - This approach for authenticating state-changing messages enables the system designer to perform a tradeoff among authentication bits per packet, application level latency, tolerance to invalid MACs and probability of an induced failure. Based upon the criticality of the message, the designer trades increased authentication bandwidth and latency for lower probability of induced failure, and trades increased tolerance to invalid MACs for increased probability of induced failure.

Additionally, system characteristics and requirements might constrain these tradeoffs. For example, in a system with hard real-time deadlines, the maximum number of samples to authenticate over might be limited to the minimum number of samples of a message type expected to contain consistent message values within the maximum tolerated delay for a state change. The number of samples might be further constrained if extra slack is needed to tolerate unexpected operating conditions such as lost packets. Adding slack for unexpected operating conditions means that there would be fewer message samples to authenticate over, decreasing the possible size of the history buffer. To authenticate over fewer samples, a system designer could increase the number of bits per MAC tag, reduce the number of invalid MAC tags to tolerate, or even ad-

just the permissible overall probability of maliciously induced failure.

Effects of lost packets and message blackouts - Each individual lost packet will cause a single message round delay before a state change can occur. In the event that the contents of a history buffer becomes stale and no longer accurately reflect the current state of the system due to a large number of consecutive packet losses (e.g., during a network blackout), a receiver can reset the contents of the history buffer and declare its contents as invalid. This work assumes that a receiver takes an appropriately safe action if it detects a network failure due to a significant number of lost packets.

3.5 Verifying reactive control messages

The verification process for reactive control messages takes advantage of the characteristic that the sampling rates of messages are much faster than the physical dynamics of the system, enabling the use of weak per-packet assurance to provide strong system level assurance against undesired actuations. Unlike state-changing message verification, nodes running feedback control loops verify and act upon each message packet as it arrives. Each correctly formed and valid message causes a controller to update its output to a physical actuator. This output in turn causes some physical change in an actuator output. However, because messages are sampled much faster than the step response time there is a damped physical response to any single message value.

For reactive control messages, the receiver does not explicitly retain an authentication history buffer in memory, but relies instead upon a damped response to messages. The system state may be forced to an unsafe value in some situation if the controller accepts too many successfully forged packets commanding the actuator to some position or action within a period of time. But, the damped response to messages requires an adversary to successfully forge multiple packets

within that period of time to compromise system operation. This creates an *implicit history buffer*, using the physical inertia in a system. The physical position or motion of actuators reflects the cumulative effects of the most recent valid message values that have been applied to the system.

Receivers verify each message value individually using the process described in Section 3.3.4. If a message value is valid, the receiver applies it as an input to the reactive control loop. If a message value is invalid or lost, the receiver applies a *safe input* to its internal control loop. What constitutes a safe input depends on the application, but when applied to a controller should not violate safety requirements (i.e., harming users, equipment, or property). Examples of safe actions might be:

- Completely cease actuator movement.
- Return actuator to a safe position.
- Use a default value that partially moves an actuator towards a safe position.
- Use a default value that does not cause the system to exert additional energy into environment.
- Ignore the lost or invalid value, and use the previous valid value, assuming correct messages will resume shortly.

Further, a safe action upon observing a lost packet is likely to be different than the safe action for invalid message values. Lost packets may be considered the results of a non-malicious fault, allowing a receiver to ignore the lost value and use the previous valid message value. However, invalid MAC tags might be considered specifically malicious. Thus, a receiver might instead actively counter the observed forgery attempt, moving an actuator to a safe position or stopping the system.

For example, consider a door controller for an elevator. When a passenger enters the elevator car and pushes a button for another floor, the doors should close. However, if the door reversal sensors detect anything in the way of the door (usually a passenger), then the doors should reopen. Similarly, if a transmission error occurs during the message carrying the door reversal sensor values, one safe thing to do is to reopen the doors. During each execution of the door controller's control loop, the door controller updates its output to the door motors, indicating whether the doors should continue closing, stop moving, or open. After each execution of the door controller's control loop, the door motors can only close a fraction of the way if the door reversal sensors indicate the doorway is clear. An unsafe situation might occur if an attacker continually spoofs the door sensor message on the network to indicate the doorway is clear, despite a person being in the way. If the attacker can successfully forge a sufficient number of door reversal sensor messages to contain a value indicating "Door way is clear," the door might not reopen and crush a passenger. For this particular example, if a door controller observes a single packet with an invalid MAC tag, a safe input to the door controller is to reopen all the way. However, for lost messages, the door controller might ignore the first few lost message values before deciding to reopen the doors. Reopening the doors does not exert energy into the environment that could injure passengers. Resetting the door to a known safe position also effectively forces an attacker to start over with their forgery attempts.

No tolerance for invalid MAC tags - First, this section considers applications in which invalid MAC tags should never occur except in the event of a malicious attack (i.e., non-malicious faults cannot cause an invalid MAC tag to be produced). The receiving controller assumes a single invalid MAC tag indicates a malicious attack and attempts to place the system into a safer state. Upon observing even a single invalid MAC tag, the receiving controller aborts any updates to a

physical actuator based on incoming message values, and instead uses a default action to cause a physical actuator to cease all movement or return to a safe position. When the controller takes this safe action, the physical effects of any successfully forged message samples do not persist in the integrated system state and any attempts at forcing an undesired actuation must start over.

The system designer defines the maximum duration that a receiving controller can tolerate arbitrary input values for a single message type before the system enters an unsafe state. For a maximum duration consisting of *n* message periods, an attacker must successfully forge *n* consecutive message values for that message type to succeed in an undetected masquerade attack. The system designer then selects the appropriate number of authentication bits per receiver such that the probability of an induced failure is sufficiently low. The probability of a successful forgery for any individual message sample containing *b* MAC tag bits to a particular receiver is equal to 2^{-b}. Again, this approach only uses a few bits per receiver. While this only provides weak per-packet assurance, each successfully forged message will only cause some increment of physical change produced by the receiving node. If a successful masquerade attack requires an attacker to forge n consecutive MAC tags, each containing *b* bit MAC tags per receiver, the probability of an attacker succeeding per message round is bounded by equation (1) in Section 3.4.

Tolerating invalid MAC tags - In some applications, continuing operation despite occasional invalid authenticators might be preferable to stopping the system and resetting it to a known safe state. As with state-changing messages, occasional invalid authenticators might occur due to non-malicious errors, such as transient time synchronization issues or network errors missed by error detection codes. A receiving controller might continue operation despite seeing one or more message values with an invalid authenticator. Then, if the receiver detects too many invalid tags

within a period of time, the receiver decides that a malicious attack is underway and acts accordingly.

For each invalid MAC tag to be ignored, the receiver still takes some safe default action temporarily. This might be to just reuse the most recent valid message value or output a default safe value to the actuator.

During an actual masquerade attack, this tolerance might effectively grant an attacker a few extra "free" tries to induce a system failure. When tolerating invalid tags in our approach, at least *k* of the *n* most recent message samples must have valid authenticators. A receiver will tolerate up to *n* - *k* invalid MAC tags, before declaring an attack is occurring and taking an appropriately safe action to deny further attack opportunities (such as ceasing actuator motion or moving to a safe position). An attacker can successfully forge at least *k* of a set *n* values with a binomial probability given by equation (2) in Section 3.4.

Tradeoffs for reactive control message verification - Like verification for state-changing messages, this approach enables multiple tradeoffs. A system designer can tradeoff among authentication bits per packet, duration before an attack should be detected, tolerance to invalid MAC tags, and probability of an induced failure. Based upon the criticality of the message, the designer trades increased authentication bandwidth for lower probability of failure. Selecting a longer duration before an attack should be detected also lowers the probability of induced failure. The system designer can also trade increased tolerance to invalid MACs for increased probability of induced failure.

3.6 Experimental analysis

In this section we discuss characteristics of our approach and experimental results of simulated attacks. The results of this section are intended as a "sanity check" to confirm the probability equations used in Sections 3.4 and 3.5.

Per our attacker model, an attacker may insert or modify packets in valid time intervals for a particular message type. Computing the MAC over the pair-wise synchronized time or TDMA round number ensures freshness of messages. At best, an attacker may only inject a packet once per message round. To be conservative in our analysis, the attacker performs masquerade attempts against a single isolated receiver, so an attacker only needs to guess one truncated MAC per packet.

We have experimentally confirmed the probability of successful forgery attacks against our approach using a software simulation written in C. In our simulation, an attacker node continually sends packets containing a known message value and randomly generated MAC values to the receiver. The receiver node verifies the packet using HMAC-SHA-256 and retains a history buffer of the *n* most recent authentication results. Once the receiver counts a sufficient number of valid MACs in its history buffer, the simulator records an attack event and the number of attempted forgeries before the successful attack occurred. After a successful attack, the simulator reset to its initial state and began again. We simulated attacks on state-changing and reactive control messages for both authentication of consecutive packets and authentication of a fraction of packets in a history buffer.

For state-changing messages, we created a simple state machine with two states. The receiver begins in the first state. When a sufficient number of values in the history buffer have a consis-

tent value, it triggers a transition to the second state. The second state automatically transitions back to the first state and clears the history buffer. Attacks on state-changing messages were considered to be successful once the attacker forced a state change, and further packet forgeries were applied to the next state change after clearing the history buffer.

For reactive control messages, we modeled a simple open loop system (no feedback). If a packet contained a valid MAC tag, the receiver would increment its output by a constant amount towards the input value. For a packet with an invalid MAC tag, the receiver would decrement its output by the same constant amount (i.e., moving to a safe position by a predefined amount). The system only accepts two inputs (zero and one). In this simple system, the attacker must successfully forge a sufficient number of samples to force the output to an "unsafe state." A successful attack was recorded for each message round the attacker was able to force the output to be an unsafe value. The physical state was not reset when the output reached an unsafe state.

We measured the number of successful attack events over a period of time long enough to record at least one hundred successful attack events per data point. We computed the *successful attack rate* as average successful attack events per message round and compared this rate to the probability of successful attack defined in equations (1) and (2) in Section 3.4. From our results we confirmed that equations (1) and (2) can be used as upper bounds on the probability of successful attacks on our approach. These equations can be used to define the required number of packets and authentication bits per packet to achieve a desired failure rate and tolerance to invalid MACs for the system.

3.6.1 No tolerance for invalid MAC tags

Figure 3.4 shows the simulated successful attack rate on both state-changing and reactive control

message types, using a fixed history buffer size of four packets containing one to six authentication bits per packet. In this experiment, a successful attack was recorded if the four most recent message samples were successfully forged. As more bandwidth is devoted to authentication, the successful attack rate decreases exponentially according to equation (1).

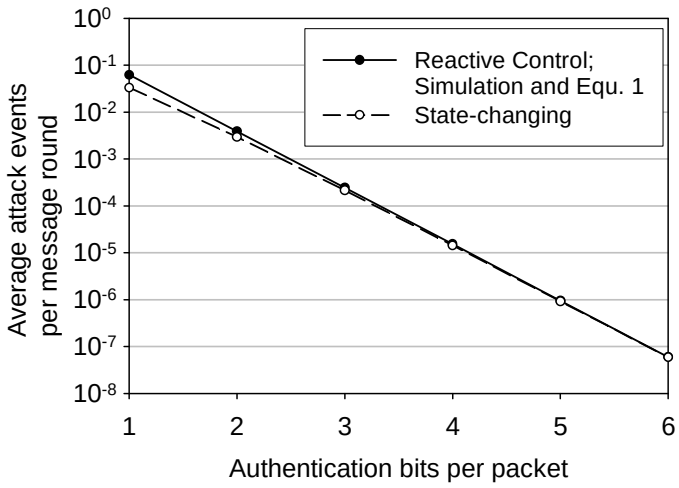

Figure 3.4. OMPR - Simulated successful attack rates for four consecutive messages.

The successful attack rates in Figure 3.4 should be no greater than the probability of successful attack defined by equation (1). As expected, the successful attack rate for reactive control messages matches equation (1) since simulated attacks were counted for any message round the attacker successfully forced the output to its desired position, and the physical state was not reset if this position was reached (the implicit history buffer was not cleared). (Equation (1) is indistinguishable from the simulated reactive control successful attack rate if plotted on Figure 3.4.)

The successful attack rate for state-changing messages is less than the rate for reactive control messages because successful attacks on reactive control messages containing few authentication bits are likely to come in bursts in consecutive message rounds. A forgery attempt on the packet after an initial attack event has a better probability of prolonging the attack in comparison to

forging a full set of *n* packets to initiate a successful attack. The simulated successful attack rate for state-changing messages is less because the history buffer is cleared after each state change.

With more bits per packet, the likelihood of successful attacks occurring on successive reactive control messages decreases, as indicated by the converging rates in Figure 3.4. We use equation (1) as a conservative upper bound on the successful attack rate for both reactive control and state-changing messages.

Typical requirements for acceptable failure rates in systems containing wired embedded networks might be defined at 10^{-3}/hr, 10^{-6}/hr, or 10^{-9}/hr of undetected message errors depending on the severity of the failure. An induced failure from a masquerade attack should occur no more often than the required rate of failure. Figure 3.5 shows the minimum number of messages in the history buffer for a given number of authentication bits per message to achieve an expected successful attack rate of 10^{-3}/hr, 10^{-6}/hr, or 10^{-9}/hr. The number of packets and bits were obtained using the three successful attack rates as expected values for one forgery attempt per millisecond over the course of an hour, each succeeding with probability given by equation (1).

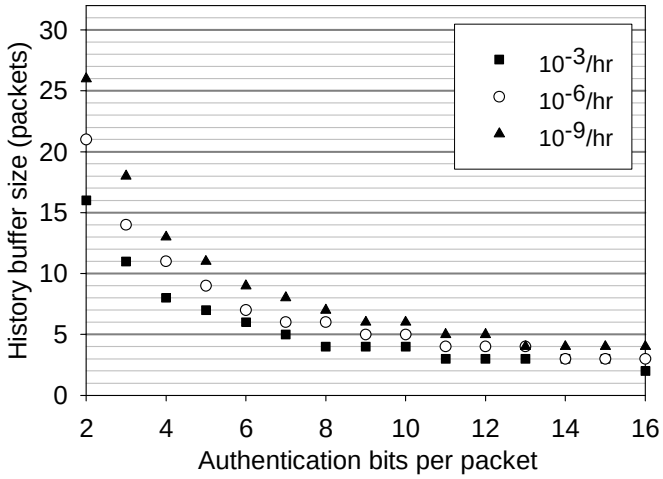

Figure 3.5. Minimum MAC bits per packet and history buffer size (consecutive messages) required to authenticate to failure rates at 1000 packets per second.

3.6.2 Tolerating invalid MAC tags

If we permit interspersed invalid MACs in the authentication history buffer, we gain tolerance to some non-malicious faults and malicious attempts to disrupt authentication of state-changing messages. But increasing this tolerance also increases the probability of an induced failure. Attacks may succeed against some control systems if the attacker forges some fraction of the most recent reactive control messages. As this fraction decreases, the probability of induced failure increases.

Figure 3.6 shows the simulated successful attack rate on state-changing and reactive control message types requiring two successful forgeries out of four packets, each containing one through six authentication bits. As the number of bits per packet for authentication increases, the probability of a successful attack decreases exponentially.

The successful attack rate on reactive control messages in Figure 3.6 matches equation (2) because attack events were counted so long as two of the four most recent message samples were successfully forged, and the output was not reset once this threshold was reached. (Equation (2) is indistinguishable from the simulated reactive control successful attack rate if plotted on Figures 3.6 and 3.7.) The successful attack rate for reactive control messages is greater than that for state-changing messages because successful attacks on reactive control messages can persist as long as the most recent n packets contain k valid MACs. The difference between lines in Figure 3.6 is greater than the difference between lines in Figure 3.4 because there are multiple combinations of successful forgeries in the most recent packets which can cause successful attacks to persist. We do not attempt to provide an equation due to the complexity of the combinations. Rather, we use equation (2) as conservative upper bound for both message types.

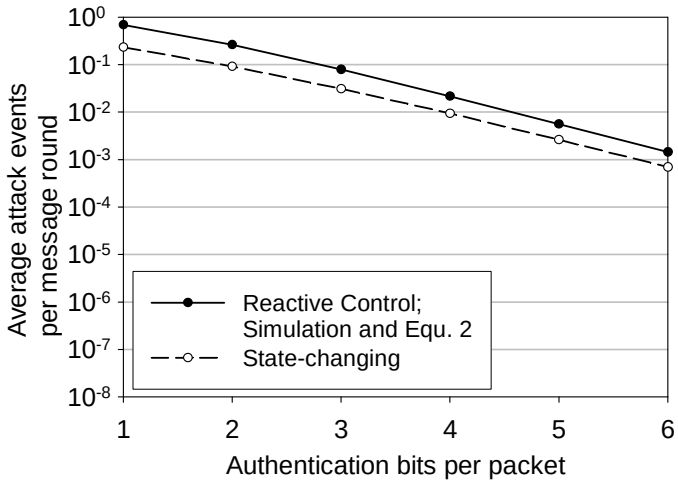

Figure 3.6. Simulated successful attack rate for two out of four messages.

Figure 3.7 illustrates how the difference between simulated successful attack rates for reactive control and state-changing messages changes as the number of required successful forgeries is varied for a buffer of eight packets each containing two authentication bits. With a lower fraction of required valid packets, there are more possible combinations which can cause a successful attack to persist for reactive control message types, causing a greater successful attack rate.

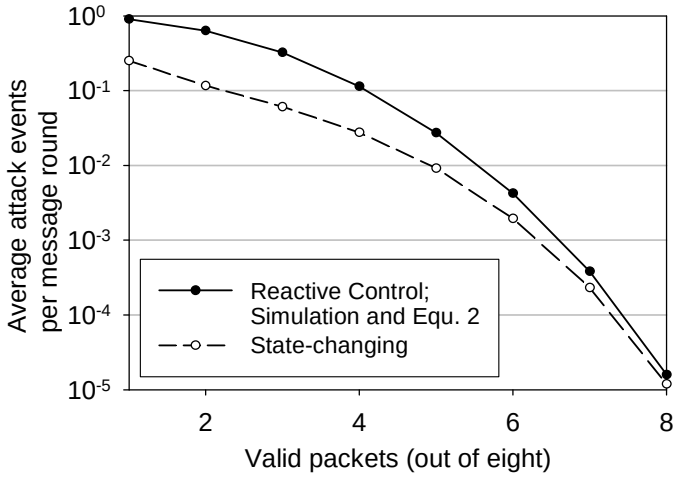

Figure 3.7. Simulated successful attack rates varying fraction of valid packets. History buffer of eight packets with two authentication bits each.

Figure 3.8 illustrates tradeoffs between history buffer size and authentication bits per packet

needed for expected successful attack rates of 10^{-3}/hr, 10^{-6}/hr, or 10^{-9}/hr, requiring all but two valid MACs. The number of packets and bits were obtained using the three successful attack rates as expected values for one forgery attempt per millisecond over the course of an hour, each succeeding with probability of equation (2). For example, with four authentication bits per message, if all packets in a history buffer must be valid, the history buffer must include at least the last eleven packets to authenticate to 10^{-6}/hr (Figure 3.5). If all but two packets must be forged in the history buffer, then the history buffer must include the past fifteen packets, in which thirteen must be valid (Figure 3.8).

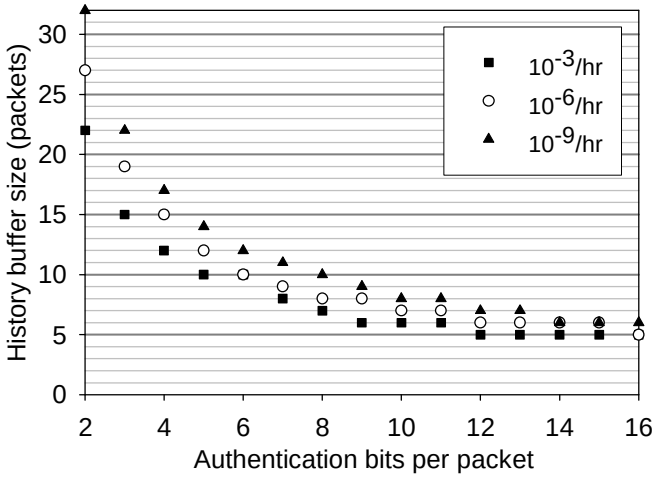

Figure 3.8. Minimum MAC bits per message and history buffer size required to authenticate to failure rates at 1000 messages per second given two invalid packets in the buffer.

3.7 Discussion

This chapter introduces time-triggered authentication. This approach takes advantage of existing temporal redundancy present in most time-triggered system designs to amortize authentication bandwidth overhead across multiple time-triggered packets. We illustrate how time-triggered authentication can provide strong assurance for state changes and actuations by verifying multiple packets, each with weak per-packet assurance.

This new approach has several advantages. Time-triggered authentication enables a fine-grained engineering tradeoff among authentication bits per packet, application level latency, tolerance to invalid MAC tags, and probability of maliciously induced failure. It also allows receivers to perform authentication on a per-packet basis (amortizing does not require batch authentication of many samples). This allows receivers to immediately resume authentication after packet losses cease. Time-triggered authentication can also be combined with many multicast authentication approaches that use MACs.

Time-triggered authentication also has several limitations. This approach relies on the periodic broadcasts of message types. Time-triggered authentication only provides advantage to the degree that messages are oversampled. This approach also requires careful handling of packet loss in conjunction with history buffers. For state-changing messages, receivers must monitor for long message blackouts and reset history buffers if the data values and verification results contained within those history buffers become stale. Further, for reactive control messages, receivers must take appropriately safe actions for both packet losses and invalid messages.

In this section, we use OMPR in conjunction with time-triggered authentication. This multicast authentication approach is uncomplicated, requiring a sender to compute one MAC tag for each receiver of a message sample. It also allows for perfect loss tolerance and tolerance to compromised nodes. However, the scalability of OMPR is limited. The processing requirements scale linearly as the number of receivers increase. Further, the bandwidth requirements scale linearly as both the per-packet assurance and number of receivers increase.

4 Validity Voting

This section introduces *validity voting* to build on the approach for time-triggered authentication using OMPR. This method integrates voting techniques to improve the bandwidth efficiency and subsequently reduce the application level latency of time-triggered authentication. While using OMPR is efficient in terms of bandwidth for a very small number of receivers or very weak per-packet assurance, the linear scaling gives poor performance for many receivers or stronger per-packet assurances. Validity voting still uses one MAC per receiver, but using voting allows it to provide stronger per-packet assurances to a larger number of receivers, for a given number of authentication bits per packet.

The main limitation of using one MAC per receiver is that each sender must redundantly transmit one MAC tag for each distinct receiver of a message value. Using one MAC tag per receiver introduces unused redundancy because each receiver only benefits from a single MAC tag. If limited to a few bytes per packet to authenticate to a large number of receivers, a sender may have to amortize authentication over too many packets to meet real-time deadlines.

Validity voting uses this redundancy to force an attacker to forge many MAC tags to fool an entire group of receivers, rather than just forging a single MAC tag to fool the targeted receiver. To force an attacker to do this, a group of receiving nodes takes a unanimous vote on whether message values were valid or not. Once a sender has transmitted a message value and MAC tag to each receiver in the group, the receivers engage in an attestation process. During this attestation process, each node in the group exchanges indications of the received value and its validity with other members of the group.

Figure 4.1 illustrates validity voting among three receivers in a network with four nodes (N_1, N_2, N_3, and N_4). In step 1, N_1 first broadcasts message m_1, authenticating it to nodes N_2, N_3, and N_4. In steps 2-4 receivers then include an indication of whether message m_1 was valid or invalid when they broadcast their respective messages. After step 4 completes, nodes N_2, N_3, and N_4 have received the authenticator from node N_1 and two votes on the authenticity of message m_1.

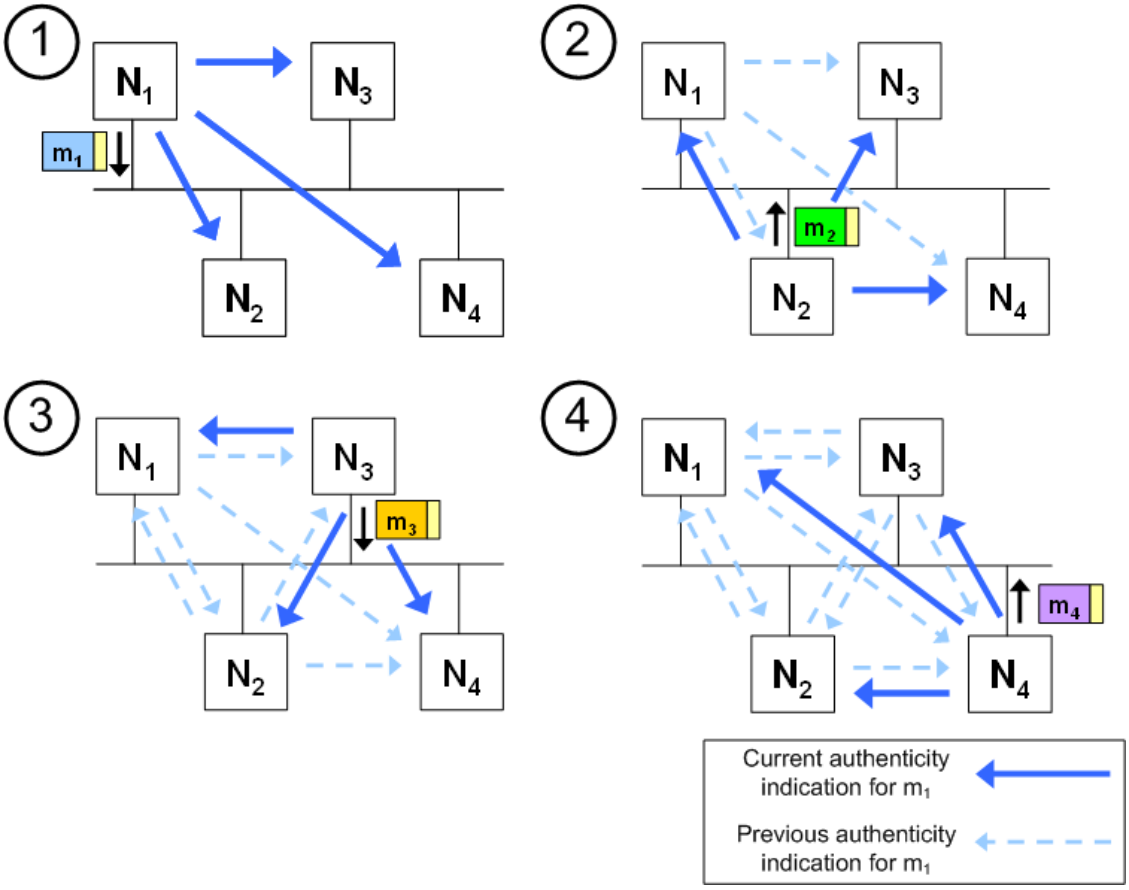

Figure 4.1. Three nodes cross checking message authenticity using validity voting.

To attest to the validity of a particular message value, each member computes its MACs (one for each receiver) over that sender's value in addition to its own transmitted value and an indicator bit to denote the attested value's validity, whether the MAC tag for that message value was valid or invalid. By using this process, voters only need to transmit a single additional bit for

each value they vote upon, "piggy-backing" votes onto messages already scheduled for transmission. They do not need to explicitly retransmit the values being voted upon.

To vote on multiple values at once, a group member uses the same process. Each message can carry multiple validity bits (one for each value being attested to) in a bit-array called a *validity vector*. A validity vector contains one bit for each message value being voted upon (those bits and the corresponding message values are all included as inputs to the MAC functions). The members of the group append this validity vector into the packets they transmit during their designated time periods in the message schedule.

Once all members of a receiving group have transmitted, each member takes a unanimous vote on the validity of the message value from the originating sender, rejecting any value the group disagrees upon or indicates as invalid. Group members accept the originating sender's value if the sender's packet contained a valid MAC tag, all packets attesting to that value also had valid MAC tags, and all attesting packets indicated the previous sender's value was valid. Any disagreement on validity or invalid MAC tags indicate a masquerade attempt, whereas unanimous agreement of validity and no invalid MAC tags indicate no such attempt. Thus, a successful forgery of a single message value requires an attacker to spoof many authenticators and fool an entire group or receivers, rather than just one authenticator when using one MAC per receiver.

4.1 Properties for detecting disagreement

Validity voting uses secure hash based MAC functions to enable voting on message validity and subsequently detect disagreement on message values. Without knowledge of the key, an attacker can at best guess the MAC tags for any message value it injects or modifies. Further, because nodes compute each tag with different keys, successfully forging one MAC tag does not assist

the attacker in forging another tag. Assuming a secure MAC function, each attempt to forge a MAC tag by an attacker succeeds *randomly* and *independently* of any other attempt on another MAC tag.

Since each MAC tag can only be successfully forged randomly and independently of another MAC tag, a receiver can vote on the results of verification of multiple MAC functions computed over the same value. The attestation process in validity voting creates a series of indirect secondary confirmation channels from the sender to each receiver, and from each receiver to all other receivers. Upon completing the attestation process, each node in a receiving group gets one authenticator from the originator of a message value and several subsequent secondary confirmation authenticators from other receivers in the group. By taking a unanimous vote on the validity of these authenticators, this approach significantly reduces the probability of successful forgery.

We also take advantage of the collision resistance of the underlying hash functions so that nodes do not have to explicitly retransmit values being compared using these secondary indirect channels. By computing the MAC function over the current value, any values being attested to, and the validity of those attestations, the MAC tags should only be valid if the sender and receiver agree on the values and validity of all packets.

4.2 Validity voting assumptions

Validity voting makes several assumptions necessary for it to be used. First, all assumptions for time-triggered authentication must hold. Further, all assumptions for using one MAC per receiver must also hold. Validity voting still requires each sender to compute one MAC for each distinct receiver of a message value.

In addition to the previous assumptions, this approach makes the following assumptions:

- If no transmission error or interference occurs during the broadcast of a message, then all receivers observe the exact same bit values transmitted on a broadcast bus by a legitimate sender during that message period. All receivers of a message type should see the same value in an error free transmission. Thus, receivers in a group do not need to explicitly retransmit a message value they are voting upon.
- The number of available bits in a packet's data payload is greater than the number of receivers of a packet plus the number of message values that packet attests to. This allows authenticators for each receiver in the packet and indicators of the validity of observed message values, leaving room for the message value in the packet.
- Any node participating in a vote is already scheduled to transmit its own message type. Validity voting does not require modification of the message schedule to add additional messages to pass along votes. Voting only requires adding a few bits to packets which are already scheduled to be transmitted.
- Only critical nodes engage in voting with each other, and we assume an attacker compromises very few, if any, critical nodes. If an attacker has already compromised one or more critical nodes, then they can likely already trigger a system failure without resorting to spoofing messages. Non-critical nodes may also engage in voting. However, non-critical nodes are less likely to be rigorously secured against compromises and failures that might allow falsified votes. Section 4.9 discusses how to tolerate a small number of compromised voters when implementing validity voting.

4.3 Initialization

Key establishment - Nodes establish key material for validity voting in the same fashion as one MAC per receiver. Each node must establish a pair-wise shared secret key with any node they communicate with.

Replay protection- Similarly, time synchronization is the same for validity voting and one MAC per receiver. Each pair of communicating nodes must be synchronized to the nearest message round to ensure freshness of messages.

Voting schedule - Validity voting also requires a *voting schedule* to be defined at design time. This fixed schedule gives each node a priori knowledge of which message types contain votes for samples of another message type. Thus, nodes know which nodes are voting on a message value and when those votes should arrive according to the network message schedule.

A voting schedule can be created so long as a set of nodes is expected to broadcast at regular intervals as described in Section 2.1.

The system designer defines voters for each message type M_V to be voted upon. Each node that consumes a message type can potentially transmit a vote on that message type. Part of a message schedule often includes the list of nodes which consume each message type. If not immediately available, this list can be reverse engineered from the design. For a node N that receives message type M_V to be selected as a voter, it must meet several requirements:

- Node N must already be scheduled to transmit a message type M_N that is consumed by other receivers of M_V. Creating a new message type to broadcast a single voting bit requires significant bandwidth. If a potential voter already communicates with other consumers of a message type, then a new message type does not need to be created; only a single voting bit

needs to be added to an existing message type. If necessary, votes can also be placed in multiple message types broadcast by N to reach more of the receivers of M_V.

- Message type M_N must have equal or higher criticality than message type M_V. In safety-critical systems, non-critical system components should not be able to induce faults in critical components.
- Message type M_N must be broadcast at a rate equal to or faster than message type M_V. If node N broadcasts M_N slower than M_V, then there are samples of M_V that will not be voted upon by M_N (e.g., if M_V is broadcast every ten milliseconds and M_N is broadcast every hundred milliseconds, then there are nine samples of M_V that cannot be voted upon). Because there are samples that cannot be trusted, this effectively reduces the sampling rate of M_V. Conversely, if the voting message type M_N is broadcast more frequently than M_V, then there will be more than one sample of M_N that vote on the same sample of M_V. This adds unnecessary voting bits. The greater the disparity in sampling rates, the less bandwidth efficient validity voting becomes. Ideally, using message types that are broadcast at the same rate provides the most efficient use of bandwidth for voting.

Assigning the best placement for votes in the schedule is somewhat subjective. There might be multiple options for assigning votes into various message types. The "best fit" depends on the application and the tradeoffs associated with validity voting. This work uses three heuristics when selecting votes to help maximize the bandwidth efficiency of validity voting:

- Minimize the number of message types transmitted by a single node carrying votes for M_V.
- Maximize the number of receivers of M_V that receive votes.
- Use message types with periods as similar as possible to that of M_V.

Chapters 6 and 7 illustrates the use of these requirements and heuristics when applying votes to a elevator network workload and industry automotive network workload.

When completed, the voting schedule contains the following information for each message type M_V being voted upon:

- A lookup table entry that lists the message types which carry votes for M_V.
- Which bit of a validity vector in a packet of a voting message type is assigned to M_V.
- Any time offsets for a receiver to locate votes for a particular sample (e.g., the votes for a sample of M_V might be assigned to be broadcast within the same message round or a subsequent message round).
- The ordering of message values a receiver is to recompute MACs over for verification.

When concluded, each node in the system is programmed with this voting schedule. Using this schedule, when a node receives a message value on the network, that node knows which future messages carry votes on that message value's validity and when they are scheduled to arrive. Thus, there is a fixed delay before each message value can be completely verified. Conversely, if the messages carrying the votes do not arrive in the expected time periods, a receiver can take appropriate action based on the lost packets.

4.4 Functions and state variables

To check for discrepancies in packet value or validity, each node *n* maintains three state vectors: a *value vector* R_n, *validity vector* V_n, and *confirmation vector* C_n. We use a subscript to denote the identity of the node that produces a variable (e.g., R_n is the value vector produced by node *n*). Nodes initialize all vectors to default values (e.g., zeroes).

The value vector R_n stores the most recently received value (valid or not) for each message type defined in the message schedule that node n consumes or participates in voting on. Receivers record lost packets as a predefined error code 'lost' if they detect a transmission error (indicated by an incorrect error checking code or no packet broadcast in a message period).

The validity vector V_n contains the authentication results of each entry in the value vector. A node stores a '1' value if the most recent value for the corresponding message type was valid and a '0' value if invalid.

Finally, the confirmation vector C_n contains an array of counters for positive secondary confirmations of validity. C_n contains one counter for each message type in R_n.

We first define a function to look up which message types are voted on by a received message type M in the voting schedule. The function *getMessageTypesVotedOn*(M) looks up message type M in the voting schedule and produces a vector *ids*, which contains the indices of R_n, V_n, and C_n that correspond to the message type ID numbers voted on by M.

We define a function *getMostRecent*(*ids*, R_n, V_n, C_n) to obtain a subset of received values, their validity, and confirmations. The indices in *ids* indicate which message types to look up in R_n, V_n, and C_n. This function produces a triple $<r_n, v_n, c_n>$ of vectors; where r_n is an ordered subset of R_n containing $|ids|$ values recently received by node n, v_n is a subset of V_n containing the validity bits for each element in r_n, and c_n is a subset of C_n containing confirmation counters for each element in r_n. The order of values in r_n, v_n, and c_n is the same as the indices of *ids*. Two nodes executing *getMostRecent* during the same message round should obtain the set of message types, because they share the same message schedule and should have received the same set of message samples on the broadcast bus.

The function *setNewest*(*M*, *msg*, *validity*, R_n, V_n, C_n) replaces the element of R_n for the message type *M* broadcast in the current message period with value *msg*. The corresponding element in V_n is set to '1' if *validity* is 'valid', and '0' if *validity* is 'invalid'. The corresponding element in C_n is set to zero.

The functions *updateValidity*(*ids*, v_n, V_n) and *updateConfirmations*(*ids*, c_n, C_n) overwrite the |*ids*| elements in V_n or C_n with the elements of v_n or c_n respectively, using the indices of *ids*.

4.5 Run-time verification

4.5.1 Producing a per-packet authenticator

Validity voting modifies the sending process for time-triggered packets (Section 3.3) to allow senders to attest to the validity of the most recently received message value samples of a set of message types (as defined by the voting schedule) in addition to authenticating the current message value (Figure 4.2).

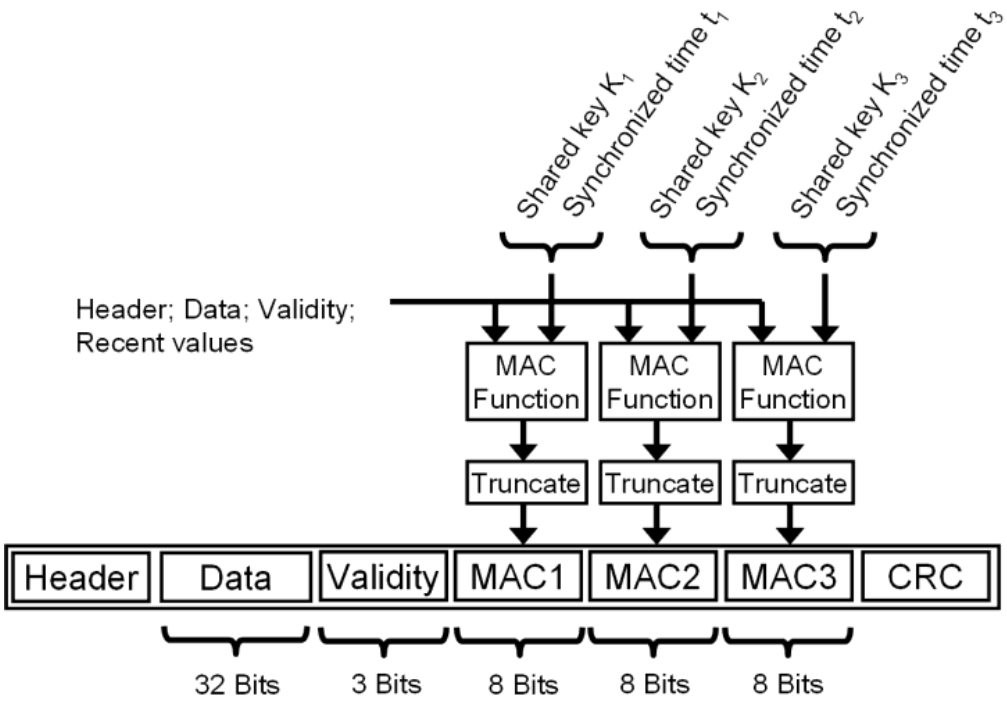

Figure 4.2. Validity voting - multicast authenticator generation. Message generation process for 32 bits of data and three 8-bit MACs, using unique shared keys and synchronized times for three receivers. This packet includes three validity bits, attesting to three prior message values.

When transmitting message type M_S, the sender obtains the message types the packet will attest to using *getMessageTypesVotedOn*(M_S) to produce *ids*. For each receiver *i*, a sender *S* computes the MAC function over the current header and message value, shared secret key k_{iS}, synchronized time *t*, and vectors r_S and v_S produced by *getMostRecent*(*ids*, R_S, V_S, C_S). Before computing the MAC functions, the sender replaces any element of r_S with an 'invalid' value if the validity vector v_S indicates the that value's packet contained an invalid authenticator. We use MMAC as a short hand notation for a function that computes an array of MAC tags (one per receiver) and truncates each MAC tag to just a few bits.

The sender includes the array of truncated MAC tags in the data payload as before, but also includes the validity vector v_S. This allows receivers to recompute the MAC function over the

same values as the sender, replacing values with 'invalid' for those indicated by v_S. After broadcasting their packet, the sender optimistically sets its own validity vector assuming its packet containing the current sample of M_S is received correctly and contains a valid authenticator. Figure 4.3 provides pseudo-code for the send process.

Send process, performed by node *S*:

- Ready to send message value m_S of type M_S to all nodes
- *ids* ← getMessageTypesVotedOn(M_S)
- <r_S, v_S, c_S> ← getMostRecent(*ids*, R_S, V_S, C_S)
- For any element of v_S that is '0', replace the corresponding element of r_S with 'invalid'
- tag_array_S ← MMAC(m_S | t | r_S | v_S)
- Broadcast {m_S | v_S | tag_array_S}
- setNewest(m_S, 'valid', R_S, V_S, C_S)

Receive process, performed by node *i*:

- Receive {m_S | v_S | tag_array_S}
- *ids* ← getMessageTypesVotedOn(M_S)
- If transmission error occurs
  - setNewest('lost', 'valid', R_i, *Vi*, *Ci*)
  - Return from receive process
- <r_i, v_i, c_i > ← getMostRecent(*ids*, R_i, $V_{i,}$ C_i)
- For any element of sender's v_S that is '0', replace the corresponding element of receiver's r_i with 'invalid'
- tag_i ← MAC_{kis}(m_S | t | r_i | v_S)
- If (tag_i = tag_array_S[i])
  Accept new value as valid
  - setNewest(m_S, 'valid', R_i, V_i, C_i)
  - v_i ← bitwiseAnd(v_i, v_S)
  - updateValidity(*ids*, v_i, V_i)
  - For each element in v_i that is '1', increment c_i counters
  - updateConfirmations(*ids*, c_i, C_i)
- Else,
  Reject previous values the current MAC tag included
  - setNewest(m_S, 'invalid', R_i, V_i, C_i)
  - Set all elements in v_i to '0'
  - updateValidity(*ids*, v_i, V_i)

Final Verification process, performed by receiver *i*:
After Receive process is completed, perform final verification step for each message type that node *i* should have received all *z* secondary confirmations:

- Reject value as masquerade attempt if bit in V_i is '0'
- Accept value as lost if bit in V_i is '1' and (value from R_i is "lost" or confirmations in C_i < *z*)
- Accept value (valid and not lost) if the corresponding bit from V_i is '1' and number of confirmations in C_i equals *z*.

Figure 4.3. Pseudo-code for validity voting. Message generation and verification processes during time *t*.

Attesting to message values in this way has several benefits:

- A sender does not need to explicitly retransmit any values it is attesting to. By including them as inputs to the MAC function, two nodes observing different values from the network, each will get a different resulting MAC tag after computing the same MAC function. With multiple attestations, there is an increased probability that a group of receivers will detect any differences in observed message values.
- A single invalid message value (detected and recorded by a voting receiver) cannot cause future messages to be marked as invalid. By replacing any invalid message value with the predefined 'invalid' error code and explicitly including a validity bit prevents further message values from being falsely marked as invalid. Both the sender and receiver will compute the MAC function over the same error code, instead of potentially different values.
- A symmetric packet loss does not cause any message values to be marked as invalid. Similar to the way invalid message values are handled, using a predefined error code for any 'lost' values allows all nodes to compute their MAC functions over the same set of values, rather than whatever erroneous value might have been observed from the network.

4.5.2 Verifying a packet

We break down the message verification into two processes. Each time a receiving node gets new messages from the network, the receiver executes the Receive process for each new message value it has received. Once the Receive process is completed for all new values, it executes the Final Verification process on each value for which all of its *z* secondary confirmations should have been received (Figure 4.3).

Receive process - First, for the received message type M_S, the receiving node *i* executes *getMessageTypesVotedOn*(M_S) to produce *ids*, which contains the message types voted on by M_S. If a transmission error occurs, receiver *i* records a 'lost' value for the received message type, marks it as valid, commits this information using the setNewest function, and exits the receive process without incrementing any confirmation counters. Otherwise, the receiver executes *getMostRecent* to obtain the most recent set of message values r_i received from the network that are voted on by this sample of M_S, corresponding validity vector v_i, and confirmation vector c_i. The receiver replaces any element of r_i with an 'invalid' value if the sender's transmitted validity vector v_S indicates the sender believes that value's packet contained an invalid authenticator. The receiver recomputes the MAC function, and compares the MAC tags.

The MAC tags should only be equal if the sender and receiver agree on the current and prior values (with the infrequent exception of MAC collisions). If they match, the receiver accepts the current value as valid. If the tags do not match, the receiver rejects the current value and all prior values that the sender is attesting to. Because the attested values are sent implicitly as inputs to the MAC function, the receiver cannot determine which value caused the disagreement and conservatively rejects all attested values.

For a valid packet, receivers update their validity vectors for each attested value. Receivers record an attested value as invalid if either the sender's valid packet indicated it was invalid or the receiver originally saw that value as invalid. Receivers perform a bitwise logical And operation on the v_i and v_S vectors. For any value in r_i that is still considered valid in v_i after the vote, the receiver increments the corresponding counter in the confirmation vector c_i. This process only allows a value to remain valid if all voters unanimously agree the message value is valid.

Once this process is complete, the results are committed to the complete vectors R_i, V_i, and C_i.

Final Verification process - Once the Receive process is completed for any new message values, the receiver checks any message values for which all of its z secondary confirmations should have been received. A receiver checks for three possible outcomes for a value in the following order:

Invalid - First, if the bit in the validity vector V_i is '0', then the receiver rejects the value as invalid. Either the original packet containing that message value had an invalid authenticator, at least one of the attesting packets had an invalid authenticator, or at least one voting node claimed that the packet was a masquerade attempt.

Lost - Second, if the bit in V_i is '1', and the value is 'lost', then the receiver accepts that the packet suffered a transmission error and no other receivers claimed it to be a masquerade attempt. Similarly, receivers accept a value as lost if it is valid, but an insufficient number of positive confirmations were received (i.e., confirmations in $C_i < z$).

Valid - Finally, if V_i indicates the value is valid, the value is not 'lost', and the counter in the confirmation vector C_i indicates a sufficient number of positive confirmations from other voting nodes, then the value is accepted as valid.

For a received value to be accepted as valid, there must be a unanimous vote on the authenticity of the value. The packet originally containing that value must have a valid MAC tag, all z attesting packets must also have valid MAC tags, and the validity vectors of each attesting message must indicate each voter observed a valid MAC tag in the original packet. To fool a single receiver into accepting an injected value, an attacker must successfully forge not only the MAC

tag for that receiver, but must also successfully forge the *z* other tags to or from the rest of the voting nodes.

We emphasize that successfully forging one or two packets, then provoking receivers to drop further attestation packets does not increase an attacker's chance of forging a message. Verification of a message value requires a node to receive all packets containing votes for that value. By dropping any attesting packets, the packets targeted for forgery will also be dropped by receivers.

4.6 Integrating with time-triggered authentication

Validity voting can be added to OMPR to verify individual packets in time-triggered authentication. Once a value is transmitted and received by a group of receivers, at subsequent times in the same message round (or subsequent message round), each voter in the group transmits with its vote. This process then repeats according to the message schedule for the network. Figure 4.4 shows an example where message types M_2 and M_3 vote on message type M_1. Each round, receivers obtain the current sample of M_1 then must wait for the next sample of M_2 and M_3 before executing the Final Verification process on a sample of M_1.

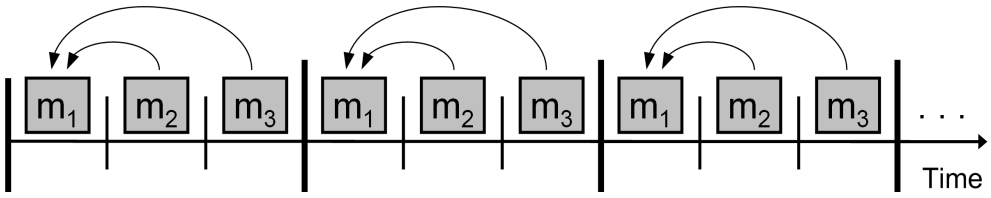

Figure 4.4. Example validity voting with non-overlapping attestations. Receivers complete verification of m_1 values using votes contained in m_2 and m_3 by the time the next value of type m_1 is sent.

Nodes verify state changes and actuations over the final verification results of several message samples, as described in Section 3 (each final verification result is a single entry of the history buffer). There are no changes to the process for verifying state changing and reactive con-

trol messages. However, when using validity voting, the verification results of each individual message sample are delayed slightly, since a receiver must wait for votes to arrive.

The message generation and verification processes for validity voting described in Section 4.5 enable quick recovery from transient network errors or masquerade attacks. As soon as the source of transmission interference or attack ceases, receivers simply resume authenticating over new values. Final verification can then be performed again after a short delay once all votes on the new value are received. Old corrupted values cannot interfere with authentication of future values. This approach limits the effects of a single packet loss or masquerade attempt to only the few previous packets that are voted upon. The effects cannot extend to any values for which receivers have already accepted or rejected based on the final verification process.

4.7 Potential complications and tradeoffs

4.7.1 Packet loss

This approach introduces a design tradeoff between loss tolerance and probability of successful packet forgery. By requiring more secondary confirmations, this approach reduces the probability that an attacker successfully forges individual packets. However, this also increases the number of packets lost by a single transmission error. If a packet is lost by all nodes due to a symmetric fault, the number of positive confirmations for the values attested to by the lost packet will not be high enough for those values to be accepted. Nodes will drop all packets attested to by the lost packet. Section 8 shows several ways to improve this approach's tolerance to transient packet losses.

Another limitation of our approach is that an asymmetric packet loss (some receivers see a well-formed packet, while others drop the packet) will be interpreted as invalid. MAC tags will

disagree because two nodes observed and recorded different sets of values. Section 8 shows methods to resolve this, including the use of an additional bit vector (similar to the validity vector). This vector allows voting nodes to indicate which packets were lost, reducing the impact of an asymmetric packet loss to that of a symmetric packet loss.

4.7.2 Tolerating compromised nodes

Relying on secondary confirmations from other nodes introduces a tradeoff between tolerance to compromised nodes and probability of successful per-packet forgery. Compromised nodes could assist in forgery attempts, attesting that a forged packet from an attacker is valid. The probability that this secondary confirmation is successfully forged is equal to one. To tolerate a fixed number of compromised nodes *w*, a node must receive *w* additional positive confirmations before finally accepting a value (in addition to the *z* confirmations already expected). System designers may trade tolerance to node compromise for increased probability of successful forgery. However, it is important to avoid adding vulnerable (more likely to be targeted for node compromise attacks) nodes to the vote simply to increase the number of votes.

This work assumes the number of compromised nodes is limited to one or two nodes. If an attacker controls multiple critical nodes in the system, then the attacker can likely cause the system to fail in other ways without resorting to masquerade attacks.

4.7.3 Node failure

While this approach automatically recovers once transient faults cease, this approach (as described in this chapter) cannot continue to operate in the event of a permanent node failure. Such a permanent failure could cause all samples of a message type carrying votes to be repeatedly lost. Section 8 shows several ways to handle a permanent voter failure.

4.8 Verification using model checking

To confirm that this voting technique for authentication is secure, we implemented and model-checked this technique using the Automated Validation of Internet Security Protocols and Applications (AVISPA) framework [AVISPA12]. Model-checking is a formal method based technique for verifying properties of concurrent finite-state systems. Model-checking security protocols allows designers to identify flaws which allow an attacker to circumvent the protocol. Our goal is to use model-checking to ensure an attacker cannot successfully forge a packet despite full control over the network, and control over some nodes. This requires verification that validity voting provides data origin authenticity and data integrity. When AVISPA tests for data origin authenticity, it tests for data integrity implicitly as well.

AVISPA uses a Dolev Yao attacker model [Dolev81], giving the attacker full control over the network. This is similar to our attacker model in Section 2. However, the Dolev Yao model assumes that all cryptographic primitives are unforgeable unless the attacker obtains the correct key material. This work addresses the probability the attacker successfully guesses authenticators in Section 4.9.

4.8.1 Model description

The model is a simple network configuration (Figure 4.5) consisting of three nodes N_1, N_2, and N_3, broadcasting message types m_1, m_2, and m_3 respectively. Each node is modeled as an independent process, broadcasting and receiving according to a fixed schedule. We model the broadcast bus using point-to-point channels, sending a copy of every message simultaneously on each

channel. However, all messages in AVISPA are passed through the attacker [AVISPA12] regardless of channel definitions, resulting in a bus-like topology.

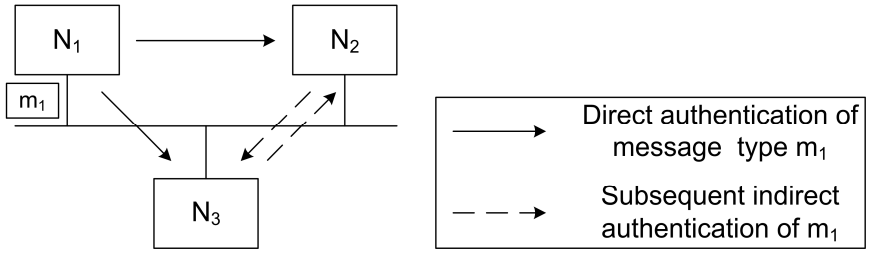

Figure 4.5. AVISPA model of three nodes authenticating message m_1 with validity voting. Node N_1 directly authenticates m_1 to N_2 and N_3. In subsequent time slots, N_2 and N_3 exchange indirect confirmations of m_1's validity and vote on the results.

Nodes communicate according to a round-robin TDMA schedule, in which each node takes a turn broadcasting, then the cycle repeats (as per Figure 4.6). The schedule is hard-coded into the model for simplicity. The three nodes execute over five time slots, allowing each node to complete the final verification process on one value of each message type (Figure 4.6). In each slot, one node sends while the other two receive and update their vote based on the received message. In this model, nodes transmit the current value of their message type, and attest to the validity of the most recent value of the other two. Nodes compute MAC functions over the current value of their message type, the two previous values transmitted by the other nodes, and the validity of those two other message types. Each node receives a direct authenticator and one indirect secondary confirmation of validity for each message type.

When transmitting message values, the state machines for all nodes are hard-coded to accept messages in two specific formats. First, the format of received data in the model can be that of a well-formed transmission (free of errors) specified in Section 4.5. Second, a message can be lost. To model a transmission error, the second allowed format for received data in the model is a sin-

gle constant value "lost." When a node receives this constant, it records the message value as "lost" and records its validity as true. Because the model only specifies these two message formats, the attacker model can inject a well-formed packet or drop a packet. Message losses can be asymmetric, as there is no limitation on the attacker model to inject symmetric message traffic to all receivers.

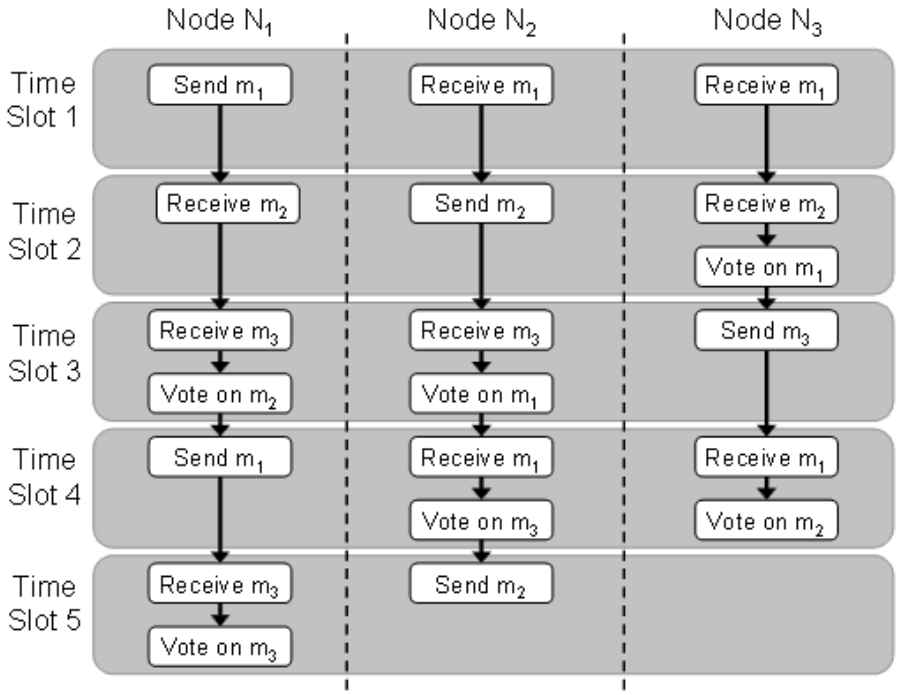

Figure 4.6. AVISPA validity voting model execution over five time slots. This allows each node to cross-check each of three message types.

The model assumes valid m_2 and m_3 values have been previously transmitted at the start of the model without attacker interference (for simplicity, nodes in our model do not vote on these previous values). During time slot one, N_1 sends m_1 with authenticators for N_2 and N_3, attesting to the validity of prior values of m_2 and m_3. Nodes N_2 and N_3 receive m_1 and check its authenticity. If m_1 is valid, N_2 updates its value and validity vectors for m_1 and m_3, while N_3 updates its own vectors for m_1 and m_2. If m_1 is invalid, N_2 and N_3 reject m_1 and the previous values of m_2

and m_3 as invalid. In time slot two, N_2 broadcasts m_2 and attests to whether m_1 and m_3 were valid. N_1 and N_3 update their vectors accordingly. At the conclusion of time slot two, N_3 has received both its direct authenticator for m_1 and the secondary confirmation from N_2. N_3 performs a unanimous vote on its validity vector entry for m_1 and the validity included in N_2's transmission. N_3 accepts the value of m_1 if the direct authenticator was valid, the packet containing the secondary confirmation was valid and indicated m_1 was valid, and the value of m_1 was not received as 'lost.' This process continues over the next three time slots, each node voting once it has received the direct authenticator and secondary confirmation for each message type.

4.8.2 Properties and results

AVISPA verified the data origin authenticity property for each message type for all receivers using OFMC and Cl-Atse, backend components of AVISPA that check this property [AVISPA12]. To provide data origin authenticity, MAC functions can be modeled as keyed hash functions. To test a transmitted variable for data origin authenticity, AVISPA uses a pair of functions: *witness* and *request*. These functions also implicitly test for data integrity. For each transmitted message, the sender executes the *witness* function. This indicates to the model-checker a node with a specific identity transmitted that value. Upon voting and accepting a message as valid, a receiver executes the *request* function. This function tests that the identity of the supposed sender and the value itself are the same as the ones specified in the corresponding *witness* function. If not, then the attacker has managed to successfully forge a packet.

AVISPA detected one trivial attack using parallel sessions starting in the same message round. This attack requires nodes the execute the same protocol twice simultaneously, accepting two values in each time slot. This occurred because both instances of the network configuration used the same set of keys (e.g., the same key K_{12} was shared between nodes N_1 and N_2 of the

first instance and between N_1 and N_2 of the second instance, allowing nodes of one instance to forge messages on a second instance). This attack was detected because of an error in the design of the model. In an implementation on a real system, two networks would use different sets of keys to prevent such an attack from occurring. Further, such an attack could not be performed on a single network. Existing embedded network protocols do not allow transmission of multiple packets over a bus within a time slot.

After modifying the model to disallow multiple parallel sessions, AVISPA reported that the protocol was safe. AVISPA was not able to find any masquerade attacks, including tests where the attacker controlled one of the three nodes. The attacker was not able to successfully forge either the explicitly transmitted value or the validity vector in each packet. Further, adding an indirect secondary confirmation from another receiver does not permit an attacker successfully "over ride" the validity of an explicitly transmitted message value (even when that secondary confirmation comes from a node under the complete control of the attacker in AVISPA). Similarly, allowing an attacker to drop packets does not enable an attacker to successfully forge a value. This confirms our expectations, as a receiver only accepts a value if all direct and indirect authenticators agree on the value of a valid packet.

4.8.3 Model limitations

The model has several limitations.

First, the network configuration in the model is limited to three nodes. The number of nodes was limited to keep the model simple and allow the model checker to verify the model in a reasonable amount of time. Using three nodes allows AVISPA to check if adding a secondary confirmation of an explicitly transmitted value introduces any vulnerabilities that allow an adversary

to successfully forge values. Adding additional nodes to enable more votes beyond the first should not produce different results, since validity voting requires a unanimous vote on the validity of a message.

Second, the model only executes a round robin TDMA schedule over five time slots. This allows each node to verify a single copy of each of the three message types (one from each node). Extending the duration of the model to include more time slots and verify multiple samples of each message type should not change the verification results. Values transmitted in time slots subsequent to the first five cannot interfere with the authentication of the first sample of each of the three message types. All voting completes after node N_2 transmits in time slot five. Further values cannot alter the results of a completed vote. Also, values which are already voted upon are not used in further validity voting. A more complex voting schedule should not change the results either, since nodes share a message and voting schedule. Thus, they know which message samples a message type carries votes for.

Also, transmissions are instantaneous in the model, and nodes can act upon messages without any delay. In real hardware, often the network interface controller on a node executes independently of the processor running the main control loop. The network interface reads messages from the network and stores the most current copy of a message type in mailboxes. Then once the microcontroller starts the next iteration of the main control loop, it accesses those mailboxes to get the most up to date sample of each message type the node consumes. This simplification in the model should not change the results. This aspect can cause an offset in time before a message can be voted upon by a receiver. However, such a time delay is finite in a real-world time-triggered network application where nodes are time-synchronized. We perform this modification in the elevator example in Chapter 6.

The model also limits an attacker to inject only a well-formed packet or a "lost" constant value. The model assumes nodes in a real-world application detect transmission errors using error checking codes within the packet, nodes can detect when no messages have been transmitted on the network during a time slot, and nodes can detect malformed packets that do not conform to the network protocol standards. All transmission errors in the model are represented by a node receiving a "lost" constant value.

Lastly, because AVISPA assumes MAC tags are unforgeable unless an attacker holds the key, AVISPA cannot analyze the probability that an attacker successfully guesses truncated authenticators. Section 4.9 shows a probability analysis and results of simulated attack.

4.9 Probability analysis

To spoof an individual packet to a single receiver, an attacker must successfully forge the authenticator designated for that receiver in the packet and all subsequent confirmations of validity. The probability of successfully forging a single secure MAC tag of b bits in length is 2^{-b}. When attempting to forge a subsequent confirmation, the attacker has two opportunities to succeed. First, the attacker may succeed in forging a MAC tag in the initial packet intended for a receiver that votes on that message. For each initial attempt that fails (indicated by validity vectors in packets), the attacker must attempt to forge each subsequent confirmation and alter the appropriate bit in the validity vector when the voter transmits. Thus, a secondary confirmation can be forged with probability $2^{-b} + 2^{-b} (1-2^{-b})$. If a voter updates its validity vector with another voter's validity before transmitting its own, the probability of successfully forging each confirmation beyond the first decreases slightly with each confirmation. We do not attempt to assign an exact probability based on these tertiary interactions in subsequent confirmations; instead we use $2^{-b} + 2^{-b} (1-2^{-b})$ as a conservative upper bound for each confirmation. This probability is also the

same for a receiver that does not constantly update its validity vectors as soon as votes arrived, and instead simply waits till all votes were received before performing the unanimous vote (in some cases this is easier to implement).

The probability $P_{p\text{-}vv}$ of successfully forging an individual packet with z subsequent confirmations and at most w compromised nodes is bounded by:

$$P_{p-vv} \leq 2^{-b}\left(2^{-b} + 2^{-b}(1 - 2^{-b})\right)^{z-w} \quad (3)$$

Using time-triggered authentication, receivers validate state-changing and reactive control messages over multiple packets for each message type they consume. Since each sample of a message type is verified independently, adding votes will decrease the probability of per-packet forgery (strengthen per-packet assurance). Equations (1) and (2) in Section 3 show that the upper bound on the probability P_A of successful masquerade attack requiring n out of n or k out of n valid time-triggered packets.

4.9.1 Experimental results

We experimentally confirmed the probability of successful forgery attacks against our approach using an embedded CAN network simulator written in Java [Koopman12] (Section 6 describes the simulator in detail). We modified the simulator to allow masquerade attacks. As per our attacker model, the simulated attacker may examine, modify, or replace any transmitted packet, so long as they obey the network schedule. The implemented attacker model does not drop packets.

The simulated network consists of a set of six nodes, broadcasting according to a round-robin schedule. Each node takes a turn sending, then the cycle repeats. The attacker targets one message type to forge, and attempts to fool a single receiver. After attempting to forge the initial

packet, the attacker examines subsequent packets which attest to their forged packet. The attacker modifies any packets that indicate the initial forgery failed (visible to the attacker in the validity vector in packets). If the targeted receiver completes the Final Verification process and accepts the forged packet as valid and not lost, the simulator increments a counter for successful packet forgeries.

We measured the number of successful packet forgeries over a period of time long enough to record at least one hundred successful attack events per data point. We computed the *successful forgery rate* as average successful packet forgeries per message round and compared this rate to the probability of successful attack defined in equation (3).

Figure 4.7 shows the successful attack rate and the expected rate given by equation (3), varying the number of indirect secondary confirmations from zero to four and using two bits per receiver in each packet. Using four confirmations decreases the probability of per-packet forgery by almost three orders of magnitude, requiring four extra bits (one in the validity vector of each packet carrying a vote). To achieve a similar probability using only one MAC per receiver with zero confirmations, each MAC tag would need to be at least eleven bits. By using our voting mechanism, we only need three bits per receiver and four bits for the validity vector if we use four secondary confirmations, reducing authentication bandwidth costs by eight bits per receiver.

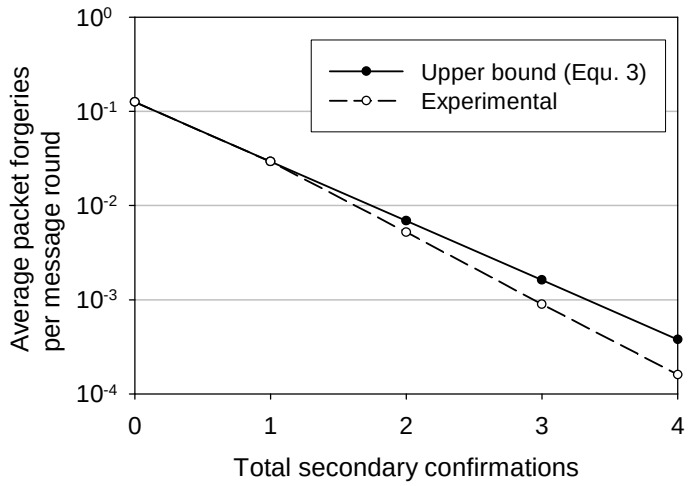

Figure 4.7. Simulated per-packet forgery rates varying secondary confirmations. MAC tags are three bits per receiver.

Figure 4.7 also shows the experimental results initially match the upper bound, then diverge from the upper bound as the number of confirmations increases. This is due to each node updating its validity vector with each received confirmation (taking a unanimous vote between the two) before transmitting its own confirmation, rather than simply sending whether the initial authenticator as valid or not. We also carried out experiments using one to four bits per receiver, varying confirmations from zero to four, with results that similarly support equation (3). These experiments assumed zero compromised nodes.

We also tested the effect of compromised nodes on the probability of successful forgery. Figure 4.8 shows the effect of increasing the number of compromised nodes on average attack events per message round. These experiments used three bits per receiver with a total of four secondary confirmations. The resulting successful packet forgery rates correspond to the same rates as those shown in Figure 4.7. Increasing the number of compromised nodes has the same effect on the probability of successful packet forgery as removing the same number of confirmations.

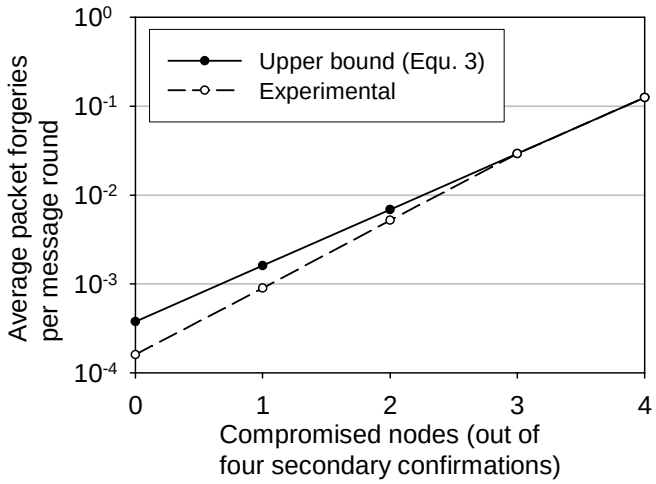

Figure 4.8. Simulated per-packet forgery rates varying the number of compromised nodes. MAC tags are three bits per receiver with four total secondary confirmations.

Figure 4.9 illustrates the effect of integrating our voting technique with our time-triggered authentication approach. Typical required failure rates for safety-critical systems might be defined at 10^{-3}/hr, 10^{-6}/hr, or 10^{-9}/hr. Figure 4.9 shows the number of authentication bits per packet and number of valid time-triggered packets to achieve a failure rate of 10^{-9}/hr using our time-triggered authentication approach alone (zero confirmations) and when combined with our voting technique (one, four, and eight confirmations). The number of packets and bits were obtained using the 10^{-9}/hr as an expected value for one forgery attempt per millisecond over the course of an hour, each succeeding with probability given by equations (1) and (3). For example, given four secondary confirmations, we can achieve an induced failure rate of 10^{-9}/hr using 3 bits per receiver over five time-triggered packets.

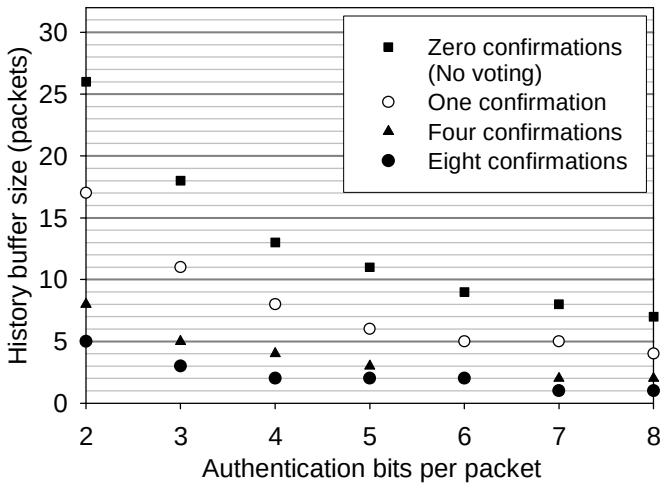

Figure 4.9. Reductions in history buffer size using validity voting. Authentication bits per packet and total packets to authenticate over required to achieve induced failure rate of 10^{-9}/hr on one message type broadcast once per millisecond.

4.10 Discussion

This chapter introduces validity voting, a technique that integrates voting techniques to improve the bandwidth efficiency of OMPR, or reduce the application level latency of time-triggered authentication.

The main benefit of validity voting is that it enables several tradeoffs. By increasing the number of votes on a message, the system designer can decrease the number of authentication bits per receiver, increase the number of receivers, or decrease the number of time-triggered samples to authenticate over. Increasing the number of votes also decreases the loss tolerance of this approach.

Validity voting also has several limitations:

- First, like OMPR, the per-packet bandwidth overhead scales nearly linearly with the number of receivers, limiting the maximum number of receivers in practice. It's main virtue is that

votes can be used to produce a smaller scaling constant than OMPR. With limited bandwidth for authentication, this approach cannot scale to hundreds or thousands of receivers. However, embedded networks typically have only tens of receivers.

- Increasing the number of votes also increases sensitivity to packet losses. If one message sample suffers a symmetric transmission error, then any messages it carries votes for will also be declared lost. In the event of an asymmetric packet loss, a message may be declared invalid, since nodes will disagree on the message value. Chapter 8 describes methods to improve tolerance to asymmetric packet loss.
- This approach also assumes a fixed number of compromised nodes to tolerate when determining the number of authentication bits, history buffer size, and secondary confirmations. If the number of compromised nodes exceeds this assumed number, no guarantees can be made about induced failure rates. However, in an embedded network containing critical nodes, if the attacker compromises more than one or two critical nodes they can likely cause the system to fail without resorting to masquerade attacks.
- Lastly, this section does not address permanent faults (i.e., node failure) that permanently disrupt authentication of multiple message types. Chapter 8 discusses methods to improve tolerance to node failure.

5 Comparisons to other multicast authentication techniques

This section compares four multicast authentication techniques that can be used in conjunction with time-triggered authentication.

One of the advantages of time-triggered authentication is that it can be combined with many multicast authentication techniques. Any multicast authentication technique using MACs can be used so long as the MAC outputs can be truncated without compromising the security of the underlying functions or key (e.g., hash-based MAC functions). This allows the system designer to perform tradeoffs among different authentication techniques to find which best satisfies the requirements of the system.

This work identifies four low overhead mechanisms to authenticate time-triggered messages on a per-packet basis. Each technique spans a range of tradeoffs, which might influence whether it is suitable for authenticating time-triggered messages in a particular system. The first technique we examine is OMPR (Section 3), which we use as a baseline multicast authentication technique in our initial work on time-triggered authentication. The second approach is validity voting (Section 4). Voting adds complexity, but allows a group of nodes to cross-check the validity of messages amongst themselves to reduce authentication overhead. Third, we apply TESLA [Perrig00], which uses time-delayed release of keys. Lastly, we introduce a master-slave authentication approach, based on Chan and Perrig's hash tree broadcast authentication using a trusted base station node [Chan08].

To determine which of these approaches are most suitable for embedded control networks, we compare them in terms of scalability with respect to number of receivers and per-packet assurance, and the effects of packet loss, node failure, and node compromise on each. We also note

any tradeoffs unique to each approach which may make it more or less desirable for certain system applications.

One of our overall goals is to minimize the number of authentication bits required per packet. This overhead is primarily affected by two system factors: number of receivers and required per-packet assurance. Authenticating to more receivers might require more symmetric authenticators per packet, depending on the approach. The assurance probability required for a packet determines how many authentication bits are needed for each symmetric authenticator. This section shows how bandwidth overhead scales for each approach with respect to these two factors.

System requirements regarding permanent and transient faults may also make one approach more desirable over another. This section shows the effects of transient packet loss on authentication and how long each approach takes to recover from such faults. Also, tolerance to node compromise and failure can be affected by reliance of receivers on other nodes to authenticate messages (e.g., using a trusted master to authenticate all messages). We discuss each approach's tolerance to node compromise and failure. We do not discuss full denial of service attacks intended to prevent all transmissions on the network.

5.1 Metrics for comparisons

This section defines our metrics for comparing multicast authentication approaches for use in time-triggered authentication. Our primary goal, beyond preventing malicious faults, is to minimize bandwidth consumed by authentication. Depending upon the multicast authentication technique used, authentication overhead can be sensitive to the number of receivers and the required level of per-packet assurance. Second, in a safety-critical application, authentication approaches should be able to recover quickly from transient faults and resume authentication. Last-

ly, some tolerance to permanent faults and node compromise is desirable. We identify potential single points of failure for these approaches.

Other metrics for comparison are also possible based on our design criteria in Section 2 (e.g., processing and memory overhead). We assume senders have sufficient computational resources to compute one MAC function per receiver and have sufficient memory capacity to store symmetric keys for nodes they communicate with. We also assume a node is able to store temporary values as well as all key material. Key chains for TESLA can be stored using an efficient construction, such as the one described by Jakobsson [Jakobsson02]. Systems that have severely constrained nodes (in terms of processing and memory) and do not conform to these assumptions require further tradeoff analyses at design time.

Scalability with per-packet assurance level - For each approach, we examine how the number of per-packet authentication bits increases with respect to the per-packet assurance level. Per-packet assurance is defined in Section 3 as the acceptable probability of successful forgery per packet. A weaker per-packet assurance level gives an attacker a higher probability of successfully forging each packet, but requires fewer authentication bits per packet. Conversely, using a stronger per-packet assurance level creates a lower probability of successful packet forgery, but requires more authentication bits per packet.

Scalability with receivers - We also examine how per-packet authentication overhead increases as the number of receivers increases. Many multicast authentication approaches are designed to scale well to hundreds or thousands of receivers based on certain assumptions. Some techniques we discuss scale poorly to large numbers of receivers, and are only suited to networks with few nodes. Other techniques scale well to thousands of receivers, but have a high baseline overhead that makes them scale poorly to very few receivers.

Loss tolerance - We show how overall network throughput decreases as packet loss increases to illustrate the impact of inter-node and inter-packet dependencies for authentication. Schemes such as validity voting and master-slave authentication require a sender to rely on one or more other nodes to confirm the authenticity of packets. We define *fragility* as the number packets lost due to a single transmission error affecting one packet.

We also examine the *robustness* of each approach, showing how much time must pass before an approach recovers from a transient network error and receivers can resume authentication.

Tolerance to node compromise and failure - Lastly, we discuss the impact of node failures and compromises on each approach. Schemes that require a higher level of inter-node dependency for authentication are more sensitive to node compromise and failure. Node failures can prevent the authentication of more message types than those sent by the failed node. Further, for these schemes with dependencies, attackers can forge any message if they control a sufficient number of nodes in the network.

5.2 TESLA

TESLA [Perrig00] uses time-delayed release of keys for multicast communications. This approach requires loose time-synchronization between a sender and receivers that consume that sender's message types. During each time interval, the sender uses a different key to authenticate broadcast messages within that interval of time. TESLA generates a chain of MAC keys using a one-way hash function. Each key is kept secret by the sender until after all receivers should have obtained the messages authenticated with that key, then the sender releases the key during a pre-

defined subsequent time interval. Receivers then use the disclosed key to verify messages release during its corresponding time interval.

By releasing keys at a pre-specified delay after a message and MAC tag are released (in a time-synchronized network), receivers can confirm the authenticity of the data from a sender. An attacker cannot obtain a secret key before other receivers to use it to forge messages on behalf of a valid sender. An attacker also cannot use a key after its designated release interval. Receivers discard late messages that arrive after the time at which the corresponding key should have been released, since receivers cannot trust an attacker did not attempt to forge those messages on the sender's behalf.

This key release approach requires a single MAC tag to authenticate each value regardless of the number of receivers, so long as they are time-synchronized with the sender. As a security requirement, TESLA requires that a sender and all receivers be loosely time-synchronized, so receivers can detect keys and messages that arrive at irregular times (indicating an attacker may have tampered with the message). Our assumption that nodes synchronize to the current message round fulfills this security requirement.

5.2.1 Modifications to TESLA

This approach does not modify the TESLA protocol other than to truncate the size of the MAC tags released based on the needed assurance level of individual samples of a message type.

5.2.2 Initialization

Key chains - During initialization, each sender generates a key chain of some predetermined length *N* [Perrig00]. The sender first selects a random seed value K_N as the last key of the chain. The sender then iterates a public one-way (pre-image resistant) hash function F. The sender

computes the chain by recursively using $K_i = F(K_{i+1})$ for $i = N\text{-}1, N\text{-}2, ..., 0$. To avoid cryptographic weaknesses, TESLA avoids using the same keys for deriving the next key in the chain and for computing MACs. A sender computes keys for MACs using another one-way function F': $K'_i = F'(K_i)$. During runtime, the sender will release keys in the order $K'_0, K'_1, ..., K'_{N-1}, K'_N$.

This work assumes nodes are able to store key chains that are sufficiently long for regular operations. Jakobsson [Jakobsson02] describes a storage efficient mechanism for one-way chains of length N only requiring $\log(N)$ storage at a cost of $\log(N)$ computations to access an element.

Time synchronization - To maintain security, TESLA requires receivers to be loosely time synchronized with each sender. At minimum, receivers must know the upper bound on each sender's clock. This upper bound defines how quickly a sender can release a key for a previous MAC tag. The strict time synchronization used in OMPR and validity voting satisfies this requirement.

When using TESLA in time-triggered authentication, we assume the time synchronization error between a sender and receivers is much less than the broadcast period for a message type. A sender and receivers synchronize time to a sufficiently fine clock-tick granularity, such that the sender can release the key on the next sample of the same message type. The time synchronization approaches discussed in Section 3 already provides this level of synchronization.

See TESLA [Perrig00] for further details on timing requirements.

Key establishment - At initialization, receivers must be loosely time synchronized with each sender, know the disclosure schedule of keys, and receive an authenticated key of the one-way key chain. The sender transmits all key disclosure schedule information and the first key to be released from the one-way key chain using an authenticated channel (e.g., digitally signed broadcast or unicast to individual receivers).

5.2.3 TESLA in time-triggered authentication

During runtime, a sender computes a MAC tag for the value transmitted during the current time interval using the key corresponding to that interval. When transmitting message *m* during the interval corresponding to key K'_i, the sender computes tag $t_i = \text{MAC}_{K'i}(m)$. The sender does not need to include the current time or message round in the MAC computation since keys are already assigned to specific time intervals. Because we are authenticating periodic messages, the sender truncates the MAC tag based upon the required per-packet assurance needed for that message type. The sender transmits the value and corresponding tag during the current interval *i*. In a subsequent interval, the sender releases the key for a prior interval K_i (Figure 5.1). Once the key is received in that subsequent interval, all receivers can compute K'_i and verify the tag in the prior interval and accept or reject those values.

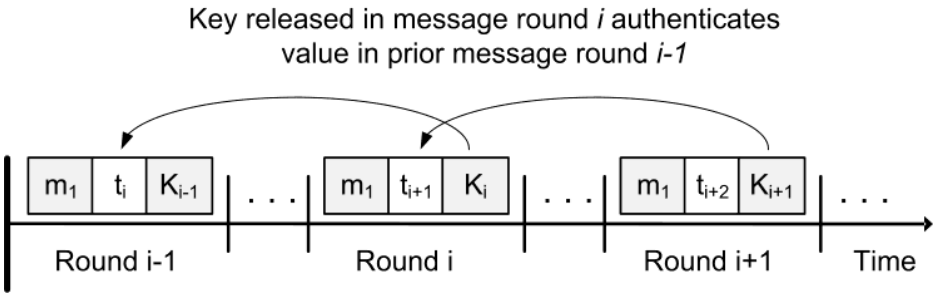

Figure 5.1. TESLA used in time-triggered authentication. TESLA uses time-delayed release of keys to provide asymmetry. The key released in the current message round for a message type authenticates the data value for that message type in the previous round.

As in time-triggered authentication for periodic messages, we truncate each MAC tag based on assurance required for the sample it authenticates. However, we cannot truncate the released keys. Truncating the key exponentially reduces the security of this approach. We assume each time-triggered sample requires the release of a complete key; senders do not "batch authenticate"

multiple samples of the same message type at once. For this work, we assume a secure symmetric key size is eighty bits [Lenstra01]. Other approaches exist that allow release of smaller keys by regularly initializing new key chains [Hu03], but we do not address those in this work. The approach described by Hu et al. enables a tradeoff between key size and the frequency of establishing new key chains (establishment of a key chain requires the sender to broadcast messages as described in Section 5.2.2).

A message is recorded as valid if the receiver computed MAC tag matches the tag received for from the sender. If the receiver computed MAC tag does not match the sender's tag, the receiver records the message value as invalid. A receiver may drop the message value if either the packet containing the message value and tag or the subsequent packet containing the key suffer a transmission error.

If a key is lost due to a transmission error, TESLA requires recomputation of those lost keys once a subsequent key is received to verify that the received keys are indeed part of the key chain committed to by the sender during key initialization.

Recovery of lost keys also allows receivers to verify message samples for which the corresponding key material was lost. Once a receiver obtains a later key, the receiver can iterate the one-way function to recover any previously lost keys. This allows a receiver to authenticate a previously lost message value if that message value and its tag are correctly received but the packet containing the corresponding key is lost. This recovery mechanism is useful for state-changing message types, since a receiver waits for several consistent values before committing to the state change. Conversely, key recovery is less useful for reactive control message types. Receiving controllers use the most recent value to update controller outputs; old values are typically discarded.

5.2.4 Tradeoffs with respect to key chains

For nodes which broadcast multiple message types, TESLA enables a tradeoff among the number of key chains maintained by a transmitting node, processing and memory overhead for key chains, bandwidth for authentication, and loss tolerance (and associated recovery of lost keys). A transmitting node might maintain one key chain for all message types that it broadcasts to minimize authentication bandwidth overhead. It also reduces processing and memory overhead for keys. Each message round, a sender computes one MAC tag for each message sample it transmits using one key. In the subsequent message round, it releases that single key. This approach has the advantage of amortizing the bandwidth cost of transmitting key material over samples of many message types. One disadvantage of maintaining a single key chain is that if a single key is lost, a receiver cannot verify any of the message samples until a subsequent key is correctly received and the lost key is recovered. For state-changing messages, this creates a delay in verifying multiple message samples. For reactive control messages, loss of a key may cause a receiver to also lose a sample of multiple message types (increasing the fragility).

Alternatively, a sender can maintain one distinct key chain for each message type it broadcasts. This approach requires more authentication bandwidth overhead, processing, and memory overhead for keys. Each message round, a sender computes MAC tags using the respective key chains for each message type. In the subsequent round, the sender releases multiple keys, one for each message type. The advantage of maintaining many key chains over maintaining a single key chain is decreased fragility. Loss of a key only affects one message type instead of all message types broadcast by the sender.

Lastly, a system designer can group message types and maintain a key chain for each group of message types. For example, message types could be grouped by required assurance, safety criticality, message period, or system function.

5.2.5 Discussion

TESLA performs well in terms of bandwidth requirements for large numbers of receivers, requiring only one MAC tag and key per transmitted packet. The required per-packet assurance of the authenticated message value determines the size of the single MAC tag. The disadvantage with respect to embedded networks is that a key must be sent for each interval. This creates a high minimum bandwidth requirement per packet even for message types with few receivers or requiring low per-packet assurance. However, this bandwidth can be amortized if a sender transmits multiple message types.

TESLA also has excellent tolerance to packet loss (low fragility) and node failures. If a receiver drops a packet, that packet does not affect any other packet. This approach is also robust to transient faults, recovering as soon as the next message value and subsequent key are received. Once correct transmissions resume, the receiver will have to perform extra hash computations during that message round to recover dropped keys and verify the current key is part of the key chain. Additionally, node failure or compromise does not affect messages from any other node.

The time synchronization requirement for TESLA is not a disadvantage when comparing TESLA to other approaches in conjunction with time-triggered authentication. The time synchronization requirement of TESLA might be considered a limitation in enterprise systems. However, time-triggered authentication already has tighter time synchronization requirements than the loose time synchronization requirements of TESLA.

5.3 Master-slave

In a master-slave authentication approach, a trusted "master" node attests to the authenticity of all messages transmitted by other nodes on a broadcast bus. This approach is similar to using a base station node to perform the same function in a wireless sensor network. Transmitter and receiver nodes on the broadcast bus are "slaves" in the sense that they rely completely upon the master node for trust in messages. Each slave node establishes a symmetric key with the master node. Slave nodes authenticate each value they transmit to the master node using a single MAC tag. The master node is responsible for regularly computing a broadcast authenticator over values observed on the bus and transmitting this broadcast authenticator to all slave nodes. This differentiates this approach from a master-slave communication protocol where the master explicitly controls which node transmits next.

The master node could use any method for broadcast authentication, depending on the requirements of the system. For example, the master could simply compute one MAC tag per receiver, compute a digital signature, or use TESLA. The cost of the broadcast authentication is amortized by authenticating a batch of messages values (of different message types) at once to the receivers.

In this work, for master-slave authentication, we use hash tree broadcast authentication, proposed by Chan and Perrig [Chan08]. All nodes are organized into a tree topology, with the base station as the root of the tree. When the base station transmits, the base station first computes a MAC tag for each receiver, using a symmetric key shared with that receiver. However, the base station does not transmit all the MAC tags. Instead, it computes a hash tree where all MAC tags are placed at the leaf nodes of the tree. The base station then transmits the single resulting root hash value. Each receiver then releases its MAC tag. Once all the tags have been released and all

receivers can recompute the base station's hash tag and confirm the authenticity of the entire batch of messages. Since only the base station knows all the keys used to compute the leaves of the hash tree, an attacker cannot derive the root hash value.

We modify and apply this idea to a broadcast bus topology, whereas Chan and Perrig applied hash tree broadcast authentication for linear, tree, and fully-connected network topologies [Chan08][Chan10].

5.3.1 Hash tree broadcast authentication

In Chan and Perrig's work [Chan08], a spanning tree is first constructed over the network topology. All communication occurs over the links of this tree. The tree is anchored with its root at the base station. Sensor nodes are placed as the leaf nodes. Intermediate nodes between the root and leaf nodes act as aggregators and disseminators of information.

When the base station at the root transmits a message *msg* to the leaf nodes, it computes a MAC tag for each receiver *i* using the key shared between the base station and that receiver: $tag_i = \mathrm{MAC}_{Ki}(N \| msg)$, where N is a nonce or timestamp and K_i is a symmetric key established between node *i* and the base station. To avoid congesting the links in the tree, the base station does not broadcast all tags. Instead it computes a hash tree over the set of MAC tags computed using each of the keys shared with nodes in the network. For example, for nodes 1 through *n*, the base station would compute a hash tree over the values $\{\mathrm{MAC}_{K1}(N \| msg), \mathrm{MAC}_{K2}(N \| msg), \ldots, \mathrm{MAC}_{Kn}(N \| msg)\}$. The base station then distributes the root *r* of this hash tree, message *msg*, and nonce N to its intermediate child nodes, which subsequently disseminate it to the leaf nodes.

Once the leaf nodes have received the root of the hash tree, each leaf disseminates its own tag. As the tags from the leaf nodes pass up through the intermediate nodes, the intermediate nodes

exchange any "off path" vertices of the tree amongst themselves and then to their child nodes. Leaf nodes do not need every MAC tag used to compute the root of the hash tree, each merely needs to confirm that its MAC tag was included in the hash tree [Chan08]. By using intermediate nodes to exchange off path vertices of the hash tree, this approach reduces message traffic congestion across any single link in the network.

Eventually, each leaf node will be able to recompute the root hash value using its own MAC tag along with the other off path vertices of the hash tree. Each node can then verify that its MAC tag was included in the base station's root hash.

5.3.2 Modifications to hash tree broadcast authentication

This work uses a variation on hash tree broadcast authentication to allow a trusted master node to perform a batch authentication of a set of message values. First, each slave node authenticates its message value to the master node. The other slave nodes attached to the broadcast bus are able to observe the values transmitted by other slaves, but cannot immediately authenticate those values. The master node then broadcasts a message indicating whether the set of message values from the slave nodes were all valid or not using a variation of hash tree broadcast authentication.

Since a broadcast bus uses only a single communication link, intermediate nodes are not needed to exchange off-path vertices of the hash tree to minimize message traffic congestion on any single link for two reasons. First, all message traffic is already exchanged over the single link of the network. Second, all nodes connected to the broadcast bus are able to observe every message transmitted on the bus. Instead of creating a binary spanning tree over the network, we use a spanning tree with only two levels: a master node at the root and all slave nodes as leaf nodes on the level below the master node.

When broadcasting a message, the root node computes a MAC for each receiver. However, instead of computing a binary hash tree over the MAC tags, it simply computes a single hash of all the MAC tags.

5.3.3 Initialization

Key establishment – Each transmitting slave node *i* in the network establishes a key K_i with the master node. This key is used to compute MAC tags to authenticate messages to the master node.

Each slave node *j* that consumes any message types establishes key K'_j with the master node. The master node computes a MAC tag for receiver *j*, using this key. The master includes this tag in the hash tree.

Any node that both transmits and receives messages establishes both keys with the master.

Time synchronization – As in Section 3, each slave node performs pair-wise time synchronization with the master node.

5.3.4 Verifying messages

This master-slave approach using the modified hash tree broadcast authentication executes over three phases to authenticate message values from *u* transmitters to *v* receivers. This approach does not require that the set of *u* transmitters be the same as the set of *v* receivers.

Phase 1 - Slaves authenticate messages to master - In this phase, each transmitting slave node *i* computes a MAC over the message m_i it will broadcast during time *t* (synchronized with the master node) using key K_i it shares with the master node: $slave\_tag_i = MAC_{Ki}(t \parallel m_i)$. During the time *t*, node *i* broadcasts $<m_i, tag_i>$. The master node and all slave nodes record message m_i. However, only the master node is able to verify the authenticity of m_i. The slave nodes must wait

for the master node to attest to the validity of messages. If a message from a slave node suffers a transmission error, a receiver (both master and slave nodes) records a predefined error code 'lost' for that message.

Phase 2 - Master broadcasts hash tree broadcast authenticator - In the second phase, the master node computes a hash tree broadcast authenticator (as described in Section 5.3.2) to attest to the validity of the all messages broadcast by slave nodes. If all messages from the slave nodes were valid, the master broadcasts a single-bit 'valid' message along with the hash value *r*. To attest to messages from *u* transmitting slaves nodes at time $t + 1$, the master node computes a MAC tag for slave node *i* using the shared key K'_i: $master\_tag_i = \text{MAC}_{K'i}(t + 1 \parallel \text{'valid'} \parallel m_1 \parallel m_2 \parallel ... \parallel m_u)$. The master does not need to retransmit the messages it is attesting to, since all slave nodes attached to the bus should observe the same messages as the master. For any message the master did not receive due to a transmission error, the master replaces that message value with the 'lost' error code. The master then computes hash *r* over these tags for slave nodes that consume any of the messages from the u senders. For *v* receivers, the master computes hash *r* over $\{master\_tag_1, master\_tag_2, ..., master\_tag_v\}$. The master node then broadcasts <'valid', *r*> onto the bus at time $t + 1$.

If any of the messages from transmitting slave nodes had invalid authenticators, the master instead broadcasts a single-bit 'invalid' message and the MAC tags are computed over only the single-bit 'invalid' message and the synchronized time with each receiver. The master then hashes the tags together and broadcasts the message and hash as per normal.

Phase 3 - Slaves exchange MAC tags to verify master's hash - In the last phase, each receiving slave nodes releases its MAC tag so that all receivers can verify the master's hash and subse-

quently validate messages from the *u* transmitters. Phase three results in one of three outcomes for all *u* messages: valid, invalid, or lost.

Valid - If the master node broadcast the 'valid' confirmation message in phase two, each receiving slave node *i* computes $master\_tag_i = MAC_{K'i}(t + 1 \parallel \text{'OK'} \parallel m_1 \parallel m_2 \parallel ... \parallel m_u)$, using its key K'_i and time $t + 1$. If a transmission error prevented a slave node from receiving one of the messages (from phase one) that the tag is computed over, the node replaces that value with the 'lost' error code when computing the MAC tag. Each slave node then releases this MAC tag. Once all *v* slave nodes transmit their MAC tags, those nodes compute the hash over that set of tags and compares to the master's hash *r*. If the hashes match, the receiver accepts all *u* messages attested to as valid (with the exception of those messages that were lost in phase one, which are recorded as such).

Invalid - If the master's message contained a 'valid' bit and the received hash does not match the computed hash, it indicates that an attacker might have attempted to tamper with the master's message in phase two. Since the receiving slave nodes cannot determine the validity of those messages broadcast in phase one, they reject all *u* message values as invalid.

If the master's message contained the 'invalid' message, indicating some of the messages from transmitters were not valid, the receivers can then reject all *u* messages from phase one as invalid.

The received hash and computed hash may also not match in the event of an asymmetric packet loss; some nodes recorded one or more the u message values as lost, while other nodes received the value correctly.

Lost - The *u* messages from phase 1 are recorded as 'lost' in two cases. First, if the master node's attestation message from phase two suffers a transmission error, the receivers simply record all *u* messages as being 'lost.' Second, if any tag from any of the *v* nodes broadcasting in phase 3 suffers a transmission error and the master's message contained a 'valid' message bit, then no receiver can verify the master's hash. In this case, receivers also record all u message values as 'lost.' If the master's message contained an 'invalid' bit, receivers always conservatively reject the *u* message values as invalid.

5.3.5 Master-slave in time-triggered authentication

When using this master-slave approach in a system application where messages are broadcast at regular intervals, each execution of the three phases can be overlapped. Each slave node that is broadcasting a message value during a message round (and also receives messages from the previous round) can also include the MAC tag for the previous round in their transmission. This halves the number of transmissions by any slave nodes needed for verifying a round of message values (Figure 5.2). Thus, a slave node transmits only two truncated MAC tags in a data payload. The first MAC tag authenticates its current value to the master node. The second MAC tag is computed over the values observed on the network in the previous round.

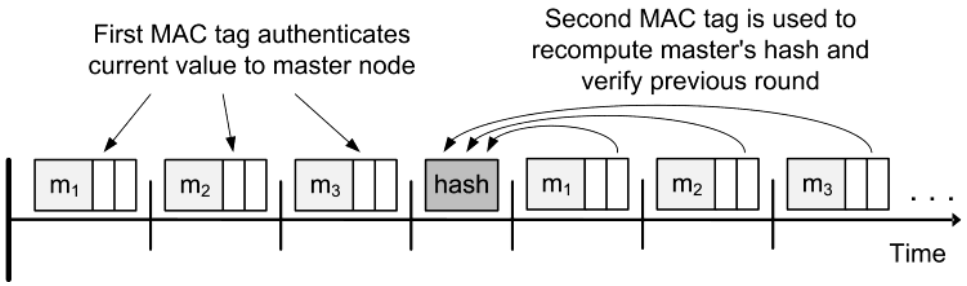

Figure 5.2. Master-slave used in time-triggered authentication. Each packet contains two MAC tags. The first authenticates the current broadcast value to the master node. The second tag is used to verify the master's hash from the previous message round.

Slave nodes truncate the MAC tags to a few bits based on required per-packet assurance. The master truncates each of the MAC tags and resulting hash it computes based on the same required per-packet assurance. The approach is only as secure as the MAC or hash with the fewest bits; all MACs and the hash for each execution of the three phases of this approach should be the same number of bits. If the MAC tag produced by the sender in phase one is smaller than the master's hash, the attacker could potentially guess that MAC tag more easily than the master's hash to inject a forged message value from that sender. Similarly, if the master's hash in phase two has fewer bits than the sender's MAC tag in phase one, then an attacker could inject a forged value from a sender and attempt to spoof the master's hash instead.

When creating a message schedule (or defining broadcast periods for message types), the master node should be scheduled to broadcast its attestation message sufficiently quickly to allow verification of individual samples for each message type. Thus, the master node broadcast period should be less than or equal to that of the message type with the shortest period being broadcast on the network. Further, the master node should be scheduled to authenticate messages to slave node receivers that are able to promptly broadcast their tag so that other receivers can

confirm the master's hash. Receivers cannot verify the hash until all receivers of the master's hash have released their tags.

5.3.6 Discussion

This approach has two primary advantages. First, authenticating via a master or base station node is very efficient in terms of bandwidth on a broadcast bus in comparison to all nodes broadcasting multicast authenticators. Having one node authenticate messages from all nodes requires a single broadcast authenticator. In this approach based on hash tree broadcast authentication, slave nodes only transmit two MAC tags, each of which can be truncated.

Using hash tree broadcast authentication also has the advantage of distributing the authentication bits of the master's attestation amongst the slave node transmissions instead of only placing them in a transmission from the master node. If the master node used another multicast authentication mechanism (e.g., OMPR or TESLA to send the master's attestation), it might have to introduce extra packets to broadcast the authenticator. Since the master's authenticator should be broadcast at same frequency as the fastest message types, adding additional packets from the master would significantly increase bandwidth costs (each packet in CAN uses a minimum overhead of 90 bits per packet). Using hash tree broadcast authentication, the master only needs to send a single hash.

Using a trusted master node also introduces several disadvantages, regardless of the broadcast authentication mechanism used. First, the master node is a single point of failure. If the master suffers a permanent failure, no authentication can be performed. Second, this approach has high fragility, being very sensitive to packet losses.

Using a trusted master also allows an attacker to attempt two guesses to forge an authenticator. First, the attacker can attempt to forge the tag for the master in phase one. Second, if the master's message indicates that one of the tags was invalid, the attacker can attempt to forge the master's broadcast authenticator. If the probability of successful forgery on either authenticator is 2^{-b}, where each authenticator is truncated to *b* bits, then the probability of successful per-packet forgery is given by equation (4). This is the same probability as forgery attempts on secondary confirmations in validity voting (Section 4.9).

$$P_{p-ms} = 2^{-b} + 2^{-b}(1 - 2^{-b}) \quad (4)$$

Using an approach based on hash tree broadcast authentication exacerbates the impact of node failure. If a single receiving node suffers a failure, that node might not transmit the MAC tag that would allow the rest of the network to verify the master's hash value. A single failed node might prevent all authentication. Chapter 8 describes approaches to improve tolerance to node failures. Similar approaches could be used in conjunction with master-slave.

This multicast authentication approach may not be suitable for networks with a large number of silent receivers that would otherwise never transmit; each of those receivers must now transmit a message on the network to participate in verifying the master's hash. This would significantly increase bandwidth requirements for authentication. For those types of networks, this master-slave approach using hash tree broadcast authentication should not be used. The master node can authenticate the messages from the previous round using another mechanism such as TESLA.

Node compromise is only a concern for this approach if the attacker gains control over the master node. If the master node is compromised, the attacker can forge any message desired. Compromised slave nodes can only forge messages they already are expected to send.

Lastly, there is a potential security vulnerability when authenticating all messages through a trusted master in conjunction with time-triggered authentication. In time-triggered authentication, packet losses are considered non-malicious; invalid packets are considered malicious. As described in section 5.3.4, the master explicitly attests to the validity of the messages transmitted in phase one of this approach. The attack is executed as follows:

1. During phase one, the attacker selects a message type to attempt to spoof and attempts to guess the authenticator attached on a sample. Because we use few authentication bits, there is a moderate probability of successful forgery per packet.
2. In phase two, the attacker intercepts and observes the master's message. If it indicates no forgery attempts, the attacker knows they guessed the MAC tag correctly during phase one. If otherwise, the attacker drops the master's packet.
3. The attacker then repeats steps one and two until a sufficient number of forgeries have been successful to induce a system failure.

While this attack might technically allow an attacker to successfully forge many message samples over time, the attacker is forced to drop many messages from the master. Even with single bit tags, the attacker will drop about every other (50 percent) messages from the master on average. If tags are four bits, the attacker is forced to drop about fifteen out of sixteen (93.75 percent) messages from the master. With more bits the percentage of dropped packets is even higher. Since at most two tags per packet are needed, system designers are likely to use relatively large tags for a high per-packet assurance. With so many dropped packets, a receiver is likely to

assume the network has suffered a blackout and take an appropriate safe action, precluding an attack from achieving the desired effect. If this attack is a concern, the validity bit in the master's attestation message can be omitted. The receiver then always computes the MAC tag in phase three over the messages observed during phase one, assuming none have been tampered with. Thus, an attacker cannot watch a master's messages to see if they successfully guessed the MAC tag, and the verification in phase three will only be successful if the master and slave nodes observed the same messages during phase one.

5.4 Comparisons

In this section, we compare each of the four techniques in terms of scalability with per-packet assurance, scalability with receivers, loss tolerance, and node failure/compromise.

5.4.1 Scalability with per-packet assurance

We first show how the required authentication overhead of each approach scales as we vary the per-packet assurance. We calculate the per-packet authentication overhead assuming a fixed number of receivers. For OMPR and TESLA, we assume that each has a probability of successful per-packet forgery P_p of 2^{-b} requires b bits per MAC tag. For validity voting, equation (3) gives an upper bound on the probability of per-packet forgery success after receiving z confirmations from other voting receivers, where w of the voters are compromised. We use this equation to determine the number of MAC tag bits required to achieve a probability of per packet forgery equal to or less than those for the other schemes. For master-slave, the probability is $2^{-b} + 2^{-b}(1-2^{-b})$, as discussed in Section 5.3.6.

Table 5.1 provides the per-packet authentication bandwidth cost for each of the four techniques when applied to various per-packet assurance probabilities given R receivers. This table

shows the number of authentication bits per MAC tag, multiplied by the number of required tags, plus any added overhead. For validity voting, each transmitter includes v validity bits in their packet (one for each message type it carries a vote for). Table 5.1 also assumes zero compromised nodes ($w = 0$). TESLA requires a key of size K. Master slave always uses at most two MAC tags in each packet. These calculations omit any packet fragmentation that may occur if a value and authenticator cannot fit within a single physical packet for a given network protocol.

Table 5.1. Authentication bits per packet vs. per-packet assurance

	Per-packet authentication overhead (bits)					
Per-packet assurance	***OMPR***	***Validity Voting***			***TESLA***	***Master/ Slave***
		1 Vote	***2 Votes***	***4 Votes***		
2^{-2}	$2\times R$	$2\times R + v$	$2\times R + v$	$1\times R + v$	$2 + K$	3×2
2^{-4}	$4\times R$	$3\times R + v$	$2\times R + v$	$2\times R + v$	$4 + K$	5×2
2^{-8}	$8\times R$	$5\times R + v$	$4\times R + v$	$3\times R + v$	$8 + K$	9×2
2^{-16}	$16\times R$	$9\times R + v$	$6\times R + v$	$4\times R + v$	$16 + K$	17×2
2^{-32}	$32\times R$	$17\times R + v$	$12\times R + v$	$8\times R + v$	$32 + K$	33×2
2^{-64}	$64\times R$	$33\times R + v$	$22\times R + v$	$14\times R + v$	$64 + K$	66×2
2^{-128}	$128\times R$	$65\times R + v$	$44\times R + v$	$27\times R + v$	$128 + K$	130×2

In Figures 5.3 through 5.5, we assume all transmitting nodes broadcast to all R receivers and transmit their messages according to a round-robin schedule. Key size K is eighty bits for TESLA. For simplicity in validity voting, we assume nodes transmit a vote for the v message values most recently received from the bus. For all receivers to obtain one vote on each sample of each message type, each node votes on the two most recent message values ($v = 2$). For two votes, each node votes on the three most recent message values ($v = 3$). For four votes, each node votes on the five most recent message values ($v = 5$). Chapters 6 and 7 detail application of techniques to more complex message schedules.

Figure 5.3 shows an example of the per-packet authentication overhead required as we vary the per-packet assurance, using ten receivers (R = 10). This example omits any added bandwidth due to packet fragmentation.

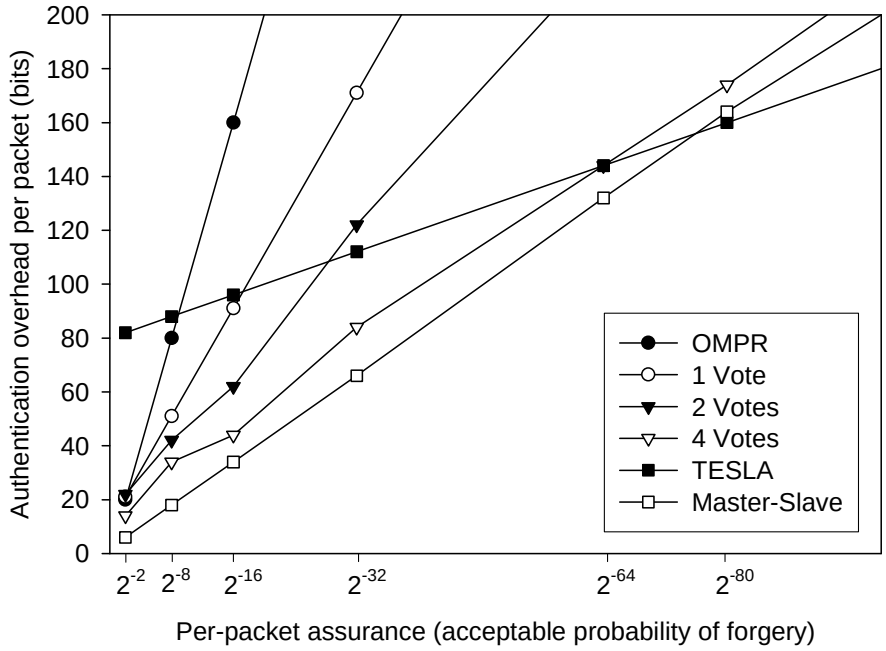

Figure 5.3. Authentication bits per packet varying per-packet assurance. Ten total receivers.

With no trusted master, Figure 5.3 shows that one MAC per receiver and validity voting have lower bandwidth consumption than TESLA in networks requiring weak per-packet assurance. This characteristic makes one MAC per receiver and validity voting most applicable to time-triggered embedded control networks with sampling rates faster than time constants in the system. Eventually, stronger per-packet assurance levels forces the size of the MAC tags to require more bits than the key that TESLA releases. For ten receivers, one MAC per receiver requires less bandwidth than TESLA for per-packet assurances of 2^{-8} or higher. Figure 5.3 also illustrates how voting can be used to lower the required authentication bandwidth at the cost of some added complexity and lower loss tolerance. By using one or two votes, we can achieve a 2^{-16} (or even

2^{-32} if using four votes) probability of per packet forgery success while using less authentication bandwidth than required by TESLA.

Figure 5.3 shows that for an embedded control network with ten receivers, using typical sampling rates, one MAC per receiver and validity voting can be used to achieve strong system level assurances using our time-triggered authentication approach. Typical embedded control networks often follow the rule of thumb of sampling message types at least ten times within a system deadline or the rise time of a control output [Franklin02][Kopetz97]. Using an assurance probability of 2^{-8} per packet, we achieve a probability of induced system failure of 10^{-9} per message round if receivers authenticate over at least four message samples. For a message type sampled once per millisecond, an expected failure rate of 10^{-9}/hour can be achieved if the receiver authenticates over at least seven samples using a 2^{-8} per packet assurance probability. For systems requiring stronger per-packet assurance, such as 2^{-16}, an induced system failure probability of 10^{-9} per message round can be achieved by authenticating over just two samples. For a message type sampled once per millisecond, a receiver would need to authenticate over at least four samples using this level of per-packet assurance to achieve an expected failure rate of 10^{-9}/hour. Embedded control networks following the rule of thumb of sampling message types at least ten times within a system deadline or the rise time of a control output should be able to authenticate state changes and actuations over four to seven samples, while still maintaining a margin of error for unanticipated operating conditions (e.g., transient packet losses). We use 2^{-8} and 2^{-16} as examples in our analysis in following sections. Systems with less stringent system level assurance requirements could use weaker levels of per-packet assurance (e.g., 2^{-2} or 2^{-4}).

Figure 5.3 also shows that TESLA has the lowest per-packet authentication cost for networks with ten receivers requiring per-packet assurance stronger than 2^{-80}. This makes TESLA the most

efficient approach of the four for systems where a change in system state must be authenticated over a single or very few packets with iron-clad security guarantees.

For systems in which a trusted master node is available, using a master-slave approach can achieve extremely low authentication overhead per packet as compared to the three other approaches. While a master-slave approach scales well with respect to per-packet assurance, the size of the authenticators eventually becomes larger than the bandwidth required by TESLA. Note that Table 5.1 and Figure 5.3 show the per-packet bandwidth overhead incurred by each slave node. The master's transmitted hash requires an additional message type to be broadcast on the network.

5.4.2 Scalability with respect to receivers

Next, we examine how each approach scales with respect to the number of receivers while fixing the per-packet assurance level. While a typical embedded network only has tens of nodes, approaches that scale linearly with the number of receivers eventually become inefficient in comparison to schemes like TESLA. Again, we calculate the total number of authentication bits per packet in the same way as the previous section (Table 5.1). In the example in Figure 5.4, we fix the per-packet assurance at a probability of 2^{-8} and vary the number of receivers. In Figure 5.5, we fix the per-packet assurance at a probability of 2^{-16}. For this example, we also ignore added bandwidth overhead due to packet fragmentation.

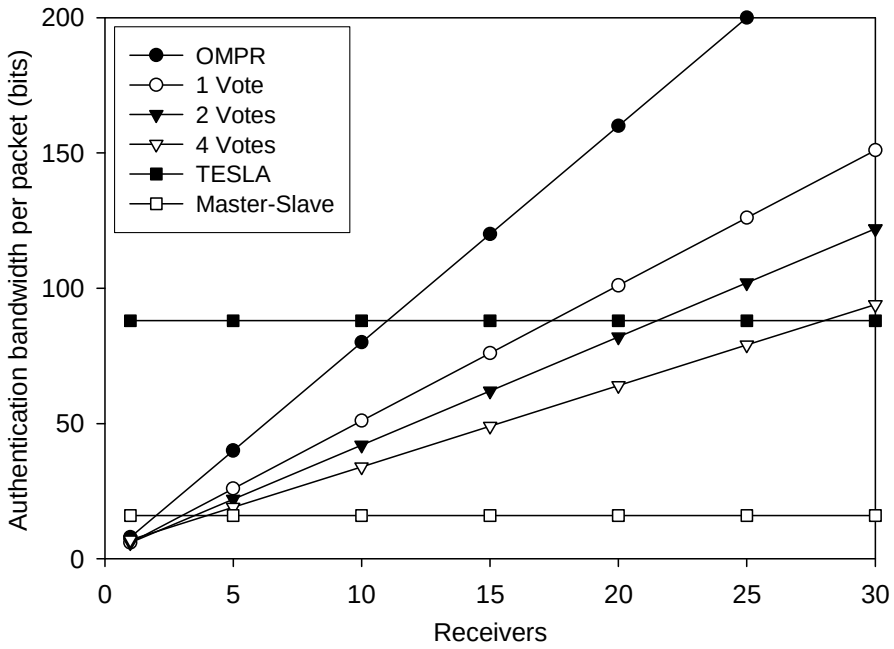

Figure 5.4. Authentication bits per packet varying number of receivers. Probability of per-packet forgery success fixed at 2^{-8}.

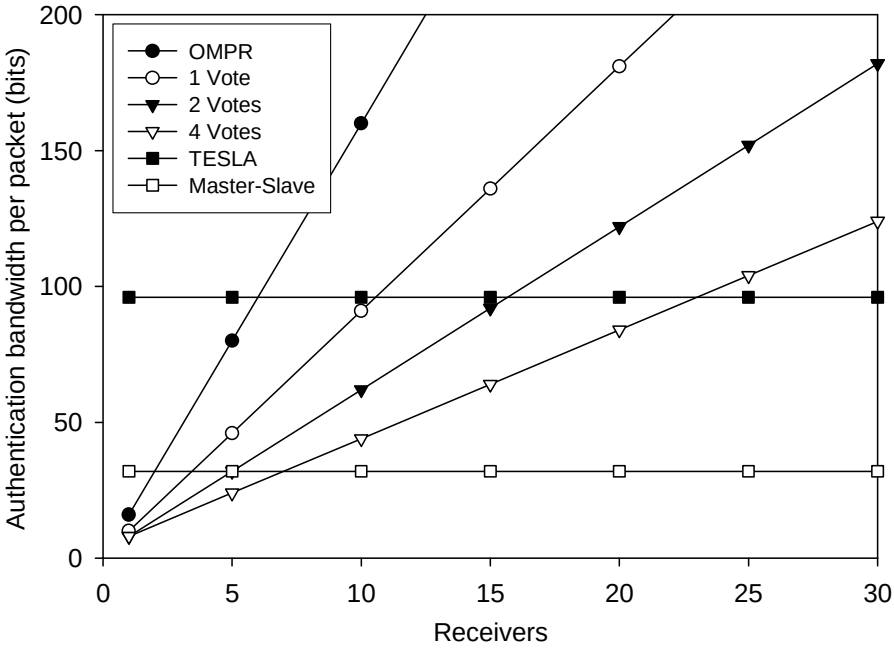

Figure 5.5. Authentication bits per packet varying number of receivers. Probability of per-packet forgery success fixed at 2^{-16}.

For networks with no trusted master, Figures 5.4 and 5.5 shows that one MAC per receiver and validity voting require low authentication bandwidth per packet in networks with moderate numbers of receivers. These approaches can be applied in embedded control networks with moderately few receivers, commonly eight to sixteen (the example automotive network workload in Section 7 has a maximum of twelve receivers for any message type).

Similarly to Figure 5.3, Figures 5.4 and 5.5 also illustrates how validity voting enables better scaling with respect to receivers. This allows a sender to authenticate a value to more receivers for a given bandwidth than when using one MAC per receiver alone.

These figures also illustrate that validity voting reduces MAC tag sizes by a greater number of bits for stronger per-packet assurances. For a per-packet assurance of 2^{-8}, adding a single vote decreases the number of authentication bits per receiver by three bits. Whereas, for 2^{-16}, adding a single vote decreases the authentication bits per receiver by seven bits.

This figure shows that TESLA is the most bandwidth-efficient non-master approach for high numbers of receivers. In the example with a fixed per-packet assurance of 2^{-16}, using OMPR becomes less efficient than TESLA after there are more than six receivers. However, using four votes remains more efficient than TESLA until there are more than 23 receivers in this example. For a per-packet assurance of 2^{-8}, OMPR becomes less efficient than TESLA after more than 11 receivers, and validity voting with four votes becomes less efficient than TESLA after more than 28 receivers.

For networks with the option of using a trusted master, a low per-packet authentication overhead can be maintained regardless of the number of receivers. However, this also assumes that each of these receivers also transmits meaningful data upon which it can "piggy-back" the necessary authenticators. Receivers which would otherwise not transmit are required to send authenti-

cation data. Figures 5.4 and 5.5 show only the authentication bits required for each slave node's transmissions (where each packet contains two MAC tags). A master's message would use fewer authentication bits, only containing a hash the size of a single MAC tag.

5.4.3 Loss tolerance

We experimentally tested the loss tolerance of each approach using an embedded CAN network simulator written in Java [Koopman12]. For this work, we use a network of six nodes broadcasting according to a round-robin schedule. Each node takes turns broadcasting a single message type consisting of a sixteen bit data value, associated authentication data and CAN packet overhead. We selected a per-packet assurance of 2^{-8} as an example of a per-packet assurance probability one might assign to message types in an embedded control.

We implemented all four authentication schemes in the simulator. Every node authenticates each of its packets to all five other nodes in the network. For one MAC per receiver and validity voting, each sender includes one MAC tag for each receiver. With a per-packet assurance of 2^{-8}, at most eight authentication bits are required per tag. Thus, all value and authentication data for these two approaches fit within a single data payload. We tested validity voting using one vote, two votes, and four votes. Master-slave authentication required an additional master node to be added to the simulation, for a total of seven nodes and message types. With two tags per packet in the master-slave approach, no packet fragmentation occurred. TESLA required two CAN packets to transmit the value, MAC tag, and key for each message type in each round. We simulated TESLA with recovering previously lost keys, and without recovering previously lost keys.

We simulated the effects of a symmetric omissive fault model [Azadmanesh00] on network packets to observe each approach's sensitivity to packet losses and how long each takes to recov-

er after transient faults cease. In a symmetric omissive fault model, either all nodes receive a packet broadcast on the network or none receive it. This type of fault may occur due to network blackouts or if a node simply fails to transmit during a message period. We apply this fault model by having the network drop a percentage of packets during execution. We use the built-in fault injection capabilities of the simulator to inject omissive faults during execution. The simulator applies this drop percentage uniformly across all message types. All injected faults affect only a single packet; prolonged effects require multiple faults.

Fragility - Sensitivity to packet losses - During execution, nodes recorded the overall ratio of authenticated data values to the total number of values transmitted (i.e., network goodput normalized over total execution time) to identify sensitivity to packet loss. Increasing authentication dependencies among nodes and packets make approaches more sensitive to each packet loss, causing the loss of multiple data values when a single packet is lost. Figure 5.6 shows the ratio of authenticated data values to the total transmitted as we vary the percentage of dropped packets. For each data point, we ran the simulator for a sufficient number of message rounds to observe at least one hundred drop events.

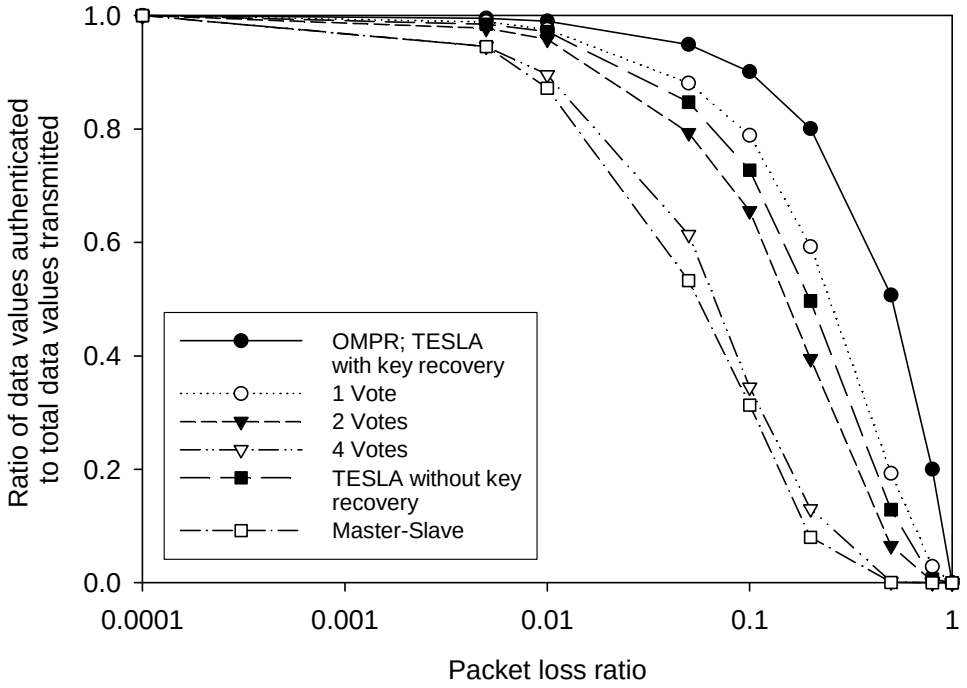

Figure 5.6. Ratio of packets authenticated to total transmitted varying packet loss.

One MAC per receiver has the highest ratio of accepted data values to transmitted values as packet loss increases, because it has no inter-node or inter-packet dependencies. Thus, this scheme represents an ideal bound on the maximum ratio of processed data values to transmitted data values.

TESLA (with key recovery) had the same loss tolerance as OMPR. Since keys are initially computed by iterating a hash function, a receiver can simply recompute lost keys if a subsequent key is eventually received. Thus, a message value was only lost if the packet containing the value suffered a transmission error. Loss of only the key temporarily prevented verification of a value until another key was successfully received.

TESLA (without key recovery) is more sensitive to packet loss than one MAC per receiver due to the use of time-delayed key release for authentication. A data value will be lost if the

packet containing that data value, or either of the two fragment-bearing packets containing the subsequent key material are lost.

When using validity voting, increasing the number of votes increases sensitivity to network faults. If any vote that confirms a value is lost, then the packet containing the value voted upon is also lost. If one of these confirmation packets is lost, then all values attested to will also be marked as lost.

Master-slave authentication suffers the greatest degradation in processed data values as packet losses increases, because all packets from one round must be received to verify the previous round. If the master node's hash value or any subsequent MAC tag used to verify the hash are lost, then all values in the prior message round are also lost.

Figure 5.6 illustrates that schemes which have few or no dependencies among nodes or packets for authentication, such as one MAC per receiver, TESLA, and validity voting with one or two votes, are best suited for lossy networks. For systems deployed in environments where little network interference is anticipated, approaches which require more interaction among nodes and message types for authentication can be used. Validity voting where most nodes participate in voting on a majority of each others' messages or master-slave authentication could be used in systems where transmission errors are sufficiently rare.

Robustness - Recovery time from transient faults - While most approaches have some sensitivity to packet losses, all approaches recover quickly from transient faults. When used in safety-critical applications, an authentication approach must be able to resume authentication quickly after a transient fault ceases. An example of such a fault is a temporary network blackout due to an electric motor starting. We experimentally tested the length of time from the point at which a transient packet losses ceased to the point at which the first message value transmitted after the

fault was successfully authenticated. To simulate these types of faults, we deterministically injected a single lost packet for a message type and measured the time until authentication resumed for that message type. Additionally, we dropped packets for all message types long enough to stop all authentication, then ceased all interference simultaneously to simulate the end of a network blackout. We observed similar recovery times in both cases and recorded the worst case.

We do not consider recovery of packets lost during the fault, as nodes in embedded control networks act on the freshest data values concerning the current system state to update outputs and actuator positions. Nodes discard stale data values after short period of time. Also, this work does not address malicious denial of service attacks on these networks.

One MAC per receiver resumes authentication immediately upon the receipt of the next message value after a transient network fault ceases. This recovery time is ideal due to no dependencies.

While validity voting is more sensitive to packet losses because of inter-node dependencies, it automatically resumes authentication after a message value and all subsequent votes are received. In this simulation, validity voting recovered within one message round. Verification of a value never depends on prior packets. Only subsequently received packets are used to authenticate a value. Votes are scheduled to be received within one message round of the value they are associated with.

TESLA recovers from transient network faults as soon as a data value and the subsequent key are received. In our simulation this occurs in just slightly over one message round. However, this delay could be reduced by scheduling a message type's keys to be released later in the message round after its value is released. Upon receiving a data value for any message type, a receiver can authenticate that value once the associated key is released in the subsequent message round. Re-

ceivers must also recompute any lost keys in order to verify the authenticity of keys and values transmitted after the fault ceases. Thus, sufficiently long network blackouts might increase recovery time.

Master-slave takes at most three message rounds to recover from a transient fault in our simulation. Once the fault ceases, receivers must wait until the beginning of a new message round. Receivers can begin verifying message values again once all values in that round, the master node's hash, and the tags in all packets in the following round have been received.

5.4.4 Node compromise and failure

Reliance on other nodes for authentication also reduces an approach's tolerance to node compromise or failure. TESLA and one MAC per receiver have perfect tolerance to node compromises or failures. An attacker controlling a compromised node can only spoof message values that would be sent from that node. A node failure would not prevent any other messages from being authenticated other than ones transmitted by the failed node.

Validity voting (Section 4) can only tolerate a fixed number of compromised nodes, specified at design time. An attacker might use a compromised node to assist in a message forgery attempt by casting a positive vote for that message. By tolerating *w* votes are compromised out of a *z* total votes, the per-packet assurance is defined by *z* - *w* useable votes (see equation (3) in Section 4). If an attacker is able to compromise more than *w* voting nodes in the system, they might be able to cause message forgeries to succeed more often than defined failure requirements for the system.

Baseline validity voting as described in Section 4 does not account for permanent node failures. Tolerance to failed nodes needs to be added. Section 8 describes how to tolerate failed nodes, using methods like group membership.

Master-slave authentication's tolerance to node compromise relies primarily on the master node remaining uncompromised. The master node is a single point of failure. If an attacker compromises the master node, the attacker has complete control over the network and can transmit any value it wishes. However, if the master node remains uncompromised, the approach retains perfect tolerance to compromise of any slave node. A compromised slave node can only spoof messages that would be sent from that node.

Permanent node failures have a more severe impact on master-slave authentication. In the event the master node fails, no authentication is possible. If a slave node fails, it will not release the MAC tags necessary for other nodes in the network to validate previous message rounds. To resolve this, a master node might periodically broadcast the current set of nodes it believes to be operating correctly. Thus, receivers can recompute the master's hash value over the tags released by correctly operating nodes.

5.5 Discussion

Table 5.2 summarizes the results of this chapter, discussing characteristics of each technique and types of embedded networks they best apply to.

Our analysis shows that the most bandwidth efficient approach depends primarily on the number of receivers, and is influenced to a lesser extent by per-packet assurance levels in networks where no trusted master is available. For example, one MAC per receiver and validity voting are the most bandwidth efficient approaches for networks characterized by few receivers and

weak per-packet assurance levels. TESLA and validity voting using many votes are the most bandwidth efficient approaches for very large numbers of receivers or strong per-packet assurance levels. A master-slave approach is also very bandwidth efficient, assuming a trusted master node is available. We also show that despite some approaches being more sensitive to transient packet losses, all approaches recover automatically within one to three message rounds. Lastly we find approaches with no inter-node dependencies for authentication, such as one MAC per receiver and TESLA, are most robust to node compromises or failures.

Table 5.2. Summary of authentication technique characteristics

	Summary
One MAC per receiver	• Best applied to embedded control networks characterized by very few receivers and weak per-packet assurance levels. • Perfect tolerance to transient packet losses. • Nodes resume authentication immediately after transient network failures cease. • Perfect tolerance to node compromise or failure.
Validity voting	• Best for systems with few receivers; can provide strong per-packet assurance by increasing votes. Enables authentication to more receivers or stronger per-packet assurances than one MAC per receiver using the same number of bits. If using many votes, validity voting is competitive with TESLA for scalability even to strong per-packet assurance levels. • Increasing voting also makes this approach more sensitive to packet losses. • Authentication resumes within one message round after transient network faults cease. • Only tolerates a fixed number compromised nodes. Node failures might require network reconfiguration.
TESLA	• Best for systems characterized by many receivers and very strong per-packet assurance levels. • Has higher per-packet authentication overhead than one MAC per receiver and validity voting when applied to few receivers and weak per-packet assurance levels. • Time-delayed key release slightly decreases loss tolerance due to inter-packet dependencies. • Authentication can resume within one message round. • Perfect tolerance to node compromise or failure.
Master-slave	• Requires a trusted master node. • Scales well to any number of receivers (so long as none are silent receivers). Also scales well with respect to per packet assurance levels. • Very sensitive to packet losses, because all nodes participate in verifying each message round. • Recovers from transient packet losses within three message rounds. • Master node is a single point of failure. Perfect tolerance to node compromise so long as only slave nodes are compromised. Failed slave nodes require reconfiguration out of the system by the master node.

6 Evaluation - Simulated elevator control network

We implemented time-triggered authentication in a simulated embedded control network of an elevator system. The embedded network simulator is a bit-level accurate CAN protocol simulator, allowing controllers to communicate using periodic messages [Koopman12]. In this proof of concept, we first identify safety requirements of the system and possible attacks in which message forgeries could induce system failures that could violate those safety requirements or cease elevator operations. We then apply time-triggered authentication in conjunction with all four techniques described in previous sections (one MAC per receiver, validity voting, TESLA, and master-slave) to prevent such attacks and examine the performance impacts of each.

The embedded network simulator has been used in several research projects as well as the project component of graduate level course work in the Electrical Engineering Department of Carnegie Mellon University. Examples of research projects that have used the embedded network simulator (or variations thereof) include graceful degradation of distributed embedded systems [Nace02][Shelton03] and embedded network gateway survivability [Ray09]. The simulator is also used in the project component of the Carnegie Mellon University graduate course 18-649 Distributed Embedded Systems. Students use the embedded network simulator to design an elevator system. Associated project tasks include development of system requirements, design and implementation of state machines for controllers, analysis of bandwidth consumption, and robustness testing of system designs to injected faults. All code and design documentation for the elevator and the underlying network simulation framework are available [Koopman12].

While not the most obvious example for a security analysis, we selected the elevator system as an implementation platform because it is representative of safety-critical embedded networks

in industry. This system contains nodes running control loops that consume both reactive control messages and state-changing messages. The simulation uses a CAN bus to broadcast periodic messages to receivers. Any node (malicious or not) which spoofs an input to a safety-critical node could cause the system to violate safety requirements. Such a violation could potentially result in injury or death of users in a real system. The elevator design also has performance and passenger comfort requirements that a system designer may also protect from malicious attacks to a lesser degree.

6.1 Network simulator framework overview

The elevator system executes on top of a CAN network simulation framework, written in the Java programming language.

The network simulation is built around an event queue that acts as the physical layer of the network and controls all time-related actions (e.g., passenger behaviors, control loop executions and message broadcasts) in the simulation. The queue maintains an ordered set of events and associated times to execute them. During execution, the event queue increments an internal counter that represent clock ticks (each clock tick represents one nanosecond) that have passed. At each clock tick, the event queue executes the events for that time.

At initialization, the simulation builds a set of nodes (sensors, actuators, and controllers) and registers them with the central event queue. Each node connected to the network is an instance of a Java class that exchanges data through the event queue. To simulate periodic control loop execution, each node registers a function callback and simulation time with the event queue. When the event queue reaches that time, it executes the function callback. The node then reads new inputs, updates internal state variables, and updates outputs. Once the control loop function com-

pletes, the node re-registers with the event queue for its next control loop execution. One limitation of the simulator is that control loop executions occur "instantly." The simulator does not attempt to model execution time.

To minimize processing requirements, the simulation models all physical signals and CAN network packets as semi-public state variables that are updated at discrete intervals. All network messages and physical signals are propagated to receivers at predefined periods. Typically, these periods match the control loop period of the message's source. While this might be considered a limitation of the simulator, in real embedded control networks most inputs (including continuous analog inputs) are sampled periodically. Nodes register message types they output and their periods with the event queue. Nodes also subscribe to a set of messages through the event queue. At the predefined frequency, the event queue pulls data from the source node and propagates the messages to mailbox variables. Each node accesses the newest copy of each network message type and physical signal through these mailboxes. The event queue can also model the CAN protocol at the bit-level of the physical layer. It models bus arbitration when multiple nodes attempt to broadcast on the bus at the same time. It will also models other aspects of the CAN protocol, such as bit stuffing. This allows for detailed bandwidth analysis for projects using the network simulation framework.

6.2 Elevator system overview

The simulated elevator system services a building with eight floors. Passengers "arrive" in the simulation at predefined floors and times, and then press the hall call button (up or down) at that floor. A centralized control system, called the dispatcher, monitors hall and car call button messages on the network and determines which floor to service next for optimum performance to minimize overall passenger delivery time. The dispatcher updates the desired floor, which is

broadcast at regular intervals over the network to the other controllers. Based on the desired floor, the drive controller commands the drive motor to raise or lower the car within the hoistway to that floor. Once the elevator arrives at the desired floor, the door controllers command the door motors to open and close the doors for each hallway appropriately. The car has front and back doorways (each with left and right doors that open simultaneously), allowing the elevator to service a front hallway, a back hallway, or both at each floor. Once inside the elevator car, a passenger waits for the doors to close, and then presses a car call button to be delivered to the corresponding floor. Again, the dispatcher determines the next floor to be visited and the drive moves the car to that floor. The car position indicator displays the current floor to the passengers so they can exit the car at the correct floor.

The elevator simulation consists of a set of nodes (sensors and controllers) that communicate over a simulated CAN bus using periodic messages. Table 6.1 lists the source nodes, message types they broadcast, message period, replication of source nodes, and a brief description of the contents of the message. Some nodes (and their respective messages) are replicated. The replication column indicates how many copies of that message are broadcast. "Floor" indicates that a copy of that node is present at every floor. "Hall" indicates that a copy of that node exists for both front and back halls. "Side" indicates that a copy of the node is present for both left and right sides, referring to the two doors of each entrance to the elevator car. "Direction" indicates that a copy of the node exists for both up and down directions.

Table 6.1. Elevator message dictionary. Contains message types, source nodes, periods, replication, and descriptions. [Koopman12]

Sensors				
Source Node Name	**Message Name**	**Period (ms)**	**Replication**	**Description**
At Floor Sensor	AtFloor	50	10 (floor, hall)	Boolean value indicating if the car is currently at that floor and hallway.
Car Level Position Sensor	Car Level Position	50	1 (single)	Integer value that provides the vertical position of the car within the hoistway in millimeters.
Door Closed Sensor	Door Closed	50	4 (hall, side)	Boolean value indicating if a particular car door is completely closed.
Door reversal Sensor	Door Reversal	10	4 (hall, side)	Boolean value indicating whether an object is blocking a door, preventing it from closing.
Weight Sensor	Car Weight	50	1 (single)	Integer value that provides the current weight of the contents of the elevator car.
Door Opened Sensor	Door Opened	50	4 (hall, side)	Boolean value indicating if a particular car door is completely open.
Hoistway Limit Sensor	Hoistway Limit	50	2 (direction)	Boolean value indicating if the car has exceeded the top or bottom of the hoistway.
Controllers				
Source Node Name	**Message Name**	**Period (ms)**	**Replication**	**Description**
Safety Monitor	Emergency Brake	50	1 (single)	Boolean value indicating if the safety monitor has detected a safety violation and engaged the emergency brake.
Drive Control	Drive Command	10	1 (single)	Contains integer values providing the current speed and direction of the drive (i.e., how fast the car is moving).
Door Control	Door Motor Command	10	4 (hall, side)	Integer value indicating the current command from the door controller to door motor (stop, close, or open).
Car Position Indicator	Car Position	50	1 (single)	Integer value providing the current floor being displayed to passengers within the elevator car.
Dispatcher	DesiredFloor	50	1 (single)	Integer values providing the next floor and direction the car should be commanded to travel to.
Hall Button	Hall Call	100	17 (floor, hall, direction)	Boolean value indicating if the call button in the corresponding floor/hallway/direction has been pressed by a passenger waiting in a hall.
Car Button	Car Call	100	10 (floor, hall)	Boolean value indicating if the call button for the corresponding floor/hallway has been pressed by a passenger inside the car.

6.3 Supporting system requirements

The elevator system design we built our authentication mechanisms into fulfills a series of high level system requirements and safety requirements. The creators of the simulated elevator de-

signed it to deliver passengers efficiently while still being robust to non-malicious failures. However, the design does not prevent maliciously induced failures due to message forgeries.

6.3.1 Safety requirements

Our highest priority for message authentication is to prevent induced failures that violate safety requirements. At no point should an attacker be able to successfully forge a sufficient number of message values to induce such a failure. We use the following requirements from original set of safety requirements for the elevator [Koopman12]:

R-S1. All doors shall remain closed while the elevator is between floors.

R-S2. Doors shall remain closed if there is no landing for that hallway at a floor.

R-S3. Door motors shall not be commanded to any value other than open for any longer than 200 milliseconds if a door reversal is detected.

R-S4. If the elevator car is overweight, the drive speed shall be set to zero, and the direction to stop.

R-S5. The elevator car shall not exceed hoistway limits.

To support requirements R-S1 through R-S5, we address attacks where the attacker creates or modifies messages with values that do not reflect the real state of the elevator system, controllers subsequently act upon those falsified messages and place the system in a state which violates one or more of these safety requirements. We identify state transitions in controllers that might be targeted to violate these requirements and apply authentication to the associated message types those transitions are based upon. We also identify time bounds in terms of the number of message samples if an attack must be detected within a certain amount of time (e.g., a door controller can expect to receive twenty samples of the Door Reversal message type within the 200 millise-

cond time limit defined by R-S3). This gives us maximum sizes for history buffers when using time-triggered authentication.

We omit one safety requirement from the original list of elevator safety requirements [REF 18-649]. The omitted requirement defines an acceptable drive acceleration profile and is not affected by network message traffic. It is addressed completely within the design of the drive controller.

6.3.2 High level system requirements

Our second priority for message authentication is to prevent induced failures which prevent the elevator system from accomplishing its mission: delivering passengers. We use the following system level requirements from the initial creators of the elevator [Koopman12]:

R-T1. The elevator shall deliver all passengers eventually.

R-T2. Any unsafe condition shall cause an emergency stop.

R-T3. An emergency stop should never occur.

To support requirement R-T1, we use authentication to ensure that an attacker cannot stealthily perform a denial of service attack using message forgeries. Without authentication, a falsified value for a message type could cause the elevator to stop delivering passengers without triggering an observable failure. Further, an attacker could cease falsifying messages to allow the system to return to normal operation without being detected. As with safety requirements we identify state transitions and associated message types that should be protected to support this requirement. There is no bounds on time limits for detecting these types of forgeries. Authenticating many samples will eventually allow receivers to detect such attacks. However, we do not de-

fine explicit or implicit history buffer sizes for message types that could be used for such denial of service attacks.

To support requirements R-T2 and R-T3, we authenticate messages to the safety monitor. The safety monitor node watches physical signals and network messages to detect when the system state violates these requirements. The safety monitor then triggers the emergency brake. If the safety monitor detects too many invalid authenticators on falsified inputs, it can trigger the emergency brake. While this could be a denial of service attack on the elevator, triggering the emergency brake is considered a safe action.

We omit high level requirements related to passenger satisfaction and optimization, as we are primarily concerned with safely delivering passengers.

6.4 Identifying messages and state transitions to protect

Next, we identify transitions within internal state machines of controllers which could violate our requirements due to spoofed messages. Thus, we can define which messages need to be authenticated.

There are six safety critical nodes in the system that require authentication of their inputs. The controllers responsible for actuations related to safety requirements are the door controllers and the drive controller. The safety monitor is responsible for engaging the emergency brake if necessary. Spoofed inputs to these nodes could cause undesired state transitions which could violate safety requirements.

There are also two mission critical nodes in the system that require authentication of inputs: the dispatcher and car position indicator. If an attacker spoofs messages to these nodes, they might cause the elevator to cease operation or deliver passengers to incorrect floors.

In the following sections, we perform a brief analysis to show the possible effects of message forgeries against each of these eight nodes. Then we provide the list of messages and the nodes to which they should be authenticated.

6.4.1 Door Controller

The general behavior of the door controllers is as follows:

- Monitor the Desired Floor message from the dispatcher to determine which floor and hallway the doors are expected to open at next.
- Once at the desired floor and hallway, open doors for that hallway completely.
- Wait until the dwell count down completes.
- Close the doors, unless the door reversal occurs or the car is overweight.

Figure 6.1 shows the state diagram for a door controller. Table 6.2 provides the guard conditions for each state transition. Door controllers execute their control loops and update their outputs every ten milliseconds.

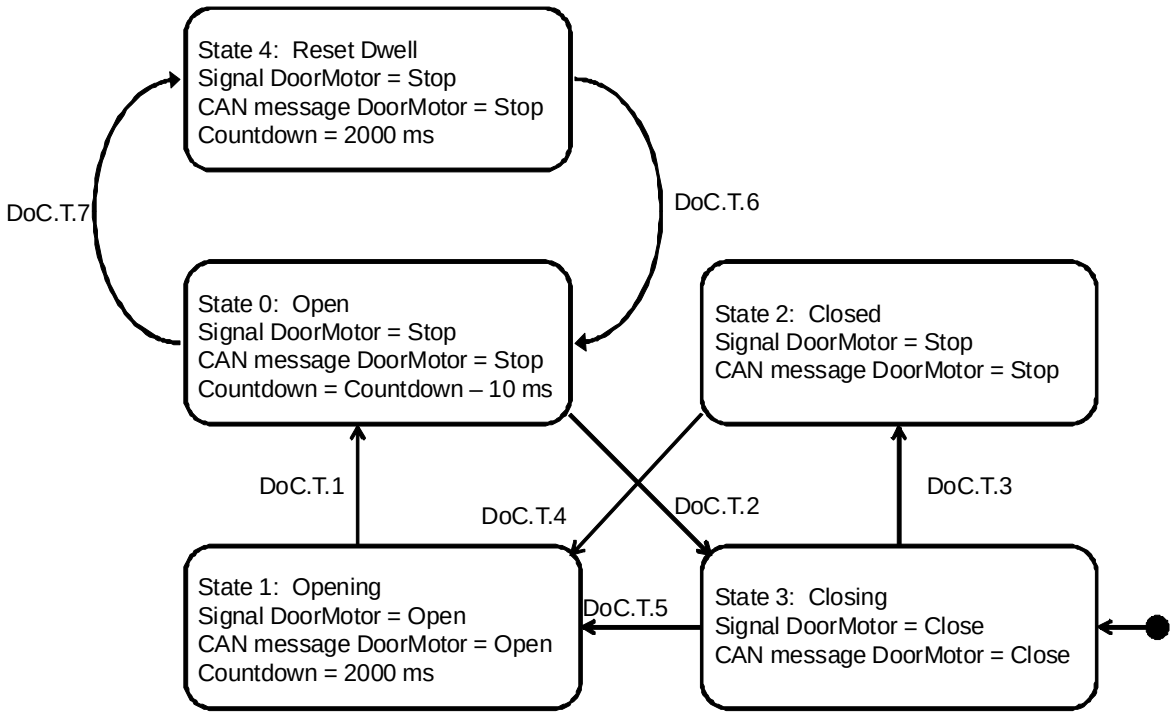

Figure 6.1. Door controller state diagram [Martin10].

Table 6.2. Door controller state transition guard conditions [Martin10].

Transition	Guard Condition
DoC.T.1	Door Open message for is true.
DoC.T.2	Dwell count down reaches zero AND All Door Reversal messages are false AND Car Weight message is less than max elevator capacity.
DoC.T.3	All Door Closed messages are true AND All Door Reversal messages are false AND Car Weight message is less than max elevator capacity.
DoC.T.4	At Floor message corresponding to Desired Floor message's floor and hallway is true AND All Door Motor Command messages indicate doors have stopped AND Drive Command message speed is zero.
DoC.T.5	Any Door Reversal message is true OR Car Weight message is greater than or equal to max elevator capacity.
DoC.T.6	No condition. Always take this transition.
DoC.T.7	Any Door Reversal message is true OR Car Weight message is greater than or equal to max elevator capacity.

To determine the effects of using message forgeries to force undesired state changes or deny normal state changes, we examined the effects of each. Table 6.3 summarizes the effects of forcing or denying each state transition and related messages. For each, we determine whether an attack could cause a possible denial of service (undetected by the system), violate a safety requirement, or have no effect.

Table 6.3. Effects of message forgeries to force or deny state transitions in door controllers.

State transition	Effects of forced transition	Effects of denied transition	Associated message types
DoC.T.1	Possible denial of service. Door only opens partially, if at all.	Possible denial of service. Door could remain closed indefinitely.	Door Opened
DoC.T.2	No effect. Door starts the closing process. However, closing occurs in next state.	Possible denial of service. Door could remain open indefinitely.	Door Reversal Car Weight
DoC.T.3	Possible denial of service or safety violation (Requirement R-S3). Doors could remain locked in position while door reversal is true.	Possible denial of service. Door controllers continue to command door motors to close, which prevents the drive from moving the car.	Door Closed Door Reversal Car Weight
DoC.T.4	Possible safety violation (Requirements R-S1, R-S2). Doors could open between floors or while car is in motion.	Possible denial of service. Car arrives at desired floor, but doors never open.	At Floor Desired Floor Door Motor Command Drive Command
DoC.T.5	Possible denial of service. Doors could be forced to repeatedly open.	Possible safety violation (Requirement R-S3). Doors could close on a passenger or object when a door reversal should occur.	Door Reversal Car Weight
DoC.T.6	No effect. Transition always taken.	No effect. Transition always taken.	None
DoC.T.7	Possible denial of service. Doors could be forced to remain open.	No effect. Door could start closing sooner.	Door Reversal Car Weight

State transitions DoC.T.3 and DoC.T.4 could be forced or denied to create a system state which violates safety requirements. For transitions DoC.T.3 and DoC.T.4, forcing the transition causes a discrete state change could be a safety violation. Triggering DoC.T.3 could violate safety requirement R-S1 and R-S2, while DoC.T.4 could violate R-S3. Using time-triggered authen-

tication, we designate the associated messages as state changing message types for these transitions. For these transitions to complete, the door controller must be sure that the values of each message type have not been tampered with prior to committing the transitions. Door controllers will retain an explicit history buffer for the associated message types in memory for use with these two transitions. There is no maximum number of samples for explicit history buffers; increasing the size of these buffers simply delays the corresponding state transitions.

For transition DoC.T.5, during each control loop execution that an attacker successfully denies the transition will cause the controller to command the door motor to continue to close the door on a passenger obstructing the doorway. After a sufficient amount of time (200 milliseconds), an attacker will have successfully caused the system to violate safety requirement R-S3. Using time-triggered authentication, we designate the associated message types as reactive control message types for DoC.T.5. Door controllers do not keep an explicit history buffer for the associated message types for this transition. Instead, we rely on the implicit history buffer that exists for each door. An attacker must successfully forge at least 20 consecutive samples of the Door Reversal message type to deny this state transition long enough to violate a safety requirement. There is no maximum time limit before the doors reopen if the car is overweight.

Transitions DoC.T.1, DoC.T.2, DoC.T.3, DoC.T.4, DoC.T.5, DoC.T.7 could be forced or denied to perform a stealthy denial of service attack, stopping the doors from performing their normal function. Any of these could be exploited to violate requirement R-T1, however there is no time bounds for detecting denial of service attacks.

Implementation notes for time-triggered authentication in door controllers - For the door controllers, we implemented explicit history buffers for the At Floor, Door Closed, Door Motor Command, Door Reversal, Desired Floor, and Drive Command message types for use with tran-

sitions DoC.T.3 and DoC.T.4. The dynamics of the elevator system create an implicit history buffer for the Door Reversal message type for use with DoC.T.5. Lastly, we also authenticate the Door Open message type to monitor for forgery attempts designed to perform denial of service. All of these message types are authenticated to the door controller.

If a door controller detects forgery attempts (invalid authenticators) on messages attempting to force state transitions DoC.T.3 or DoC.T.4, the door controller resets their history buffer and aborts the state change. For DoC.T.5, if it detects forgery attempts on its associated message types, it reopens the doors to avoid a safety violation. In an implementation in a real system, the system designer can take whatever action is appropriate and safe if such a malicious fault is detected.

6.4.2 Drive Controller

The general behavior of the door controllers is as follows:

- Monitor Desired Floor message from the dispatcher to determine which floor to travel to next.
- If not currently at the desired floor, accelerate to slow speed (0.25 m/s) towards the desired floor.
- Once at slow speed, accelerate to fast speed (5.0 m/s) towards the desired floor.
- Once the car has reached the commit point, begin decelerating to slow speed.
- Continue at slow speed until the desired floor has been reached, then stop the car.

Figure 6.2 shows the state diagram for the drive controller. Table 6.4 provides the guard conditions for each state transition. The drive controller executes its control loop and updates its output every ten milliseconds.

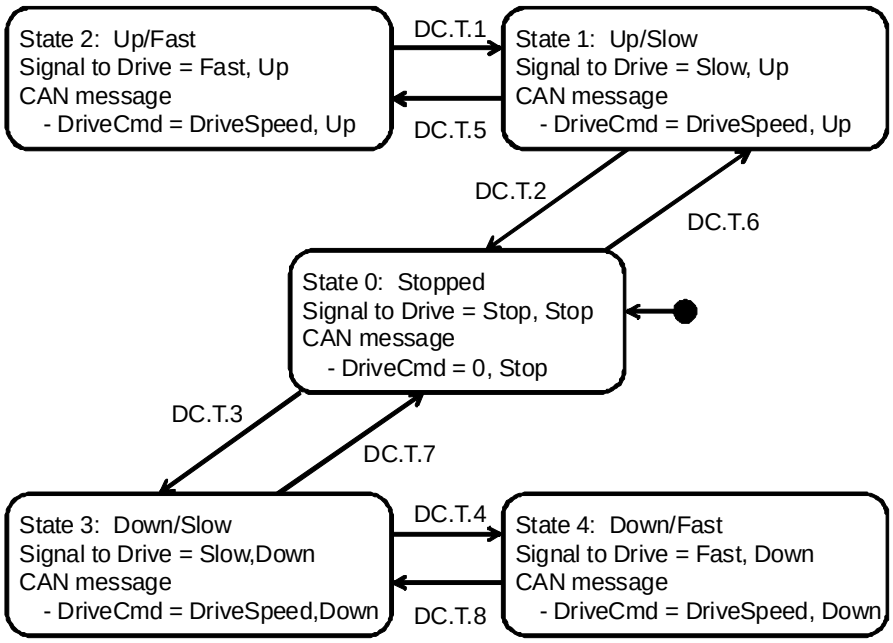

Figure 6.2. Drive controller state diagram [Martin10]

Table 6.4. Drive controller state transition guard conditions [Martin10].

Transition	Guard Condition
DC.T.1	Car Level Position message indicates commit point reached OR Any Door Closed message is false OR Any Door Motor Command message is not stop OR Emergency Brake message is true OR Hoistway Limit message is true OR Car Weight message is greater than max car capacity
DC.T.2	At Floor message for desired floor is true OR Any Door Closed message is false OR Any Door Motor Command message is not stop OR Emergency Brake message is true OR Hoistway Limit message is true OR Car Weight message is greater than max car capacity
DC.T.3	At Floor message for desired floor is false AND Desired floor is below current position AND All Door Closed messages are true AND All Door Motor Command messages are stop AND Emergency Brake message is false AND Hoistway Limit message is false AND Car Weight message is less than or equal to max car capacity
DC.T.4	Car Level Position message indicates commit point not reached AND Drive Command message indicates slow speed has been reached AND All Door Closed messages are true AND All Door Motor Command messages are stop AND Emergency Brake message is false AND Hoistway Limit message is false AND Car Weight message is less than or equal to max car capacity
DC.T.5	Car Level Position message indicates commit point not reached AND Drive Command message indicates slow speed has been reached AND All Door Closed messages are true AND All Door Motor Command messages are stop AND Emergency Brake message is false AND Hoistway Limit message is false AND Car Weight message is less than or equal to max car capacity
DC.T.6	At Floor message for desired floor is false AND Desired floor is above current position AND All Door Closed messages are true AND All Door Motor Command messages are stop AND Emergency Brake message is false AND Hoistway Limit message is false AND Car Weight message is less than or equal to max car capacity
DC.T.7	At Floor message for desired floor is true OR Any Door Closed message is false OR Any Door Motor Command message is not stop OR Emergency Brake message is true OR Hoistway Limit message is true OR Car Weight message is greater than max car capacity
DC.T.8	Car Level Position message indicates commit point reached OR Any Door Closed message is false OR Any Door Motor Command message is not stop OR Emergency Brake message is true OR Hoistway Limit message is true OR Car Weight message is greater than max car capacity

Again, we examined the effect of message forgeries intended to force or deny state transitions. Table 6.5 summarizes the effects of forcing or denying each state transition and related messages. For each, we determine whether an attack could cause a possible denial of service (undetected by the system), violate a safety requirement, or have no effect. Undetected denial of service attacks violate requirement R-T1.

Table 6.5. Effects of message forgeries to force or deny drive controller state transitions.

Transition	Effects of forced transition	Effects of denied transition	Associated message types
DC.T.1	No effect. Drive decelerates to slow speed early before reaching commit point. This is safe, but slows performance.	Potential safety violation (R-S1, R-S4, R-S5, R-T2) Car might exceed hoistway limit or move while doors opening, emergency brake being engaged, or weight exceeding capacity.	Car Level Position Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.2	Potential denial of service. Drive could stop between floors.	Potential safety violation Same as denying DC.T.1.	At Floor Desired Floor Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.3	Potential safety violation (R-S1, R-S4, R-S5, R-T2) Car might begin moving while doors are open, emergency brake engaged, hoistway limit tripped, or weight exceeded.	Potential denial of service. Drive could never move to next desired floor.	At Floor Desired Floor Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.4	No effect. Drive will always attempt to go to fast speed between floors during normal operation. Drive does not instantly change speed. Next state tests whether drive should begin slowing down.	No effect. Drive will never reach fast speed. This is safe, but slows performance.	Car Level Position Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.5	No effect. Same as forcing DC.T.4.	No effect. Same as denying DC.T.4.	Car Level Position Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.6	Potential safety violation. Same as forcing DC.T.3.	Potential denial of service. Same as denying DC.T.3.	At Floor Desired Floor Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.7	Potential denial of service. Same as forcing DC.T.2.	Potential safety violation Same as denying DC.T.2.	At Floor Desired Floor Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight
DC.T.8	No effect. Same as forcing DC.T.1.	Potential safety violation. Same as denying DC.T.1.	Car Level Position Door Closed Door Motor Emergency Brake Hoistway Limit Car Weight

Forcing transitions DC.T.3 and DC.T.6 could trigger potential safety violations. Both of these initiate drive motor acceleration, causing a discrete change in elevator behavior (stopped to moving). Both of these could violate requirements R-S1, R-S4, R-S5, and R-T2. Using time-triggered authentication, we designate the associated messages as state-changing messages. The drive controller will retain an explicit history buffer for the associated message types for use with these two transitions. There is no maximum number of samples for explicit history buffers; increasing the size of these buffers simply delays the corresponding state transitions.

Denying transitions DC.T.1, DC.T.2, DC.T.7, and DC.T.8 could also cause the system to violate requirements R-S1, R-S4, R-S5, and R-T2. Using time-triggered authentication, we authenticate the message types associated with these state transitions as reactive control messages. The drive controller will not keep an explicity history buffer. Instead, we rely on the implicit history buffer that exists for these messages. An attacker must successfully forge multiple samples of a message to actually violate one of the requirements. For example, if attempting to cause the car to exceed the hoistway limits of the elevator shaft, an attacker could forge messages to force the car to travel an extra meter beyond the top or bottom floor. An attacker could either forge the Car Level Position message to force the car to travel at fast speed for an extra second (due to extra slack time built into the design for stopping). Alternately, they could forge the At Floor message to force it to travel an extra four seconds at slow speed. The sensors broadcasts the Car Level Position and At Floor messages every fifty milliseconds. This creates a maximum implicit history buffer size of twenty samples for the Car Level Position message and eighty samples for At Floor for their respective transitions. The other messages associated with these state transitions do not have defined maximum delays before slowing or stopping the elevator car. For example, while the emergency brake is engaged, there is no requirement defined for the simulation for

maximum time before the drive shuts off. For simplicity, we use the same maximum history buffer size as for Car Level Position.

Transitions DoC.T.2, DoC.T.3, DoC.T.6, or DoC.T.7 could be forced or denied to perform a stealthy denial of service attack, stopping the doors from performing their normal function. Any of these could be exploited to violate requirement R-T1, however there is no time bounds for detecting denial of service attacks.

Implementation notes for time-triggered authentication in door controllers - For the drive controller, we implemented explicit history buffers for verifying the authenticity of the At Floor, Desired Floor, Door Closed, Door Motor Command, Emergency Brake, Hoistway Limit, and Car Weight message types for use as with transitions DC.T.3 and DC.T.6. The elevator dynamics provide implicit history buffers for the At Floor, Car Level Position, Desired Floor Door Closed Door Motor Command, Emergency Brake, Hoistway Limit, and Car Weight message types for use with transitions DC.T.1, DC.T.2, DC.T.7, and DC.T.8. Lastly, we also authenticate these message types for the purpose of monitoring for forgery attempts intended to deny elevator operations.

If the drive controller detects forgery attempts (invalid authenticators) on messages attempting to force state transitions DC.T.3 or DC.T.6, the drive controller resets their history buffers and aborts the state change. For DC.T.1, DC.T.2, DC.T.7, and DC.T.8, if it detects forgery attempts on its associated message types, it slows or stops the drive accordingly to avoid a safety violation. In an implementation in a real system, the system designer can take whatever action is appropriate and safe if such a malicious fault is detected.

6.4.3 Safety monitor

In the simulation, the safety monitor is an omnipotent node that is able to access all system state variables. It does not have an internal state machine; it only monitors signals and messages and outputs a signal to inform the network if the emergency brake has been engaged.

The safety monitor in the simulation was originally created as a debugging mechanism to assist students in identifying safety violations. The safety monitor engages the emergency brake instantly if it detects a violation based on system state variables. The simulation represents the engagement of the emergency brake by throwing an exception which causes the simulation to halt with a description of the violation outputted to the screen. There is no maximum delay defined for engaging the emergency brake.

Implementation notes for time-triggered authentication in the safety monitor - In our implementation, the safety monitor verifies the authenticity of all message types it monitors: At Floor, Car Weight, Door Closed, Door Motor Command, Door Reversal, Drive Control, and Hoistway Limit. We do not implement any explicit history buffers. If the safety monitor detects an invalid authenticator, the simulation currently only logs the invalid authenticator and prints it to standard output. In a real system, such a safety monitor might trigger the emergency brake if too many invalid authenticators are observed. However, in the simulation, we do not trigger the emergency brake since it would cause the simulation to throw an exception and halt.

6.4.4 Dispatcher

The dispatcher is responsible for deciding what floor to go to and informing the rest of the network. Forging messages to this node can only cause a denial of service. The dispatcher consumes At Floor, Car Weight, Door Closed, Door Open, and Car Position messages. For brevity, we

omit a detailed analysis. However, forgery of any of these messages could prevent elevator operation; forgeries cannot induce a failure to violate safety requirements.

Implementation notes for time-triggered authentication in the dispatcher - In our implementation, the dispatcher verifies the authenticity of the At Floor, Car Weight, Door Closed, Door Open, and Car Position message types to monitor for forgeries intended to stop elevator operations. If the dispatcher detects invalid authenticators, the simulation currently logs the invalid authenticator and prints the detection to standard output. The dispatcher does not use any explicit history buffers.

The dispatcher also consumes the Hall Call and Car Call message types. However, the dispatcher does not authenticate these messages. The dispatcher design tolerates failed hall call and car call buttons. After a predefined time limit, the dispatcher will travel to a floor that it has not received a call for to ensure that no passengers have been waiting at that floor. Forging Hall Call and Car Call messages could reduce elevator performance by forcing the elevator to visit all floors in an operating scenario where there are few passengers. However, the performance is no different than the worst case operating scenario where high volumes of passengers are constantly arriving at and traveling to all floors.

6.4.5 Car position indicator

The car position indicator displays the floor the elevator car is currently at to the passengers. This indicator must display the correct floor so that passengers will exit the car at the correct floor. Forging messages to this node can only cause a denial of service. The car position indicator consumes Drive Command, At Floor, Desired Floor, and Car Level Position messages. For

brevity, we omit a detailed analysis. However, forgery of any of these messages could prevent elevator operation.

Implementation notes for time-triggered authentication in the car position indicator - In our implementation, the car position indicator verifies the authenticity of the At Floor, Car Level Position, Drive Command, and Desired Floor message types to monitor for forgeries intended to stop elevator operations. The car position indicator does not implement any explicit history buffers.

6.4.6 Messages to authenticate and receivers

Table 6.6 defines the set of messages to be authenticated from the source node to receiver nodes within the elevator network. Based on our analysis in Sections 6.4.1 through 6.4.5, we authenticate any message which could be forged to violate any of our safety or high level requirements.

Table 6.6. Messages to be authenticated in the elevator, senders, and receivers.

Message	Sender	Receivers							
		Door Contr. (F/L)	**Door Contr. (F/R)**	**Door Contr. (B/L)**	**Door Contr. (B/R)**	**Drive Contr.**	**Safety Monitor**	**Dispatcher**	**Car Position Indicator**
Door Motor Command	Door Controllers	X	X	X	X	X	X		
Door Reversal	Door Reversal Sensors	X	X	X	X		X		
Drive Command	Drive Controller	X	X	X	X		X		X
At Floor	At Floor Sensors	X	X	X	X	X	X	X	X
Car Weight	Car Weight Sensor	X	X	X	X	X	X	X	
Desired Floor	Dispatcher	X	X	X	X	X			X
Door Closed	Door Closed Sensors	X	X	X	X	X	X	X	
Door Open	Door Open Sensors	X	X	X	X			X	
E-Brake	Safety Monitor					X			
Hoistway	Hoistway Limit Sensors					X	X		
Car Level Position	Car Level Position Sensor					X			X
Car Position	Car Position Indicator					X		X	

We implemented time-triggered authentication for each message in Table 6.6 for each receiver within the elevator simulation as described in the implementation notes for each controller in Sections 6.4.1 through 6.4.5. No receivers verify the authenticity of the Hall Call and Car Call message types from the button controllers. Forgeries against these message types can be addressed as described in Section 6.4.4.

6.5 Implementation of time-triggered authentication

This section briefly discusses pertinent implementation details related to each multicast authentication technique (one MAC per receiver, validity voting, TESLA, and master-slave) for use with time-triggered authentication.

6.5.1 Selecting time-triggered authentication parameters

We chose parameters for time-triggered authentication such that attacks should successfully induce failures no more often than a rate of 10^{-9} failures per hour. Equation (1) in Chapter 3 gives an upper bound on the probability during each message round of having successfully forged the n most recent consecutive message samples in a history buffer, each with probability 2^{-b}. We use this equation to define the number of samples and required per-packet assurance. First, we use this probability of successful attack per message round as an expected rate of attack success per message round. For message types broadcast at ten millisecond periods, our desired failure rate becomes approximately 2.777×10^{-15} failures per message period. We then use Equation (1) and select for appropriate values of n samples in the history buffer and b bits per MAC tag for each technique, such that the result is less than our desired failure rate. For simplicity, we use the same failure rate for messages with fifty millisecond periods as well. Achieving the same failure rate for fifty millisecond messages requires approximately the same values for n and b.

From our analysis in sections 6.4.1 through 6.4.5, our maximum history buffer size is twenty samples for Car Level Position and Door Reversal message types. Other message types with less stringent timing requirements can be verified over more samples if desired, though we use twenty as the largest history buffer size we implemented for all message types. This conforms to our assumption that message types are sampled sufficiently quickly to allow us to verify messages

over multiple message samples. For discrete state transitions, such as opening doors or engaging the drive motor, this will create a delay of no more than one second. When authenticating over twenty message samples in a history buffer, the required per-packet assurance is 2^{-3} to achieve our desired failure rate ($n = 20$, $b = 3$).

To define the minimum number of samples to verify messages over, we used OMPR to determine the largest authenticators that could be placed within one packet along with the data values using the CAN communication protocol. Table 6.7 shows the number of data bits already used and the number of remaining bits if the packet size were increased to the full eight bytes. For simplicity, we treat each message and receiver with equal criticality. However, a system designer has the option of devoting more of the available payload bits to authenticating messages related to safety critical functionality over those that are only related to performance characteristics of the system. For our maximum per-packet assurance, we use the maximum tag size defined by the At Floor message type. With a per-packet assurance of 2^{-7}, we must verify messages over at least seven message samples in a history buffer ($n = 7$, $b = 7$).

Table 6.7. Identifying largest tag size among all message types for OMPR. Highlighted table cells show largest tag size we use in the system.

Message Type	Receivers to be authenticated to	Data bits in payload	Available bits in payload	Maximum bits per tag	Number of samples to verify over
Door Motor Command	6	2	62	10	5
Door Reversal	5	1	63	12	5
Drive Command	5	16	48	8	7
At Floor	6	1	63	7	7
Car Weight	6	8	56	8	7
Desired Floor	5	16	48	8	7
Door Closed	6	1	63	9	6
Door Open	4	1	63	12	5
E-Brake	1	1	63	63	1
Hoistway	2	1	63	31	2
Car Level Position	1	32	32	32	2
Car Position	1	8	56	28	2

We implemented each of the four multicast authentication techniques using three sets of parameters for time-triggered authentication ($n = 7$, $b = 7$), ($n = 10$, $b = 5$), and ($n = 20$, $b = 3$). For simplicity, we used the same per-packet assurance for all message types. Thus, we also used the same history buffer size for all message types.

While this section will primarily focus on the trade-off among per-packet assurance and number of samples to verify over, time-triggered authentication also allows significant customization of these parameters on a per-receiver, per-message type, and per-state transition basis. Using the same per-packet assurance for each message type simplifies implementation significantly, but may not provide optimal performance for a system. For example, suppose two nodes broadcast at different sampling rates; one transmits every ten milliseconds, and the other at every twenty milliseconds. If a receiver consumes both of those message types for a state transition, the system designer can verify the faster message over more samples (e.g., ten samples of the ten millisecond message would arrive in the same time it takes to receive five samples of the twenty millisecond message type). This would allow them to use smaller authenticators for the ten millisecond period message to save bandwidth. An in-depth analysis of these customizations within the elevator system is beyond the scope of this work.

6.5.2 One MAC per receiver

For OMPR, each sender computers a MAC tag for each receiver using a corresponding symmetric secret key. We used the Java Cryptography Extension library to define key material and MAC functions within the simulation. We used the Mac class to use the HMAC algorithm for computing all MAC tags. Specifically, we used HMAC in conjunction with the MD5 algorithm. At startup, the underlying network simulation framework creates and assigns symmetric keys to

nodes and MAC functions defined by each key. Our implementation does not perform key establishment or time synchronization. We assume these are already in place at simulation start time.

At the beginning of each control loop execution, receiving nodes verify message authenticity and record message values and their validity within history buffers. Since all authenticators are placed within the same packet as the message value, receivers immediately verify and store the verification results. Nodes also record whether the value suffered a transmission error. If a message is lost, the receiver does not update the contents of their history buffer for that message type.

Once new output values have been determined, nodes compute MAC tags at the end of their control loop execution. A transmitting node calls the MAC function defined by the key corresponding to each receiver of a message. The sender's inputs to the function include the data values within the payload and current simulation time. All tags fit within a single data payload for all message types. The simulation then inserts the MAC tags into the predefined locations within the payload. Execution of these functions within the simulation is "instantaneous" because processing time for nodes is not modeled within the simulation.

For a message type with a period of T milliseconds, the network simulation propagates a new sample of that message type on the network after every T milliseconds pass. Since we assume a fixed transmission schedule for messages, each node should always have the most up to date message value every T milliseconds, unless a controller executes at the same time the new sample is broadcast. Without coordinating control loop executions and message broadcasts, for a controller executing every T milliseconds, the worst case delivery time to receivers in the simulation is $2T$ after that node executes its control loop (e.g., control loops execute and just miss transmissions occurring at the same time). However, since we assume a static message schedule,

we scheduled message broadcasts to occur between control loop executions (this also assumes control loop executions and message periods do not drift out of synch). Thus, receivers have the latest message value after T milliseconds. A receiving node is able to verify and act on n message samples and the corresponding votes after nT milliseconds from the time the transmitter sends the first.

6.5.3 Validity voting

For validity voting, we first examined what message types could carry votes on others. The main limitation in validity voting is that only receivers that share authentication channels with a sender (i.e., the sender computes MAC tags to those receivers) can vote on messages from that sender to one another. Further, for one of those receivers to attest to the validity of a message from that sender, it must also share an authentication channel with the other receiver it attests to.

We only use existing authentication channels for validity voting, as listed in Table 6.6. We do not add new authenticators to any message types to other receivers that are not already listed in Table 6.6. For several message types, adding new MAC tags to a packet would exceed the payload size for the parameters we selected.

While we were limited in the number of votes that could be added in this network, we were able to implement validity voting using one, two, and three votes. Nodes only attest to messages consumed by the door controllers and drive controller. Nodes vote on the message types as described in Table 6.8 and Table 6.9. Table 6.8 lists each transmitting node, that node's message type, the number of voting bits added (one for each message type it votes on), and the message types it votes upon. Table 6.9 shows how many votes each node receives for each message type. Numbers in brackets indicate messages from multiple nodes that broadcast the same time.

Table 6.8. Message types voted upon in validity voting

Sender	Sender's message type	Voting bits	Message types voted upon
Door Controllers	Door Motor Command	5	Drive Speed, Door Reversal [4]
Drive Controller	Drive Command	0	None
Car Position Indicator	Car Position	1	Car Level Position
Safety Monitor	Emergency Brake	7	Door Closed [4], Car Weight, Hoistway Limit [2]
Dispatcher	Desired Floor	5	Door Closed [4], Car Weight

Table 6.9. Number of votes received for each message type by each node

Receiver	Message type	Votes received for message type	Nodes votes are received from
Door Controllers	Door Reversal [4]	3	Three other Door Controllers
	Drive Speed	3	Three other Door Controllers
	Car Weight	1	Dispatcher
	Door Closed [4]	1	Dispatcher
Drive	Car Level Position	1	Car Position Indicator
	Hoistway Limit [2]	1	Safety Monitor
	Car Weight	2	Safety Monitor, Dispatcher
	Door Closed [4]	2	Safety Monitor, Dispatcher

Message broadcast periods also limited the number of votes we could implement in the elevator. Messages only carry votes for other message types with the same broadcast period. For example, the Door Motor Command message type only votes on other message types that are broadcast at ten millisecond periods. Similarly, the Car Position, Emergency Brake, and Desired Floor messages carry votes for other message types with fifty millisecond periods.

In our implementation, we again have nodes compute MAC tags at the end of their control loops using the same key material and MAC functions defined for one MAC per receiver. We modified the inputs to the functions to include the most current message value for the message

types being voted upon, along with a bit vector indicating which message values were valid or invalid. For invalid and lost message values, the transmitter uses predefined error codes as inputs for message values, as described in Chapter 4.

For messages that are voted upon, receivers can verify the validity of a value in a packet, but must wait for the confirmations carried in other message types before using the value being voted upon. This creates an extra message period delay before a receiver can use the message value being voted upon. As with one MAC per receiver (not coordinating message transmissions and control loop executions), for a controller executing every T milliseconds, the worst case delivery time to receivers (including voters) in the simulation is $2T$ after that node executes its control loop. Voting nodes (which also execute every T milliseconds) then include their votes in their own messages after their next control loop execution, which the network simulation propagates to receivers every T milliseconds. Again, the worst case delay to receive votes is $2T$ milliseconds. Nodes receive a message sample and all votes in the simulation after no more than $4T$ milliseconds, regardless of the number of votes. In our implementation, we scheduled node transmissions to occur just after control loop executions completed. Thus the delay for receiving each transmission was only T milliseconds. A node in our implementation is able to verify and act on n message samples and the corresponding votes after $(n + 1)T$ milliseconds from the time the transmitter sends the first.

6.5.4 TESLA

For TESLA, nodes must also transmit a key in addition to a message value and its MAC tag. In our implementation, we transmit an eighty bit key for each sample of a message type. We transmit the key corresponding to each message sample in the subsequent message round. Since we are limited to sixty-four bit data payloads in CAN, we used two data payloads to transmit the

message value, MAC tag, and key for the previous sample. In our implementation, the first payload includes the value, MAC tag, and the first portion of the key for the previous message round. The second data payload contains the remainder of that key. Our implementation omits the key establishment and time synchronization procedures required for TESLA.

We used a simplified abstraction of key chains in our implementation. We assume that key generation and storage algorithms of TESLA are correct and secure. A full implementation of TESLA would require each transmitting node to iterate a hash function to generate a sequence of keys along with an appropriate storage algorithm. Based on time constraints for development and debugging, we did not implement the key chain generation and storage. Instead, our "key chain" in the simulation is a series of incrementing eighty-bit integers whose values correspond to the message round number (i.e., 0, 1, 2, 3, ...). While such a simplified implementation would not be secure in a real system, our attacker model does not attack key material for any schemes. Implementations of TESLA in a real-world safety-critical system should implement the correct algorithm for key chain generation, as published [Perrig00].

At the beginning of control loop executions, nodes store new message values and tags that are received. If the node receives the key for the prior message round, it will verify the authenticity of that message value. The node then stores the value and its validity in corresponding history buffers. Receivers also store the most recently received key. If a receiver stores an explicit history buffer for state-changing message types, it will attempt to recover values for which it received the message value and tag, but not the subsequent key. In TESLA, once a node receives a key, it can always compute prior keys that might have been lost to transmission errors. Thus, a node can verify any prior message value for which the corresponding key was initially lost. The receiver records these recovered values in history buffers for state-changing messages as per normal. For

reactive control message types, receivers do not attempt to recover old values, since only the most recent value is used to update controller outputs.

Similarly to validity voting, TESLA creates a one message round delay before receivers obtain both a value and its key. After coordinating controller executions and message transmission schedules in our implementation, a node is able to verify and act on n message samples after $(n + 1)T$ milliseconds from the time the transmitter sends the first.

6.5.5 Master-slave

For master-slave, we added a trusted master node to the network. Each transmitting node authenticates its message to the master node. This master node verifies the authenticity of each message broadcast on the network. It then transmits a single bit message along with a hash tree broadcast authenticator (as described in Chapter 5) indicating if all messages observed in the previous round were valid or any were invalid. In our implementation, the master node executes and transmits this broadcast authenticator every ten milliseconds to allow receivers to verify all messages.

The most challenging aspect of implementing master-slave in the elevator network was coordinating verification of messages that were broadcast at two different rates (ten and fifty millisecond periods) using a single master node. Slave nodes that execute control loops at ten millisecond intervals (fast slave nodes) can easily participate and verify messages. However, nodes executing at fifty millisecond intervals (slow slave nodes) posed several challenges:

- During four of five ten-millisecond periods, the master node only attested to messages being broadcast every ten milliseconds. However, every fifth ten-millisecond period, the master attested to message types broadcast at both the ten-millisecond and fifty-millisecond.

- Slave nodes could only exchange messages to verify the broadcast authenticator at the rate at which they executed their control loops. Thus, for four of five ten-millisecond periods, the master node would only compute MAC tags for receivers executing every ten-milliseconds and hashed those together. On the fifth ten-millisecond period, the master computed MAC tags for both fast and slow receivers.
- Slower receivers do not necessarily obtain every sample of message types broadcast faster than their control loop periods. The network simulation framework creates a mailbox for each CAN message type each node receives. This mailbox only records the most recent sample of each message type. Thus, slower receivers might attempt to verify a master's hash tree broadcast authenticator using out-of-date message values.

We resolved the first two challenges by storing a predefined table indicating when samples of a message type was expected to be transmitted on the network. This allowed each node to verify the master's hash tree broadcast authenticator over the correct set of values. Every fifth ten-millisecond period, faster slave nodes would verify values from the slower slave nodes as well.

For the third challenge, we altered the slower receivers control loop periods. Every ten milliseconds, the slower slave nodes would execute and store a copy of the message types broadcast at this faster interval. Then, every fifty milliseconds, these nodes would execute their full control loop.

Another option to address these concerns would have been to use multiple messages from the master node to authenticate messages to groups of receivers executing at different rates. One message type could have been transmitted every ten milliseconds for the ten millisecond nodes, and the second message type could have been transmitted every fifty milliseconds for the fifty millisecond nodes.

Using master-slave creates a one message round delay before receivers obtain both a value and its subsequent confirmation. After coordinating controller executions and message transmission schedules in our implementation, a node is able to verify and act on n message samples after $(n + 1)T$ milliseconds from the time the transmitter sends the first.

6.6 Analysis

6.6.1 Bandwidth comparison

We first compared bandwidth consumption for each technique. Table 6.10 shows the bandwidth consumption of the messages transmitted within for the elevator network without authentication. We computed all packet sizes using the worst case CAN message size equation provided in the analysis of worst case CAN message delays performed by Ellims et al. [Ellims02].

Table 6.10. Baseline elevator bandwidth required with no authentication

Message Type	Period (msec)	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	1	90	9000	4	36000
Door Reversal	10	1	90	9000	4	36000
Drive Speed	10	2	100	10000	1	10000
AtFloor	50	1	90	1800	10	18000
Car Weight	50	1	90	1800	1	1800
Desired Floor	50	2	100	2000	1	2000
Door Closed	50	1	90	1800	4	7200
Door Open	50	1	90	1800	4	7200
EBrake	50	1	90	1800	1	1800
Hoistway	50	1	90	1800	2	3600
Car Level Position	50	4	120	2400	1	2400
Car Position	50	1	90	1800	1	1800
Hall Call	100	1	90	900	17	15300
Car Call	100	1	90	900	10	9000
					Total Bandwidth (bits/sec)	**152100**

We then computed the additional bandwidth consumed when applying each of the four multicast authentication techniques. We computed the required bandwidth for the three sets of time-triggered authentication parameters we defined in Section 6.5.1:

1. Per-packet assurance = 2^{-7}, number of samples = 7.
2. Per-packet assurance = 2^{-5}, number of samples = 10.
3. Per-packet assurance = 2^{-3}, number of samples = 20.

One MAC per Receiver - We assigned MAC tags of equal size to each receiver for each message type. Tables 6.11, 6.12, and 6.13 shows the required bandwidth for each parameter set. Payload bytes include both the data values and authentication.

Table 6.11. OMPR history buffer size, required per-packet assurance, and MAC tag size.

History buffer size (samples)	Required per-packet assurance	MAC tag size (bits)
7	2^{-7}	7
10	2^{-5}	5
20	2^{-3}	3

Table 6.12. OMPR required bandwidth (Per-packet assurance = 2^{-7}, number of samples = 7)

Message Type	Period (msec)	Total authentication bits	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	42	6	140	14000	4	56000
Door Reversal	10	35	5	130	13000	4	52000
Drive Speed	10	42	8	160	16000	1	16000
AtFloor	50	56	8	160	3200	10	32000
Car Weight	50	49	8	160	3200	1	3200
Desired Floor	50	42	8	160	3200	1	3200
Door Closed	50	49	7	150	3000	4	12000
Door Open	50	35	5	130	2600	4	10400
EBrake	50	7	1	90	1800	1	1800
Hoistway	50	14	2	100	2000	2	4000
Car Level Position	50	14	6	140	2800	1	2800
Car Position	50	14	3	110	2200	1	2200
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**219900**
						Authentication Bandwidth (bits/sec)	**20800**

Table 6.13. OMPR required bandwidth (Per-packet assurance = 2^{-5}, number of samples = 10)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	30	4	120	12000	4	48000
Door Rever-sal	10	25	4	120	12000	4	48000
Drive Speed	10	30	6	140	14000	1	14000
AtFloor	50	40	6	140	2800	10	28000
Car Weight	50	35	6	140	2800	1	2800
Desired Floor	50	30	6	140	2800	1	2800
Door Closed	50	35	5	130	2600	4	10400
Door Open	50	25	4	120	2400	4	9600
EBrake	50	5	1	90	1800	1	1800
Hoistway	50	10	2	100	2000	2	4000
Car Level Position	50	10	6	140	2800	1	2800
Car Position	50	10	3	110	2200	1	2200
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**198700**
						Authentication Bandwidth (bits/sec)	**15800**

Table 6.14. OMPR required bandwidth (Per-packet assurance = 2^{-3}, number of samples = 20)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	18	3	110	11000	4	44000
Door Rever-sal	10	15	2	100	10000	4	40000
Drive Speed	10	18	5	130	13000	1	13000
AtFloor	50	24	4	120	2400	10	24000
Car Weight	50	21	4	120	2400	1	2400
Desired Floor	50	18	5	130	2600	1	2600
Door Closed	50	21	3	110	2200	4	8800
Door Open	50	15	2	100	2000	4	8000
EBrake	50	3	1	90	1800	1	1800
Hoistway	50	6	1	90	1800	2	3600
Car Level Position	50	6	5	130	2600	1	2600
Car Position	50	6	2	100	2000	1	2000
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**177100**
						Authentication Bandwidth (bits/sec)	**10800**

Validity voting (VV)- For this technique, we reduced the size of authenticators based on the number of votes for each message type. Voting bits were added as discussed in Section 6.5.3. Table 6.15 shows tag sizes for each history buffer size for each level of voting. In some cases, adding extra votes did not reduce MAC tag size. We included these votes in our implementation to examine the effects of additional votes in the experimental analysis (Section 6.6.2-4). Tables 6.16, 6.17, and 6.18 shows the required bandwidth for each parameter set. Payload bytes include both the data values and authentication.

Table 6.15. VV history buffer size, required per-packet assurance, and MAC tag size.

History buffer size (samples)	Required per-packet assurance	MAC tag size w/ zero votes (bits)	MAC tag size w/ one vote (bits)	MAC tag size w/ two votes (bits)	MAC tag size w/ three votes (bits)
7	2^{-7}	7	4	3	3
10	2^{-5}	5	3	3	2
20	2^{-3}	3	2	2	2

Table 6.16. VV required bandwidth (Per-packet assurance = 2^{-7}, number of samples = 7)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	47	7	150	15000	4	60000
Door Rever-sal	10	19	3	110	11000	4	44000
Drive Speed	10	26	6	140	14000	1	14000
AtFloor	50	56	8	160	3200	10	32000
Car Weight	50	33	6	140	2800	1	2800
Desired Floor	50	47	8	160	3200	1	3200
Door Closed	50	33	5	130	2600	4	10400
Door Open	50	35	5	130	2600	4	10400
EBrake	50	14	2	100	2000	1	2000
Hoistway	50	11	2	100	2000	2	4000
Car Level Position	50	11	6	140	2800	1	2800
Car Position	50	15	3	110	2200	1	2200
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
					Total Bandwidth (bits/sec)		**212100**
					Authentication Bandwidth (bits/sec)		**14300**

Table 6.17. VV required bandwidth (Per-packet assurance = 2^{-5}, number of samples = 10)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	35	5	130	13000	4	52000
Door Rever-sal	10	13	2	100	10000	4	40000
Drive Speed	10	18	5	130	13000	1	13000
AtFloor	50	40	6	140	2800	10	28000
Car Weight	50	25	5	130	2600	1	2600
Desired Floor	50	35	7	150	3000	1	3000
Door Closed	50	25	4	120	2400	4	9600
Door Open	50	25	4	120	2400	4	9600
EBrake	50	12	2	100	2000	1	2000
Hoistway	50	8	2	100	2000	2	4000
Car Level Position	50	8	5	130	2600	1	2600
Car Position	50	11	3	110	2200	1	2200
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**192900**
						Authentication Bandwidth (bits/sec)	**10380**

Table 6.18. VV required bandwidth (Per-packet assurance = 2^{-3}, number of samples = 20)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-packet bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	18	3	110	11000	4	44000
Door Rever-sal	10	11	2	100	10000	4	40000
Drive Speed	10	14	4	120	12000	1	12000
AtFloor	50	24	4	120	2400	10	24000
Car Weight	50	16	3	110	2200	1	2200
Desired Floor	50	18	5	130	2600	1	2600
Door Closed	50	16	3	110	2200	4	8800
Door Open	50	15	2	100	2000	4	8000
EBrake	50	3	1	90	1800	1	1800
Hoistway	50	5	1	90	1800	2	3600
Car Level Position	50	5	5	130	2600	1	2600
Car Position	50	6	2	100	2000	1	2000
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**175900**
						Authentication Bandwidth (bits/sec)	**6460**

TESLA - This scheme required us to add an additional message type to transmit keys for the message types being authenticated. All keys were eighty bits in size. Each sample required only a single MAC tag (Table 6.19 shows tag and key sizes). All messages except Hall Call and Car Call message types require two packets. Tables 6.20, 6.21, and 6.22 shows the required bandwidth for each parameter set. Payload bytes include both the data values, key, and authentication.

Table 6.19. TESLA history buffer size, required per-packet assurance, and MAC tag size.

History buffer size (samples)	Required per-packet assurance	MAC tag size (bits)	Key size (bits)
7	2^{-7}	7	80
10	2^{-5}	5	80
20	2^{-3}	3	80

Table 6.20. TESLA required bandwidth (Per-packet assurance = 2^{-7}, number of samples = 7)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Sample size (bits)	Per-sample bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	87	12	280	28000	4	112000
Door Rever-sal	10	87	11	270	27000	4	108000
Drive Speed	10	87	13	290	29000	1	29000
AtFloor	50	87	11	270	5400	10	54000
Car Weight	50	87	12	280	5600	1	5600
Desired Floor	50	87	13	290	5800	1	5800
Door Closed	50	87	11	270	5400	4	21600
Door Open	50	87	11	270	5400	4	21600
EBrake	50	87	11	270	5400	1	5400
Hoistway	50	87	11	270	5400	2	10800
Car Level Position	50	87	15	310	6200	1	6200
Car Position	50	87	12	280	5600	1	5600
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**409900**
						Authentication Bandwidth (bits/sec)	**41760**

Table 6.21. TESLA required bandwidth (Per-packet assurance = 2^{-5}, number of samples = 10)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Sample size (bits)	Per-sample bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	85	11	270	27000	4	108000
Door Rever-sal	10	85	11	270	27000	4	108000
Drive Speed	10	85	13	290	29000	1	29000
AtFloor	50	85	11	270	5400	10	54000
Car Weight	50	85	12	280	5600	1	5600
Desired Floor	50	85	13	290	5800	1	5800
Door Closed	50	85	11	270	5400	4	21600
Door Open	50	85	11	270	5400	4	21600
EBrake	50	85	11	270	5400	1	5400
Hoistway	50	85	11	270	5400	2	10800
Car Level Position	50	85	15	310	6200	1	6200
Car Position	50	85	12	280	5600	1	5600
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**405900**
						Authentication Bandwidth (bits/sec)	**40800**

Table 6.22. TESLA required bandwidth (Per-packet assurance = 2^{-3}, number of samples = 20)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Sample size (bits)	Per-sample bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Door Motor	10	83	11	270	27000	4	108000
Door Rever-sal	10	83	11	270	27000	4	108000
Drive Speed	10	83	13	290	29000	1	29000
AtFloor	50	83	11	270	5400	10	54000
Car Weight	50	83	12	280	5600	1	5600
Desired Floor	50	83	13	290	5800	1	5800
Door Closed	50	83	11	270	5400	4	21600
Door Open	50	83	11	270	5400	4	21600
EBrake	50	83	11	270	5400	1	5400
Hoistway	50	83	11	270	5400	2	10800
Car Level Position	50	83	15	310	6200	1	6200
Car Position	50	83	12	280	5600	1	5600
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
						Total Bandwidth (bits/sec)	**405900**
						Authentication Bandwidth (bits/sec)	**39840**

Master-Slave (MS)- Master-slave required a single additional message type to be added for transmissions from the master node. Each message type required one or two MAC tags. Message types transmitted by the master or sensors required only a single tag. Tag size is one bit higher than used for OMPR (Table 6.23). Messages from controllers that must verify the master's broadcast authenticator must transmit two. Tables 6.24, 6.25, and 6.26 shows the required bandwidth for each parameter set. Payload bytes include both the data values and authentication.

Table 6.23. MS history buffer size, required per-packet assurance, and MAC tag size.

History buffer size (samples)	Required per-packet assurance	MAC tag size (bits)
7	2^{-7}	8
10	2^{-5}	6
20	2^{-3}	4

Table 6.24. MS required bandwidth (Per-packet assurance = 2^{-7}, number of samples = 7)

Message Type	Period (msec)	Total authentication bits	Payload (bytes)	Packet size (bits)	Per-sample bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Master	10	8	2	100	10000	1	10000
Door Motor	10	16	3	110	11000	4	44000
Door Reversal	10	8	2	100	10000	4	40000
Drive Speed	10	16	4	120	12000	1	12000
AtFloor	50	8	2	100	2000	10	20000
Car Weight	50	8	2	100	2000	1	2000
Desired Floor	50	16	4	120	2400	1	2400
Door Closed	50	8	2	100	2000	4	8000
Door Open	50	8	2	100	2000	4	8000
EBrake	50	16	3	110	2200	1	2200
Hoistway	50	8	2	100	2000	2	4000
Car Level Position	50	8	5	130	2600	1	2600
Car Position	50	16	3	110	2200	1	2200
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
Total Bandwidth (bits/sec)							**181700**
Authentication Bandwidth (bits/sec)							**6720**

Table 6.25. MS required bandwidth (Per-packet assurance = 2^{-5}, number of samples = 10)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-sample bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Master	10	6	1	90	9000	1	9000
Door Motor	10	12	2	100	10000	4	40000
Door Rever-sal	10	6	1	90	9000	4	36000
Drive Speed	10	12	4	120	12000	1	12000
AtFloor	50	6	1	90	1800	10	18000
Car Weight	50	6	2	100	2000	1	2000
Desired Floor	50	12	4	120	2400	1	2400
Door Closed	50	6	1	90	1800	4	7200
Door Open	50	6	1	90	1800	4	7200
EBrake	50	12	2	100	2000	1	2000
Hoistway	50	6	1	90	1800	2	3600
Car Level Position	50	6	5	130	2600	1	2600
Car Position	50	12	3	110	2200	1	2200
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
Total Bandwidth (bits/sec)							**168500**
Authentication Bandwidth (bits/sec)							**5040**

Table 6.26. MS required bandwidth (Per-packet assurance = 2^{-3}, number of samples = 20)

Message Type	Period (msec)	Total authen-tication bits	Payload (bytes)	Packet size (bits)	Per-sample bandwidth (bits/sec)	Replication	Message type bandwidth (bits/sec)
Master	10	4	1	90	9000	1	9000
Door Motor	10	8	2	100	10000	4	40000
Door Rever-sal	10	4	1	90	9000	4	36000
Drive Speed	10	8	3	110	11000	1	11000
AtFloor	50	4	1	90	1800	10	18000
Car Weight	50	4	2	100	2000	1	2000
Desired Floor	50	8	3	110	2200	1	2200
Door Closed	50	4	1	90	1800	4	7200
Door Open	50	4	1	90	1800	4	7200
EBrake	50	8	2	100	2000	1	2000
Hoistway	50	4	1	90	1800	2	3600
Car Level Position	50	4	5	130	2600	1	2600
Car Position	50	8	2	100	2000	1	2000
Hall Call	100	0	1	90	900	17	15300
Car Call	100	0	1	90	900	10	9000
Total Bandwidth with CAN protocol overhead (bits/sec)							**167100**
Authentication Bandwidth (bits/sec)							**3360**

Comparisons - In Table 6.27, we show the total authentication bandwidth and total message bandwidth (including CAN protocol overhead) for each of the four techniques for our three sets of time-triggered authentication parameters.

Table 6.27. Total authentication bits per second

Technique	**Time-triggered authentication parameters PPA = Per-packet assurance, n = history buffer size (samples)**		
	PPA = 2^{-7}, n = 7	**PPA = 2^{-5}, n = 10**	**PPA = 2^{-3}, n = 20**
One MAC per receiver	20800	15800	10800
Validity voting	14300	10380	6460
TESLA	41760	40800	39840
Master-slave	6720	5040	3360

Table 6.28. Total bits per second transmitted on bus (including CAN protocol overhead)

Technique	**Time-triggered authentication parameters PPA = Per-packet assurance, n = history buffer size (samples)**		
	PPA = 2^{-7}, n = 7	**PPA = 2^{-5}, n = 10**	**PPA = 2^{-3}, n = 20**
One MAC per receiver	219900	198700	177100
Validity voting	212100	192900	175900
TESLA	409900	405900	405900
Master-slave	181700	168500	167100

*Total bits per second without authentication is: 152100 bits per sec (same for all values of *n*)

Table 6.29. Percent increase in required bandwidth with authentication (including CAN protocol overhead)

Technique	**Time-triggered authentication parameters PPA = Per-packet assurance, n = history buffer size (samples)**		
	PPA = 2^{-7}, n = 7	**PPA = 2^{-5}, n = 10**	**PPA = 2^{-3}, n = 20**
One MAC per receiver	44 %	31 %	16 %
Validity voting	39 %	27 %	16 %
TESLA	170 %	167 %	167 %
Master-slave	20 %	11 %	10 %

Table 6.27 shows a reduction in authentication bandwidth overhead as we use weaker per-packet assurance and amortize authentication over more samples. Master-slave has the lowest authentication overhead, requiring only one MAC tag for each message type authenticated to the master and another MAC tag for each message type from a receiver that verifies the hash tree broadcast authenticator. Master-slave also has very low impact on overall network bandwidth,

since only one message type was added for the master's broadcast authenticator, and there are no silent receivers in the system. Validity voting requires the second lowest bandwidth for authentication. Voting on message authenticity (despite the limitations in the number of possible votes) saved between four to six kilobits per second in authentication data over one MAC per receiver. Validity voting provides a greater reduction in authentication bandwidth when MAC tag sizes are larger. TESLA adds approximately forty kilobits per second of authentication data for all parameters, primarily due to the key material that must be transmitted.

Table 6.28 shows similar decreases in overall bandwidth for one MAC per receiver and validity voting. However, for TESLA, two sets of parameters require the same bandwidth. This is due to the quantization of payload sizes in CAN. The protocol defines payload size by the number of bytes, rather than the number of bits in the payload. Since there are only one or two MAC tags in messages for these techniques, reducing a MAC tag by a few bits may not reduce the overall payload size by more than one byte. Table 6.29 shows the percent increase in required bandwidth after incorporating each authentication technique.

6.6.2 Effects of history buffer size on system performance

After implementing each technique in the elevator, we examined the effect of each technique on elevator performance. Specifically, we measured delivery times for passengers for our three history buffer sizes (n = 7, 10, and 20). Figure 6.3 shows the average passenger delivery times as we vary the history buffer size for each technique. For this experiment, the elevator car begins at the first floor, a passenger makes a hall call at the seventh floor and wants to travel to the first floor. For each data point, we executed this single-passenger workload one hundred times. While the transition delays and elevator dynamics remained constant for each run of the simulator, the passenger behaviors can affect delivery times. Passengers update their internal variables at discrete

intervals (e.g., they check doors every 100 milliseconds and check/press call buttons every 200 milliseconds). The simulation also adds a randomized offset for passenger actions of up to a few hundred milliseconds. Thus, we averaged the delivery times over many executions of the passenger workload. Delivery times varied no more than two seconds from one another for each data point.

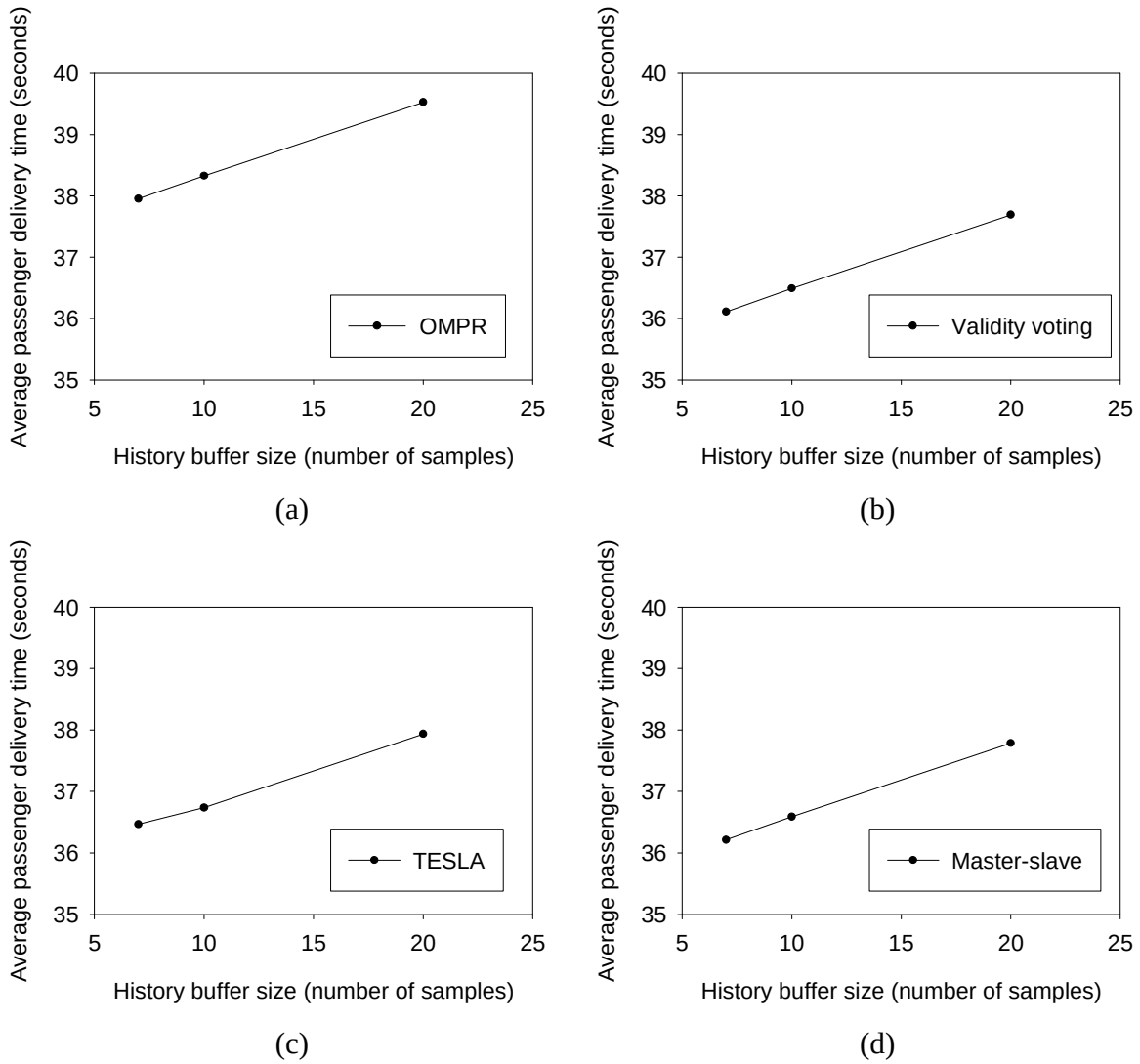

Figure 6.3. Effects of buffer size on single passenger delivery times. (a) One MAC per receiver, (b) validity voting, (c) TESLA, and (d) master-slave. History buffer size varied from seven samples to twenty samples.

One interesting side-effect of the using different authentication schemes is that each technique affects elevator dynamics slightly differently due to delays in verification of various message types. In particular, passenger delivery times are slightly less for the techniques that have a per-packet verification delay. We emphasize that adding a per-packet delay does not increase the verification speed of individual samples or history buffers. The decrease in delivery time is due to the effects of per-packet verification delays on drive controller transitions to slow the car as it approaches a floor. Thus, we do not compare delivery times between techniques.

Instead of comparing overall delivery times between techniques, we instead focus on the increase in delivery times for individual techniques. The effects of elevator dynamics do not change by varying time-triggered authentication parameters. Drive transitions for slowing the car are treated as reactive control and require only one sample to trigger the transition. We observed that the delivery times showed a linear increase as we increased history buffer size. This is as expected, since increasing history buffer sizes for state-changing messages creates a delay before each associated state transition can occur. Increases in delays should be similar for each technique. For each technique, the average delivery time increased by approximately 1.5 seconds as we increase delays in transitions from seven samples to twenty samples. This increase is primarily due to delays in verifying message types broadcast at fifty millisecond intervals for starting drive motion to and from the passenger's floor, as it must verify the desired floor before moving. Delays in the door controller state transitions have less effect on overall delivery delays. These delays in opening doors are mostly due to waiting for several samples of the Drive Command message (ten millisecond period) to indicate the drive is stopped. The values fifty millisecond period messages the door controllers rely on for opening doors actually satisfy the door controller's state transition conditions before the drive actually comes to a complete stop.

6.6.3 Symmetric packet loss effects on history buffer output readiness

Our second set of experiments examined the effect of packet loss on state transition delays. In a symmetric omissive fault model, either all nodes attached to the network receive the message or none receive it [Azadmanesh00]. To do this, we injected symmetric omissive faults into the network simulation framework as it propagated messages to receivers. Thus, all nodes drop the affected packet.

We varied the packet loss rate between zero and twenty percent for all message types broadcast on the network. For each technique, we measured the number of message periods that passed before a node received and verified a sufficient number of messages for a history buffer of size twenty to allow a node to commit to a state change. Once the history buffer output was ready, we recorded the number of message rounds that had passed, reset the buffer, and restarted the experiment. We repeated this experiment for at least one thousand history buffers, and computed the average results. For validity voting, we examined the packet loss effects on message types that received one vote, two votes, and three votes.

We can calculate how many message periods a receiver can expect to wait to receive n samples of data, and L is the fraction of packets lost. If x different message types must be received to verify our desired message, then the probability that a node will receive and be able to verify a sample is $(1\text{-}L)^x$. The number of message periods until we receive n error-free samples is $n/(1\text{-}L)^x$. Figure 6.4 shows the expected number of samples and the experimental results for each technique. For one MAC per receiver, we observed an average time till the state-changing history buffer output was ready increased by a factor of $1/(1\text{-}L)$ times the number of samples in the history buffer. For validity voting with one vote, it increased by a factor of $1/(1\text{-}L)^2$; for two votes, it increased by a factor of $1/(1\text{-}L)^3$; and for three votes, it increased by a factor of $1/(1\text{-}L)^4$.

The delay for TESLA was less than a factor of $1/(1-L)^2$ despite requiring two packets to verify each sample. The delay was only slightly higher than that for one MAC per receiver, since previously lost keys can be recovered corresponding to received, but unverified message values could not be until the next key arrived. The delays we observed for messages verified using the master-slave scheme do not conform to this equation for all message types. We experimentally tested the delay for a ten millisecond period message type. During most message rounds, verification depends only on the ten millisecond message types broadcast in the network. Every fifth message period, verification of that message also requires receipt of messages from nodes transmitting every fifty milliseconds. In the worst case, verifying messages using master-slave require broadcasts from eight other nodes. Thus, the delay factor should be no greater than $(1-L)^8$.

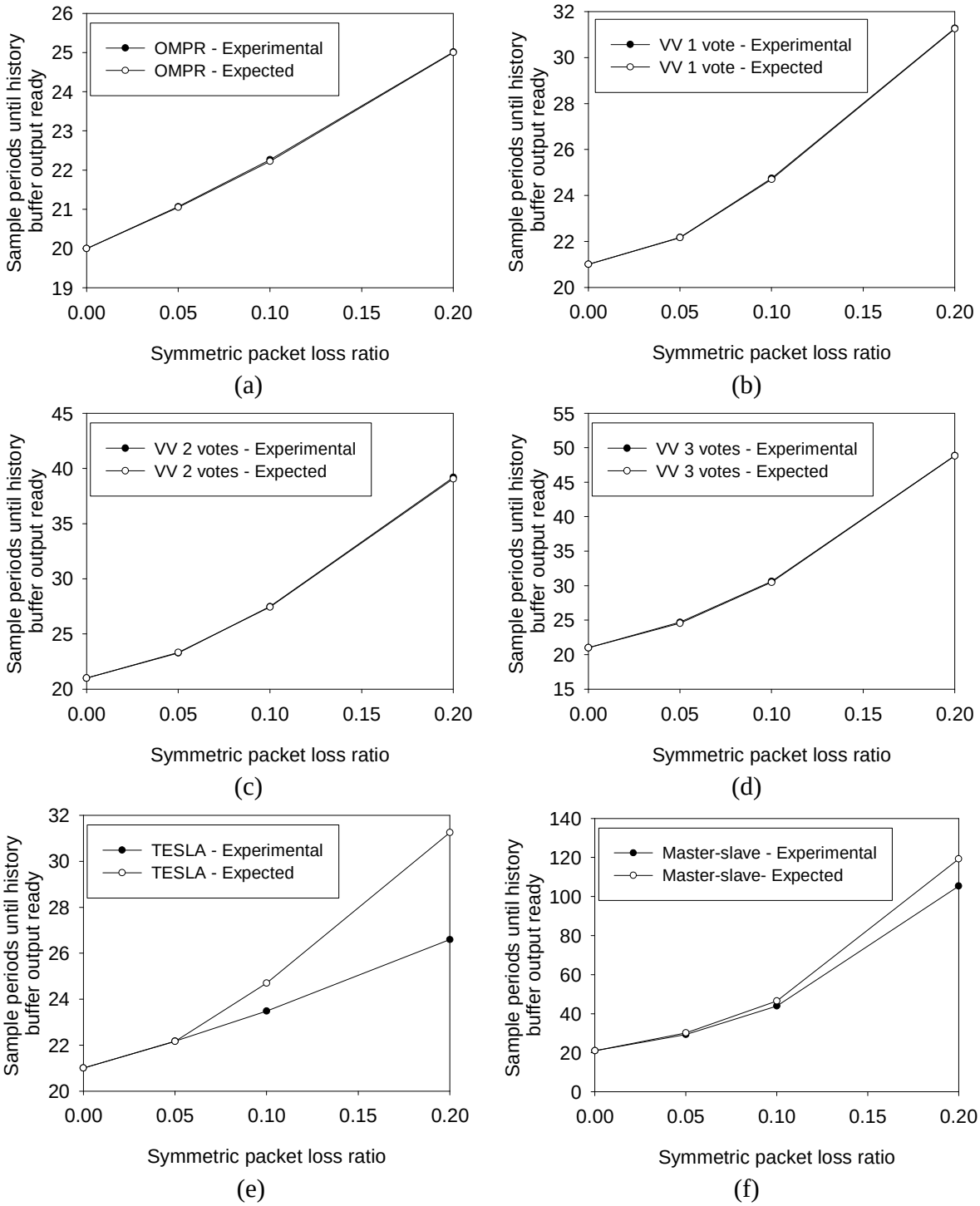

Figure 6.4. Average delay of history buffer output readiness due to symmetric packet loss. Symmetric packet loss rate was varied from zero to twenty percent. History buffer size was fixed at twenty samples. Techniques are (a) one MAC per receiver, (b) validity voting - one vote, (c) validity voting - two votes, (d) validity voting - three votes, (e) TESLA, (f) master-slave.

Figures 6.5 shows the average number of message periods that pass before a history buffer output is ready as we vary the symmetric packet loss rate for all techniques together for comparison.

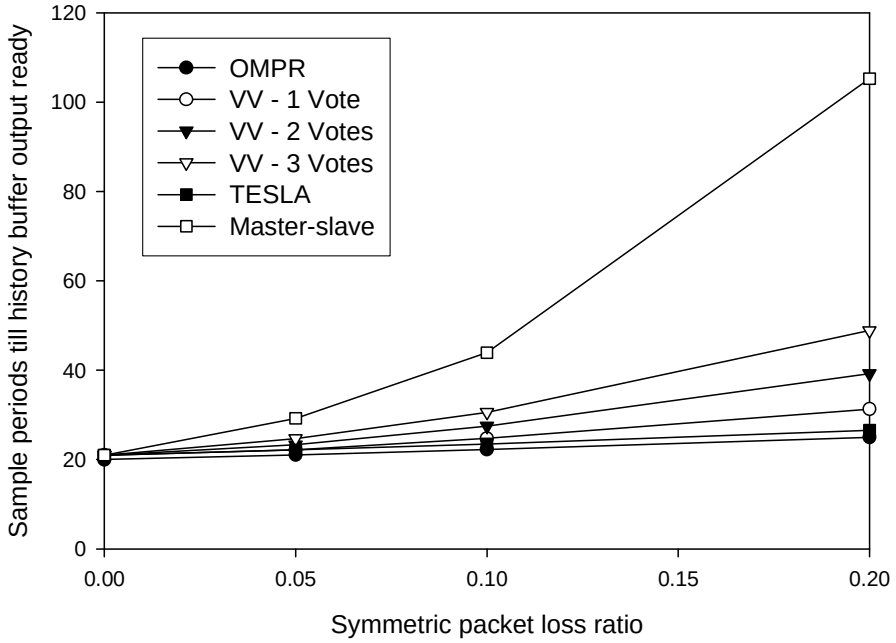

Figure 6.5. Average delay of history buffer output readiness due to symmetric packet loss (combined figures 6.4.a through 6.4.f). Symmetric packet loss rate was varied from zero to twenty percent. History buffer size was fixed at twenty samples.

We observed an exponential increase in the time until a history buffer output was ready as we increased the symmetric packet loss rate, as expected from the equations. One MAC per receiver suffered the least delays since all data and authentication is stored within the same packet. TESLA had only slightly longer delays than one MAC per receiver. In TESLA, despite requiring a key to be transmitted for each message value, a receiver will be always eventually be able to recover message values with lost keys. Once the receiver obtains another key, the receiver can compute all previous keys to recover any unverified message values for which a corresponding key was not received. However, if there are several dropped keys in a row, when a receiver re-

covers unverified message values, it might recover more samples than are required for the buffer output to be ready. Thus, a few samples go unused for a state change. For validity voting, as we increase the number of votes, the average time until the history buffer output increased with respect to the number of votes being used. Lastly, master-slave suffered the highest delays. If either the master's message or any slave's message carrying a MAC tag necessary to verify the master's hash is lost, then all values in the previous round are also lost.

We only performed these experiments on message types being used as state-changing messages. Losses of reactive control message values will trigger a system to perform a safe action. If a receiver observes too many packet losses for a reactive control transition (e.g., too many Door Reversal message values are lost when closing the doors) the system should perform a safe action. The number of lost samples to tolerate is up to the system designer.

6.6.4 Symmetric packet loss effects on system performance

Next, we experimentally examined the effects of packet loss on system performance for each multicast authentication technique we implemented in the elevator. We use a modified symmetric omissive fault model for this set of experiments. If our fault model were to drop packets at any time during execution, there are points in time where dropping packets will actually speed up passenger delivery times. This occurs when the a packet loss delays drive controller state transitions to slow or stop the drive speed; dropping a packet at this instant delays the transition to reduce speed and the car continues longer at a higher speed. Triggering this elevator-specific behavior can speed up elevator performance significantly, masking the delays these experiments are intended to observe. For example, a single packet loss while at a speed of 5 m/s allows the drive to travel up to an additional 0.25 meters, reducing delivery time by 0.25 seconds.

If the elevator is designed to stop at the absolute minimum distance, then a single packet loss could cause the elevator to miss the desired floor and exceed hoistway limits. Thus, the original developers of the elevator system added slack time to make the system more robust to packet losses or other delays that might propagate through the system and trigger this failure. In a real elevator system, slack time is likely to be programmed into the system. Further, the system is likely to begin slowing down and/or stopping if too many packet losses occur before the slack time is used up during normal operation.

Instead of attempting to account for these elevator-specific effects, we modify the fault model so it does not drop packets if the elevator should be slowing down to avoid triggering this accidental speed up in performance.

We varied the rate of packet loss from zero to twenty percent, and measured the average passenger delivery time over one hundred executions of the first passenger workload in Section 6.6.2. We fixed the history buffer size at twenty samples. Figure 6.6 shows the average delivery times for each technique.

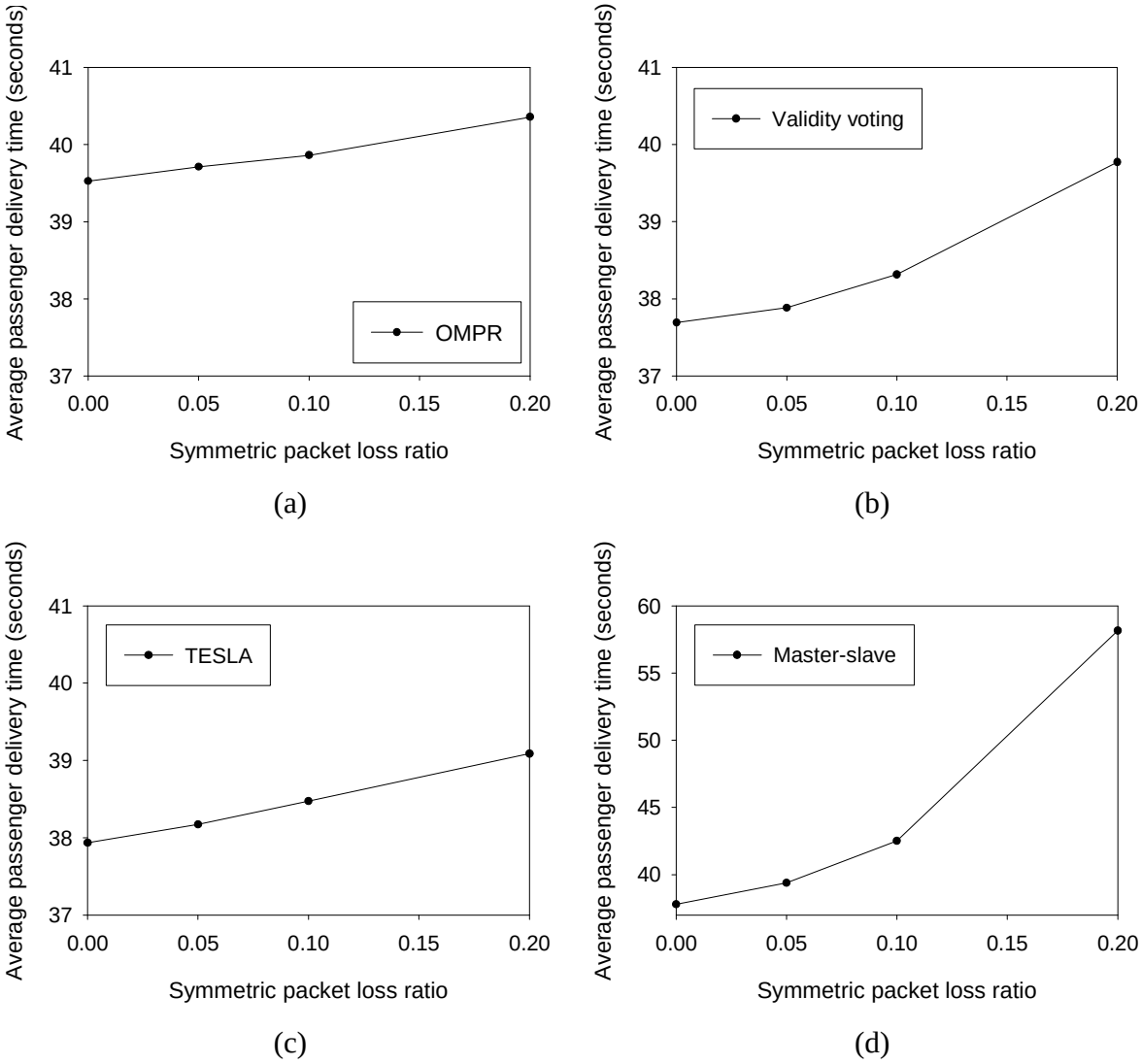

Figure 6.6. Average passenger delivery times varying symmetric packet loss rate. Symmetric packet loss rate was varied from zero to twenty percent. History buffer size was fixed at twenty samples. Techniques are (a) one MAC per receiver, (b) validity voting, (c) TESLA, and (d) master-slave.

As we increase the packet loss rate up to twenty percent, the average delivery time for one MAC per receiver increases by 0.83 seconds. TESLA increases by slightly more (1.15 seconds at twenty percent packet loss), though this was likely due to the recovery behavior discussed in Section 6.6.3 where a few extra message periods pass before unverified messages can be recov-

ered and a state transition executed. The validity voting implementation delivery times increased by 2.08 seconds at twenty percent packet loss. This was likely due to the implementation being a mixture of one MAC per receiver and validity voting using one, two, and three votes. Lastly, as expected, the master-slave implementation suffers the worst delivery delays of an additional 20.37 seconds at twenty percent packet loss (more than a fifty percent increase in delivery time due to few state transition delays). This is due to the high degree of inter-packet dependencies for this technique.

6.6.5 Forgery test

Our final set of experiments consisted of simple brute force guessing attacks against a state-changing message type for each technique. Once a successful state transition was forced, we reset the history buffer and the state machine and allowed the attacker to begin again with no delay. We recorded the number of successful attacks (triggering a state transition) per message round. The purpose of these experiments was simply to verify the probability of successful attack and successful per-packet forgery is less than or equal to the expected success rates described in the equations of Chapters 3, 4, and 5. For brevity, we omit a detailed review of the results. The resulting success rate for brute force guessing attacks were slightly less than the equations in Chapters 3, 4, and 5. This is due to the history buffer being reset and an attack requiring all samples in the history buffer to be successfully forged. Thus, there are message rounds where an attack cannot yet have occurred after the history buffer is reset. This same result is demonstrated in Chapter 3 for attacks against state-changing messages. The experimental attacks on reactive control message types produced similar results.

The attacker model used against OMPR and TESLA was the same as the one used in Chapter 3. The attacker model used against validity voting was the same as the one in Chapter 4. For

master-slave, we used a slightly different attacker model. The attacker would first attempt to alter the message value from a slave node that is authenticated to the master node. The attacker then intercepts the master's message in the next message round, and examines the validity bit. If the bit is a '1,' then the attacker knows it successfully forged the tag on the initial attempt. If not, the attacker attempts to alter the master's validity bit before passing the message along to the slave nodes. The results were slightly less than equation (4) in Chapter 5, due to the attack being performed on a state-changing message type.

6.7 Discussion

In this chapter, we showed our analysis of the elevator system to identify which message types to authenticate along with time triggered parameter selection. We compared bandwidth consumption for each technique, varying time-triggered authentication parameters. Finally, our experimental results showed effects of varying time-triggered authentication parameters on system performance (passenger delivery time in the elevator) for each technique, and effects of symmetric packet loss on history buffer output readiness and system performance. We also performed brute force forgery attacks to confirm equations in Chapters 3, 4, and 5.

By varying time-triggered authentication parameters (per-packet assurance and history buffer size) we illustrate several tradeoffs for all techniques. Increasing per-packet assurance (decreasing history buffer size) increases bandwidth costs for authentication but decreases application level latency. Conversely, reducing per-packet assurance (increasing history buffer size) reduces bandwidth costs but increases application level latency.

Adjusting time-triggered authentication parameters had a similar effect on elevator system performance for all techniques. Varying the history buffer size from seven samples up to twenty increased passenger delivery times by approximately 1.5 seconds for all techniques.

In the presence of packet losses, we showed that system performance and history buffer output readiness for one MAC per receiver and TESLA were least affected. The implementation of validity voting (which built upon one MAC per receiver to introduce one, two and three votes on messages) was more sensitive to packet losses. We showed that increasing the number of inter-dependencies amongst packets for verification significantly increased the amount of time before a history buffer output was useable. The master-slave scheme was extremely sensitive to packet losses and suffered long delays in both history buffer output readiness and system performance.

Our analysis also illustrates some of the tradeoffs among techniques. While the bandwidth analysis shows that master-slave has very low authentication bandwidth overhead, the experimental analysis shows it has very high sensitivity to packet losses. Validity voting allows us to reduce the authentication bandwidth overhead of one MAC per receiver at the cost of increased sensitivity to packet losses. One MAC per receiver required higher authentication bandwidth overhead than master-slave and validity voting, but is the least sensitive to packet losses. TESLA required the highest authentication bandwidth overhead (which remained relatively constant regardless for our three sets of time-triggered authentication parameters), but was can also recover unverified state-changing message values for which key material has been lost. Recovering reactive control message values is possible, but may not be useful if the system acts only upon the most recent message values.

7 Evaluation - Automotive network

Our second proof of concept analyzes the impacts on bandwidth consumption when applying time-triggered authentication (for each of the four techniques) to an automotive network workload. The workload is from a high-speed CAN bus in an industry production automotive system.

The workload contains almost all of the information required for our analysis: node identifier numbers (both senders and receivers), message identifier numbers, message periods, and payload sizes. However, the workload has been sanitized of system data; it does not include any node or message names. Also, the identifier numbers have been randomized, such that ID numbers of the workload provided in this work do not reveal CAN bus identifier numbers for messages, removing priority information. The workload also does not include information related to what any of the messages are used for. We did not have access to the system or a model of the system these messages are used within. We also do not have requirements associated with the system.

Our bandwidth consumption analysis in this section requires a few pieces of information not included in the workload provided by industry: requirements for system failure rates and per-packet assurance (i.e., we need to know how many samples that can be authenticated over). Since we do not have access to the system and design information related to the workload, we used typical values commonly found in embedded control networks.

For system failure rates, we selected three common rates used in industry: 10^{-9} failures per hour, 10^{-6} failures per hour, and 10^{-3} failures per hour. These failure rates were not part of the provided workload. As in earlier sections, we assume successful forgery of a single message type could induce a system failure. These failure rates were selected based on common standards, such as IEC 61508 [IEC61508]. We elected to assign different failure rates to illustrate the flex-

ibility of our time-triggered authentication approach and effects of using different failure rates on parameter selection. The workload is divided into four levels of assurance:

- High - For messages in this group we selected parameters such that forgery success rates should be no higher than 10^{-9}/hr. There are twenty-four message types in this group. These messages have periods between ten and one hundred milliseconds.
- Medium - For messages in this group we selected parameters such that forgery success rates should be no higher than 10^{-6}/hr. This group contains thirty-two message types. Most message periods in this group are similar to those in the high assurance group, with some longer periods up to one second.
- Low - We selected parameters such that forgery success rates should be no higher than 10^{-3}/hr. There are twenty-two message types in this group. Message periods for this group range from twenty milliseconds up to five seconds.
- None - We did not apply authentication to these message types, nor do these message types participate in authentication. This group consists of messages with periods mostly slower message periods and messages broadcast in response to non-periodic events. There are eighty-seven message types in this group.

We emphasize that for this analysis, these ratings do not represent security risks in the automotive system this workload is from. We do not speculate on the failure modes of the system this workload is from, since we have limited information about the workload. The failure rates were selected arbitrarily. Appropriately assigning requirements for system-level and per-packet assurance levels requires analysis of the system design (which we did not have access to).

Tables 7.1 shows the list of message types in the high assurance level along with broadcast period, sender number, receivers, and payload size. Table 7.2 shows the medium assurance level group of messages. Table 7.3 shows the low assurance level group. Finally, Table 7.4 shows the list of message types that we did not apply authentication to.

Table 7.1. High assurance automotive messages.

Message ID	Period (ms)	Sender ID	Payload (bits)	Number of Receivers	Receivers
ID_009	10	ECU_05	44	8	ECU_02, ECU_03, ECU_04, ECU_06, ECU_07, ECU_09, ECU_11, ECU_13
ID_008	10	ECU_07	49	1	ECU_09
ID_047	10	ECU_07	49	9	ECU_01, ECU_04, ECU_05, ECU_06, ECU_08, ECU_09, ECU_12, ECU_13, ECU_14
ID_040	12	ECU_07	62	1	ECU_09
ID_001	12	ECU_09	55	2	ECU_02, ECU_07
ID_007	12	ECU_09	64	12	ECU_01, ECU_02, ECU_03, ECU_04, ECU_05, ECU_06, ECU_07, ECU_08, ECU_10, ECU_11, ECU_13, ECU_14
ID_039	20	ECU_07	36	2	ECU_09, ECU_11
ID_042	20	ECU_07	24	1	ECU_04
ID_025	25	ECU_02	52	1	ECU_09
ID_029	25	ECU_02	64	1	ECU_09
ID_030	25	ECU_02	64	4	ECU_05, ECU_07, ECU_09, ECU_11
ID_038	25	ECU_07	56	1	ECU_09
ID_036	25	ECU_09	64	3	ECU_05, ECU_07, ECU_11
ID_074	25	ECU_09	16	1	ECU_02
ID_046	30	ECU_05	52	2	ECU_09, ECU_11
ID_057	30	ECU_05	60	2	ECU_02, ECU_09
ID_076	35	ECU_11	52	1	ECU_09
ID_077	35	ECU_11	34	1	ECU_09
ID_078	35	ECU_11	34	1	ECU_09
ID_058	50	ECU_07	33	4	ECU_05, ECU_06, ECU_11, ECU_13
ID_081	50	ECU_07	45	4	ECU_02, ECU_04, ECU_05, ECU_09
ID_061	50	ECU_13	46	3	ECU_05, ECU_07, ECU_11
ID_098	100	ECU_09	37	1	ECU_07
ID_060	100	ECU_13	12	1	ECU_07

Table 7.2. Medium assurance automotive messages.

Message ID	Period (ms)	Sender ID	Payload (bits)	Number of Receivers	Receivers
ID_006	6	ECU_02	32	1	ECU_04
ID_004	10	ECU_07	64	10	ECU_02, ECU_04, ECU_05, ECU_06, ECU_08, ECU_09, ECU_10, ECU_12, ECU_13, ECU_14
ID_005	10	ECU_07	64	11	ECU_01, ECU_02, ECU_04, ECU_05, ECU_06, ECU_08, ECU_09, ECU_10, ECU_12, ECU_13, ECU_14
ID_010	12	ECU_02	61	4	ECU_04, ECU_06, ECU_07, ECU_09
ID_003	12	ECU_09	9	1	ECU_02
ID_026	12	ECU_09	31	1	ECU_02
ID_027	12	ECU_09	62	2	ECU_02, ECU_04
ID_048	12	ECU_09	59	1	ECU_10
ID_052	12	ECU_09	61	1	ECU_10
ID_041	20	ECU_04	26	3	ECU_02, ECU_07, ECU_11
ID_045	20	ECU_04	27	1	ECU_07
ID_024	20	ECU_07	11	5	ECU_02, ECU_05, ECU_06, ECU_13, ECU_14
ID_049	20	ECU_07	62	12	ECU_01, ECU_02, ECU_03, ECU_04, ECU_05, ECU_06, ECU_08, ECU_09, ECU_11, ECU_12, ECU_13, ECU_14
ID_028	25	ECU_02	16	1	ECU_09
ID_033	25	ECU_02	45	1	ECU_09
ID_106	25	ECU_05	17	1	ECU_09
ID_031	25	ECU_09	54	1	ECU_02
ID_034	25	ECU_09	62	1	ECU_02
ID_035	25	ECU_09	57	8	ECU_04, ECU_05, ECU_06, ECU_07, ECU_08, ECU_11, ECU_13, ECU_14
ID_037	25	ECU_09	48	2	ECU_07, ECU_11
ID_075	50	ECU_09	40	2	ECU_04, ECU_07
ID_018	100	ECU_05	24	1	ECU_01
ID_020	100	ECU_05	34	2	ECU_07, ECU_08
ID_053	100	ECU_05	54	12	ECU_01, ECU_02, ECU_03, ECU_04, ECU_06, ECU_07, ECU_09, ECU_10, ECU_11, ECU_12, ECU_13, ECU_14
ID_059	100	ECU_06	9	2	ECU_05, ECU_08
ID_023	100	ECU_07	18	1	ECU_05
ID_021	100	ECU_08	18	1	ECU_05
ID_102	250	ECU_05	58	6	ECU_04, ECU_06, ECU_07, ECU_09, ECU_13, ECU_14
ID_101	250	ECU_08	44	1	ECU_09
ID_083	500	ECU_06	16	3	ECU_05, ECU_07, ECU_08
ID_017	1000	ECU_05	17	2	ECU_01, ECU_11
ID_117	1000	ECU_05	45	3	ECU_02, ECU_04, ECU_09

Table 7.3. Low assurance automotive messages.

Message ID	Period (ms)	Sender ID	Payload (bits)	Number of Receivers	Receivers
ID_044	20	ECU_04	3	1	ECU_07
ID_002	25	ECU_02	53	1	ECU_09
ID_056	25	ECU_02	64	11	ECU_01, ECU_04, ECU_05, ECU_06, ECU_07, ECU_08, ECU_09, ECU_11, ECU_12, ECU_13, ECU_14
ID_082	25	ECU_06	60	3	ECU_01, ECU_06, ECU_07
ID_032	25	ECU_09	1	2	ECU_02, ECU_07
ID_054	30	ECU_05	16	2	ECU_02, ECU_09
ID_088	35	ECU_11	16	2	ECU_05, ECU_07
ID_089	35	ECU_11		3	ECU_02, ECU_05, ECU_07
ID_084	50	ECU_07	36	8	ECU_03, ECU_04, ECU_05, ECU_06, ECU_11, ECU_12, ECU_13, ECU_14
ID_085	50	ECU_07	36	8	ECU_03, ECU_04, ECU_05, ECU_06, ECU_11, ECU_12, ECU_13, ECU_14
ID_087	50	ECU_07	28	1	ECU_05
ID_043	100	ECU_04	6	6	ECU_02, ECU_05, ECU_07, ECU_08, ECU_09, ECU_11
ID_013	100	ECU_05	57	8	ECU_01, ECU_02, ECU_06, ECU_07, ECU_09, ECU_10, ECU_11, ECU_13
ID_016	100	ECU_05	9	2	ECU_07, ECU_11
ID_022	100	ECU_07	47	10	ECU_01, ECU_03, ECU_04, ECU_05, ECU_06, ECU_08, ECU_10, ECU_11, ECU_12, ECU_13
ID_080	100	ECU_07	40	1	ECU_09
ID_113	500	ECU_09	56	2	ECU_05, ECU_08
ID_136	500	ECU_09	64	1	ECU_02
ID_014	1000	ECU_05	3	1	ECU_10
ID_120	1000	ECU_07	25	9	ECU_04, ECU_05, ECU_06, ECU_08, ECU_10, ECU_11, ECU_12, ECU_13, ECU_14
ID_118	1000	ECU_09	44	8	ECU_02, ECU_04, ECU_05, ECU_07, ECU_10, ECU_11, ECU_12, ECU_13
ID_012	5000	ECU_05	33	1	ECU_04

Table 7.4. Non-authenticated automotive messages.

Message ID	Period (ms)	Sender ID	Payload (bits)	Number of Receivers	Receivers
ID_011	Event	ECU_08	64	14	ECU_01, ECU_02, ECU_03, ECU_04, ECU_05, ECU_06, ECU_07, ECU_08, ECU_09, ECU_10, ECU_11, ECU_12, ECU_13, ECU_14
ID_015	100	ECU_05	56	5	ECU_01, ECU_06, ECU_07, ECU_11, ECU_12
ID_019	1000	ECU_14	1	2	ECU_05, ECU_13
ID_050	12	ECU_09	28	1	ECU_05
ID_051	12	ECU_10	13	1	ECU_09
ID_055	25	ECU_09	33	2	ECU_05, ECU_07
ID_062	20	ECU_07	47	6	ECU_04, ECU_05, ECU_06, ECU_08, ECU_10, ECU_14
ID_063	Event	ECU_08	64	1	ECU_05
ID_064	Event	ECU_08	64	1	ECU_14
ID_065	Event	ECU_08	64	1	ECU_07
ID_066	Event	ECU_08	64	1	ECU_03
ID_067	Event	ECU_08	64	1	ECU_01
ID_068	Event	ECU_08	64	1	ECU_11
ID_069	Event	ECU_08	64	1	ECU_06
ID_070	Event	ECU_08	64	1	ECU_12
ID_071	Event	ECU_08	64	1	ECU_13
ID_072	Event	ECU_08	64	1	ECU_04
ID_073	Event	ECU_08	64	1	ECU_10
ID_079	50	ECU_09	10	1	ECU_07
ID_086	100	ECU_03	3	1	ECU_05
ID_090	100	ECU_01	8	1	ECU_05
ID_091	100	ECU_01	16	1	ECU_05
ID_092	100	ECU_01	8	1	ECU_05
ID_093	1500	ECU_01	6	2	ECU_05, ECU_11
ID_094	100	ECU_09	64	2	ECU_05, ECU_08
ID_095	100	ECU_05	64	1	ECU_09
ID_096	100	ECU_05	33	1	ECU_09
ID_097	100	ECU_09	60	6	ECU_02, ECU_05, ECU_06, ECU_08, ECU_10, ECU_11
ID_099	100	ECU_02	11	1	ECU_05
ID_100	100	ECU_09	64	12	ECU_01, ECU_02, ECU_03, ECU_04, ECU_05, ECU_06, ECU_07, ECU_08, ECU_10, ECU_12, ECU_13, ECU_14
ID_103	250	ECU_09	63	3	ECU_05, ECU_08, ECU_10
ID_104	250	ECU_09	14	3	ECU_05, ECU_08, ECU_10
ID_105	250	ECU_09	29	2	ECU_05, ECU_07
ID_107	500	ECU_09	61	8	ECU_01, ECU_02, ECU_04, ECU_05, ECU_06, ECU_07, ECU_08, ECU_10
ID_108	1000	ECU_09	49	2	ECU_05, ECU_08
ID_109	500	ECU_05	16	1	ECU_09
ID_110	500	ECU_09	23	1	ECU_02
ID_111	500	ECU_02	30	4	ECU_04, ECU_05, ECU_08, ECU_09
ID_112	1000	ECU_09	24	1	ECU_05
ID_114	500	ECU_05	34	1	ECU_09
ID_115	500	ECU_10	17	1	ECU_09
ID_116	1000	ECU_05	64	6	ECU_02, ECU_04, ECU_07, ECU_09, ECU_13, ECU_14
ID_119	1000	ECU_09	8	1	ECU_11
ID_121	500	ECU_10	27	1	ECU_05

Message ID	Period (ms)	Sender ID	Payload (bits)	Number of Receivers	Receivers
ID_122	1000	ECU_05	64	4	ECU_04, ECU_07, ECU_13, ECU_14
ID_123	1000	ECU_05	48	1	ECU_07
ID_124	1000	ECU_05	32	1	ECU_10
ID_125	Event	ECU_05	64	1	ECU_08
ID_126	Event	ECU_14	64	1	ECU_08
ID_127	Event	ECU_07	64	1	ECU_08
ID_128	Event	ECU_03	64	1	ECU_08
ID_129	Event	ECU_01	64	1	ECU_08
ID_130	Event	ECU_11	64	1	ECU_08
ID_131	Event	ECU_06	64	1	ECU_08
ID_132	Event	ECU_12	64	1	ECU_08
ID_133	Event	ECU_13	64	1	ECU_08
ID_134	Event	ECU_04	64	1	ECU_08
ID_135	Event	ECU_10	64	1	ECU_08
ID_137	Event	ECU_09	64	1	ECU_08
ID_138	Event	ECU_02	64	1	ECU_08
ID_139	Event	ECU_05	64	1	ECU_08
ID_140	Event	ECU_14	64	1	ECU_08
ID_141	Event	ECU_07	64	1	ECU_08
ID_142	Event	ECU_03	64	1	ECU_08
ID_143	Event	ECU_01	64	1	ECU_08
ID_144	Event	ECU_11	64	1	ECU_08
ID_145	Event	ECU_06	64	1	ECU_08
ID_146	Event	ECU_12	64	1	ECU_08
ID_147	Event	ECU_13	64	1	ECU_08
ID_148	Event	ECU_04	64	1	ECU_08
ID_149	Event	ECU_10	64	1	ECU_08
ID_150	1000	ECU_09	56	1	ECU_08
ID_151	1000	ECU_07	56	1	ECU_08
ID_152	1000	ECU_14	56	1	ECU_08
ID_153	1000	ECU_04	56	1	ECU_08
ID_154	1000	ECU_01	56	1	ECU_08
ID_155	1000	ECU_13	56	1	ECU_08
ID_156	1000	ECU_06	56	1	ECU_08
ID_157	1000	ECU_02	56	1	ECU_08
ID_158	1000	ECU_07	56	1	ECU_08
ID_159	1000	ECU_10	56	1	ECU_08
ID_160	1000	ECU_12	56	1	ECU_08
ID_161	Event	ECU_08	64	2	ECU_02, ECU_09
ID_162	Event	ECU_08	64	1	ECU_09
ID_163	Event	ECU_08	64	1	ECU_02
ID_164	Event	ECU_09	64	1	ECU_08
ID_165	Event	ECU_02	64	1	ECU_08

Our analysis also requires us to assign per-packet assurance levels to message types. Since we did not have access to the characteristics of the physical dynamics of the system, we performed a sensitivity analysis based on common sampling rates. Sensor inputs are typically sampled faster

than the time constraints of control stability requirements. As a rule of thumb, ten or more samples are sent within the rise time of a control system or prior to a system deadline [REF Kopetz][REF Controls book]. This number of samples gives us our history buffer size (the number of messages we can verify state changes and actuations over). For our sensitivity analysis, we used history buffer sizes of five, ten, and twenty. We assume all messages use the same history buffer size. Thus, all message types within a group use the same per-packet assurance level.

The number of nodes and numbers of receivers for each message type in this network conforms to our assumptions in Section 2. In an embedded network, there are typically at most tens for receivers for a message. In this network, there are fourteen total nodes. The number of receivers for each message type ranges from one to twelve. Only nine nodes broadcast messages that require authentication (ECU_2, ECU_4, ECU_5, ECU_6, ECU_7, ECU_8, ECU_9, ECU_11, and ECU_13). There are five nodes which only consume messages and do not broadcast authenticated messages (ECU_1, ECU_3, ECU_10, ECU_12, and ECU_14). These five nodes do, however, broadcast non-authenticated messages. In this analysis, non-authenticated messages transmitted by these nodes do not participate in voting or master-slave authentication schemes.

Another note of interest is that many message types already have full data payloads, which will require a second (or third) CAN packet to transmit authenticators. In Section 3 and 4, we assumed that data payloads were small enough such that at least one MAC tag bit could be placed within a packet for each receiver. The message types for the elevator network also had room for one MAC tag per receiver in the data payloads (at least seven bits could be placed in a payload for each receiver without exceeding the sixty-four bit payload size of CAN); TESLA was the only technique that required an additional packet (due to the key). The bandwidth im-

pacts of authentication in this workload will be greater than those in the elevator since nodes must transmit additional packets for authentication for all techniques.

The baseline automotive network workload (with no authentication applied) consumes 478782 bits per second. This value only includes periodic message types; it omits the impacts of the non-periodic message types, since we do not have information on mean inter-arrival times for those message types. In the following sections, we apply each authentication technique while varying the history buffer size. We then summarize the impacts of each authentication technique on network bandwidth.

7.1 One MAC per receiver

For OMPR, we first determined the MAC tag size for each receiver based on the failure rate associated with each message type along with the number of samples for the history buffer. Tables 7.5. lists the history buffer size, per-packet assurance and number of bits per MAC tag for each assurance level group (high, medium, and low).

Table 7.5. OMPR history buffer size, required per-packet assurance, and MAC tag size.

History buffer size (samples)	Desired failure rate	Required per-packet assurance	MAC tag size (bits)
5	10^{-9}/hr	2^{-10}	10
	10^{-6}/hr	2^{-8}	8
	10^{-3}/hr	2^{-6}	6
10	10^{-9}/hr	2^{-5}	5
	10^{-6}/hr	2^{-4}	4
	10^{-3}/hr	2^{-3}	3
20	10^{-9}/hr	2^{-3}	3
	10^{-6}/hr	2^{-2}	2
	10^{-3}/hr	2^{-2}	2

Tables in Appendix A provide a detailed breakdown of bandwidth required for authentication and the messages of the workload.

7.1.1 One MAC per receiver - summary

Table 7.15 summarizes the results of applying one MAC per receiver to the automotive workload. Using one MAC per receiver, as we increase the number of samples in a history buffer, there is a exponential decrease in the bandwidth consumed by authentication. Similarly, there is an exponential decrease in total bandwidth consumption (including CAN protocol overhead). As we increase the history buffer size, it approaches the baseline workload bandwidth of 478782 bits per second. However, one MAC per receiver will always require at least one bit per receiver no matter how samples messages are verified over.

Table 7.6. OMPR bandwidth summary.

	History buffer size (samples)		
	5	10	20
Bandwidth increase due to authentication (bits per second)	90525	45292	25482
Total bandwidth including CAN protocol overhead (bits per second)	745662	629660	588735
Percent increase in total bandwidth over baseline	56 %	32 %	23 %

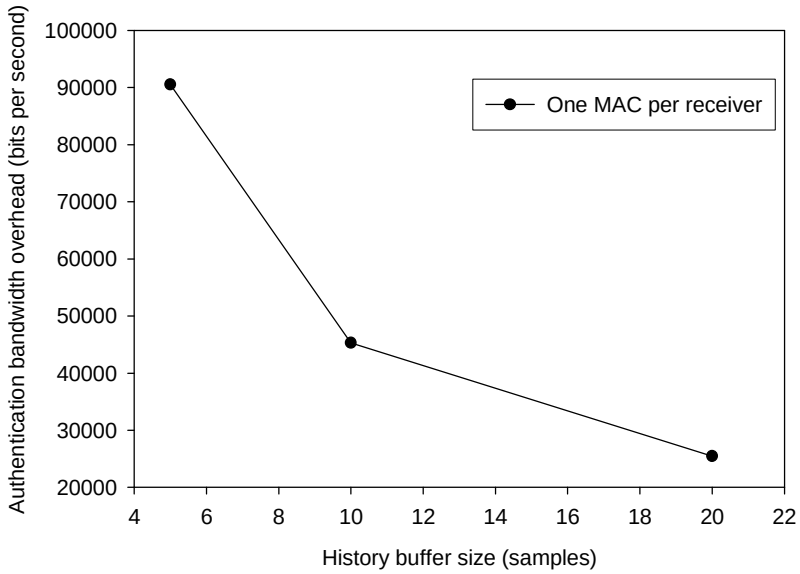

Figure 7.1. OMPR authentication bits per second (no CAN protocol overhead, varying history buffer size from five samples to twenty samples).

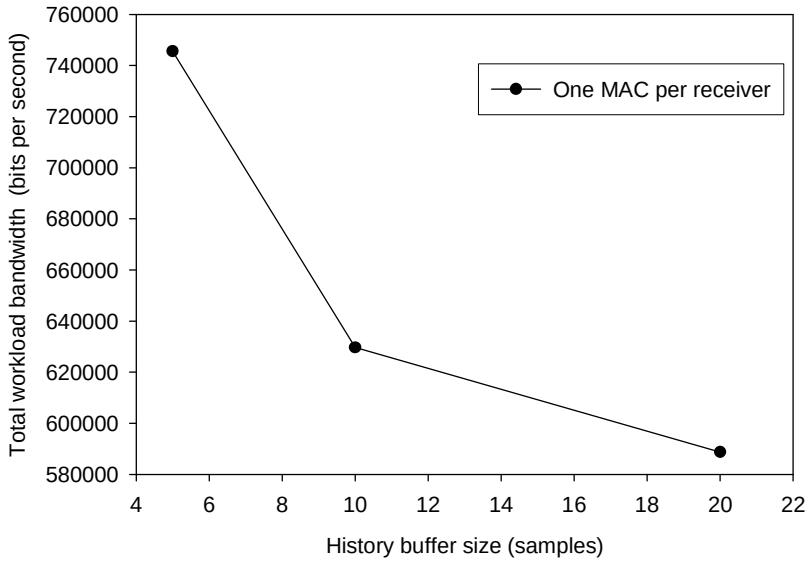

Figure 7.2. OMPR total bits per second transmitted on CAN bus (includes CAN protocol overhead)

7.2 Validity voting

For validity voting, we applied the maximum number of votes for every message type. All votes were used to reduce the size of authenticators (optionally, votes can also be used to reduce the number of message samples to verify over). We included votes only if they provided some bandwidth savings. Table 7.16 shows the size of each MAC tag when zero votes are received for that message type, one vote is received for the message type, and two votes are received for that message type. Table 7.16 also shows occurrences where increasing the number of votes did not decrease the MAC tag bit size (marked N/A). When adding a vote to decrease tag size while maintaining per-packet assurance, the tag size decreases by a fraction based on the number of votes. Using v votes reduces tag size by a factor of slightly less than $1/(v+1)$. For example, using one vote decreases tag size to almost half the bits; two votes decreases tag size to a little more than one third the bits; three votes decreases the tag size to a little more than one quarter. Thus, votes save more bandwidth when used to reduce tag sizes for higher per-packet assurances.

Table 7.7. VV history buffer size, required per-packet assurance, and MAC tag sizes (for zero votes, one vote, and two votes). N/A indicates votes do not provide any bandwidth reduction.

History buffer size (samples)	Desired failure rate	Required per-packet assurance	MAC tag size zero votes (bits)	MAC tag size one vote (bits)	MAC tag size two votes (bits)
5	10^{-9}/hr	2^{-10}	10	6	4
	10^{-6}/hr	2^{-8}	8	5	4
	10^{-3}/hr	2^{-6}	6	4	3
10	10^{-9}/hr	2^{-5}	5	3	N/A
	10^{-6}/hr	2^{-4}	4	3	2
	10^{-3}/hr	2^{-3}	3	2	N/A
20	10^{-9}/hr	2^{-3}	3	2	N/A
	10^{-6}/hr	2^{-2}	2	N/A	N/A
	10^{-3}/hr	2^{-2}	2	N/A	N/A

As with the elevator network workload, we applied votes based on the message types consumed by each message type. For one receiver to vote on a message to another, both receivers

must consume that message type from the sender. Nodes receiving votes must consume at least one message type from the voting node. Further, any message types carrying votes must be broadcast at a rate greater than or equal to the message type they vote upon.

We also limited message types to vote on messages of equal or lower criticality (a message in the high assurance group can vote on a message in the medium assurance group, but not vice versa). This limitation was primarily based on the tag sizes for messages of each assurance level. In a message that carries a vote, the tag designated for a receiver should be at least as many bits as the tag designated for the same receiver in the message being voted upon. The tag sizes for message types requiring low assurance have fewer bits than those requiring higher assurance. Thus, if a lower assurance message type carries a vote for a higher assurance message, it could create a vulnerability that could allow an attacker to more easily forge a samples of the higher assurance message.

Based on these limitations, we were able to apply at most two votes for any message type. However, messages can carry votes for any number of other message types.

Tables in Appendix A provide a detailed breakdown of bandwidth required for authentication and the messages of the workload.

7.2.1 Validity voting - summary

Table 7.8 summarizes the results of applying validity voting to the automotive network workload. Validity voting uses one MAC per receiver as a base for multicast authentication, and uses voting to reduce bandwidth consumption (or history buffer size). Figures 7.3 shows the authentication bits per second added to the workload bandwidth for each history buffer size, while Figure

7.4 shows the total bandwidth used by the workload (including CAN protocol overhead). Figures 7.3 and 7.4 also show the results of applying one MAC per receiver to allow comparison.

Table 7.8. Validity voting bandwidth summary.

	History buffer size (samples)		
	5	10	20
Bandwidth increase due to authentication (bits per second)	72368	38856	24039
Total bandwidth including CAN protocol overhead (bits per second)	702694	619584	587702
Percent increase in total bandwidth over baseline	47 %	29 %	23 %

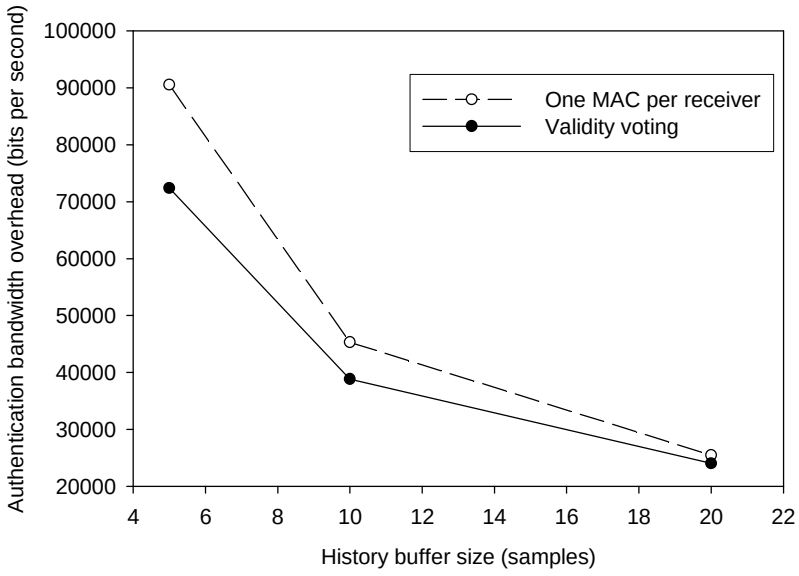

Figure 7.3. Validity voting authentication bits per second (no CAN protocol overhead, varying history buffer size from five samples to twenty samples.

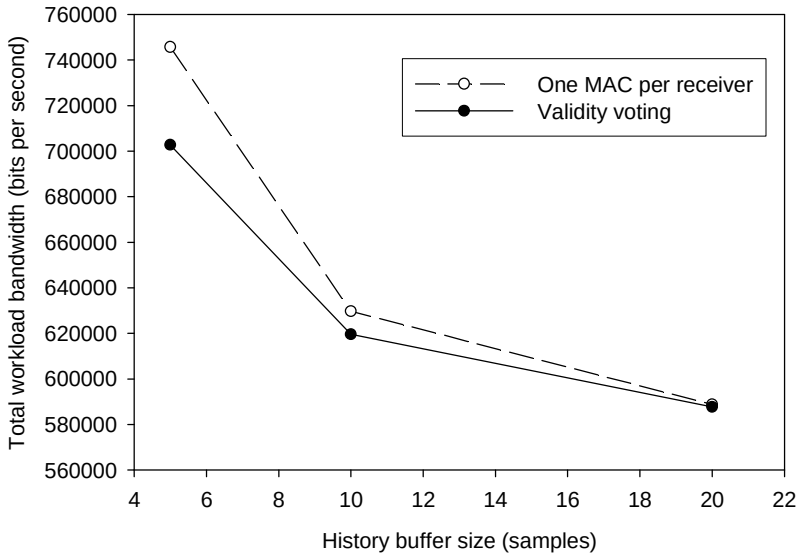

Figure 7.4. Validity voting total bits per second transmitted on CAN bus (includes CAN protocol overhead).

Table 7.8 and Figures 7.3 and 7.4 confirms that voting produces the greatest reduction in authentication bandwidth for stronger per-packet assurance levels (i.e., smaller history buffer sizes). Voting provides the greatest reduction in authentication bandwidth overhead for stronger per-packet assurance levels and greater numbers of receivers. For a history buffer size of five samples, more votes could be applied that reduced bandwidth consumption. Reducing total bandwidth from a 56% increase to 47%. For a history buffer size of ten samples, the reduction was from 32% to 29%. For twenty samples, there was less than one percent difference. With weaker per-packet assurances (larger history buffer sizes), fewer votes could be applied to reduce authentication bandwidth. Appendix A provides all the votes applied to reduce bandwidth.

While validity voting provides less bandwidth savings as MAC tag size decreases, it can instead be used to strengthen per-packet assurance (reducing the number of samples a receiver must verify state-changes or actuations over) without increasing bandwidth overhead. This as-

pect of validity voting is likely more useful to reduce application level latency using the same MAC tag sizes as those for history buffer sizes of twenty samples.

7.3 TESLA

For TESLA we used the same per-packet assurance levels and history buffer sizes as those for one MAC per receiver and a key size of eighty bits (Table 7.9). To allow verification of each message sample individually, transmitters must also send a key used to compute the corresponding MAC tag. As in elevator implementation, nodes are scheduled to transmit the key during the message round after the round in which the corresponding value and MAC tag are transmitted. Transmitting a key required at least one additional packet to be broadcast for each message type. This analysis did not examine the bandwidth required for transmitting one key for multiple message types from the same sender.

For simplicity, we assume each node maintains one key chain for each message type they transmit. This workload does not contain information about what messages are used for by each receiver, how often nodes execute their control loops, or whether batch-authenticating multiple message types together using a single key chain would be acceptable. Unfortunately, this approach requires transmitting an eighty bit key for every sample of every message type, which creates a very high bandwidth requirement. In Section 5.2.4, we discuss some tradeoffs associated with using fewer key chains.

Table 7.9. TESLA history buffer size, per-packet assurance, MAC tag size, and key size.

History buffer size (samples)	Desired failure rate	Required per-packet assurance	MAC tag size (bits)	Key size (bits)
5	10^{-9}/hr	2^{-10}	10	80
	10^{-6}/hr	2^{-8}	8	80
	10^{-3}/hr	2^{-6}	6	80
10	10^{-9}/hr	2^{-5}	5	80
	10^{-6}/hr	2^{-4}	4	80
	10^{-3}/hr	2^{-3}	3	80
20	10^{-9}/hr	2^{-3}	3	80
	10^{-6}/hr	2^{-2}	2	80
	10^{-3}/hr	2^{-2}	2	80

Tables in Appendix A provide a detailed breakdown of bandwidth required for authentication and the messages of the workload.

7.3.1 TESLA - summary

Table 7.10 summarizes the results of applying TESLA to the automotive network workload. The authentication overhead is mostly constant. Figures 7.5 and 7.6 show the decrease in bandwidth as history buffer size increases. Although the tag sizes decreases exponentially as the history buffer size increases, the transmitted key material makes up a majority of bandwidth required for authentication. Thus, altering the size of the history buffer does not provide much benefit when authenticating with TESLA (Percent increase over baseline workload bandwidth is between 147% and 140%). Also, since the changes in tag sizes from one history buffer parameter value to another are at most a few bits in size, this reduction often is not large enough to reduce the payload size by a byte. This further reduces the effects of changing history buffer sizes for TESLA.

As shown in Section 5, TESLA is best suited for applications which require very high per-packet assurance (e.g., event-triggered systems which must verify state changes or actuations

over single samples), or applications which must scale to hundreds or thousands of receivers (e.g., enterprise systems which distribute media to thousands of consumers).

Bandwidth for this approach could be reduced by authenticating multiple message types from a single sender using one key chain, rather than using one key chain for each message type.

Table 7.10. TESLA bandwidth summary.

	History buffer size (samples)		
	5	10	20
Bandwidth increase due to authentication (bits per second)	263537	250917	245377
Total bandwidth including CAN protocol overhead (bits per second)	1184004	1170289	1149747
Percent increase in total bandwidth over baseline	147 %	144 %	140 %

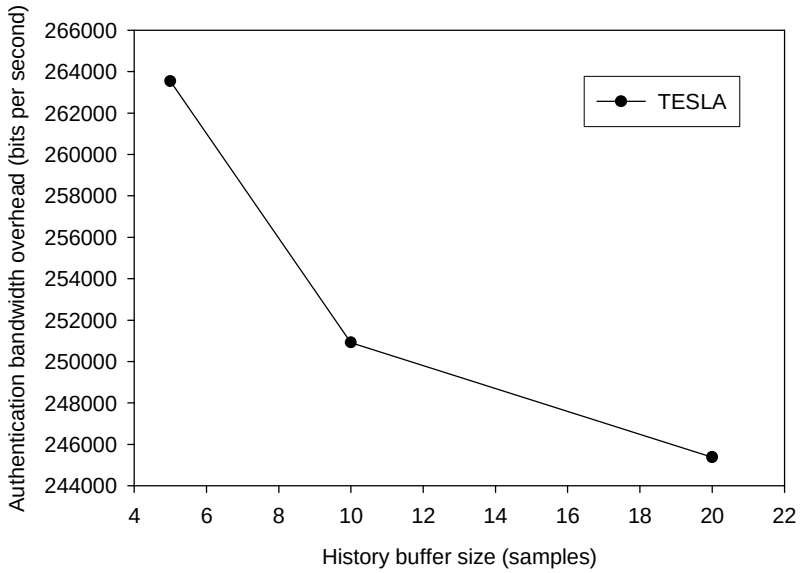

Figure 7.5. TESLA authentication bits per second (no CAN protocol overhead), varying history buffer size from five samples to twenty samples.

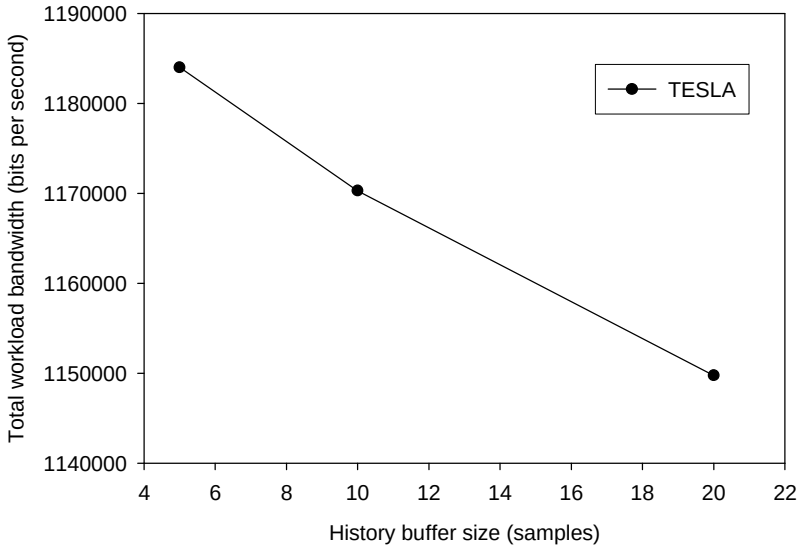

Figure 7.6. TESLA total bits per second transmitted on CAN bus (includes CAN protocol overhead)

7.4 Master-slave

For our bandwidth analysis for master-slave, we again used the same per-packet assurance levels and history buffer sizes as other techniques. Table 7.11 shows the MAC tag sizes for each assurance level and history buffer size.

Table 7.11. Master-slave history buffer size, required per-packet assurance, and MAC tag size.

History buffer size (samples)	Desired failure rate	Required per-packet assurance	MAC tag size (bits)
5	10^{-9}/hr	2^{-10}	11
	10^{-6}/hr	2^{-8}	9
	10^{-3}/hr	2^{-6}	7
10	10^{-9}/hr	2^{-5}	6
	10^{-6}/hr	2^{-4}	5
	10^{-3}/hr	2^{-3}	4
20	10^{-9}/hr	2^{-3}	4
	10^{-6}/hr	2^{-2}	3
	10^{-3}/hr	2^{-2}	3

The master-slave approach described in Section 5 requires that all receivers broadcast a message as part of verifying the hash tree broadcast authenticator from the master node. There were three issues with in applying the master-slave approach to this workload:

- First, the workload does not contain data regarding node control loop execution periods. Thus, we do not know how often they must verify messages they consume. To resolve this, we assume that all broadcasting nodes execute their control loops at approximately the same period as the most frequent message type they broadcast. Thus, the message type from each node with the shortest broadcast period contains the MAC tag for verifying the hash-tree broadcast authenticator from the master node.
- Second, we did not allow lower assurance message types to participate in authentication of higher assurance message types. This limitation was primarily based on the tag sizes for messages of each assurance level (similar to our application of validity voting in Section 7.2). To resolve this, we included a message type from the master node for each assurance level (high, medium, and low assurance message types). For each assurance group that a node consumes at least one message type from, that node participates in verifying the hash tree broadcast authenticator for that assurance group.
- Third, some nodes did not broadcast messages in the high, medium, or low assurance message type groups. We treated these nodes as "silent receivers" for those assurance groups that they did not transmit messages in. We added a message type to the workload for any node that did not already broadcast a message type within that group. Again, we assume each of these nodes executes their control loops at approximately the same period as the most frequent message type they broadcast. Alternatively, there are a few message types in the non-authenticated group whose data payloads did not already contain the maximum amount of

data for a CAN payload. These message types could be moved from the non-authenticated group to one of the other assurance groups. For simplicity, we did not attempt to move message types from the non-authenticated group to another assurance group.

In addition to resolving these issues, we added a trusted master node and one message type to carry its confirmation bit and hash-tree broadcast authenticator for each assurance level (three total message types were added for the master node).

Tables in Appendix A provide a detailed breakdown of bandwidth required for authentication and the messages of the workload. Appendix A also identifies the added message types and which message types carry MAC tags for verification of the hash tree broadcast authenticators.

7.4.1 Master-slave - summary

Table 7.11 summarizes the results of applying master-slave to the automotive network workload. Figures 7.5 and 7.6 show the decrease in bandwidth as history buffer size increases. The bandwidth required specifically for authentication data is extremely small in comparison to that for other approaches. However, since several message types were added for each assurance level, the overall bandwidth is about the same as OMPR or validity voting.

Table 7.12. Master-slave bandwidth summary.

	History buffer size (samples)		
	5	10	20
Bandwidth increase due to authentication (bits per second)	45153	24980	16221
Total bandwidth including CAN protocol overhead (bits per second)	707175	638491	606741
Percent increase in total bandwidth over baseline	48 %	33 %	27 %

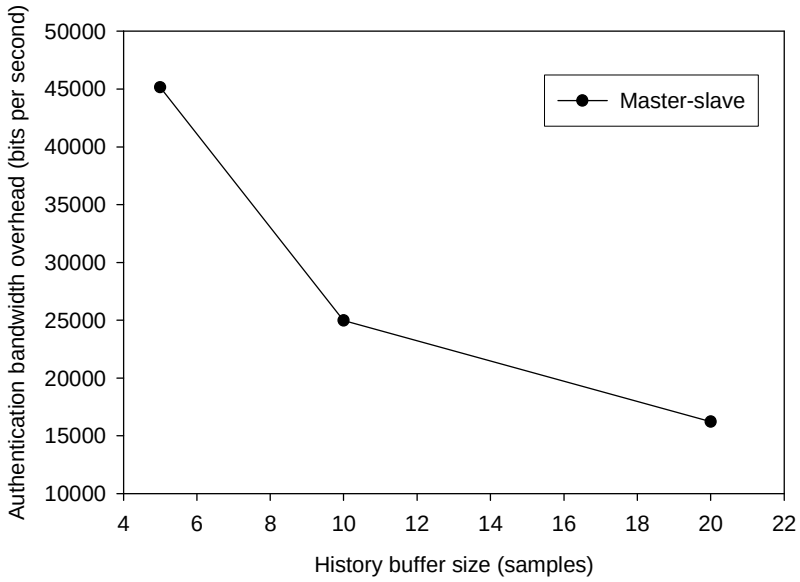

Figure 7.7. Authentication bits per second (no CAN protocol overhead), varying history buffer size from five samples to twenty samples.

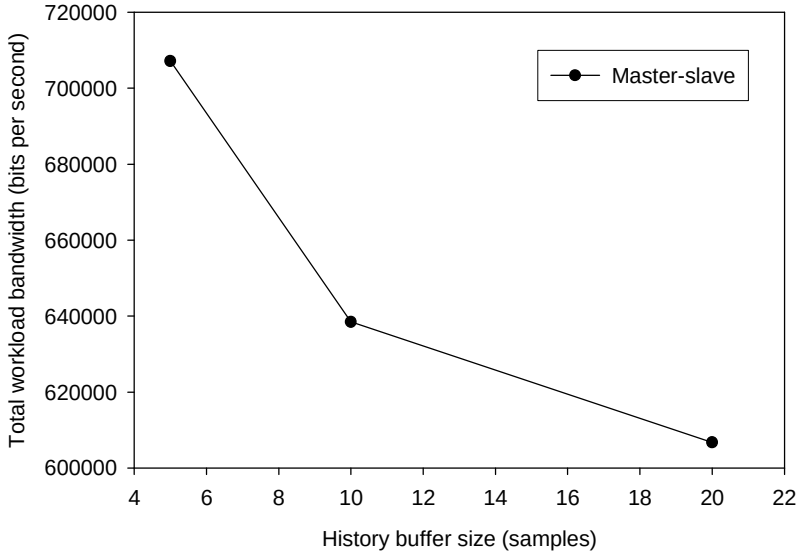

Figure 7.8. Total bits per second transmitted on CAN bus (includes CAN protocol overhead)

7.5 Discussion

Table 7.13, 7.14, and 7.15 compare all four techniques in terms of the bandwidth required for authentication, the total bandwidth for the entire workload, and the percent increase in bandwidth over baseline workload (techniques with lowest values are highlighted). For reference, the baseline network workload without authentication is 478782 bits per second. After applying all four techniques, the technique that required the least bandwidth for authentication was master-slave. However, the technique requiring the least overall total bandwidth for the workload was validity voting. In this case study, OMPR also required less overall bandwidth than master-slave (for ten and twenty sample history buffers). The reason that master-slave required a more significant increase in bandwidth than in the elevator case study is that multiple message types had to be added for verification of the hash tree broadcast authenticators from the master. This illustrates a fundamental practical limit of master-slave. If nodes do not already broadcast (in this case within each assurance level), then new messages must be added to carry authenticators. Adding new message types to carry authenticators for nodes is expensive in terms of bandwidth.

Table 7.13. Comparison of authentication bandwidth (bits per second) overhead as history buffer size is varied.

Technique	History buffer size (samples)		
	5	10	20
One MAC per receiver	90525	45292	25482
Validity voting	72368	38856	24039
TESLA	263537	250917	245377
Master-slave	45153	24980	16221

Table 7.14. Comparison of total bandwidth (bits per second) required for workload (including CAN protocol overhead) as history buffer size is varied.

Technique	History buffer size (samples)		
	5	10	20
One MAC per receiver	745662	629660	588735
Validity voting	702694	619584	587702
TESLA	1184004	1170289	1149747
Master-slave	707175	638491	606741

Table 7.15. Comparison of percent increase in total bandwidth required for workload (including CAN protocol overhead) as history buffer size is varied.

Technique	History buffer size (samples)		
	5	10	20
One MAC per receiver	56 %	32 %	23 %
Validity voting	47 %	29 %	23 %
TESLA	147 %	144 %	140 %
Master-slave	48 %	33 %	27 %

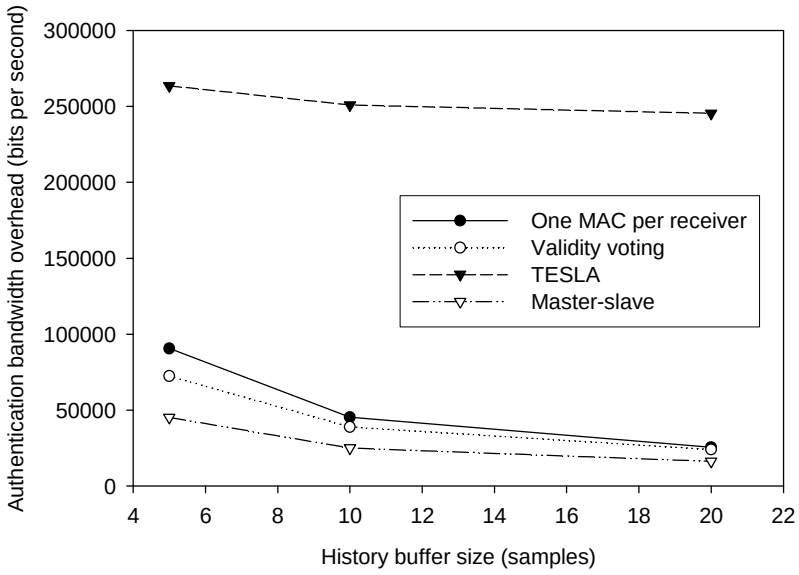

Figure 7.7. All techniques, authentication bits per second (no CAN protocol overhead) for all authentication techniques, varying history buffer size from five samples to twenty samples.

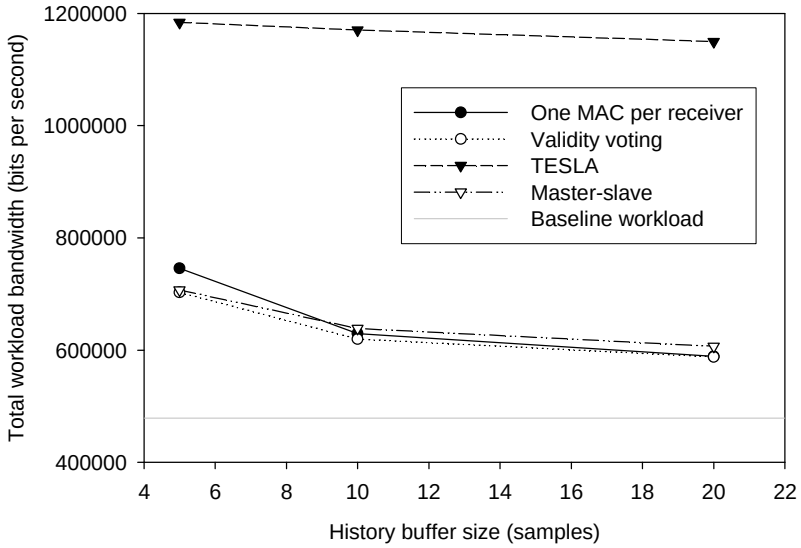

Figure 7.8. All techniques, total bits per second transmitted on CAN bus for all authentication techniques (includes CAN protocol overhead)

While one MAC per receiver had the third highest authentication bandwidth overhead, the total bandwidth required for the workload was similar to that validity voting. While this workload did not allow voting on all message types, it is reasonable to assume that votes are likely to be limited in some industry network workloads since they are not necessarily designed to support validity voting. However, in some cases, adding new authentication channels between nodes may introduce more options for votes which outweigh the cost of adding a single MAC tag. In workloads where nodes in the network receive a majority of message types, more options will exist for votes.

These results show that for systems whose sampling rates allow authentication over ten to twenty message samples, one MAC per receiver is likely the best option of the four presented here. Validity voting might reduce bandwidth consumption, but it also carries a disadvantage of reduced increased sensitivity to packet loss, and node compromise or failure. Master-slave also

has similar bandwidth consumption, but carries the disadvantage of being very sensitive to packet losses (illustrated in Section 6).

In this analysis, TESLA increases the bandwidth required for the workload to well over the one megabit per second bandwidth limit of the CAN protocol. This indicates that it might not be suitable for a typical embedded network workload where bandwidth is extremely limited. TESLA would be best applied if the application required large numbers of receivers (hundreds or thousands) or required strong per-packet assurances for event-triggered messages instead of periodic messages. Reducing the number of keys transmitted by each sender could significantly decrease the authentication bandwidth overhead.

This analysis also illustrates one of the disadvantages of the master-slave approach using hash-tree broadcast authentication: silent receivers require the addition of new messages to be broadcast on the network. Adding new message types to the network requires significant bandwidth. Thus, this approach did not perform well in terms of bandwidth. The analysis of the elevator system in Section 6 shows an example where the network has no silent receivers.

7.5.1 Limitations

The primary limitation of this analysis is that the workload included very limited information about the system being analyzed. However, the workload provided almost all information required to apply our time-triggered authentication approach in conjunction with each of the four multicast authentication techniques. With further information, selection of parameters for time-triggered authentication could be improved. This limitation required assumptions about per-packet assurance levels for all techniques. For TESLA, we did not explore possible tradeoffs for maintaining and sending fewer key chains for each sender. In master-slave, we also had to make

assumptions about control loop execution periods to make reasonable estimates of how often each node would have to participate in verifying the hash tree broadcast authenticators.

Another limitation is that we do not analyze the resulting workload (with authenticators) in terms of schedulability. All techniques increase the bandwidth required for the workload significantly. Such increases in bandwidth are unavoidable if the system must be protected from masquerade attacks intended to maliciously induce system failures. Future work may include analysis of the impacts of authentication techniques on schedulability.

8 Technique modifications and variations

This section describes some modifications and alternatives for some of the multicast authentication techniques used in this work. These ideas were not implemented. We leave implementation of each variation along with associated analyses for future work.

8.1 OMPR - Shared keys within groups

One of limitations of using OMPR is that the processing and authentication bandwidth scales linearly with the number of receivers. One way to address this limitation is to reduce the number of MAC tags to be computed of a message type by grouping receivers. A set of receivers might be grouped together based on criticality or function. Each group shares one symmetric key used for communication within the group and shares a different key for each external group to be communicated with.

For example, if partitioning by criticality, the system designer might partition the nodes in a network into critical and non-critical nodes. In this case, at most two MAC tags are required for any broadcast message. Since no node in the non-critical group knows the key shared among the critical nodes, a non-critical node cannot spoof messages (maliciously or accidentally) to a node in the critical group.

This may be useful if a security analysis determines that the most likely node to be compromised does not directly control safety critical functionality (e.g., an Internet or wireless gateway node), or physical access to critical nodes is limited to trusted personnel. Nodes that are more likely to be compromised can be partitioned into another group that does not have the required key material to authenticate messages among critical nodes.

The benefit of this approach is that it can decrease the number of MAC tags required per packet (in our example, at most two tags per packet are needed). This reduces processing time as well as authentication bandwidth. Nodes can be partitioned into any number of groups. Further, communications between groups can be limited based on key material held (as illustrated in our example).

The limitation of this approach is that by sharing symmetric keys among a set of nodes, it is no longer possible to determine which node within a group actually broadcast a message. This concern is not limited to a compromised node spoofing messages within a group. In the event that a node suffers a non-malicious failure and accidentally masquerades as another node, it may no longer be possible to identify that node for fault isolation purposes.

8.2 OMPR - Tuning on a per-message type and per-receiver basis

OMPR allows tuning of time-triggered authentication parameters on a per-message type and per-receiver basis. When selecting the number of authentication bits per MAC tag and the number of message samples to verify across, each message type and receiver can be considered individually. The per-packet assurance requirements may differ among receivers for the same message type.

When tuning on a per-receiver basis, a system designer can examine what functionality each message type is used to support (e.g., is it used by a safety-critical function, system performance, or convenience feature). For example, a message containing the vehicle speed may be used in optimizing system performance, but is also consumed by nodes such as door locks (doors might automatically lock once a vehicle reaches a certain speed) or the infotainment system (displaying current vehicle performance characteristics). A system designer can devote more authentication

bits in a data payload for receivers that are optimizing performance characteristics, while other receivers like the infotainment system might only require a single bit to be able to eventually detect a masquerade attack. Similarly, those receivers might verify state changes and actuations over differing numbers of message samples based on timing requirements.

The system designer can also divide message types into different criticality levels (similar to the partitioning in Section 7); each set may have different requirements for failure rates. Time-triggered authentication parameters can then be selected for each message type.

The benefit of tuning on a per-message type or per-receiver basis is that it allows more efficient use of system resources (which are likely already limited in an embedded control network). This approach can also improve system performance (e.g., creating equal state transition delays for two message types that are broadcast at different periods).

The limitation of such tuning is that it can significantly increase design complexity. Further, this complexity increases again if validity voting is applied to the design, introducing new tradeoff parameters.

8.3 Validity voting - Tolerating asymmetric packet loss

One of the limitations of the baseline validity voting scheme described in Section 4 is that asymmetric packet losses can cause invalid authenticators, creating a false alarm of a masquerade attack. Asymmetric packet losses can cause one node to receive a correctly formatted packet, while another node receives a malformed packet. The first node will record the value in the packet, while the second node will record the packet as lost. If these two nodes participate in validity voting on the observed message, they will not be able to compute a correct MAC tag (since

they are computing over a different set of values). Thus, they might misinterpret an otherwise non-malicious fault as a malicious one.

In baseline validity voting (described in Section 4), an invalid authenticator due to asymmetric packet loss can occur in two cases. Consider two receivers N_1 and N_2 that consume a message *m*. Node N_1 then broadcasts a vote m_v that N_2 consumes. An asymmetric omissive fault affecting message *m* can cause N_2's verification result of m_v to be invalid.

- Case 1: N_1 drops message *m* and N_2 correctly receives *m*. N_1 will record *m* as a predefined error code 'lost' while N_2 records the actual value. When verifying m_v, N_2 assumes that N_1 computed its authenticators over the same set of message values that N_2 observed from the network. In baseline validity voting, N_1 does not have a channel to communicate to N_2 which messages were lost. Thus, N_2 will record both *m* and m_v as invalid.
- Case 2: N_1 correctly receives message *m* and N_2 drops *m*. N_1 records the actual value of message *m*, while N_2 records *m* as 'lost.' Again, N_2 has no way of knowing that N_1 did not lose the value (even if N_2 did know that N_1 did not lose *m*, N_2 still would not know the value of *m*). Thus, N_2 will also record *m* and m_v as invalid in this case as well.

To address case 1, we propose each voter include a *loss vector* in the payload of its transmitted messages. The loss vector contains one bit for each message value being voted upon, indicating whether each message value was recorded as 'lost,' or received correctly. In our example for case 1, this would create a channel by which N_1 can communicate the set of message values that were lost to N_2.

Loss vectors work the same way validity vectors work in Section 4. When transmitting a message, for each message value inputted to the MAC functions that is 'lost', the sender sets the corresponding bit in the loss vector to a '1.' If the sender recorded the message as invalid (with a '0' in the validity vector), then the corresponding loss vector bit should be a '0' as well. When receiving and verifying a message, for any bit that is a '1' in the loss vector, the receiver replaces that message value input to the MAC function with the 'lost' error code. The receiver's computed MAC tag will then match the sender's since each was computed over the same set of values.

Loss vectors require additional bits to be placed within a data payload, but prevents one case in which an asymmetric packet causes an invalid MAC tag.

Addressing case 2 is more difficult, because no backwards channel (from N_2 to N_1) exists for N_2 to communicate to N_1 the set of messages N_2 did not receive. One way to handle this case is for N_2 to record m_v as 'lost' (along with all other packets containing votes on m) in addition to recording m as 'lost.' This method will cause all values that would normally be recorded as invalid (using baseline validity voting) to be instead recorded as 'lost.' This includes all the messages being voted upon by m_v, in addition to m. One exception to this however, is that if m was lost, but m_v indicates m was invalid, then N_2 should still reject m as invalid.

A disadvantage of addressing case 2 in this way is that it increases the number of packet losses in the event of a symmetric packet loss. Baseline validity voting allows receivers to continue authenticating voting messages despite symmetric packet losses affecting the messages being voted upon. However, with this modification, a receiver drops any message carrying votes for any message suffering any type of packet loss.

Another option (though not recommended) to handle asymmetric packet losses might be for a receiver to speculate on which packets have been lost asymmetrically. However, this is would also increase the probability of successful packet forgery by an attacker. If a receiver detects an invalid MAC tag, the receiver can sequentially replace each message value being voted upon by the 'lost' error code. If one combination of values and 'lost' error codes results in a valid MAC tag, this might indicate an asymmetric loss has occurred. This requires up to 2^x MAC function computations to check all possible combinations, where x is the number of message values being voted upon. The disadvantage is that this speculation increases the probability of a successful packet forgery by an attacker. During an actual masquerade attack, a receiver might misinterpret a forgery attempt as an asymmetric packet loss for any of those 2^x combinations. Each combination inputs to a MAC function has a 2^{-b} probability of producing a valid authenticator, where b is the number of MAC tag bits. Another issue is that multiple speculated combinations might result in valid MAC tags. The receiver would then have to guess which is the correct

In future work, these approaches should be analyzed to ensure that an attacker cannot exploit these mechanisms to successfully inject message forgeries undetected.

8.4 Validity voting - Improving tolerance to packet loss and node failure

In this section, we propose two methods to improve tolerance to packet loss and node failures. If a message carrying votes is dropped, then all message values being voted upon will also be dropped by receivers. A persistent fault could permanently prevent any authentication of multiple message types.

The techniques in this section could also be applied to our master-slave approach using hash tree broadcast authentication to reduce the impact of packet losses and node failures.

8.4.1 Assume a fixed level of packet loss

To allow validity voting more tolerance to permanent node failures, nodes could accept a valid packet after receiving a fraction of the confirmation packets carrying votes. However, accepting a value with only partial confirmation from the rest of the group increases the probability of per-packet forgery, requiring more bits in MAC tags to compensate.

To tolerate at most *y* lost votes, a receiver accepts a packet as valid so long as no more than *y* confirmation packets carrying votes are lost (from transient or permanent faults). If the receiver drops more than *y* confirmations for a value, then the receiver drops the value being voted upon as lost. To tolerate this fixed level of packet loss, a system designer will have to increase the number of authentication bits per MAC tag or increase the number of messages to authenticate over.

We do not attempt to assign a specific probability of successful forgery for this approach, leaving this analysis for future work.

Using this approach also grants an attacker new options when attempting to forge a packet. Typically, in baseline validity voting with *z* voting nodes, a node must receive all *z* votes before a message can be recorded as valid. Thus, an attacker would have to successfully forge messages to or from all *z* voting nodes. However, by expecting *y* of these *z* votes to be lost, this approach effectively grants an attacker *y* free tries to forge MAC tags in a sender's initial packet containing the value being voted upon. The attacker first attempts to forge a sufficient number of tags correct in the initial value packet. The attacker then examines the validity bits in confirmation packets containing votes to determine how many initial guesses were correct and how many more are needed. If the attacker gets an insufficient number of tags correct in the initial value packet, it attempts to forge a sufficient number of the confirmation packets to force the value to be ac-

cepted by the targeted receiver. The attacker can drop up to *y* confirmation packets that indicating the initial forgery attempt for that tag failed. Thus, a system designer will need to use more bits per MAC tag or verify state changes and actuations over more message samples.

The benefit of this method is that the number of authentication rounds remains constant for time-triggered authentication. The disadvantage is that the number of lost packets tolerated is fixed, limiting the system to suboptimal performance. The receiver gets no benefit from any additional confirmations past those expected. Also, if more confirmation packets are lost than the maximum tolerated number, then the packet containing the value being voted upon is still recorded as lost.

8.4.2 Group membership to remove sources of failure

As an additional optional service on top of authentication, we can monitor each message type and remove those that repeatedly interfere in the voting process using group membership techniques. Typically, group membership techniques allow a group of nodes to agree on the subset of those nodes which are present and operating correctly [Cristian88]. For validity voting, we can use group membership to determine the set of correct and present message types in the schedule in addition to the sending nodes themselves. A message type may interfere with voting if it is repeatedly dropped or is repeatedly invalid (e.g., from a masquerade fault). Because the transmitter of a message type might not be the source of the fault, nodes remove message types from validity voting as a form of task reconfiguration after agreeing on the set of correctly operating nodes.

The Multicomputer Architecture for Fault Tolerance (MAFT) provides a group membership service that can be used to monitor and remove faulty message types in addition to faulty nodes

[Kiechafer98]. In MAFT, nodes execute the membership service at periodic intervals. To track the faulty behaviors of other nodes, each node keeps two penalty counters for each other node based upon their message traffic. A base penalty counter (BPC) indicates the current value of accrued penalties for every node at the point of the last membership period. An incremental penalty counter (IPC) contains a proposed penalty assessment for each node based on detected errors since the last membership period. At the beginning of each membership period, all nodes exchange and reach Byzantine agreement on these counters using an Interactive Consistency algorithm (e.g., [Pease80]). Once completed, each node compares the new BPC values to an exclusion threshold and then broadcasts a new suggested membership. Nodes perform a second execution of the Interactive Consistency algorithm to agree upon the new membership of the group.

The IPC can be incremented for any faulty behavior defined for the system. For validity voting, a receiver could increment a message type's IPC error counter for any reason which may interfere with voting. For example, the IPC could be incremented for packet loss, invalid authenticators, or disagreement with authentication results for messages voted upon.

Repeatedly dropped packets of a particular message type may indicate the node that broadcasts the message type has silently failed or persistent network interference against that message type. Removing the affected message types from the voting scheme would prevent those message types from repeatedly causing other message types to be lost.

Similarly, invalid authenticators might indicate a particular message type is targeted by masquerade attacks or affected by persistent asymmetric packet losses. Removing message types that are repeatedly invalid from the voting process prevents additional non-targeted message types from being repeatedly invalidated.

If a voting node repeatedly disagrees with other voters, it may indicate that the node has suffered some failure. For example, it may always indicate samples of some message type are invalid, when all other voters indicate they are valid. Conversely, a majority of nodes might repeatedly marked the values of a message type as invalid due to masquerade faults, but one of the voting nodes repeatedly broadcasts a positive confirmation with a correct authenticator. These two cases might also indicate a node has been compromised either to propagate message forgeries or create a denial of service attack.

This list of reasons to increment error counters is not intended to be exhaustive. There may be other types of observable faults that would warrant incrementing an error counter.

Error counters can also be maintained for each node in addition to message types. MAFT describes how to maintain error counters on a node-by-node basis to determine which nodes are present and operating correctly. We do not explore tracking errors on a per-node basis in this work.

Periodically, group members exchange error counters, to determine if message types should be removed from the voting scheme. Once nodes exchange and agree upon error counters, each node proposes a new node membership and list of correctly operating message types. Nodes vote to remove any node or message type whose counter exceed some predefined exclusion threshold. If a node is found to be faulty during the membership exchanges, the accusing nodes might also vote to remove any message types originating from the offending node from voting. Nodes agree on these two lists via an Interactive Consistency algorithm. Once completed, nodes remove any node convicted as faulty from membership, and reconfigure the number of votes to remove any message type considered to be faulty. The group will also have to agree on new time-triggered

authentication parameters (bits per MAC tag and number of message samples to authenticate across).

The main benefit of using group membership in conjunction with validity voting is that it allows a system to recover from permanent (or persistent) faults that would prevent authentication of messages. The approach is limited in that it cannot identify and remove faulty nodes or message types immediately, and executing the membership service will require additional processing and bandwidth. Further, the messages related to the Interactive Consistency algorithm must also be authenticated using a scheme that will provide strong per-packet assurance. We leave further analysis and implementation for further work. This section is only intended to discuss the possible benefits and limitations of applying group membership to validity voting.

8.4.3 Variable number of confirmations (not secure)

Allowing a variable number of confirmations is not secure. Instead of assuming a fixed number of votes will be lost, it is tempting to allow a receiver to act on a variable number of votes. This would potentially allow a receiver to act on a particular sample of a message type with more or less assurance that it is valid, depending on the number of positive votes received.

However, using the same attack listed above for a fixed level of packet loss, an attacker can simply examine which forgeries on the initial packet containing a value succeeded, and drop any confirmation packets that contain a negative vote. Thus, allowing a variable number of confirmation packets grants an attacker up to z free tries to forge votes. It does not necessarily allow a receiver to detect that votes were tampered with.

8.5 TESLA - Using fewer key chains

In some systems, a system designer can perform tradeoffs for TESLA with respect to the number of key chains that each node maintains. In both case studies, we assumed that each sender would maintain a distinct key chain for authenticating each message type it broadcasts.

In the elevator control network case study, no tradeoffs were possible; all nodes only broadcast a single message type. Thus, each node maintained a single key chain. In the automotive case study, we did not have sufficient system information to perform a tradeoff analysis to determine whether one key chain per message type, one key chain per sender, or some number in between would be best. Thus, for simplicity, we limited our analysis to one key chain per message type.

However, in the automotive case study, transmitting fewer keys would reduce the added bandwidth for authentication. In Section 5.2.4, we briefly discussed tradeoffs associated with maintaining different numbers of key chains in each transmitting node. In the automotive example, maintaining a key chain for each message type increased the number of packets transmitted on the network. Using fewer key chains could eliminate many of those extra packets.

In future work, tradeoffs related to key chains should be performed to minimize system resources consumed by authentication while also minimizing the impacts to loss tolerance (e.g., avoiding batch authentication of too many message types) and system performance in an embedded control network. Perrig et al. have already explored some aspects related to using different numbers of key chains (e.g., using different key chains to authenticate messages to receivers consuming messages at different rates [Perrig00]), though these analyses are not specifically focused on embedded control networks. Groza and Murvay also explore some tradeoffs of different

numbers of key chains, focusing on memory and processing overhead rather than bandwidth overhead for authentication [Groza11].

8.6 Master-slave - Using different multicast authentication techniques

The master-slave approach described in Section 5.3 can use any multicast authentication technique to distribute the master's hash tag to all receivers. The disadvantages of hash tree broadcast authentication limits its suitability in embedded control network applications. In particular, hash tree broadcast authentication has high sensitivity to packet loss, node failure, and passive receivers. The master node could use other multicast authentication techniques (such as OMPR or TESLA) to attest to the authenticity of a set of messages. System designers can explore the tradeoffs associated with each technique to identify a technique that best fits their system constraints.

8.7 Multiple techniques in one system

Lastly, it is also possible to use multiple authentication techniques within a single network. In Section 5, we showed that each of the multicast authentication techniques performs best depending on the system configuration (e.g., TESLA requires much less bandwidth for hundreds of receivers than OMPR). Thus, it is useful to identify when one technique may be better suited to authenticating one message type vs. another.

For example, consider a case where all message types except one are broadcast to one or two receivers and require weak per-packet assurance. The last message requires strong per-packet assurance and is broadcast to fifty receivers. OMPR is best suited for most of the messages, requiring only one or two MAC tags per packet of just a few bits each. However, for the last mes-

sage type, TESLA can be used to provide strong per-packet assurance to the large number of receivers.

In our implementations in Sections 6 and 7, we already use both OMPR in addition to validity voting; OMPR is simply validity voting with zero votes.

The benefit to this approach is that it allows more efficient use of system resources. However, using multiple multicast authentication schemes will also increase complexity in the design.

8.8 Alternate response to forgery attempts

Receivers can take any appropriately safe response to invalid packets. Another option is to increase the number of valid packets required for state changes or to update actuators in the event that invalid authenticators occur. For example, in the baseline time-triggered authentication approach described in Section 3, a receiver applies each reactive control message input if it is valid, and takes a safe action for invalid packets. Alternately, if an invalid packet is received, then a receiver can wait for two consistent valid packets before applying that input to an actuator. Similarly, a state change may occur after *n* consistent packets that are valid. If an invalid authenticator is received, then a receiver may require more than *n* consistent packets before committing to subsequent state change commands.

8.9 Composability with fault tolerance techniques

In this work, we have shown several ways to compose authentication with fault tolerance techniques. Assuming secure cryptographic functions, an attacker can only successfully forge a MAC tag randomly and independently of other MAC tags. This key property enables the use of many fault tolerance techniques in conjunction with authentication.

For example, Section 3 introduces the idea of filtering over multiple authenticated input samples which drive state changes and actuations. Section 4 shows how to vote on verification results of a message sample among multiple receivers.

Another approach that could be used is retransmitting a value multiple times for stronger assurance. A receiver could verify a repeated message sample multiple times, each with their own authenticators to strengthen per-packet assurance.

Error detection codes can also be used in conjunction with authentication to detect non-malicious transmission errors. Communication protocols often already incorporate error detection codes. We use these error detection codes to differentiate between malicious and non-malicious faults affecting packets.

Section 8.4 shows an example application of group membership to remove nodes or message types which interfere with authentication. However, group membership protocol exchanges likely require strong assurance, so an attacker cannot force a node to agree to an incorrect membership list.

8.10 Summary

This chapter discusses several possible modifications or variations of the techniques proposed in this thesis. For OMPR, we discuss the possibility of sharing keys among groups of nodes to reduce authentication overhead, and tuning time-triggered authentication parameters on a per-receiver and per-message type basis. For validity voting, we discuss methods to improve tolerance to packet loss and to prevent an asymmetric packet loss from producing in invalid authenticators. For TESLA, we discuss using fewer key chains. For master-slave, other multicast authentication techniques can be used to distribute the master's confirmation of message validity. We

also discuss the possibility of combining multiple multicast authentication techniques within an embedded control network. Finally, we discuss alternate responses to forgery attempts and composability with fault tolerance techniques.

9 Conclusion

A successful masquerade attack in an embedded control network can make a system unsafe in nearly limitless ways; multicast authentication is needed to prevent these attacks. This thesis has presented time-triggered authentication: a new method for efficiently authenticating periodic messages in an embedded control network to prevent masquerade and replay attacks. We first apply time-triggered authentication to OMPR, our baseline multicast authentication scheme. We then improved one MAC per receiver using validity voting: a method which uses voting to make more efficient use of authentication bandwidth or reduce application level latency. We also showed how to adapt TESLA and hash tree broadcast authentication (using a trusted master node) to time-triggered authentication, and compared the four multicast authentication techniques. We demonstrated the applicability of time-triggered authentication in conjunction with each of the four techniques in two representative embedded control network workloads.

9.1 Thesis contributions

To address masquerade and replay attacks in embedded control networks, this thesis has made the following contributions:

9.1.1 Time-triggered authentication using OMPR

This thesis first proposes time-triggered authentication in Chapter 3. The main idea behind this approach is that individual packets in an embedded control network typically do not need strong assurances of authenticity (i.e., hundreds or thousands of authentication bits). Instead, time-triggered authentication can provide strong system level assurance of the authenticity of state

change and actuation commands by verifying multiple message samples, each with weak per-packet assurances of authenticity (i.e., MAC tags truncated to just a few bits).

Time-triggered authentication takes advantage of the existing temporal redundancy in embedded control networks to enable verification across multiple periodic message samples; system state variables and sensor inputs are typically sampled faster than the time constraints of control stability requirements. This temporal redundancy grants the system tolerance to transient faults. An undetected fault affecting a single message sample is unlikely to cause the system to fail. More likely, it will result in some vibration, slight delay in updating control outputs, or less smooth control.

We first combine time-triggered authentication with OMPR, our baseline multicast authentication technique. A sender computes one truncated MAC tag for each receiver of a message. In Chapter 3, we show that OMPR can produce authenticators just a few bytes in size for embedded control networks requiring weak per-packet assurances and few receivers. We also verified the probability of forgery success for state-changing and reactive control message types using simulated masquerade attacks.

One of the main benefits of time-triggered authentication is that it enables a tradeoff among authentication bits per packet, application level latency, tolerance to invalid MAC tags, and probability of induced system failure. Using OMPR granted this approach perfect tolerance to packet losses, node compromise, and node failure. The main limitations are that time-triggered authentication only provides advantage to the degree of temporal redundancy in message sampling rates and the authentication overhead of OMPR scales linearly with respect to per-packet assurance and number of receivers.

9.1.2 Validity voting

We proposed validity voting as an improvement to OMPR. The main idea behind this approach is that forging multiple MAC tags to a group of receivers has lower probability of success than only forging one MAC tag to a single receiver. Validity voting takes advantage of the multiple MAC tags used in OMPR; forgery attempts on each MAC tag in OMPR succeed randomly and independently of one another. This property allows a group of nodes to cross check the validity of a message value that was authenticated to each of them using OMPR. In validity voting, each node in the group broadcasts an authenticated bit to the other nodes in the group attesting to whether a particular message value was valid or not. Once all nodes have transmitted their votes, each node takes a unanimous vote on the authenticity of the message value. Using this attestation process reduces the probability that an attacker will successfully forge a message value, for a given number of MAC tag bits.

In Chapter 4, we showed how to define votes for a set of message types in a network workload, how to implement validity voting, and how to modify the approach to tolerate a fixed number of compromised voters. We also model-checked this approach using the security model checker AVISPA. Lastly, we simulated the probability of successful forgery using simulated attacks on messages verified with one to four votes.

Validity voting adds new tradeoffs to time-triggered authentication. Increasing the number of votes allows the system designer to make more efficient use of authentication bandwidth, either decreasing the number of bit per MAC tag or the number of message samples that must be verified over in time-triggered authentication. However, increasing the number of votes also decreases the loss tolerance of this approach. If a message containing a vote suffers a transmission

error, a receiver also drops any message value that was voted upon. Further, a single invalid authenticator will cause a receiver to reject any messages being voted upon as invalid as well.

This approach also has several limitations. Validity voting only handles a fixed number of compromised nodes (set at design time). If the number of compromised nodes exceeds this number, an attacker will have a greater probability of successfully forging messages. The baseline version of validity voting in Chapter 4 also requires modifications to address asymmetric packet losses and node failures. These are discussed in Chapter 8.

9.1.3 Comparisons with TESLA and hash tree broadcast authentication

We compared OMPR and validity voting to two existing multicast authentication schemes that also use symmetric authentication functions: TESLA and master-slave (hash tree broadcast authentication using a trusted master). In Chapter 5, we first described how to apply TESLA and master-slave in conjunction with time-triggered authentication. This illustrates one way to adapt TESLA and hash tree broadcast authentication for use in an embedded control network. We then compared these four techniques in terms of scalability with respect to per-packet assurance, scalability with respect to number of receivers, sensitivity to packet loss, and tolerance to compromised or failed nodes. These comparisons illustrate tradeoffs among techniques which can be integrated with time-triggered authentication.

In this tradeoff analysis, we showed that the most bandwidth efficient approach depends primarily on the number of receivers, and is influenced to a lesser extent by per-packet assurance levels in networks where no trusted master is available. OMPR and validity voting with few votes are the most bandwidth efficient approaches for networks characterized with few receivers and weak per-packet assurance. TESLA and validity voting using many votes are the most bandwidth efficient approaches for very large numbers of receivers or strong per-packet assur-

ance levels. A master-slave approach is also very bandwidth efficient, assuming a trusted master node is available. In this analysis we experimentally demonstrated that OMPR and TESLA were least sensitive to packet losses. Master-slave was the most sensitive to packet loss; a single transmission error can cause an entire message rounds worth of values to be dropped. Validity voting's sensitivity to packet loss depended on the number of votes. A single transmission error forced a receiver to drop more packets as the number of votes increased. We also showed that despite some approaches being more sensitive to transient packet losses, all approaches are robust and recover automatically from transient faults. Using hash tree broadcast authentication in our master-slave approach also resulted in sensitivity to passive nodes; new messages must be added for any passive receiver that does not already broadcast a message of its own. Lastly we find approaches with no inter-node dependencies for authentication, such as one MAC per receiver and TESLA, are most robust to node compromises or failures. The master node in master-slave is a single point of failure.

9.1.4 Two case studies

We demonstrated the applicability of time-triggered authentication in conjunction with all four techniques using two representative network workloads. First, we implemented these techniques in a simulated distributed elevator control network and examined the impact on bandwidth and system performance. Second, we applied these techniques to an industry automotive workload and examined the impacts on bandwidth.

In the elevator, we first examined each state transition in the door controllers and drive controller to determine the effects of a masquerade attack to force or deny each transition. We identified attacks which could force the system to violate safety requirements and the associated message types that would be targeted for those attacks. We identified the minimum and maximum

time-triggered authentication parameters for per-packet assurance and number of messages to verify over (history buffer size). We used these parameters to determine the additional bandwidth required for each authentication technique. Master-slave added the least bandwidth overhead for authentication, followed by validity voting and OMPR. TESLA added the most, due to the requirement to transmit key material for each message to verify. We also experimentally tested the effects of each set of parameters on delays in state transitions, and delays in average passenger delivery times. We also applied varying levels of symmetric packet losses and examined the resulting effects on state transition delays and passenger delivery times. We observed that varying the history buffer size created a similar delay in state transitions for all four techniques. Similarly, passenger delivery times increased by approximately the same amount for all techniques. However, when applying symmetric packet losses, OMPR and TESLA had the least increase in state transition and delivery time delays. These delays increased correspondingly with the number of inter-packet dependencies for validity voting. Master-slave suffered the worst delays, since so many packets could be lost due to a single transmission error. We also performed simulated masquerade attacks to confirm the forgery success rate matches the expectations from the equations in Chapters 3 and 4.

In the automotive workload, message types were divided into four groups: high assurance, medium assurance, low assurance, and no authentication applied. For the high, medium, and low message types we assigned failure rate requirements typical to safety-critical systems. We then examined the authentication bandwidth overhead for each technique as we varied the time-triggered authentication parameters. We observed that validity voting had the least authentication bandwidth overhead, followed by OMPR. Master-slave required additional message types to be

added to transmit MAC tags for verification of the master's hash value. TESLA required the highest bandwidth overhead, again, due to the key material being transmitted.

9.2 Future work

This work is a first step in identifying multicast authentication techniques that conform to embedded control network design constraints. There are several paths that future work could take from this work.

First, this work only discusses methods to provide message authentication and data integrity; embedded control networks will also likely require approaches for key management, tamper resistance, secrecy, privacy, access control, and prevention of denial of service.

Second, Chapter 8 discusses numerous possibilities for modifications and variations of the techniques we used in this work. For example, partitioning nodes into groups for sharing authentication keys might be very useful in embedded networks where compromise of critical nodes can be restricted in some fashion. This approach is similar to some existing fault tolerance methods [Morris03]. Also, using TESLA with fewer key transmissions could amortize authentication bandwidth overhead over multiple message types.

This work also could not explore all of the possible design space for embedded control networks. The case studies in this work were limited to embedded control networks using the CAN protocol. Other protocols, such as FlexRay, offer greater bandwidth and may be able to tolerate higher authentication overhead. Further, our tradeoff analyses focused primarily on authentication bandwidth overhead and loss tolerance. Future work could include analysis of processing and memory requirements and associated impacts on system performance. Implementations and analyses on other systems using embedded control networks may also reveal new design considerations specific to those types of systems.

In this work, all time-triggered authentication and voting parameters were selected by hand. Many of the associated tasks could be automated using tools to significantly reduce the time it takes to perform these analyses. This would be especially useful for updating these parameters during system development. For example, over many design iterations message types might be added or removed, senders and receivers of message types might change, or system characteristics could change that affect the maximum number of message samples that can be authenticated across.

Lastly, we limited our analyses to multicast authentication approaches that use symmetric authenticators that can be truncated. One option for future work would be to create MAC functions that are optimized to produce outputs of just a few bits in size (our approach throws away a majority of the MAC output). Another research path is to explore digital signatures or one-time digital signatures which could produce outputs a few bits in size without compromising the security of the cryptographic functions or key material.

10 References

[AVISPA12] The AVISPA Project. Retrieved April 2012 from http://avispa-project.org/.

[Azadmanesh00] M. Azadmanesh and R. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

[Bergadano00] F. Bergadano, D. Cavagnino, and B. Crispo. Individual Single-Source Authentication on the MBONE. In *Proc. of the 2000 IEEE Int'l Conf. on Multimedia and Expo*, volume 1, pp. 541–544. IEEE, 2000.

[Bosch91] R. Bosch GmbH, CAN Specification, Version 2, Sept. 1991.

[Brown00] M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M. Kirkup, and A. Menezes. PGP in constrained wireless devices. In *SSYM'00: Proc. of the 9th Conf. on USENIX Security Symposium*, p. 19, Berkeley, CA, USA, 2000. USENIX Association.

[Canetti99] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: a taxonomy and some efficient constructions. In *INFOCOM '99: Proc. 18th Annual Joint Conf. of the IEEE Computer and Communications Societies*, volume 2, pp. 708–716. IEEE, 1999.

[Chan08] H. Chan and A. Perrig. Efficient security primitives derived from a secure aggregation algorithm. In *Proc. ACM Conf. on Computer and Communications Security*, pp. 521–534, 2008.

[Chan10] H. Chan and A. Perrig. Round-effcient broadcast authentication protocols for fixed topology classes. In *Proc. of the IEEE Symposium on Security and Privacy*, pp. 257–272, 2010.

[Chavez05] M. L. Chavez, C. H. Rosete, and F. R. Henriquez. Achieving Confidentiality Security Service for CAN. In *CONIELECOMP '05: Proc. of the 15th Int'l Conf. on Electronics, Communications and Computers*, pp. 166–170. IEEE, 2005.

[Cristian88] F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proc. of the Eighteenth Int'l Symp. on Fault-Tolerant Computing*, pp. 206 –211. IEEE, 1988.

[Diffie76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, vol. 22, 1976.

[Dolev81] D. Dolev and A. C. Yao. On the security of public key protocols. In *SFCS '81: Proc. of the 22nd Annual Symp. on Foundations of Computer Science*, pp. 350–357. IEEE, 1981.

[Even89] S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. In *CRYPTO '89: Proc. on Advances in cryptology*, pp. 263–275. Springer-Verlag, 1989.

[Ewing10] G. Ewing. Reverse Engineering a CRC Algorithm. Retrieved April 2012 from http://www.cosc.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html. March 2010.

[FIPS 180-3] Federal Information Processing Standards Publication (FIPS PUB) 180-3. Secure Hash Standard (SHS). October 2008.

[FIPS 198-1] Federal Information Processing Standards Publication (FIPS PUB) 198-1. The Keyed-Hash Message Authentication Code (HMAC). July 2008.

[FlexRay05] FlexRay Consortium. FlexRay Communications System Protocol Specification, Version 2.1, Revision A, December 2005.

[Freescale12] Freescale Semiconductor. S12XD Product Summary Page. Retrieved April 2012 from http://www.freescale.com/.

[Franklin02] G. Franklin, J. Powell, and A. Emami-Naeini. Feedback Control of Dynamic Systems. Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[Führer00] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel and M.Walther. Time Triggered Communication on CAN (Time Triggered CAN - TTCAN). 7th International CAN Conference (ICC), 2000.

[Ganeriwal05] S. Ganeriwal, S Capkun, C.-C. Han, and M. B. Srivastava. Secure time synchronization service for sensor networks. In *WiSe '05: Proc. of the 4th ACM workshop on Wireless security*, pp. 97–106. ACM, 2005.

[Gennaro97] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. In *CRYPTO '97: Proc. of the 17th Annual Int'l Cryptology Conf. on Advances in Cryptology*, pp. 180–197. Springer-Verlag, 1997.

[Groza11] B. Groza and P. Murvay. Higher Layer Authentication for Broadcast in Controller Area Networks. In *SECRYPT '11: Proc. of the Int'l Conf. on Security and Cryptography*, pp. 188-197. 2011.

[Groza11_2] B. Groza and P. Murvay. Secure Broadcast with One-Time Signatures in Controller Area Networks. In *ARES '11: Proc. of the Int'l Conf. on Availability, Reliability, and Security*, pp. 188-197. 2011.

[Herrewege11] A. Van Herrewege, D. Singelée, and I. Verbauwhede. CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography 2011*, 2011.

[Hoppe07] T. Hoppe and J. Dittman. Sniffng/Replay Attacks on CAN Buses: A simulated attack on the electric window lift classifed using an adapted CERT taxonomy. In *Proc. of the 2nd Workshop on Embedded Systems Security (WESS)*, 2007.

[Hu03] Y. Hu, M. Jakobsson, and A. Perrig. Efficient constructions for one-way hash chains. In *Applied Cryptography and Network Security*, pp. 423–441, 2003.

[IEEE610.12] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

[IEC61508] International Electrotechnical Commission. Functional Safety of electrical / electronic / programmable electronic systems. IEC 61508. 1998.

[Jakobsson02] M. Jakobsson. Fractal hash sequence representation and traversal. In *Proc. of the IEEE Int'l Symp. on Information Theory*, page 437. IEEE, 2002.

[Karlof04] C. Karlof, N. Sastry, and D. Wagner. TinySec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 162–175. ACM, 2004.

[Kiechafer98] R. Kiechafer, C. J. Walter, A. M. Finn, P. M. Thambidurai. The MAFT Architecture for Distributed Fault Tolerance. IEEE Trans. on Computers, Vol. 37, No. 4, April 1988.

[Koopman12] P. Koopman. Carnegie Mellon University. 18-649 Distributed Embedded Systems. Retrieved April 2012 from http://www.ece.cmu.edu/ ece649/.

[Koopman05] P. Koopman, J. Morris, and P. Narasimhan. Challenges in Deeply Networked System Survivability. *NATO Advanced Research Workshop on Security and Embedded Systems*, pp. 57–64, 2005.

[Kopetz97] H. Kopetz. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Koscher10] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Sha-cham, S. Savage. Experimental Security Analysis of a Modern Automobile, In Proc. of the IEEE Symposium on Security and Privacy, pp.447-462, 2010.

[Krawczyk97] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed Hashing for Message Authentication," Feb. 1997, RFC 2104.

[Lamport82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, 1982.

[Lang07] A. Lang, J. Dittman, S. Kiltz, and T. Hoppe. Future Perspectives: The car and its IP address - A potential safety and security risk assessment. In *Proc. of the 26th Int'l Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, 2007.

[Lenstra01] A. Lenstra and E. Verheul. Selecting Cryptographic Key Sizes. Journal of Cryptology vol. 14(no. 4):pp. 255-293, 2001.

[Luk06] M. Luk, A. Perrig, and B. Whillock. Seven Cardinal Properties of Sensor Network Broadcast Authentication. In *Proc. of the 4th ACM Workshop on Security of Ad Hoc and Sensor Networks*, pp. 147-156. ACM, 2006.

[Martin10] T. Martin, N. White, and A. Jameson. 18-649 Course Project: Java Simulated Elevator Controller Implementation and Design. Carnegie Mellon University, May 2010.

[Menezes96] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.

[Miner01] S. Miner and J. Staddon. Graph-Based Authentication of Digital Streams. In *SP '01: Proc. of the 2001 IEEE Symposium on Security and Privacy*, pp. 232–246, 2001.

[Morris03] J. Morris and P. Koopman. Critical Message Integrity Over A Shared Network. *5th IFAC Int'l Conf. on Fieldbus Systems and their Applications*, 2003.

[Nace02] W. Nace. Graceful Degradation via System-wide Customization for Distributed Embedded Systems. Ph.D. dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, May 2002.

[Neumann56] J. von Neumann. Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components. In Automata Studies (Annals of Mathematics Studies, no. 34), pp. 43-99. Princeton Univ. Press, Princeton NJ, USA, 1956.

[Nilsson08] D. Nilsson and U. Larson. Simulated Attacks on CAN Buses: Vehicle virus. *5th IASTED Asian Conf. on Communication Systems and Networks*, 2008.

[Nilsson08_2] D. Nilsson, U. Larson, E. Jonsson. Efficient In-Vehicle Delayed Data Authentication Based on Compound Message Authentication Codes. In *Proc. of the Vehicular Technology Conference*, pp. 1-5. IEEE, 2008

[Park02] J. M. Park, E. K. P. Chong, and H. J. Siegel. Efficient Multicast Packet Authentication Using Signature Amortization. In *SP '02: Proc. of the Symposium on Security and Privacy*, pp. 227–240. IEEE, 2002.

[Pease80] M. Pease, R. Shostak, L. Lamport. Reaching Agreement in the Presence of Faults. Journal of the ACM vol. 27(no. 2), April 1980.

[Perrig00] A. Perrig, J. D. Tygar, D. Song, and R. Canetti. Efficient Authentication and Signing of Multicast Streams over Lossy Channels. In *SP '00: Proc. of the 2000 IEEE Symposium on Security and Privacy*, pp. 56–73. IEEE, 2000.

[Perrig01] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *CCS '01: Proc. of the 8th ACM Conf. on Computer and Communications Security*, pp. 28–37. ACM, 2001.

[Perrig02] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: security protocols for sensor networks. *Wireless Networks*, vol. 8(no. 5):pp. 521–534, 2002.

[Ray09] J. Ray, P. Koopman. Data Management Mechanisms for Embedded Systems Gateways. In *DSN '09: Proc. of the Int'l Conference on Dependable Systems and Networks*, pp. 175-184, 2009.

[Rivest92] R. Rivest, "The MD5 Message-Digest Algorithm," April 1992, RFC 1321.

[Schneier95] B. Schneier. Applied Cryptography (2nd ed.): Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Shelton03] C. Shelton. Scalable Graceful Degradation for Distributed Embedded Systems. Ph.D. dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, June 2003.

[Shirey00] R. Shirey. "Internet Security Glossary," May 2000, RFC 2828.

[SuperCoupe12] Super Coupe Club of Iowa. 0-60 and ¼ mile times for factory stock vehicles. Retrieved April 2012 from http://www.albeedigital.com/ supercoupe/articles/0-60times.html.

[TTTech03] TTTech. Time-Triggered Protocol Specification TTP/C, Version 1.1, November 2003.

[Wolf04] M. Wolf, A. Weimerskirch, and C. Paar. Security in Automotive Bus Systems. *Workshop on Embedded Security in Cars*, 2004.

[Wong98] C. K. Wong and S. S. Lam. Digital Signatures for Flows and Multicasts. In *ICNP '98: Proc. of the 6th Int'l Conf. on Network Protocols*, pp. 198–209. IEEE, 1998.

10.1 Thesis Publications

[Szilagyi08] C. Szilagyi and P. Koopman. A flexible approach to embedded network multicast authentication. In *Proc. of the 2nd Workshop on Embedded Systems Security (WESS)*. 2008.

[Szilagyi09] C. Szilagyi and P. Koopman. Flexible multicast authentication for time-triggered embedded control network applications. In *DSN '09: Proc. of the Int'l Conference on Dependable Systems and Networks*, pp. 165–174, 2009.

[Szilagyi10] C. Szilagyi and P. Koopman. Low cost multicast authentication via validity voting in time-triggered embedded control networks. In *WESS '10: Proc. of the Workshop on Embedded Systems Security*, 2010.

Appendix A - Automotive network workload analysis data

A.1 One MAC per receiver

A.1.1 One MAC per receiver - history buffer size = 5 samples

Table A.1. High assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 10 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	8	80	16	8000	32000
ID_008	10	49	1	10	8	1000	16000
ID_047	10	49	9	90	18	9000	42000
ID_040	12	62	1	10	9	833.3333	20833.33333
ID_001	12	55	2	20	10	1666.667	21666.66667
ID_007	12	64	12	120	23	10000	39166.66667
ID_039	20	36	2	20	7	1000	7500
ID_042	20	24	1	10	5	500	6500
ID_025	25	52	1	10	8	400	6400
ID_029	25	64	1	10	10	400	10400
ID_030	25	64	4	40	13	1600	11600
ID_038	25	56	1	10	9	400	10000
ID_036	25	64	3	30	12	1200	11200
ID_074	25	16	1	10	4	400	4800
ID_046	30	52	2	20	9	666.6667	8333.333333
ID_057	30	60	2	20	10	666.6667	8666.666667
ID_076	35	52	1	10	8	285.7143	4571.428571
ID_077	35	34	1	10	6	285.7143	4000
ID_078	35	34	1	10	6	285.7143	4000
ID_058	50	33	4	40	10	800	5200
ID_081	50	45	4	40	11	800	5400
ID_061	50	46	3	30	10	600	5200
ID_098	100	37	1	10	6	100	1400
ID_060	100	12	1	10	3	100	1100

Table A.2. Medium assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 8 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	1	8	5	1333.333	21666.66667
ID_004	10	64	10	80	18	8000	42000
ID_005	10	64	11	88	19	8800	43000
ID_010	12	61	4	32	12	2666.667	23333.33333
ID_003	12	9	1	8	3	666.6667	9166.666667
ID_026	12	31	1	8	5	666.6667	10833.33333
ID_027	12	62	2	16	10	1333.333	21666.66667
ID_048	12	59	1	8	9	666.6667	20833.33333
ID_052	12	61	1	8	9	666.6667	20833.33333
ID_041	20	26	3	24	7	1200	7500
ID_045	20	27	1	8	5	400	6500
ID_024	20	11	5	40	7	2000	7500
ID_049	20	62	12	96	20	4800	22000
ID_028	25	16	1	8	3	320	4400
ID_033	25	45	1	8	7	320	6000
ID_106	25	17	1	8	4	320	4800
ID_031	25	54	1	8	8	320	6400
ID_034	25	62	1	8	9	320	10000
ID_035	25	57	8	64	16	2560	12800
ID_037	25	48	2	16	8	640	6400
ID_075	50	40	2	16	7	320	3000
ID_018	100	24	1	8	4	80	1200
ID_020	100	34	2	16	7	160	1500
ID_053	100	54	12	96	19	960	4300
ID_059	100	9	2	16	4	160	1200
ID_023	100	18	1	8	4	80	1200
ID_021	100	18	1	8	4	80	1200
ID_102	250	58	6	42	13	168	1160
ID_101	250	44	1	7	7	28	600
ID_083	500	16	3	21	5	42	260
ID_017	1000	17	2	18	5	18	130
ID_117	1000	45	3	21	9	21	250

Table A.3. Low assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 6 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	1	6	2	300	5000
ID_002	25	53	1	6	8	240	6400
ID_056	25	64	11	66	17	2640	16400
ID_082	25	60	3	18	10	720	10400
ID_032	25	1	2	12	2	480	4000
ID_054	30	16	2	12	4	400	4000
ID_088	35	16	2	12	4	342.8571	3428.571429
ID_089	35	48	3	18	3	514.2857	7142.857143
ID_084	50	36	8	48	11	960	5400
ID_085	50	36	8	48	11	960	5400
ID_087	50	28	1	6	5	120	2600
ID_043	100	6	6	36	6	360	1400
ID_013	100	57	8	48	14	480	3000
ID_016	100	9	2	12	3	120	1100
ID_022	100	47	10	60	14	600	3000
ID_080	100	40	1	6	6	60	1400
ID_113	500	56	2	10	9	20	500
ID_136	500	64	1	5	9	10	500
ID_014	1000	3	1	5	1	5	90
ID_120	1000	25	9	45	9	45	250
ID_118	1000	44	8	40	11	40	270
ID_012	5000	33	1	4	5	0.8	26

A.1.2 One MAC per receiver - history buffer size = 10 samples

Table A.4. High assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 5 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	8	40	11	4000	27000
ID_008	10	49	1	5	7	500	15000
ID_047	10	49	9	45	12	4500	28000
ID_040	12	62	1	5	9	416.6667	20833.33333
ID_001	12	55	2	10	9	833.3333	20833.33333
ID_007	12	64	12	60	16	5000	26666.66667
ID_039	20	36	2	10	6	500	7000
ID_042	20	24	1	5	4	250	6000
ID_025	25	52	1	5	8	200	6400
ID_029	25	64	1	5	9	200	10000
ID_030	25	64	4	20	11	800	10800
ID_038	25	56	1	5	8	200	6400
ID_036	25	64	3	15	10	600	10400
ID_074	25	16	1	5	3	200	4400
ID_046	30	52	2	10	8	333.3333	5333.333333
ID_057	30	60	2	10	9	333.3333	8333.333333
ID_076	35	52	1	5	8	142.8571	4571.428571
ID_077	35	34	1	5	5	142.8571	3714.285714
ID_078	35	34	1	5	5	142.8571	3714.285714
ID_058	50	33	4	20	7	400	3000
ID_081	50	45	4	20	9	400	5000
ID_061	50	46	3	15	8	300	3200
ID_098	100	37	1	5	6	50	1400
ID_060	100	12	1	5	3	50	1100

Table A.5. Medium assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 4 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	1	4	5	666.6667	21666.66667
ID_004	10	64	10	40	13	4000	29000
ID_005	10	64	11	44	14	4400	30000
ID_010	12	61	4	16	10	1333.333	21666.66667
ID_003	12	9	1	4	2	333.3333	8333.333333
ID_026	12	31	1	4	5	333.3333	10833.33333
ID_027	12	62	2	8	9	666.6667	20833.33333
ID_048	12	59	1	4	8	333.3333	13333.33333
ID_052	12	61	1	4	9	333.3333	20833.33333
ID_041	20	26	3	12	5	600	6500
ID_045	20	27	1	4	4	200	6000
ID_024	20	11	5	20	4	1000	6000
ID_049	20	62	12	48	14	2400	15000
ID_028	25	16	1	4	3	160	4400
ID_033	25	45	1	4	7	160	6000
ID_106	25	17	1	4	3	160	4400
ID_031	25	54	1	4	8	160	6400
ID_034	25	62	1	4	9	160	10000
ID_035	25	57	8	32	12	1280	11200
ID_037	25	48	2	8	7	320	6000
ID_075	50	40	2	8	6	160	2800
ID_018	100	24	1	4	4	40	1200
ID_020	100	34	2	8	6	80	1400
ID_053	100	54	12	48	13	480	2900
ID_059	100	9	2	8	3	80	1100
ID_023	100	18	1	4	3	40	1100
ID_021	100	18	1	4	3	40	1100
ID_102	250	58	6	24	11	96	1080
ID_101	250	44	1	4	6	16	560
ID_083	500	16	3	12	4	24	240
ID_017	1000	17	2	8	4	8	120
ID_117	1000	45	3	12	8	12	160

Table A.6. Low assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 3 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	1	3	2	1	4500
ID_002	25	53	1	3	8	7	6000
ID_056	25	64	11	33	17	13	11600
ID_082	25	60	3	9	10	9	10000
ID_032	25	1	2	6	2	1	3600
ID_054	30	16	2	6	4	3	3666.666667
ID_088	35	16	2	6	4	3	3142.857143
ID_089	35	48	3	9	3	8	4571.428571
ID_084	50	36	8	24	11	8	3200
ID_085	50	36	8	24	11	8	3200
ID_087	50	28	1	3	5	4	2400
ID_043	100	6	6	18	6	3	1100
ID_013	100	57	8	24	14	11	2700
ID_016	100	9	2	6	3	2	1000
ID_022	100	47	10	30	14	10	2600
ID_080	100	40	1	3	6	6	1400
ID_113	500	56	2	6	9	8	320
ID_136	500	64	1	3	9	9	500
ID_014	1000	3	1	3	1	1	90
ID_120	1000	25	9	27	9	7	150
ID_118	1000	44	8	24	11	9	250
ID_012	5000	33	1	3	5	5	26

A.1.3 One MAC per receiver - history buffer size = 20 samples

Table A.7. High assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 3 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	8	24	9	2400	25000
ID_008	10	49	1	3	7	300	15000
ID_047	10	49	9	27	10	2700	26000
ID_040	12	62	1	3	9	250	20833.33333
ID_001	12	55	2	6	8	500	13333.33333
ID_007	12	64	12	36	13	3000	24166.66667
ID_039	20	36	2	6	6	300	7000
ID_042	20	24	1	3	4	150	6000
ID_025	25	52	1	3	7	120	6000
ID_029	25	64	1	3	9	120	10000
ID_030	25	64	4	12	10	480	10400
ID_038	25	56	1	3	8	120	6400
ID_036	25	64	3	9	10	360	10400
ID_074	25	16	1	3	3	120	4400
ID_046	30	52	2	6	8	200	5333.333333
ID_057	30	60	2	6	9	200	8333.333333
ID_076	35	52	1	3	7	85.71429	4285.714286
ID_077	35	34	1	3	5	85.71429	3714.285714
ID_078	35	34	1	3	5	85.71429	3714.285714
ID_058	50	33	4	12	6	240	2800
ID_081	50	45	4	12	8	240	3200
ID_061	50	46	3	9	7	180	3000
ID_098	100	37	1	3	5	30	1300
ID_060	100	12	1	3	2	30	1000

Table A.8. Medium assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 2 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	1	2	5	333.3333	21666.66667
ID_004	10	64	10	20	11	2000	27000
ID_005	10	64	11	22	11	2200	27000
ID_010	12	61	4	8	9	666.6667	20833.33333
ID_003	12	9	1	2	2	166.6667	8333.333333
ID_026	12	31	1	2	5	166.6667	10833.33333
ID_027	12	62	2	4	9	333.3333	20833.33333
ID_048	12	59	1	2	8	166.6667	13333.33333
ID_052	12	61	1	2	8	166.6667	13333.33333
ID_041	20	26	3	6	4	300	6000
ID_045	20	27	1	2	4	100	6000
ID_024	20	11	5	10	3	500	5500
ID_049	20	62	12	24	11	1200	13500
ID_028	25	16	1	2	3	80	4400
ID_033	25	45	1	2	6	80	5600
ID_106	25	17	1	2	3	80	4400
ID_031	25	54	1	2	7	80	6000
ID_034	25	62	1	2	8	80	6400
ID_035	25	57	8	16	10	640	10400
ID_037	25	48	2	4	7	160	6000
ID_075	50	40	2	4	6	80	2800
ID_018	100	24	1	2	4	20	1200
ID_020	100	34	2	4	5	40	1300
ID_053	100	54	12	24	10	240	2600
ID_059	100	9	2	4	2	40	1000
ID_023	100	18	1	2	3	20	1100
ID_021	100	18	1	2	3	20	1100
ID_102	250	58	6	12	9	48	1000
ID_101	250	44	1	2	6	8	560
ID_083	500	16	3	6	3	12	220
ID_017	1000	17	2	4	3	4	110
ID_117	1000	45	3	6	7	6	150

Table A.9. Low assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 2 bits.

Message ID	Period (ms)	Payload bits	Number of receivers	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	1	2	1	100	4500
ID_002	25	53	1	2	7	80	6000
ID_056	25	64	11	22	11	880	10800
ID_082	25	60	3	6	9	240	10000
ID_032	25	1	2	4	1	160	3600
ID_054	30	16	2	4	3	133.3333	3666.666667
ID_088	35	16	2	4	3	114.2857	3142.857143
ID_089	35	48	3	6	7	171.4286	4285.714286
ID_084	50	36	8	16	7	320	3000
ID_085	50	36	8	16	7	320	3000
ID_087	50	28	1	2	4	40	2400
ID_043	100	6	6	12	3	120	1100
ID_013	100	57	8	16	10	160	2600
ID_016	100	9	2	4	2	40	1000
ID_022	100	47	10	20	9	200	2500
ID_080	100	40	1	2	6	20	1400
ID_113	500	56	2	4	8	8	320
ID_136	500	64	1	2	9	4	500
ID_014	1000	3	1	2	1	2	90
ID_120	1000	25	9	18	6	18	140
ID_118	1000	44	8	16	8	16	160
ID_012	5000	33	1	2	5	0.4	26

A.2 Validity voting

A.2.1 Validity voting - history buffer size = 5 samples

Tables A.10-15 show the validity vector size (number of message types voted upon), a list of message types each node votes upon, and which messages vote upon them. Similar tables are provided for each history buffer size to show which votes were applied.

Table A.10. High assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 5 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_009	10	ECU_05	12	ID_004, ID_005, ID_007, ID_030, ID_035, ID_036, ID_047, ID_049, ID_058, ID_061, ID_084, ID_085
ID_008	10	ECU_07	0	
ID_047	10	ECU_07	4	ID_007, ID_009, ID_030, ID_036
ID_040	12	ECU_07	0	
ID_001	12	ECU_09	0	
ID_007	12	ECU_09	4	ID_010, ID_039, ID_049, ID_081
ID_039	20	ECU_07	1	ID_037
ID_042	20	ECU_07	0	
ID_025	25	ECU_02	0	
ID_029	25	ECU_02	1	ID_057
ID_030	25	ECU_02	2	ID_032, ID_081
ID_038	25	ECU_07	1	ID_030
ID_036	25	ECU_09	2	ID_030, ID_046
ID_074	25	ECU_09	1	ID_057
ID_046	30	ECU_05	0	
ID_057	30	ECU_05	1	ID_081
ID_076	35	ECU_11	0	
ID_077	35	ECU_11	0	
ID_078	35	ECU_11	0	
ID_058	50	ECU_07	2	ID_061, ID_102
ID_081	50	ECU_07	0	
ID_061	50	ECU_13	2	ID_058, ID_084
ID_098	100	ECU_09	1	ID_102
ID_060	100	ECU_13	0	

Table A.11. High assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 5 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_009	10	ID_047	ECU_07	ECU_04, ECU_06, ECU_09, ECU_13
ID_008	10			
ID_047	10	ID_009	ECU_07	ECU_04, ECU_06, ECU_09, ECU_13
ID_040	12			
ID_001	12			
ID_007	12	ID_047	ECU_07	ECU_01, ECU_04, ECU_05, ECU_06, ECU_08, ECU_13, ECU_14
		ID_009	ECU_05	ECU_02, ECU03, ECU_04, ECU_06, ECU_07, ECU_11, ECU_13
ID_039	20	ID_007	ECU_09	ECU_11
ID_042	20			
ID_025	25			
ID_029	25			
ID_030	25	ID_038	ECU_07	ECU_09
		ID_036	ECU_09	ECU_05, ECU_07, ECU_11
		ID_047	ECU_07	ECU_05
		ID_009	ECU_05	ECU_07, ECU_09, ECU_11
ID_038	25			
ID_036	25	ID_009	ECU_05	ECU_07, ECU_11
		ID_047	ECU_07	ECU_05
ID_074	25			
ID_046	30	ID_036	ECU_09	ECU_11
ID_057	30	ID_074	ECU_09	ECU_02
		ID_029	ECU_02	ECU_09
ID_076	35			
ID_077	35			
ID_078	35			
ID_058	50	ID_061	ECU_13	ECU_05, ECU_11
		ID_009	ECU_05	ECU_06, ECU_11, ECU_13
ID_081	50	ID_007	ECU_09	ECU_02, ECU_04, ECU_05
		ID_030	ECU_02	ECU_05, ECU_09
		ID_057	ECU_05	ECU_02, ECU_09
ID_061	50	ID_058	ECU_07	ECU_05, ECU_11
		ID_009	ECU_05	ECU_07, ECU_11
ID_098	100			
ID_060	100			

Table A.12. Medium assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 5 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_006	6	ECU_02	1	ID_027
ID_004	10	ECU_07	1	
ID_005	10	ECU_07	0	
ID_010	12	ECU_02	1	ID_041
ID_003	12	ECU_09	0	
ID_026	12	ECU_09	0	
ID_027	12	ECU_09	0	
ID_048	12	ECU_09	0	
ID_052	12	ECU_09	0	
ID_041	20	ECU_04	0	
ID_045	20	ECU_04	1	ID_075
ID_024	20	ECU_07	1	ID_088
ID_049	20	ECU_07	6	ID_032, ID_035, ID_041, ID_053, ID_056, ID_082
ID_028	25	ECU_02	0	
ID_033	25	ECU_02	0	
ID_106	25	ECU_05	0	
ID_031	25	ECU_09	0	
ID_034	25	ECU_09	0	
ID_035	25	ECU_09	1	ID_053
ID_037	25	ECU_09	0	
ID_075	50	ECU_09	0	
ID_018	100	ECU_05	0	
ID_020	100	ECU_05	1	ID_059
ID_053	100	ECU_05	1	ID_022
ID_059	100	ECU_06	1	ID_022
ID_023	100	ECU_07	0	
ID_021	100	ECU_08	1	ID_059
ID_102	250	ECU_05	2	ID_083, ID_120
ID_101	250	ECU_08	0	
ID_083	500	ECU_06	1	ID_120
ID_017	1000	ECU_05	1	ID_120
ID_117	1000	ECU_05	1	ID_118

Table A.13. Medium assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 5 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_006	6			
ID_004	10	ID_009	ECU_05	ECU_02, ECU_04, ECU_06, ECU_09, ECU_13
ID_005	10	ID_009	ECU_05	ECU_02, ECU_04, ECU_06, ECU_09, ECU_13
ID_010	12	ID_004	ECU_07	ECU_04, ECU_06, ECU_09
		ID_007	ECU_09	ECU_04, ECU_06, ECU_07
ID_003	12			
ID_026	12			
ID_027	12	ID_006	ECU_02	ECU_04
ID_048	12			
ID_052	12			
ID_041	20	ID_049	ECU_07	ECU_02, ECU_11
		ID_010	ECU_02	ECU_07
ID_045	20			
ID_024	20			
ID_049	20	ID_007	ECU_09	ECU_01, ECU_02, ECU_03, ECU_04, ECU_05, ECU_06, ECU_08, ECU_11, ECU_13, ECU_14
		ID_009	ECU_05	ECU_02, ECU_03, ECU_04, ECU_06, ECU_09, ECU_11, ECU_13
ID_028	25			
ID_033	25			
ID_106	25			
ID_031	25			
ID_034	25			
ID_035	25	ID_049	ECU_07	ECU_04, ECU_05, ECU_06, ECU_08, ECU_11, ECU_13, ECU_14
		ID_009	ECU_05	ECU_04, ECU_06, ECU_07, ECU_11, ECU_13
ID_037	25	ID_039	ECU_07	ECU_11
ID_075	50	ID_045	ECU_04	ECU_07
ID_018	100			
ID_020	100			
ID_053	100	ID_049	ECU_07	ECU_01, ECU_02, ECU_03, ECU_04, ECU_06, ECU_09, ECU_11, ECU_12, ECU_13, ECU_14
		ID_035	ECU_09	ECU_04, ECU_05, ECU_06, ECU_07, ECU_11, ECU_13, ECU_14
ID_059	100	ID_021	ECU_08	ECU_05
		ID_020	ECU_05	ECU_08
ID_023	100			
ID_021	100			
ID_102	250	ID_058 {7 to 4,6,13} ID_098 {13 to 7}		
ID_101	250			
ID_083	500	ID_102 {5 to 7}		
ID_017	1000			
ID_117	1000			

Table A.14. Low assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 5 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_044	20	ECU_04	0	
ID_002	25	ECU_02	1	ID_054
ID_056	25	ECU_02	2	ID_013, ID_089
ID_082	25	ECU_06	0	
ID_032	25	ECU_09	1	ID_054
ID_054	30	ECU_05	1	ID_089
ID_088	35	ECU_11	0	
ID_089	35	ECU_11	0	
ID_084	50	ECU_07	0	
ID_085	50	ECU_07	0	
ID_087	50	ECU_07	0	
ID_043	100	ECU_04	1	ID_022
ID_013	100	ECU_05	1	ID_043
ID_016	100	ECU_05	0	
ID_022	100	ECU_07	2	ID_013, ID_043
ID_080	100	ECU_07	0	
ID_113	500	ECU_09	0	
ID_136	500	ECU_09	0	
ID_014	1000	ECU_05	1	ID_120
ID_120	1000	ECU_07	1	ID_118
ID_118	1000	ECU_09	0	
ID_012	5000	ECU_05	0	

Table A.15. Low assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 5 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_044	20			
ID_002	25			
ID_056	25	ID_049	ECU_07	ECU_01, ECU_04, ECU_05, ECU_06, ECU_08, ECU_09, ECU_11, ECU_12, ECU_13, ECU_14
ID_082	25	ID_049	ECU_07	ECU_01
ID_032	25	ID_049	ECU_07	ECU_02
		ID_030	ECU_02	ECU_07
ID_054	30	ID_032	ECU_09	ECU_02
		ID_002	ECU_02	ECU_09
ID_088	35	ID_024	ECU_07	ECU_05
ID_089	35	ID_054	ECU_02	ECU_05, ECU_07
		ID_056	ECU_05	ECU_02
ID_084	50	ID_061	ECU_13	ECU_05, ECU_11
		ID_009	ECU_05	ECU_03, ECU_04, ECU_06, ECU_11, ECU_13
ID_085	50	ID_009	ECU_05	ECU_03, ECU_04, ECU_06, ECU_11, ECU_13
ID_087	50			
ID_043	100	ID_013	ECU_05	ECU_02, ECU_07, ECU_09, ECU_11
		ID_022	ECU_06	ECU_07, ECU_08, ECU_11
ID_013	100	ID_022		ECU_01, ECU_06, ECU_10, ECU_11, ECU_13
		ID_056	ECU_02	ECU_01, ECU_06, ECU_07, ECU_09, ECU_11, ECU_13
ID_016	100			
ID_022	100	ID_053	ECU_05	ECU_01, ECU_03, ECU_04, ECU_06, ECU_10, ECU_11, ECU_12, ECU_13, ECU_14
		ID_059	ECU_06	ECU_05, ECU_08
		ID_043	ECU_04	ECU_05, ECU_08, ECU_11
ID_080	100			
ID_113	500			
ID_136	500			
ID_014	1000			
ID_120	1000	ID_102	ECU_05	ECU_04, ECU_06, ECU_13, ECU_14
		ID_014	ECU_05	ECU_10
		ID_017	ECU_05	ECU_11
		ID_083	ECU_06	ECU_05, ECU_08
ID_118	1000	ID_120	ECU_07	ECU_04, ECU_05, ECU_10, ECU_11, ECU_12, ECU_13
		ID_117	ECU_05	ECU_02, ECU_04
ID_012	5000			

Tables A.16-18 show the bandwidth consumed by each message type for a history buffer size of five samples, using validity voting. Votes were applied as per Tables A.10-15.

Table A.16. High assurance message bandwidth consumption for validity voting. History buffer size is 5 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_009	10	44	12		10	10	6		6	10		6		10		6		76	15	7600	31000
ID_008	10	49	0							10								10	8	1000	16000
ID_047	10	49	4	10			6	10	6		10	6			10	6	10	78	16	7800	32000
ID_040	12	62	0									10						10	9	833.3333	20833.33333
ID_001	12	55	0		10					10								20	10	1666.667	21666.66667
ID_007	12	64	4	6	6	6	4	6	4	6	6		10	6		4	6	74	18	6166.667	35000
ID_039	20	36	1									10		6				17	7	850	7500
ID_042	20	24	0				10											10	5	500	6500
ID_025	25	52	0									10						10	8	400	6400
ID_029	25	64	1									10						11	10	440	10400
ID_030	25	64	2					4		4		4		4				18	11	720	10800
ID_038	25	56	1									10						11	9	440	10000
ID_036	25	64	2					6		6				6				20	11	800	10800
ID_074	25	16	1		10													11	4	440	4800
ID_046	30	52	0									10		6				16	9	533.3333	8333.333333
ID_057	30	60	1		6							6						13	10	433.3333	8666.666667
ID_076	35	52	0									10						10	8	285.7143	4571.428571
ID_077	35	34	0									10						10	6	285.7143	4000
ID_078	35	34	0									10						10	6	285.7143	4000
ID_058	50	33	2					6	6					4		6		24	8	480	3200
ID_081	50	45	0		4		6	4				4						18	8	360	3200
ID_061	50	46	2					6		6				4				18	8	360	3200
ID_098	100	37	1							10								11	6	110	1400
ID_060	100	12	0							10								10	3	100	1100

Table A.17. Medium assurance message bandwidth consumption for validity voting. History buffer size is 5 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_006	6	32	1				8											9	6	1500	23333.33333
ID_004	10	64	1		5		5	8	5		8	5	8		8	5	8	66	17	6600	41000
ID_005	10	64	0	8	5		5	8	5		8	5	8		8	5	8	73	18	7300	42000
ID_010	12	61	1				4		4	5		5						19	10	1583.333	21666.66667
ID_003	12	9	0		8													8	3	666.6667	9166.666667
ID_026	12	31	0		8													8	5	666.6667	10833.33333
ID_027	12	62	0		8		5											13	10	1083.333	21666.66667
ID_048	12	59	0										8					8	9	666.6667	20833.33333
ID_052	12	61	0										8					8	9	666.6667	20833.33333
ID_041	20	26	0		5					5				5				15	6	750	7000
ID_045	20	27	1							8								9	5	450	6500
ID_024	20	11	1		8			8	8							8	8	41	7	2050	7500
ID_049	20	62	6	5	5	5	4	5	4		5	5		4	8	4	5	65	16	3250	16000
ID_028	25	16	0									8						8	3	320	4400
ID_033	25	45	0									8						8	7	320	6000
ID_106	25	17	0									8						8	4	320	4800
ID_031	25	54	0		8													8	8	320	6400
ID_034	25	62	0		8													8	9	320	10000
ID_035	25	57	1				4	5	4	5	5			4		4	5	37	12	1480	11200
ID_037	25	48	0							8				5				13	8	520	6400
ID_075	50	40	0				8			5								13	7	260	3000
ID_018	100	24	0	8														8	4	80	1200
ID_020	100	34	1							8	8							17	7	170	1500
ID_053	100	54	1	5	5	5	4		4	4		5	8	4	5	4	4	58	14	580	3000
ID_059	100	9	1					5			5							11	3	110	1100
ID_023	100	18	0					8										8	4	80	1200
ID_021	100	18	1					8										9	4	90	1200
ID_102	250	58	2				5		5	5		8				5	8	38	12	152	1120
ID_101	250	44	0									8						8	7	32	600
ID_083	500	16	1					8		5	8							22	5	44	260
ID_017	1000	17	1	8										8				17	5	17	130
ID_117	1000	45	1		8		8					8						25	9	25	250

Table A.18. Low assurance message bandwidth consumption for validity voting. History buffer size is 5 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_044	20	3	0							6								6	2	300	5000
ID_002	25	53	1									6						7	8	280	6400
ID_056	25	64	2	4			4	4	4	6	4	4		4	4	4	4	48	14	1920	12000
ID_082	25	60	0	4						6								10	9	400	10000
ID_032	25	1	1		4					4								9	2	360	4000
ID_054	30	16	1		4							4						9	4	300	4000
ID_088	35	16	0					4		6								10	4	285.7143	3428.571429
ID_089	35	48	0		4			4		4								12	8	342.8571	4571.428571
ID_084	50	36	0			4	4	4	4					3	6	4	6	35	9	700	5000
ID_085	50	36	0			4	4	6	4					4	6	4	6	38	10	760	5200
ID_087	50	28	0					6										6	5	120	2600
ID_043	100	6	1		4			4		4	4	4		3				24	4	240	1200
ID_013	100	57	1	3	6				3	4		4	4	3		3		31	11	310	2700
ID_016	100	9	0							6				6				12	3	120	1100
ID_022	100	47	2	4		4	4	3	4		3		4	3	4	4		39	11	390	2700
ID_080	100	40	0									6						6	6	60	1400
ID_113	500	56	0					6			6							12	9	24	500
ID_136	500	64	0		6													6	9	12	500
ID_014	1000	3	1										6					7	2	7	100
ID_120	1000	25	1				4	4	4		4		4	4	6	4	4	39	8	39	160
ID_118	1000	44	0		4		3	4		6			4	4	4	4		33	10	33	260
ID_012	5000	33	0				6											6	5	1.2	26

A.2.1 Validity voting - history buffer size = 10 samples

Table A.19. High assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 10 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_009	10	ECU_05	7	ID_004, ID_005, ID_007, ID_035, ID_036, ID_047,
ID_008	10	ECU_07	0	
ID_047	10	ECU_07	2	ID_007, ID_009
ID_040	12	ECU_07	0	
ID_001	12	ECU_09	0	
ID_007	12	ECU_09	4	ID_010, ID_039, ID_049, ID_081
ID_039	20	ECU_07	0	
ID_042	20	ECU_07	0	
ID_025	25	ECU_02	0	
ID_029	25	ECU_02	1	ID_057
ID_030	25	ECU_02	0	
ID_038	25	ECU_07	1	ID_030
ID_036	25	ECU_09	2	ID_030, ID_046
ID_074	25	ECU_09	1	ID_057
ID_046	30	ECU_05	0	
ID_057	30	ECU_05	1	ID_081
ID_076	35	ECU_11	0	
ID_077	35	ECU_11	0	
ID_078	35	ECU_11	0	
ID_058	50	ECU_07	1	ID_061
ID_081	50	ECU_07	0	
ID_061	50	ECU_13	2	ID_058, ID_084
ID_098	100	ECU_09	0	
ID_060	100	ECU_13	0	

Table A.20. High assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 10 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_009	10	ID_047	ECU_07	ECU_04, ECU_06, ECU_09, ECU_13
ID_008	10			
ID_047	10	ID_009	ECU_07	ECU_04, ECU_06, ECU_09, ECU_13
ID_040	12			
ID_001	12			
ID_007	12	ID_047	ECU_07	ECU_01, ECU_04, ECU_05, ECU_06, ECU_08, ECU_13, ECU_14
		ID_009	ECU_05	ECU_02, ECU03, ECU_04, ECU_06, ECU_07, ECU_11, ECU_13
ID_039	20	ID_007	ECU_09	ECU_11
ID_042	20			
ID_025	25			
ID_029	25			
ID_030	25	ID_038	ECU_07	ECU_09
		ID_036	ECU_09	ECU_05, ECU_07, ECU_11
ID_038	25			
ID_036	25	ID_009	ECU_05	ECU_07, ECU_11
ID_074	25			
ID_046	30	ID_036	ECU_09	ECU_11
ID_057	30	ID_074	ECU_09	ECU_02
		ID_029	ECU_02	ECU_09
ID_076	35			
ID_077	35			
ID_078	35			
ID_058	50	ID_061	ECU_13	ECU_05, ECU_11
ID_081	50	ID_007	ECU_09	ECU_02, ECU_04, ECU_05
		ID_057	ECU_05	ECU_02, ECU_09
ID_061	50	ID_058	ECU_07	ECU_05, ECU_11
		ID_009	ECU_05	ECU_07, ECU_11
ID_098	100			
ID_060	100			

Table A.21. Medium assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 10 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_006	6	ECU_02	0	
ID_004	10	ECU_07	1	ID_010
ID_005	10	ECU_07	0	
ID_010	12	ECU_02	0	
ID_003	12	ECU_09	0	
ID_026	12	ECU_09	0	
ID_027	12	ECU_09	0	
ID_048	12	ECU_09	0	
ID_052	12	ECU_09	0	
ID_041	20	ECU_04	0	
ID_045	20	ECU_04	0	
ID_024	20	ECU_07	0	
ID_049	20	ECU_07	4	ID_035, ID_041, ID_053
ID_028	25	ECU_02	0	
ID_033	25	ECU_02	0	
ID_106	25	ECU_05	0	
ID_031	25	ECU_09	0	
ID_034	25	ECU_09	0	
ID_035	25	ECU_09	1	ID_053
ID_037	25	ECU_09	0	
ID_075	50	ECU_09	0	
ID_018	100	ECU_05	0	
ID_020	100	ECU_05	0	
ID_053	100	ECU_05	1	ID_022
ID_059	100	ECU_06	1	ID_022
ID_023	100	ECU_07	0	
ID_021	100	ECU_08	0	
ID_102	250	ECU_05	0	
ID_101	250	ECU_08	0	
ID_083	500	ECU_06	0	
ID_017	1000	ECU_05	0	
ID_117	1000	ECU_05	0	

Table A.22. Medium assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 10 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_006	6			
ID_004	10	ID_009	ECU_05	ECU_02, ECU_04, ECU_06, ECU_09, ECU_13
ID_005	10	ID_009	ECU_05	ECU_02, ECU_04, ECU_06, ECU_09, ECU_13
ID_010	12	ID_004	ECU_07	ECU_04, ECU_06, ECU_09
		ID_007	ECU_09	ECU_04, ECU_06, ECU_07
ID_003	12			
ID_026	12			
ID_027	12			
ID_048	12			
ID_052	12			
ID_041	20	ID_049	ECU_07	ECU_02, ECU_11
ID_045	20			
ID_024	20			
ID_049	20	ID_007	ECU_09	ECU_01, ECU_02, ECU_03, ECU_04, ECU_05, ECU_06, ECU_08, ECU_11, ECU_13, ECU_14
		ID_009	ECU_05	ECU_02, ECU_03, ECU_04, ECU_06, ECU_09, ECU_11, ECU_13
ID_028	25			
ID_033	25			
ID_106	25			
ID_031	25			
ID_034	25			
ID_035	25	ID_049	ECU_07	ECU_04, ECU_05, ECU_06, ECU_08, ECU_11, ECU_13, ECU_14
		ID_009	ECU_05	ECU_04, ECU_06, ECU_07, ECU_11, ECU_13
ID_037	25			
ID_075	50			
ID_018	100			
ID_020	100			
ID_053	100	ID_049	ECU_07	ECU_01, ECU_02, ECU_03, ECU_04, ECU_06, ECU_09, ECU_11, ECU_12, ECU_13, ECU_14
		ID_035	ECU_09	ECU_04, ECU_05, ECU_06, ECU_07, ECU_11, ECU_13, ECU_14
ID_059	100			
ID_023	100			
ID_021	100			
ID_102	250			
ID_101	250			
ID_083	500			
ID_017	1000			
ID_117	1000			

Table A.23. Low assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 10 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_044	20	ECU_04	0	
ID_002	25	ECU_02	0	
ID_056	25	ECU_02	0	
ID_082	25	ECU_06	0	
ID_032	25	ECU_09	0	
ID_054	30	ECU_05	1	ID_089
ID_088	35	ECU_11	0	
ID_089	35	ECU_11	0	
ID_084	50	ECU_07	0	
ID_085	50	ECU_07	0	
ID_087	50	ECU_07	0	
ID_043	100	ECU_04	0	
ID_013	100	ECU_05	1	ID_043
ID_016	100	ECU_05	0	
ID_022	100	ECU_07	2	ID_013, ID_043
ID_080	100	ECU_07	0	
ID_113	500	ECU_09	0	
ID_136	500	ECU_09	0	
ID_014	1000	ECU_05	0	
ID_120	1000	ECU_07	1	ID_118
ID_118	1000	ECU_09	0	
ID_012	5000	ECU_05	0	

Table A.24. Low assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 10 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_044	20			
ID_002	25			
ID_056	25	ID_049	ECU_07	ECU_01, ECU_04, ECU_05, ECU_06, ECU_08, ECU_09, ECU_11, ECU_12, ECU_13, ECU_14
ID_082	25			
ID_032	25			
ID_054	30			
ID_088	35			
ID_089	35	ID_054	ECU_02	ECU_05, ECU_07
ID_084	50	ID_061	ECU_13	ECU_05, ECU_11
ID_085	50			
ID_087	50			
ID_043	100	ID_013	ECU_05	ECU_02, ECU_07, ECU_09, ECU_11
		ID_022	ECU_06	ECU_07, ECU_08, ECU_11
ID_013	100	ID_022		ECU_01, ECU_06, ECU_10, ECU_11, ECU_13
ID_016	100			
ID_022	100	ID_053	ECU_05	ECU_01, ECU_03, ECU_04, ECU_06, ECU_10, ECU_11, ECU_12, ECU_13, ECU_14
		ID_059	ECU_06	ECU_05, ECU_08
ID_080	100			
ID_113	500			
ID_136	500			
ID_014	1000			
ID_120	1000			
ID_118	1000	ID_120	ECU_07	ECU_04, ECU_05, ECU_10, ECU_11, ECU_12, ECU_13
ID_012	5000			

Tables A.25-27 show the bandwidth consumed by each message type for a history buffer size of ten samples, using validity voting. Votes were applied as per Tables A.19-24.

Table A.25. High assurance message bandwidth consumption for validity voting. History buffer size is 10 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_009	10	44	7		5	5	3		3	5		3		5		3		39	11	3900	27000
ID_008	10	49	0							5								5	7	500	15000
ID_047	10	49	2	5			3	5	3		5	3			5	3	5	39	11	3900	27000
ID_040	12	62	0									5						5	9	416.6667	20833.33333
ID_001	12	55	0		5					5								10	9	833.3333	20833.33333
ID_007	12	64	4	3	3	3	3	3	3	3	3		5	3		3	3	42	14	3500	25000
ID_039	20	36	0									5		3				8	6	400	7000
ID_042	20	24	0				5											5	4	250	6000
ID_025	25	52	0									5						5	8	200	6400
ID_029	25	64	1									5						6	9	240	10000
ID_030	25	64	0					3		3		3		3				12	10	480	10400
ID_038	25	56	1									5						6	8	240	6400
ID_036	25	64	2					5		3				3				13	10	520	10400
ID_074	25	16	1		5													6	3	240	4400
ID_046	30	52	0									5		3				8	8	266.6667	5333.333333
ID_057	30	60	1		3							3						7	9	233.3333	8333.333333
ID_076	35	52	0									5						5	8	142.8571	4571.428571
ID_077	35	34	0									5						5	5	142.8571	3714.285714
ID_078	35	34	0									5						5	5	142.8571	3714.285714
ID_058	50	33	1					3	5					3		5		17	7	340	3000
ID_081	50	45	0		3		3	3				3						12	8	240	3200
ID_061	50	46	2					3		5				3				13	8	260	3200
ID_098	100	37	0							5								5	6	50	1400
ID_060	100	12	0							5								5	3	50	1100

Table A.26. Medium assurance message bandwidth consumption for validity voting. History buffer size is 10 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_006	6	32	0				4											4	5	666.6667	21666.66667
ID_004	10	64	1		3		3	4	3		4	3	4		4	3	4	36	13	3600	29000
ID_005	10	64	0	4	3		3	4	3		4	3	4		4	3	4	39	13	3900	29000
ID_010	12	61	0				2		2	3		3						10	9	833.3333	20833.33333
ID_003	12	9	0		4													4	2	333.3333	8333.333333
ID_026	12	31	0		4													4	5	333.3333	10833.33333
ID_027	12	62	0		4		4											8	9	666.6667	20833.33333
ID_048	12	59	0										4					4	8	333.3333	13333.33333
ID_052	12	61	0										4					4	9	333.3333	20833.33333
ID_041	20	26	0		3					4				3				10	5	500	6500
ID_045	20	27	0							4								4	4	200	6000
ID_024	20	11	0		4			4	4							4	4	20	4	1000	6000
ID_049	20	62	4	3	2	2	2	3	2		3	3		2	4	2	3	34	12	1700	14000
ID_028	25	16	0									4						4	3	160	4400
ID_033	25	45	0									4						4	7	160	6000
ID_106	25	17	0									4						4	3	160	4400
ID_031	25	54	0		4													4	8	160	6400
ID_034	25	62	0		4													4	9	160	10000
ID_035	25	57	1				2	3	2	3	3			2		2	3	21	10	840	10400
ID_037	25	48	0							4				4				8	7	320	6000
ID_075	50	40	0				4			4								8	6	160	2800
ID_018	100	24	0	4														4	4	40	1200
ID_020	100	34	0							4	4							8	6	80	1400
ID_053	100	54	1	3	3	3	2		2	3		3	4	2	3	2	2	33	11	330	2700
ID_059	100	9	1					4			4							9	3	90	1100
ID_023	100	18	0					4										4	3	40	1100
ID_021	100	18	0					4										4	3	40	1100
ID_102	250	58	0				4		4	4		4				4	4	24	11	96	1080
ID_101	250	44	0									4						4	6	16	560
ID_083	500	16	0					4		4	4							12	4	24	240
ID_017	1000	17	0	4										4				8	4	8	120
ID_117	1000	45	0		4		4					4						12	8	12	160

Table A.27. Low assurance message bandwidth consumption for validity voting. History buffer size is 10 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_044	20	3	0							3								3	1	150	4500
ID_002	25	53	0									3						3	7	120	6000
ID_056	25	64	0	2			2	2	2	3	2	2		2	2	2	2	23	11	920	10800
ID_082	25	60	0	3					3	3								9	9	360	10000
ID_032	25	1	0		3					3								6	1	240	3600
ID_054	30	16	1		3							3						7	3	233.3333	3666.666667
ID_088	35	16	0					3		3								6	3	171.4286	3142.857143
ID_089	35	48	0		3			2		2								7	7	200	4285.714286
ID_084	50	36	0			3	3	2	3					2	3	3	3	22	8	440	3200
ID_085	50	36	0			3	3	3	3					3	3	3	3	24	8	480	3200
ID_087	50	28	0					3										3	4	60	2400
ID_043	100	6	0		2			2		2	2	2		2				12	3	120	1100
ID_013	100	57	1	2	3				2	3		3	2	2		2		20	10	200	2600
ID_016	100	9	0							3				3				6	2	60	1000
ID_022	100	47	2	2		2	2	2	2		2		2	2	2	2		22	9	220	2500
ID_080	100	40	0									3						3	6	30	1400
ID_113	500	56	0					3			3							6	8	12	320
ID_136	500	64	0		3													3	9	6	500
ID_014	1000	3	0										3					3	1	3	90
ID_120	1000	25	1				3	3	3		3		3	3	3	3	3	28	7	28	150
ID_118	1000	44	0		3		2	2		3			2	2	2	2		18	8	18	160
ID_012	5000	33	0				3											3	5	0.6	26

A.2.3 Validity voting - history buffer size = 20 samples

Tables A.28-33 show the validity vector size (number of message types voted upon), a list of message types each node votes upon, and which messages vote upon them.

Table A.28. High assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 20 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_009	10	ECU_05	2	ID_007, ID_047,
ID_008	10	ECU_07		
ID_047	10	ECU_07	2	ID_007, ID_009
ID_040	12	ECU_07		
ID_001	12	ECU_09		
ID_007	12	ECU_09		
ID_039	20	ECU_07		
ID_042	20	ECU_07		
ID_025	25	ECU_02		
ID_029	25	ECU_02		
ID_030	25	ECU_02		
ID_038	25	ECU_07		
ID_036	25	ECU_09	1	ID_030
ID_074	25	ECU_09		
ID_046	30	ECU_05		
ID_057	30	ECU_05	1	ID_081
ID_076	35	ECU_11		
ID_077	35	ECU_11		
ID_078	35	ECU_11		
ID_058	50	ECU_07	1	ID_061
ID_081	50	ECU_07		
ID_061	50	ECU_13	1	ID_058
ID_098	100	ECU_09		
ID_060	100	ECU_13		

Table A.29. High assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 20 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_009	10	ID_047	ECU_07	ECU_04, ECU_06, ECU_09, ECU_13
ID_008	10			
ID_047	10	ID_009	ECU_07	ECU_04, ECU_06, ECU_09, ECU_13
ID_040	12			
ID_001	12			
ID_007	12	ID_047	ECU_07	ECU_01, ECU_04, ECU_05, ECU_06, ECU_08, ECU_13, ECU_14
		ID_009	ECU_05	ECU_02, ECU03, ECU_04, ECU_06, ECU_07, ECU_11, ECU_13
ID_039	20			
ID_042	20			
ID_025	25			
ID_029	25			
ID_030	25	ID_036	ECU_09	ECU_05, ECU_07, ECU_11
ID_038	25			
ID_036	25			
ID_074	25			
ID_046	30			
ID_057	30			
ID_076	35			
ID_077	35			
ID_078	35			
ID_058	50	ID_061	ECU_13	ECU_05, ECU_11
ID_081	50	ID_057	ECU_05	ECU_02, ECU_09
ID_061	50	ID_058	ECU_07	ECU_05, ECU_11
ID_098	100			
ID_060	100			

Table A.30. Medium assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 20 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_006	6	ECU_02	These messages do not vote on others. Voting did not reduce bandwidth.	
ID_004	10	ECU_07		
ID_005	10	ECU_07		
ID_010	12	ECU_02		
ID_003	12	ECU_09		
ID_026	12	ECU_09		
ID_027	12	ECU_09		
ID_048	12	ECU_09		
ID_052	12	ECU_09		
ID_041	20	ECU_04		
ID_045	20	ECU_04		
ID_024	20	ECU_07		
ID_049	20	ECU_07		
ID_028	25	ECU_02		
ID_033	25	ECU_02		
ID_106	25	ECU_05		
ID_031	25	ECU_09		
ID_034	25	ECU_09		
ID_035	25	ECU_09		
ID_037	25	ECU_09		
ID_075	50	ECU_09		
ID_018	100	ECU_05		
ID_020	100	ECU_05		
ID_053	100	ECU_05		
ID_059	100	ECU_06		
ID_023	100	ECU_07		
ID_021	100	ECU_08		
ID_102	250	ECU_05		
ID_101	250	ECU_08		
ID_083	500	ECU_06		
ID_017	1000	ECU_05		
ID_117	1000	ECU_05		

Table A.31. Medium assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 20 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_006	6	No messages voted on this message type. Voting did not reduce bandwidth.		
ID_004	10			
ID_005	10			
ID_010	12			
ID_003	12			
ID_026	12			
ID_027	12			
ID_048	12			
ID_052	12			
ID_041	20			
ID_045	20			
ID_024	20			
ID_049	20			
ID_028	25			
ID_033	25			
ID_106	25			
ID_031	25			
ID_034	25			
ID_035	25			
ID_037	25			
ID_075	50			
ID_018	100			
ID_020	100			
ID_053	100			
ID_059	100			
ID_023	100			
ID_021	100			
ID_102	250			
ID_101	250			
ID_083	500			
ID_017	1000			
ID_117	1000			

Table A.32. Low assurance messages. Validity vector size (number of validity votes carried), and message types voted upon. History buffer size of 20 samples.

Message ID	Period (ms)	Sender ID	Validity vector size (bits)	Other message IDs voted on by this message type
ID_044	20	ECU_04	These messages do not vote on others. Voting did not reduce bandwidth.	
ID_002	25	ECU_02		
ID_056	25	ECU_02		
ID_082	25	ECU_06		
ID_032	25	ECU_09		
ID_054	30	ECU_05		
ID_088	35	ECU_11		
ID_089	35	ECU_11		
ID_084	50	ECU_07		
ID_085	50	ECU_07		
ID_087	50	ECU_07		
ID_043	100	ECU_04		
ID_013	100	ECU_05		
ID_016	100	ECU_05		
ID_022	100	ECU_07		
ID_080	100	ECU_07		
ID_113	500	ECU_09		
ID_136	500	ECU_09		
ID_014	1000	ECU_05		
ID_120	1000	ECU_07		
ID_118	1000	ECU_09		
ID_012	5000	ECU_05		

Table A.33. Low assurance messages. Message types, nodes that vote upon them, nodes that receive those votes. History buffer size of 20 samples.

Message ID	Period (ms)	Other message types that vote on this message type	Sender of vote	Nodes that consume vote
ID_044	20	No messages voted on this message type. Voting did not reduce bandwidth.		
ID_002	25			
ID_056	25			
ID_082	25			
ID_032	25			
ID_054	30			
ID_088	35			
ID_089	35			
ID_084	50			
ID_085	50			
ID_087	50			
ID_043	100			
ID_013	100			
ID_016	100			
ID_022	100			
ID_080	100			
ID_113	500			
ID_136	500			
ID_014	1000			
ID_120	1000			
ID_118	1000			
ID_012	5000			

Tables A.34-36 show the bandwidth consumed by each message type for a history buffer size of twenty samples, using validity voting. Votes were applied as per Tables A.28-33.

Table A.34. High assurance message bandwidth consumption for validity voting. History buffer size is 20 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_009	10	44	7		3	3	2		2	3		2		3		2		22	9	2200	25000
ID_008	10	49	0							3								3	7	300	15000
ID_047	10	49	2	3			2	3	2		3	2			3	2	3	25	10	2500	26000
ID_040	12	62	0									3						3	9	250	20833.33333
ID_001	12	55	0		3					3								6	8	500	13333.33333
ID_007	12	64	4	2	2	2	2	2	2	2	2		3	2		2	2	25	12	2083.333	23333.33333
ID_039	20	36	0									3		3				6	6	300	7000
ID_042	20	24	0				3											3	4	150	6000
ID_025	25	52	0									3						3	7	120	6000
ID_029	25	64	1									3						3	9	120	10000
ID_030	25	64	0					2		2		3		2				9	10	360	10400
ID_038	25	56	1									3						3	8	120	6400
ID_036	25	64	2					3		3				3				10	10	400	10400
ID_074	25	16	1		3													3	3	120	4400
ID_046	30	52	0									3		3				6	8	200	5333.333333
ID_057	30	60	1		3							3						7	9	233.3333	8333.333333
ID_076	35	52	0									3						3	7	85.71429	4285.714286
ID_077	35	34	0									3						3	5	85.71429	3714.285714
ID_078	35	34	0									3						3	5	85.71429	3714.285714
ID_058	50	33	1					2	3					2		3		11	6	220	2800
ID_081	50	45	0		2		3	3				2						10	7	200	3000
ID_061	50	46	2					2		3				2				8	7	160	3000
ID_098	100	37	0							3								3	5	30	1300
ID_060	100	12	0							3								3	2	30	1000

Table A.35. Medium assurance message bandwidth consumption for validity voting. History buffer size is 20 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_006	6	32	0				2											2	5	333.3333	21666.66667
ID_004	10	64	0		2		2	2	2		2	2	2		2	2	2	20	11	2000	27000
ID_005	10	64	0	2	2		2	2	2		2	2	2		2	2	2	22	11	2200	27000
ID_010	12	61	0				2		2	2		2						8	9	666.6667	20833.33333
ID_003	12	9	0		2													2	2	166.6667	8333.333333
ID_026	12	31	0		2													2	5	166.6667	10833.33333
ID_027	12	62	0		2		2											4	9	333.3333	20833.33333
ID_048	12	59	0										2					2	8	166.6667	13333.33333
ID_052	12	61	0										2					2	8	166.6667	13333.33333
ID_041	20	26	0		2					2				2				6	4	300	6000
ID_045	20	27	0							2								2	4	100	6000
ID_024	20	11	0		2			2	2							2	2	10	3	500	5500
ID_049	20	62	0	2	2	2	2	2	2		2	2		2	2	2	2	24	11	1200	13500
ID_028	25	16	0									2						2	3	80	4400
ID_033	25	45	0									2						2	6	80	5600
ID_106	25	17	0									2						2	3	80	4400
ID_031	25	54	0		2													2	7	80	6000
ID_034	25	62	0		2													2	8	80	6400
ID_035	25	57	0				2	2	2	2	2			2		2	2	16	10	640	10400
ID_037	25	48	0							2				2				4	7	160	6000
ID_075	50	40	0				2			2								4	6	80	2800
ID_018	100	24	0	2														2	4	20	1200
ID_020	100	34	0							2	2							4	5	40	1300
ID_053	100	54	0	2	2	2	2		2	2		2	2	2	2	2	2	24	10	240	2600
ID_059	100	9	0					2			2							4	2	40	1000
ID_023	100	18	0					2										2	3	20	1100
ID_021	100	18	0					2										2	3	20	1100
ID_102	250	58	0				2		2	2		2				2	2	12	9	48	1000
ID_101	250	44	0									2						2	6	8	560
ID_083	500	16	0					2		2	2							6	3	12	220
ID_017	1000	17	0	2										2				4	3	4	110
ID_117	1000	45	0		2		2					2						6	7	6	150

Table A.36. Low assurance message bandwidth consumption for validity voting. History buffer size is 20 samples.

Message ID	Period (ms)	Payload bits	Validity vector bits	Tag size for each receiver (bits)														Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second
				1	2	3	4	5	6	7	8	9	10	11	12	13	14				
ID_044	20	3	0							2								2	1	100	4500
ID_002	25	53	0									2						2	7	80	6000
ID_056	25	64	0	2			2	2	2	2	2	2		2	2	2	2	22	11	880	10800
ID_082	25	60	0	2					2	2								6	9	240	10000
ID_032	25	1	0		2					2								4	1	160	3600
ID_054	30	16	0		2							2						4	3	133.3333	3666.666667
ID_088	35	16	0					2		2								4	3	114.2857	3142.857143
ID_089	35	48	0		2			2		2								6	7	171.4286	4285.714286
ID_084	50	36	0			2	2	2	2					2	2	2	2	16	7	320	3000
ID_085	50	36	0			2	2	2	2					2	2	2	2	16	7	320	3000
ID_087	50	28	0					2										2	4	40	2400
ID_043	100	6	0		2			2		2	2	2		2				12	3	120	1100
ID_013	100	57	0	2	2				2	2		2	2	2		2		16	10	160	2600
ID_016	100	9	0							2				2				4	2	40	1000
ID_022	100	47	0	2		2	2	2	2		2		2	2	2	2		20	9	200	2500
ID_080	100	40	0									2						2	6	20	1400
ID_113	500	56	0					2			2							4	8	8	320
ID_136	500	64	0		2													2	9	4	500
ID_014	1000	3	0										2					2	1	2	90
ID_120	1000	25	0				2	2	2		2		2	2	2	2	2	18	6	18	140
ID_118	1000	44	0		2		2	2		2			2	2	2	2		16	8	16	160
ID_012	5000	33	0				2											2	5	0.4	26

A.3 TESLA

A.3.1 TESLA - history buffer size = 5 samples

Table A.37. High assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 10 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	90	17	9000	41000
ID_008	10	49	90	18	9000	42000
ID_047	10	49	90	18	9000	42000
ID_040	12	62	90	19	7500	35833.33333
ID_001	12	55	90	19	7500	35833.33333
ID_007	12	64	90	20	7500	36666.66667
ID_039	20	36	90	16	4500	16000
ID_042	20	24	90	15	4500	15500
ID_025	25	52	90	18	3600	16800
ID_029	25	64	90	20	3600	17600
ID_030	25	64	90	20	3600	17600
ID_038	25	56	90	19	3600	17200
ID_036	25	64	90	20	3600	17600
ID_074	25	16	90	14	3600	12000
ID_046	30	52	90	18	3000	14000
ID_057	30	60	90	19	3000	14333.33333
ID_076	35	52	90	18	2571	12000
ID_077	35	34	90	16	2571	9142.857143
ID_078	35	34	90	16	2571	9142.857143
ID_058	50	33	90	16	1800	6400
ID_081	50	45	90	17	1800	8200
ID_061	50	46	90	17	1800	8200
ID_098	100	37	90	16	900	3200
ID_060	100	12	90	13	900	2900

Table A.38. Medium assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 8 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	88	15	14667	51666.66667
ID_004	10	64	88	19	8800	43000
ID_005	10	64	88	19	8800	43000
ID_010	12	61	88	19	7333	35833.33333
ID_003	12	9	88	13	7333	24166.66667
ID_026	12	31	88	15	7333	25833.33333
ID_027	12	62	88	19	7333	35833.33333
ID_048	12	59	88	19	7333	35833.33333
ID_052	12	61	88	19	7333	35833.33333
ID_041	20	26	88	15	4400	15500
ID_045	20	27	88	15	4400	15500
ID_024	20	11	88	13	4400	14500
ID_049	20	62	88	19	4400	21500
ID_028	25	16	88	13	3520	11600
ID_033	25	45	88	17	3520	16400
ID_106	25	17	88	14	3520	12000
ID_031	25	54	88	18	3520	16800
ID_034	25	62	88	19	3520	17200
ID_035	25	57	88	19	3520	17200
ID_037	25	48	88	17	3520	16400
ID_075	50	40	88	16	1760	6400
ID_018	100	24	88	14	880	3000
ID_020	100	34	88	16	880	3200
ID_053	100	54	88	18	880	4200
ID_059	100	9	88	13	880	2900
ID_023	100	18	88	14	880	3000
ID_021	100	18	88	14	880	3000
ID_102	250	58	88	19	352	1720
ID_101	250	44	88	17	352	1640
ID_083	500	16	88	13	176	580
ID_017	1000	17	88	14	88	300
ID_117	1000	45	88	17	88	410

Table A.39. Low assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 6 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	86	12	4300	14000
ID_002	25	53	86	18	3440	16800
ID_056	25	64	86	19	3440	17200
ID_082	25	60	86	19	3440	17200
ID_032	25	1	86	11	3440	10800
ID_054	30	16	86	13	2867	9666.666667
ID_088	35	16	86	13	2457	8285.714286
ID_089	35	48	86	17	2457	11714.28571
ID_084	50	36	86	16	1720	6400
ID_085	50	36	86	16	1720	6400
ID_087	50	28	86	15	1720	6200
ID_043	100	6	86	12	860	2800
ID_013	100	57	86	18	860	4200
ID_016	100	9	86	12	860	2800
ID_022	100	47	86	17	860	4100
ID_080	100	40	86	16	860	3200
ID_113	500	56	86	18	172	840
ID_136	500	64	86	19	172	860
ID_014	1000	3	86	12	86	280
ID_120	1000	25	86	14	86	300
ID_118	1000	44	86	17	86	410
ID_012	5000	33	86	15	17	62

A.3.2 TESLA - history buffer size = 10 samples

Table A.40. High assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 5 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	85	17	8500	41000
ID_008	10	49	85	17	8500	41000
ID_047	10	49	85	17	8500	41000
ID_040	12	62	85	19	7083	35833.33333
ID_001	12	55	85	18	7083	35000
ID_007	12	64	85	19	7083	35833.33333
ID_039	20	36	85	16	4250	16000
ID_042	20	24	85	14	4250	15000
ID_025	25	52	85	18	3400	16800
ID_029	25	64	85	19	3400	17200
ID_030	25	64	85	19	3400	17200
ID_038	25	56	85	18	3400	16800
ID_036	25	64	85	19	3400	17200
ID_074	25	16	85	13	3400	11600
ID_046	30	52	85	18	2833	14000
ID_057	30	60	85	19	2833	14333.33333
ID_076	35	52	85	18	2429	12000
ID_077	35	34	85	15	2429	8857.142857
ID_078	35	34	85	15	2429	8857.142857
ID_058	50	33	85	15	1700	6200
ID_081	50	45	85	17	1700	8200
ID_061	50	46	85	17	1700	8200
ID_098	100	37	85	16	850	3200
ID_060	100	12	85	13	850	2900

Table A.41. Medium assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 4 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	84	15	14000	51666.66667
ID_004	10	64	84	19	8400	43000
ID_005	10	64	84	19	8400	43000
ID_010	12	61	84	19	7000	35833.33333
ID_003	12	9	84	12	7000	23333.33333
ID_026	12	31	84	15	7000	25833.33333
ID_027	12	62	84	19	7000	35833.33333
ID_048	12	59	84	18	7000	35000
ID_052	12	61	84	19	7000	35833.33333
ID_041	20	26	84	14	4200	15000
ID_045	20	27	84	14	4200	15000
ID_024	20	11	84	12	4200	14000
ID_049	20	62	84	19	4200	21500
ID_028	25	16	84	13	3360	11600
ID_033	25	45	84	17	3360	16400
ID_106	25	17	84	13	3360	11600
ID_031	25	54	84	18	3360	16800
ID_034	25	62	84	19	3360	17200
ID_035	25	57	84	18	3360	16800
ID_037	25	48	84	17	3360	16400
ID_075	50	40	84	16	1680	6400
ID_018	100	24	84	14	840	3000
ID_020	100	34	84	15	840	3100
ID_053	100	54	84	18	840	4200
ID_059	100	9	84	12	840	2800
ID_023	100	18	84	13	840	2900
ID_021	100	18	84	13	840	2900
ID_102	250	58	84	18	336	1680
ID_101	250	44	84	16	336	1280
ID_083	500	16	84	13	168	580
ID_017	1000	17	84	13	84	290
ID_117	1000	45	84	17	84	410

Table A.42. Low assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 3 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	86	11	4150	13500
ID_002	25	53	86	17	3320	16400
ID_056	25	64	86	19	3320	17200
ID_082	25	60	86	18	3320	16800
ID_032	25	1	86	11	3320	10800
ID_054	30	16	86	13	2767	9666.666667
ID_088	35	16	86	13	2371	8285.714286
ID_089	35	48	86	17	2371	11714.28571
ID_084	50	36	86	15	1660	6200
ID_085	50	36	86	15	1660	6200
ID_087	50	28	86	14	1660	6000
ID_043	100	6	86	12	830	2800
ID_013	100	57	86	18	830	4200
ID_016	100	9	86	12	830	2800
ID_022	100	47	86	17	830	4100
ID_080	100	40	86	16	830	3200
ID_113	500	56	86	18	166	840
ID_136	500	64	86	19	166	860
ID_014	1000	3	86	11	83	270
ID_120	1000	25	86	14	83	300
ID_118	1000	44	86	16	83	320
ID_012	5000	33	86	15	17	62

A.3.3 TESLA - history buffer size = 20 samples

Table A.43. High assurance messages authenticated with TESLA. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 3 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	83	16	8300	32000
ID_008	10	49	83	17	8300	41000
ID_047	10	49	83	17	8300	41000
ID_040	12	62	83	19	6917	35833.33333
ID_001	12	55	83	18	6917	35000
ID_007	12	64	83	19	6917	35833.33333
ID_039	20	36	83	15	4150	15500
ID_042	20	24	83	14	4150	15000
ID_025	25	52	83	17	3320	16400
ID_029	25	64	83	19	3320	17200
ID_030	25	64	83	19	3320	17200
ID_038	25	56	83	18	3320	16800
ID_036	25	64	83	19	3320	17200
ID_074	25	16	83	13	3320	11600
ID_046	30	52	83	17	2767	13666.66667
ID_057	30	60	83	18	2767	14000
ID_076	35	52	83	17	2371	11714.28571
ID_077	35	34	83	15	2371	8857.142857
ID_078	35	34	83	15	2371	8857.142857
ID_058	50	33	83	15	1660	6200
ID_081	50	45	83	16	1660	6400
ID_061	50	46	83	17	1660	8200
ID_098	100	37	83	15	830	3100
ID_060	100	12	83	12	830	2800

Table A.44. Medium assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 2 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	82	15	13667	51666.66667
ID_004	10	64	82	19	8200	43000
ID_005	10	64	82	19	8200	43000
ID_010	12	61	82	18	6833	35000
ID_003	12	9	82	12	6833	23333.33333
ID_026	12	31	82	15	6833	25833.33333
ID_027	12	62	82	18	6833	35000
ID_048	12	59	82	18	6833	35000
ID_052	12	61	82	18	6833	35000
ID_041	20	26	82	14	4100	15000
ID_045	20	27	82	14	4100	15000
ID_024	20	11	82	12	4100	14000
ID_049	20	62	82	18	4100	21000
ID_028	25	16	82	13	3280	11600
ID_033	25	45	82	16	3280	12800
ID_106	25	17	82	13	3280	11600
ID_031	25	54	82	17	3280	16400
ID_034	25	62	82	18	3280	16800
ID_035	25	57	82	18	3280	16800
ID_037	25	48	82	17	3280	16400
ID_075	50	40	82	16	1640	6400
ID_018	100	24	82	14	820	3000
ID_020	100	34	82	15	820	3100
ID_053	100	54	82	17	820	4100
ID_059	100	9	82	12	820	2800
ID_023	100	18	82	13	820	2900
ID_021	100	18	82	13	820	2900
ID_102	250	58	82	18	328	1680
ID_101	250	44	82	16	328	1280
ID_083	500	16	82	13	164	580
ID_017	1000	17	82	13	82	290
ID_117	1000	45	82	16	82	320

Table A.45. Low assurance messages authenticated with one MAC per receiver. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 2 bits. Key size is 80 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	82	11	4100	13500
ID_002	25	53	82	17	3280	16400
ID_056	25	64	82	19	3280	17200
ID_082	25	60	82	18	3280	16800
ID_032	25	1	82	11	3280	10800
ID_054	30	16	82	13	2733	9666.666667
ID_088	35	16	82	13	2343	8285.714286
ID_089	35	48	82	17	2343	11714.28571
ID_084	50	36	82	15	1640	6200
ID_085	50	36	82	15	1640	6200
ID_087	50	28	82	14	1640	6000
ID_043	100	6	82	11	820	2700
ID_013	100	57	82	18	820	4200
ID_016	100	9	82	12	820	2800
ID_022	100	47	82	17	820	4100
ID_080	100	40	82	16	820	3200
ID_113	500	56	82	18	164	840
ID_136	500	64	82	19	164	860
ID_014	1000	3	82	11	82	270
ID_120	1000	25	82	14	82	300
ID_118	1000	44	82	16	82	320
ID_012	5000	33	82	15	16	62

A.4 Master-slave

Tables A.46-48 define the message types in which nodes include the MAC tags for verification of the master node's hash tree broadcast authenticators.

Table A.46. High assurance message types that carry tags for verifying hash-tree broadcast authenticators.

Sender ID	Message ID	Period (ms)	Added message type?
ECU_01	ID_A_01	100	Y
ECU_02	ID_025	25	N
ECU_03	ID_A_03	100	Y
ECU_04	ID_A_04	20	Y
ECU_05	ID_009	10	N
ECU_06	ID_A_06	25	Y
ECU_07	ID_008	10	N
ECU_08	ID_A_08	100	Y
ECU_09	ID_001	12	N
ECU_10	ID_A_10	12	Y
ECU_11	ID_076	35	N
ECU_12	ID_A_12	1000	Y
ECU_13	ID_061	50	N
ECU_14	ID_A_14	1000	Y

Table A.47. Medium assurance message types that carry tags for verifying hash-tree broadcast authenticators.

Sender ID	Message ID	Period (ms)	Added message type?
ECU_01	ID_B_01	100	Y
ECU_02	ID_006	6	N
ECU_03	ID_B_03	100	Y
ECU_04	ID_041	20	N
ECU_05	ID_106	25	N
ECU_06	ID_059	100	N
ECU_07	ID_004	10	N
ECU_08	ID_021	100	N
ECU_09	ID_003	12	N
ECU_10	ID_B_10	12	Y
ECU_11	ID_B_11	35	Y
ECU_12	ID_B_12	1000	Y
ECU_13	ID_B_13	10	Y
ECU_14	ID_B_14	1000	Y

Table A.48. Low assurance message types that carry tags for verifying hash-tree broadcast authenticators.

Sender ID	Message ID	Period (ms)	Added message type?
ECU_01	ID_C_01	100	Y
ECU_02	ID_002	25	N
ECU_03	ID_C_03	100	Y
ECU_04	ID_044	20	N
ECU_05	ID_054	30	N
ECU_06	ID_082	25	N
ECU_07	ID_084	50	N
ECU_08	ID_C_08	100	Y
ECU_09	ID_032	25	N
ECU_10	ID_C_10	12	Y
ECU_11	ID_088	35	N
ECU_12	ID_C_12	1000	Y
ECU_13	ID_C_13	50	Y
ECU_14	ID_C_14	1000	Y

A.4.1 Master-slave - history buffer size = 5 samples

Table A.49. High assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 11 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	22	9	2200	25000
ID_008	10	49	22	9	2200	25000
ID_047	10	49	11	8	1100	16000
ID_040	12	62	11	10	917	21666.66667
ID_001	12	55	22	10	1833	21666.66667
ID_007	12	64	11	10	917	21666.66667
ID_039	20	36	11	6	550	7000
ID_042	20	24	11	5	550	6500
ID_025	25	52	22	10	880	10400
ID_029	25	64	11	10	440	10400
ID_030	25	64	11	10	440	10400
ID_038	25	56	11	9	440	10000
ID_036	25	64	11	10	440	10400
ID_074	25	16	11	4	440	4800
ID_046	30	52	11	2	367	3333.333333
ID_057	30	60	11	9	367	8333.333333
ID_076	35	52	22	10	629	7428.571429
ID_077	35	34	11	6	314	4000
ID_078	35	34	11	6	314	4000
ID_058	50	33	11	6	220	2800
ID_081	50	45	11	7	220	3000
ID_061	50	46	22	9	440	5000
ID_098	100	37	11	6	110	1400
ID_060	100	12	11	3	110	1100
ID_A_Mstr	10	1	11	2	1100	10000
ID_A_01	10	0	11	2	110	1000
ID_A_03	10	0	11	2	110	1000
ID_A_04	10	0	11	2	550	5000
ID_A_06	10	0	11	2	440	4000
ID_A_08	10	0	11	2	110	1000
ID_A_10	12	0	11	2	917	8333.333333
ID_A_12	10	0	11	2	11	100
ID_A_14	10	0	11	2	11	100

Table A.50. Medium assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 9 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	18	7	3000	25000
ID_004	10	64	18	11	1800	27000
ID_005	10	64	9	10	900	26000
ID_010	12	61	9	9	750	20833.33333
ID_003	12	9	18	4	1500	10000
ID_026	12	31	9	5	750	10833.33333
ID_027	12	62	9	9	750	20833.33333
ID_048	12	59	9	9	750	20833.33333
ID_052	12	61	9	9	750	20833.33333
ID_041	20	26	18	6	900	7000
ID_045	20	27	9	5	450	6500
ID_024	20	11	9	3	450	5500
ID_049	20	62	9	9	450	12500
ID_028	25	16	9	4	360	4800
ID_033	25	45	9	7	360	6000
ID_106	25	17	18	5	720	5200
ID_031	25	54	9	8	360	6400
ID_034	25	62	9	9	360	10000
ID_035	25	57	9	9	360	10000
ID_037	25	48	9	8	360	6400
ID_075	50	40	9	7	180	3000
ID_018	100	24	9	5	90	1300
ID_020	100	34	9	6	90	1400
ID_053	100	54	9	8	90	1600
ID_059	100	9	18	4	180	1200
ID_023	100	18	9	4	90	1200
ID_021	100	18	18	5	180	1300
ID_102	250	58	9	9	36	1000
ID_101	250	44	9	7	36	600
ID_083	500	16	9	4	18	240
ID_017	1000	17	9	4	9	120
ID_117	1000	45	9	7	9	150
ID_B_Mstr	10	1	9	2	900	10000
ID_B_01	10	0	9	2	90	1000
ID_B_03	20	0	9	2	90	1000
ID_B_10	10	0	9	2	750	8333.333333
ID_B_11	20	0	9	2	257	2857.142857
ID_B_12	10	0	9	2	9	100
ID_B_13	10	0	9	2	180	2000
ID_B_14	10	0	9	2	9	100

Table A.51. Low assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 5 samples. Tag size is 7 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	14	3	700	5500
ID_002	25	53	14	9	560	10000
ID_056	25	64	7	9	280	10000
ID_082	25	60	14	10	560	10400
ID_032	25	1	14	2	560	4000
ID_054	30	16	14	4	467	4000
ID_088	35	16	14	4	400	3428.571429
ID_089	35	48	7	1	200	2571.428571
ID_084	50	36	14	7	280	3000
ID_085	50	36	7	6	140	2800
ID_087	50	28	7	5	140	2600
ID_043	100	6	7	2	70	1000
ID_013	100	57	7	1	70	900
ID_016	100	9	7	2	70	1000
ID_022	100	47	7	7	70	1500
ID_080	100	40	7	6	70	1400
ID_113	500	56	7	8	14	320
ID_136	500	64	7	9	14	500
ID_014	1000	3	7	2	7	100
ID_120	1000	25	7	4	7	120
ID_118	1000	44	7	7	7	150
ID_012	5000	33	7	5	1	26
ID_C_Mstr	20	1	7	1	350	4500
ID_C_01	25	0	7	1	70	900
ID_C_03	50	0	7	1	70	900
ID_C_08	25	0	7	1	70	900
ID_C_10	100	0	7	1	583	7500
ID_C_12	25	0	7	1	7	90
ID_C_13	25	0	7	1	140	1800
ID_C_14	25	0	7	1	7	90

A.4.2 Master-slave - history buffer size = 10 samples

Table A.52. High assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 6 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	12	7	1200	15000
ID_008	10	49	12	8	1200	16000
ID_047	10	49	6	7	600	15000
ID_040	12	62	6	9	500	20833.33333
ID_001	12	55	12	9	1000	20833.33333
ID_007	12	64	6	9	500	20833.33333
ID_039	20	36	6	6	300	7000
ID_042	20	24	6	4	300	6000
ID_025	25	52	12	8	480	6400
ID_029	25	64	6	9	240	10000
ID_030	25	64	6	9	240	10000
ID_038	25	56	6	8	240	6400
ID_036	25	64	6	9	240	10000
ID_074	25	16	6	3	240	4400
ID_046	30	52	6	1	200	3000
ID_057	30	60	6	9	200	8333.333333
ID_076	35	52	12	8	343	4571.428571
ID_077	35	34	6	5	171	3714.285714
ID_078	35	34	6	5	171	3714.285714
ID_058	50	33	6	5	120	2600
ID_081	50	45	6	7	120	3000
ID_061	50	46	12	8	240	3200
ID_098	100	37	6	6	60	1400
ID_060	100	12	6	3	60	1100
ID_A_Mstr	10	1	6	1	600	9000
ID_A_01	10	0	6	1	60	900
ID_A_03	10	0	6	1	60	900
ID_A_04	10	0	6	1	300	4500
ID_A_06	10	0	6	1	240	3600
ID_A_08	10	0	6	1	60	900
ID_A_10	12	0	6	1	500	7500
ID_A_12	10	0	6	1	6	90
ID_A_14	10	0	6	1	6	90

Table A.53. Medium assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 5 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	10	6	1667	23333.33333
ID_004	10	64	10	10	1000	26000
ID_005	10	64	5	9	500	25000
ID_010	12	61	5	9	417	20833.33333
ID_003	12	9	10	3	833	9166.666667
ID_026	12	31	5	5	417	10833.33333
ID_027	12	62	5	9	417	20833.33333
ID_048	12	59	5	8	417	13333.33333
ID_052	12	61	5	9	417	20833.33333
ID_041	20	26	10	5	500	6500
ID_045	20	27	5	4	250	6000
ID_024	20	11	5	2	250	5000
ID_049	20	62	5	9	250	12500
ID_028	25	16	5	3	200	4400
ID_033	25	45	5	7	200	6000
ID_106	25	17	10	4	400	4800
ID_031	25	54	5	8	200	6400
ID_034	25	62	5	9	200	10000
ID_035	25	57	5	8	200	6400
ID_037	25	48	5	7	200	6000
ID_075	50	40	5	6	100	2800
ID_018	100	24	5	4	50	1200
ID_020	100	34	5	5	50	1300
ID_053	100	54	5	8	50	1600
ID_059	100	9	10	3	100	1100
ID_023	100	18	5	3	50	1100
ID_021	100	18	10	4	100	1200
ID_102	250	58	5	8	20	640
ID_101	250	44	5	7	20	600
ID_083	500	16	5	3	10	220
ID_017	1000	17	5	3	5	110
ID_117	1000	45	5	7	5	150
ID_B_Mstr	10	1	5	1	500	9000
ID_B_01	10	0	5	1	50	900
ID_B_03	20	0	5	1	50	900
ID_B_10	10	0	5	1	417	7500
ID_B_11	20	0	5	1	143	2571.428571
ID_B_12	10	0	5	1	5	90
ID_B_13	10	0	5	1	100	1800
ID_B_14	10	0	5	1	5	90

Table A.54. Low assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 10 samples. Tag size is 4 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	8	2	400	5000
ID_002	25	53	8	8	320	6400
ID_056	25	64	8	2	320	4000
ID_082	25	60	4	9	160	10000
ID_032	25	1	8	9	320	10000
ID_054	30	16	8	3	267	3666.666667
ID_088	35	16	8	3	229	3142.857143
ID_089	35	48	4	1	114	2571.428571
ID_084	50	36	8	6	160	2800
ID_085	50	36	4	5	80	2600
ID_087	50	28	4	4	80	2400
ID_043	100	6	4	1	40	900
ID_013	100	57	4	2	40	1000
ID_016	100	9	4	7	40	1500
ID_022	100	47	4	2	40	1000
ID_080	100	40	4	6	40	1400
ID_113	500	56	4	8	8	320
ID_136	500	64	4	9	8	500
ID_014	1000	3	4	1	4	90
ID_120	1000	25	4	6	4	140
ID_118	1000	44	4	4	4	120
ID_012	5000	33	4	5	1	26
ID_C_Mstr	20	1	4	1	200	4500
ID_C_01	25	0	4	1	40	900
ID_C_03	50	0	4	1	40	900
ID_C_08	25	0	4	1	40	900
ID_C_10	100	0	4	1	333	7500
ID_C_12	25	0	4	1	4	90
ID_C_13	25	0	4	1	80	1800
ID_C_14	25	0	4	1	4	90

A.4.3 Master-slave - history buffer size = 20 samples

Table A.55. High assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 4 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_009	10	44	8	7	800	15000
ID_008	10	49	8	8	800	16000
ID_047	10	49	4	7	400	15000
ID_040	12	62	4	9	333	20833.33333
ID_001	12	55	8	8	667	13333.33333
ID_007	12	64	4	9	333	20833.33333
ID_039	20	36	4	5	200	6500
ID_042	20	24	4	4	200	6000
ID_025	25	52	8	8	320	6400
ID_029	25	64	4	9	160	10000
ID_030	25	64	4	9	160	10000
ID_038	25	56	4	8	160	6400
ID_036	25	64	4	9	160	10000
ID_074	25	16	4	3	160	4400
ID_046	30	52	4	1	133	3000
ID_057	30	60	4	8	133	5333.333333
ID_076	35	52	8	8	229	4571.428571
ID_077	35	34	4	5	114	3714.285714
ID_078	35	34	4	5	114	3714.285714
ID_058	50	33	4	5	80	2600
ID_081	50	45	4	7	80	3000
ID_061	50	46	8	7	160	3000
ID_098	100	37	4	6	40	1400
ID_060	100	12	4	2	40	1000
ID_A_Mstr	10	1	4	1	400	9000
ID_A_01	10	0	4	1	40	900
ID_A_03	10	0	4	1	40	900
ID_A_04	10	0	4	1	200	4500
ID_A_06	10	0	4	1	160	3600
ID_A_08	10	0	4	1	40	900
ID_A_10	12	0	4	1	333	7500
ID_A_12	10	0	4	1	4	90
ID_A_14	10	0	4	1	4	90

Table A.56. Medium assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 3 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_006	6	32	6	5	1000	21666.66667
ID_004	10	64	6	9	600	25000
ID_005	10	64	3	9	300	25000
ID_010	12	61	3	8	250	13333.33333
ID_003	12	9	6	2	500	8333.333333
ID_026	12	31	3	5	250	10833.33333
ID_027	12	62	3	9	250	20833.33333
ID_048	12	59	3	8	250	13333.33333
ID_052	12	61	3	8	250	13333.33333
ID_041	20	26	6	4	300	6000
ID_045	20	27	3	4	150	6000
ID_024	20	11	3	2	150	5000
ID_049	20	62	3	9	150	12500
ID_028	25	16	3	3	120	4400
ID_033	25	45	3	6	120	5600
ID_106	25	17	6	3	240	4400
ID_031	25	54	3	8	120	6400
ID_034	25	62	3	9	120	10000
ID_035	25	57	3	8	120	6400
ID_037	25	48	3	7	120	6000
ID_075	50	40	3	6	60	2800
ID_018	100	24	3	4	30	1200
ID_020	100	34	3	5	30	1300
ID_053	100	54	3	8	30	1600
ID_059	100	9	6	2	60	1000
ID_023	100	18	3	3	30	1100
ID_021	100	18	6	3	60	1100
ID_102	250	58	3	8	12	640
ID_101	250	44	3	6	12	560
ID_083	500	16	3	3	6	220
ID_017	1000	17	3	3	3	110
ID_117	1000	45	3	6	3	140
ID_B_Mstr	10	1	3	1	300	9000
ID_B_01	10	0	3	1	30	900
ID_B_03	20	0	3	1	30	900
ID_B_10	10	0	3	1	250	7500
ID_B_11	20	0	3	1	86	2571.428571
ID_B_12	10	0	3	1	3	90
ID_B_13	10	0	3	1	60	1800
ID_B_14	10	0	3	1	3	90

Table A.57. Low assurance messages authenticated with master-slave. Message type, period, authentication overhead. History buffer size is 20 samples. Tag size is 3 bits.

Message ID	Period (ms)	Payload bits	Total authentication bits	Total payload (bytes)	Authentication bits per second	Total bits per second (including CAN overhead)
ID_044	20	3	6	2	300	5000
ID_002	25	53	6	8	240	6400
ID_056	25	64	3	9	120	10000
ID_082	25	60	6	9	240	10000
ID_032	25	1	6	1	240	3600
ID_054	30	16	6	3	200	3666.666667
ID_088	35	16	6	3	171	3142.857143
ID_089	35	48	3	1	86	2571.428571
ID_084	50	36	6	6	120	2800
ID_085	50	36	3	5	60	2600
ID_087	50	28	3	4	60	2400
ID_043	100	6	3	2	30	1000
ID_013	100	57	3	1	30	900
ID_016	100	9	3	2	30	1000
ID_022	100	47	3	7	30	1500
ID_080	100	40	3	6	30	1400
ID_113	500	56	3	8	6	320
ID_136	500	64	3	9	6	500
ID_014	1000	3	3	1	3	90
ID_120	1000	25	3	4	3	120
ID_118	1000	44	3	6	3	140
ID_012	5000	33	3	5	1	26
ID_C_Mstr	20	1	3	1	150	4500
ID_C_01	25	0	3	1	30	900
ID_C_03	50	0	3	1	30	900
ID_C_08	25	0	3	1	30	900
ID_C_10	100	0	3	1	250	7500
ID_C_12	25	0	3	1	3	90
ID_C_13	25	0	3	1	60	1800
ID_C_14	25	0	3	1	3	90