MTR090281

MITRE TECHNICAL REPORT

Using Attestation to Lift Crash Resilience to Byzantine Resilience

September 2009

Jonathan Herzog
Jonathan Millen
Brian O'Hanlon
John D. Ramsdell
Ariel Segall

Sponsor: MITRE Innovation Program **Contract No.:** N/A
Dept. No.: G020 **Project No.:** 05MSR144-A9

MITRE
Center for Integrated Intelligence Systems
Bedford, Massachusetts

Abstract

This paper explores the use of attestation protocols as Byzantine failure detectors. An attestation protocol enables one node in a distributed system to obtain enough information about other nodes to detect malicious compromises. By filtering network communication, channels to Byzantine nodes are made to appear crashed. Distributed algorithms that tolerate channel failures are thus transformed into ones that tolerate Byzantine failures. Erlang modules to support filtering and attestation have been written, including a partial Trusted Platform Module (TPM) interface. A demonstration prototype for a leader election algorithm is in progress.

Contents

1 Introduction

1.1 Failures in Distributed Systems

Modern computing systems fail with depressing inevitability. Not only do computing systems simply crash, and stop operating, but they can also be subverted in a discouraging variety of ways. Much research has been and is being done on possible solutions, but it still behooves us to investigate methods for mitigating this unfortunate fact of life.

Many essential applications are implemented with a distributed system rather than a single platform. There is a famous quote from Leslie Lamport, "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." But a more optimistic view of the situation is that, once it is understood how multiple computers can cooperate to achieve a task, it becomes possible to achieve better failure resilience through effective use of redundancy in computing capacity and coordination protocols.

Despite its simplicity, this vision is actually quite difficult to achieve. The universe, it seems, it stacked against global coordination of this sort. Messages can be lost in transit. They can also be delayed an arbitrarily long time, making it impossible to know when to stop waiting for a message that will never come. Participants can crash at the most inopportune times, including *during* the execution of some distributed task.

Our concern with malicious attacks forces us to consider not only crash failures, but *Byzantine* failures as well. In this kind of failure—whether due to subversion, an operating system bug, or physical damage—a participant begins to run arbitrary code. Thus, a Byzantine participant can send any message to any other participant at any time, including messages that actively subvert the protocol; or it can simulate a crash; or it can act normally.

When viewed through the prism of previous results, Byzantine failures seem to be qualitatively different from crash failures. Consider, for example, the problem of *distributed agreement*. In its simplest form, this problem can be stated as follows (paraphrased from that of Lynch [14]):

> Suppose that all processes start with an input from some fixed value set V where each process may start with a different input. Then:
>
> 1. No two non-faulty processes output different values,
> 2. All non-faulty processes eventually output some value, and
> 3. If all non-Byzantine processes start with the same value $v \in V$, then v is the only possible output for non-Byzantine processes.
>
> Here, a 'faulty' process is either a crashed process or a Byzantine process, depending on the allowed faults.

Byzantine failures seem strictly stronger than crash failures. It is known, for example, that if the only possible failures are crash failures and participants can

detect crashes (though timeouts, for example) then agreement can be achieved so long as at least two participants remain alive [14]. But even under the most optimistic of settings—completely synchronous processes and reliable channels—agreement is not possible if less than two-thirds of the processes are honest (non-Byzantine).

Furthermore, it is known that if an agreement algorithm is resilient against Byzantine failures, then it must be time-consuming. The synchronous algorithms for Byzantine agreement proceed in rounds, and it is known that such protocols must use at least one round more than the number of Byzantine faults. The asynchronous agreement algorithms generally emulate the synchronous algorithms and proceed in rounds, but the need to simulate synchonicity in an asynchronous setting generally increases the cost of each round.

The core of the problem is that without a two-thirds majority of honest participants, the algorithm cannot detect inconsistent behavior on the part of the Byzantine processes. Therefore, no honest participant can ever be sure that it accurately knows the state of the global system– or even of any other participant. In short: no one can know who to trust, and therefore no one can trust anyone.

1.2 TPMs and Attestation

Recent technological advances promise to overcome our inability to test for Byzantine faults. Intrusions or faults in a system that permit Byzantine behavior can, in many cases, be detected and reported by protected subsystems. An *attestation protocol* is a way for a remote participant to request and obtain such a report. Various mechanisms have been suggested for implementing a protected subsystem capable of performing suitable local measurements.

In a setting proposed by the Trusted Computing Group (TCG), each computing platform would have a Trusted Platform Module (TPM): a hardware module that can store the 'fingerprint' of a platform's boot-up sequence [4]. The TPM can also sign these fingerprints with a unique key, certified as belonging to a TPM. Computers manufactured by members of the TCG already have these installed. A TPM, combined with software support, can produce a platform capable of attestation. Attestation protocols can also invoke and report run-time measurements made by tools such as LKIM for Linux kernel inspection [13].

This new technology has an immediate application to distributed algorithms. Attestation may make it possible to determine whether another participant is actually a Byzantine process. It should be admitted that, at present, no measurement and attestation technology is guaranteed to detect every Byzantine participant. Even now, however, attestation methods offer considerable improvement over their absence. For now, in the theoretical result, we make the simplifying assumption that Byzantine failures are always detectable by attestation. If this comes to pass, can we use this to circumvent the above bounds on agreement algorithms? Can we achieve agreement when more than a third of the participants are faulty, or do so in a more efficient fashion?

1.3 A Theoretical Result

In this work, we show that when attestation is possible, Byzantine fault resilience is actually *no harder* than resilience against channel failures. We do this in four steps:

- First, we define an 'ideal' network model in which crash faults are possible but Byzantine faults are not. That is, this model allows participants to crash and recover, channels to crash and recover, and individual messages can be dropped.[1] Also, the channels of this real model are unauthenticated (meaning that participants do not know the sender of a message beyond what can be determined from the message itself).
- We then define a 'real' network model that contains all of the failures of the 'ideal' model but with Byzantine failures as well. However, we also add two novel and related auxiliary processes:
  - An *attestation* process, representing an attestation protocol, and
  - A *dispatcher* process, which acts as a network proxy for participants.

  We encapsulate the general notion of an attestation protocol into an attestation process that will reliably inform one participant whether it and another participant are executing the same program.[2] Attestation protocols are used by dispatchers, which 'sit' between the network and all honest participants. Dispatchers use an attestation protocol to determine which other participants are honest and which are Byzantine, and to prevent messages from Byzantine participants from reaching honest participants. (To this end, the dispatchers will use cryptographic keys distributed by the attestation protocol.)
- We then show a kind of equivalence between two network models: a 'real' network model in which all honest participants are protected by dispatchers, and an 'ideal' model in which channels from Byzantine participants are permanently down. In particular, we show that every algorithm induces exactly the same set of traces in both network models.

As a corollary, this implies that an algorithm satisfies every trace property in the real model that it does in the ideal model. This means that the real network with dispatchers is behaviorally indistinguishable to the original participants from a network with possible channel crash failures but no Byzantine failures. Thus, in settings where attestation can be performed, channel-crash-resilient distributed algorithms can automatically become Byzantine-resilient as well.

1.4 Adding a Dispatcher to a Crash-Resilient Algorithm

With our approach, each participant uses a local 'dispatcher' process as a proxy for all network communication and failure detection. Attestation failures—

[1] Although one typically models channel failures as simply a subtype of message loss, we will actually use these two types of channel failures in different ways.

[2] Participants are state machines in our model, and so this can be rigorously defined as having the same transition table.

indicating Byzantine faults—are translated by the dispatchers into apparent crash failures and handled by the participants as such. The number of faults tolerated in the new setting (including both crashes and Byzantine failures) is exactly the number of crash failures tolerated in the ideal setting.

This means that the benefits of the dispatcher are independent of the extent of resilience offered by the original distributed algorithm. The dispatcher simply protects the honest participants from Byzantine attacks, but not from crashes. The advantage of this is that we can make effective use of existing crash-resilient algorithms in an environment with malware and Byzantine failures.

When the dispatcher engages in an attestation protocol, there are actually three possible results: (1) the attestation is a success and communication with the remote node is enabled; (2) the attestation fails and the dispatcher blocks messages from the remote node; or (3) the attestation protocol does not terminate, either because of communication delay or because of noncooperation from the remote participant.

Some distributed algorithms can distinguish between cases (2) and (3) with a *failure detector*[7, 6]. A failure detector for crashes is a function that can explicitly "suspect" a remote participant as having crashed, so that the rest of the algorithm can take appropriate action without waiting indefinitely. This is sometimes implemented by imposing a time-out. There are several possible models for the properties of a failure detector. In particular, their suspicions are not required to be correct, although they may be expected to converge on correctness if invoked repeatedly on the same target.

From the dispatcher's point of view, a failure detector is part of the algorithm, and its presence does not affect the attestation. However, a dispatcher can support a failure detector by passing along the negative result in case (2) above, so that the failure detector can report the dispatcher-induced crash as a suspicion to the participant using it.

1.5 Comparison With Other Approaches

There has been some prior work on Byzantine failure detection applicable to consensus algorithms. A paper by Doudou, Garbinato, and Guerraoui [9] proposes a protocol that detects a limited type of Byzantine misbehavior, called a *muteness failure*, in which messages to a chosen recipient may be deliberately stopped. Malkhi and Reiter [18] previously studied "quiet" processes, defined similarly, and proposed a protocol based on the assumption of an abstractly defined failure detector for them.

The work of Han, Pei, Ravindran, and Jensen [11] provides Byzantine tolerance for information dissemination with a gossip-based, real-time protocol. Gossip protocols disperse information gradually by sending messages to randomly selected group members, and their resiliency properties are probabilistic. The gossip-based protocol context makes it possible to define and detect Byzantine failures behaviorally.

Another example of an approach that deals with Byzantine failure in the context of a particular algorithm is [20]. They give an agreement and leader

election algorithm for a network in which channels may have Byzantine faults.

Our work differs from prior approaches in that it can address whatever form of Byzantine fault is detectable by remote attestation, and it can be applied to any higher-layer distributed algorithm. The resilience offered by the result depends on the crash tolerance of the higher-layer algorithm; the algorithm need not supply any Byzantine failure tolerance of its own.

1.6 Implementation Support in Erlang

To apply our theoretical result in a concrete setting, we chose Erlang as an implementation language. This language, designed for the implementation of distributed systems and algorithms, natively supports such high-level features such as crash-failure detection and reliable channels. Erlang is a parallel functional programming language designed for programming real-time control systems such as telephone exchanges and automated teller systems. The compiler and runtime environment are available under an open source license and as downloadable object code.

We implement four things in Erlang:

- A *parse transform* that can be applied to any Erlang code, forcing the dispatcher to be invoked;
- Our dispatcher, which employs an attestation protocol;
- An attestation protocol, which uses a TPM interface to obtain stored boot measurements;
- An Erlang interface to access a TPM.

The TPM interface requires an intermediate interface in C which will also be described.

A parse transform, when applied to a piece of Erlang code, rewrites the code so that it uses the dispatcher-filtered version of any command that causes network communication. The parse transform has to be applied both to the distributed system algorithm code and to Erlang libraries used by it. The Erlang compiler can conveniently be configured to apply a user-supplied parse transformation.

1.7 The Leader-Election Prototype

For purposes of demonstration, we added the dispatcher to a leader-election algorithm. A leader-election algorithm is a consensus algorithm in which the honest members of a group agree on the choice of a leader for some purpose of group coordination. If that leader crashes, then the surviving participants would choose a new leader from among their number to continue where the previous leader left off. Such an algorithm would guarantee a number of desirable properties, such as:

- Some participant eventually announces itself to be the leader, and
- No two participants simultaneously believe themselves to be the leader.

The leader does not have to be chosen by a vote. In fact, we used a known crash-resilient algorithm, the 'Bully' algorithm [10, 21, 22] which orders the group participants by priority and chooses the highest-priority non-crashed participant as the leader. We transformed the existing Erlang implementation of this algorithm (by Svensson and Arts [24]) into a Byzantine-resilient implementation by applying the parse transform to it.

1.8 Organization of the Paper

The rest of this paper is structured as follows. In Section 2, we formalize our 'ideal' network model in which the only possible failures are crash failures and message loss. In Section 3, we refine this model into a 'real' model that includes Byzantine participants. We also formalize attestation protocols and the dispatcher. We then formalize our main theorem in Section 4: when the participants use dispatchers, the two models (ideal and real) are perfectly trace equivalent. (We also comment on some possible variations of interest.) The theorem is proved in Appendix A.

The Erlang implementation, including the parse transform, dispatcher, and a basic attestation protocol, are summarized in Section 5. Finally, in Section 6, we discuss the Bully algorithm. The details of the demonstration software will be covered in a separate document.

2 A non-Byzantine Network Model

In this section, we present our 'ideal' network model. This network model is 'ideal' only in that Byzantine failures are not allowed. Participants are still allowed to crash, and channels are allowed to both crash and lose messages during normal operation. (Although it is typically unnecessary to model the first type of channel failure if the model already contains the second, our model will contain both for technical reasons that arise in the proof.) Furthermore, messages are unauthenticated, meaning that neither messages nor channels necessarily identify the sender of a message.

Our presentation is based heavily on that of Lynch [14], particularly her model for asynchronous networks. For the purposes of exposition, we will build our model in a number of steps. We begin by constructing the simplest possible network model—one with reliable authenticated channels and no faults. After this, we extend it as needed to capture our setting. In particular, we introduce crash failures for participants and for channels. We then allow channels to lose messages. We finish by removing the assumption that channels are authenticated.

2.1 A Simple Network Model

In this treatment, the network is treated as a n-node directed graph, $G = (V, E)$. The *processes* of the network (also called the *participants*) are associated with

the nodes of the graph and the edges of the graph are associated with *channels* of communication. Both the processes and the channels are modeled as input-output automata (IOA, [15, 16, 17]) over the same fixed *message alphabet* M. (Looking ahead, we note that we will extend this alphabet in our 'real' model of Section 3.) Each participant automaton P_i has some fixed input or output actions (not considered in this work) with the 'external user.' Also, it can have outputs of the form $send(m)_{i,j}$ and inputs of the form $receive(m)_{k,i}$ where $m \in M$, j is an outgoing neighbor of i, and k is an incoming neighbor of i. Aside from these interface restrictions, however, participants can be any I/O automaton the designer might like. We assume, however, that all participants are the *same* automaton. While they may start in different initial states, they must have the same state variables and transition function. In this way, we will later be able to clearly define which participants are honest and which are Byzantine: the honest participants are those executing the participant automaton, while the Byzantine participants are those executing any other automaton.

This initial model will use reliable FIFO channels, meaning that these channels do not 'lose' or re-order messages. The reliable channel $C_{i,j}$, which holds messages from P_i to P_j, has inputs of the form $send(m)_{i,j}$ and outputs of the form $receive(m)_{i,j}$ for $m \in M$.[3] These channels are *trace-equivalent* to (*i.e.*, have the same traces as) the I/O automaton in Figure 1. When a message is sent by participant P_i via the $send(m)_{i,j}$ event, the channel $C_{i,j}$ adds the message to the tail of an internal queue. Likewise, the channel $C_{i,j}$ delivers messages from the head of the queue to participant P_j via the $receive(m)_{i,j}$ event. Because these are the only ways to change the state of the message queue, it is guaranteed that

- No message will be delivered that is not sent, and
- No message will be delivered before all prior messages are delivered.

Also, these channels are implicitly authenticated. To see this, note that the only messages which can be on the message queue of channel $C_{i,j}$ are those put there by the event $send(m)_{i,j}$. And the only party which has event $send(m)_{i,j}$ as an output event is participant P_i. When the event $receive(m)_{i,j}$ occurs, therefore, P_j can know that message m was sent by P_i and no one else.

Much of interest can be said about this model of communication (and the algorithms that can be executed in it) and it underlies the network models we eventually adopt in this work. As it stands, however, it is actually too 'ideal' for our purposes. In particular, it implicitly assumes that nothing ever 'goes wrong': participants never crash, messages are never lost, *etc.* In this work, we will be creating an equivalence of sorts between two malicious and non-malicious faults. To that end, therefore, our 'ideal' model needs to contain at least non-malicious faults. We will therefore weaken the simple model of this section in three ways: participant failures, channel failures, and unauthenticated channels.

[3] We use the notation $C_{i,j}$ to distinguish these channels from others we will consider in this work under the names $C^{id}_{i,j}$ and $C^{re}_{i,j}$.

Signature:

Input:	Output:
$send(m)_{i,j}$, $m \in M$	$receive(m)_{i,j}$, $m \in M$

State Variables:
queue, a FIFO queue of messages in M, initially empty.

Transitions:

$send(m)_{i,j}$:
Effect:
add m to the *queue*.

$receive(m)_{i,j}$:
Precondition:
m is first on *queue*.
Effect:
remove first element of *queue*

Tasks:
Arbitrary.

Figure 1: Reliable FIFO channel I/O automaton $C_{i,j}$

2.2 Participant Faults

In this section, we extend the simple model of Section 2.1 so as to allow participants to crash. A *crash* failure (also called a *stop* failure) is one in which a participant simply stops execution. These failures are modeled by way of a special 'stop' event. The external interface of participant P_i is extended to include the input action $stop_i$, which will have the effect of halting P_i. These events can be thought of as happening non-deterministically, or of being 'triggered' by some arbitrary external mechanism. In this work, we also consider the possibility that processes can *recover* from the crash (perhaps by being re-started). To model this, each participant P_i also has the input action $recover_i$ in its external interface. We will make no assumptions about how participants will act upon recovery. Realistically, one possibility is that a participant resumes execution from the state it was in when it crashed. Alternately, a participants might immediately transition to a special 'recovered' state. Other options are possible as well, but the issue is not of concern here. Ultimately, the decision is that of the algorithm designer in response to the needs of the algorithm and the behavior of the computing platform. We will simply assume that participants *have* some defined behavior for recovery which can be called the 'honest' behavior.

In a given participant i, the sequence of $stop_i$ and $recover_i$ events can have two events of the form $stop_i$ or the form $recover_i$ in a row. However, we assume that if the sequence contains two or more events of a given form in a row, none of those events other than the first have any effect.

2.3 Unauthenticated Channels

In this section, we remove the assumption that channels are authenticated. For technical reasons, we cannot do this by 'removing' the sender's identity from

the channel automaton $C_{i,j}$. That is, channels must remain defined in terms of actual sender and recipient, and so the actions/events of a channel must remain in terms of those participants as well.[4] To remove the authenticated-channel assumption, therefore, we must re-examine where exactly the authenticated-channel assumption appears in the network model of Section 2.1. In particular, it arises from the fact that the event $\mathit{receive}(m)_{i,j}$ is:

1. common to both channel $C_{i,j}$ and participant P_i, and
2. contains the identity of the sender, i.

Thus, participant P_j can determine that the sender of the message m is participant P_i, and channels are therefore authenticated.

To remove this assumption, therefore, we need to change either of the two facts above. As previously stated, technical considerations require that the channel's events be defined in terms of both sender and receiver. Therefore, we cannot change the second fact in any essential way. To achieve unauthenticated channels, therefore, we change the first fact: channels and participants will no longer share a common event for message-reception. Instead, we use two new events:

- We will use the event $\mathit{deliver}(m)_{i,j}$ to represent the delivery of message m to P_j, sent by P_i.
- We will use the event $\mathit{receive}(m)_j$ to represent the reception of message m by participant P_j.

Because channels and participants no longer share a common event for message delivery and reception, we need to add a new component to the model. This new *anonymizer* component A_j^{re} will simply accept messages delivered by the channel via the event $\mathit{deliver}(m)_{i,j}$, and will then pass them through to the participant who receives them via the event $\mathit{receive}(m)_j$. Internally, the anonymizer simply holds a queue of delivered messages.

We formalize this new architecture by providing I/O automata for channels and anonymizers. Before we do so, however, we need to first consider our last extension to the simple model above: channel faults.

2.4 Channel Faults

In this section, we consider the faults that might affect channels. In particular, we consider two type of faults: the dropping of individual messages and the crash of the entire channel. Very often, these two types of misbehavior are modeled using the same mathematical 'machinery.' Because one could view a total channel failure simply as a long sequence of message drops, it is often simpler to only include message-drops in the model. We, however, will use

[4]To compose multiple I/O automata into a single automaton, it must be the case that no event can be an output event of more than one component automaton. If we redefine channels to be in terms of apparent sender instead of actual sender, therefore, or so that they are not defined in terms of sender at all, then some event must be an output event of multiple participants or channels. Therefore, channels must remain defined in terms of actual senders.

these two types of faults in different ways and so represent them using different formalisms. Furthermore, our theorem implicitly imposes different liveness conditions on these two faults. It is typically assumed, for example, that a lossy channel does not lose every message. More formally, it might be assumed that if an infinite number of messages are sent over a lossy channel, then infinitely many messages are successfully delivered [14]. We do not consider such liveness conditions for message loss in this work, but we *do* require it be possible that channels be down permanently. Thus, a similar liveness condition for channel failures will be impossible.

Just as participants crash and restart in response to 'stop' and 'recover' events, channels will do the same in response to 'channel up' and 'channel-down' events. When a channel has crashed, sent messages will automatically be lost. In terms of the I/O automaton of Figure 1, sent messages will simply not be added to the tail of the queue when the channel is crashed. The channel will return to normal operation, however, when instructed to do so via 'recover' event.

When a channel loses an individual message, on the other hand, it removes a message that has already been placed on the queue. In particular, a channel can delete an element of its queue by executing an internal 'lose' event. One still can achieve full generality by requiring the 'lose' event to delete the first message of the queue, or deleting the last. For our own convenience, however, we will allow *any* message in transit to be dropped. That is, we add two 'lose' events: one which deletes an arbitrary message from the channel, and one which deletes an arbitrary message from the anonymizer. (Messages held by the anonymizer are still in transit, only in an anonymized form.)

We can now give our new automata for channels ($C_{i,j}^{id}$) and anonymizers (A_j^{id}) in Figures 2 and 3. Note that these automata are marked as being part of the ideal model. For technical reasons, we will need slightly different automata for the 'real' model. Also, please note that we leave the 'tasks' of these automata as arbitrary. We will not use this aspect of I/O automata in this work, and so all of our automata will leave these fields in an indeterminate state for future results to refine.

2.5 Our 'Ideal' Network

In this section, we define our 'ideal' network model.

Definition 1 (Ideal network) *Given a graph $G = (V, E)$, let I_G be the I/O automaton created by composing the following components:*

- *For each node $i \in V$, a participant automaton P_i in some valid start state,*
- *An anonymizer A_i^{id} for every P_i, and*
- *A channel process $C_{i,j}^{id}$ for every edge (i, j) in E.*

Signature:

Input:
$send(m)_{i,j}$, $m \in M$
$channel\_down_{i,j}$
$channel\_up_{i,j}$

Output
$deliver(m)_{i,j}$, $m \in M$

Internal
$channel\_drop(n)_{i,j}$

State Variables:
stopped, a boolean value initially set to 'false'
queue, a FIFO queue of messages in M, initially empty.

Transitions:

$send(m)_{i,j}$:
Effect:
if *stopped* is false, add m to the *queue*. Otherwise, no effect.

$channel\_down_{i,j}$:
Effect:
set *stopped* to true.

$channel\_up_{i,j}$:
Effect:
set *stopped* to false.

$deliver(m)_{i,j}$:
Precondition:
m is first on *queue*.
Effect:
remove first element of *queue*.

$channel\_drop(n)_{i,j}$:
Precondition:
queue has at least n elements.
Effect:
remove the nth message from *queue*

Tasks:
Arbitrary.

Figure 2: The lossy channel I/O automaton $C_{i,j}^{id}$

Signature:

Input:
$deliver(m)_{i,j}$, $m \in M$

Output
$receive(m)_j$, $m \in M$

Internal
$anon\_drop(m)_j$

State Variables:
queue, a FIFO queue of messages, initially empty.

Transitions:

$deliver(m)_{i,j}$:
Effect:
add m to *queue*.

$receive(m)_j$:
Precondition:
m is the first element of *queue*.
Effect:
remove the first element of *queue*

$anon\_drop(n)_j$:
Precondition:
queue has at least n elements.
Effect:
remove the nth element of *queue*.

Tasks:
Arbitrary.

Figure 3: The *anonymizer* I/O automaton A_j^{id}

3 A Byzantine Network Model With Attestation

In this section, we refine the 'ideal' network model of Section 2 so as to allow malicious behavior. Participants can still crash, channels can still fail, and messages can still be dropped. However, participants can now undergo Byzantine faults, in which case they can deviate from the algorithm in arbitrary (and therefore arbitrarily dangerous) ways.

3.1 Modeling Byzantine Faults

We model a Byzantine fault at P_i by allowing the automaton for the P_i to be any I/O automaton with the same external interface. (Note that we will reconsider this interface at the end of Section 3.3.) Other than this, the model imposes no restrictions on the Byzantine participant, allowing them to deviate from the algorithm in unrestricted ways. Note that Byzantine failures are strictly stronger (*i.e.* more general) than stopping failures: one way the subverted participant can deviate from the algorithm is to simply stop.

Implicitly, this definition makes the strong assumption that Byzantine faults are *static*: participants cannot become Byzantine (or cease being Byzantine) during execution. As opposed to crash failures—which are dynamic in that participants can crash and recover at any time—we assume in this work that participants are always honest or always Byzantine. Because of this, we can regard the Byzantine participants as being a constant and unchanging subset of participants. In the following sections, we will use *Byz* to refer to this set. Note that our definition does not speak to the *maliciousness* of the Byzantine participants. They may, in fact, be executing the exact same algorithm as the honest participants—merely encoded in a different automaton. Once, however, a specific automaton is chosen for the participants to use, the use of any other automaton is considered to be a Byzantine fault.

3.2 Attestation Protocols

In this section, we discuss how Byzantine failures can be detected through *attestation* protocols. These protocols have been described by Coker *et al.* [8], and the material of this section is drawn from their exposition.

In an attestation protocol, one principal reliably learns properties about another principal's internal state. In particular, an *appraiser* uses an attestation protocol to evaluate some property of the *target*. In support of this, a third principal, the *attester*, will *measure* the target: observe the target's state and build evidence about this state, which will then be delivered to the appraiser.

Such a protocol can be trustworthy only if the attester shares special relationships with the other two principals. For example, the attester must inspect the target's internal state, meaning that the target and attester must reside on the same platform. On the other hand, the appraiser must trust the attester

to correctly gather and process the evidence in question, which means (in part) that the attester cannot be fooled or subverted by a malicious target.

To reconcile these two opposing requirements, attestation protocols must take place in a particular *attestation architecture*. In this setting, both the target and the attester are on the same physical machine, but running in different *domains*. The attester's domain would be privileged enough to inspect the target's domain, but the target's domain would not have the privileges necessary to interfere with the attester's domain.

One instantiation of this architecture is described by Coker *et al.* In this instantiation, the domains of the architecture are virtual machines (VMs) supported by a hypervisor such as Xen. The hypervisor runs on a platform equipped with a Trusted Platform Module (TPM). Information about the system software, including the hypervisor, is stored in the TPM when the platform is booted or launched in a secure mode. These platform measurements can be retrieved from the TPM with a digital signature for authentication. The target would be a VM containing the normal OS and applications of the user. The attester, on the other hand, would be running in a trusted VM, which has access to the TPM.

During the attestation protocol, the attester obtains the authenticated platform measurements from the TPM, and may also perform a "live" measurement of the target VM. These measurements are sent to the appraiser to evaluate.

One specific attestation protocol for this general architecture is the CAVES protocol, also by Coker *et al.* [8]. This protocol can be used to create a shared symmetric key between the target and the appraiser, if the appraiser is satisfied with the target platform measurements. Subsequent communication between the appraiser and target can be authenticated using this key.

In this work, we will encapsulate attestation and attestation protocols into a single 'functionality' module. This approach, inspired by the Universal Composability framework from cryptography [5] and the idea of failure detectors from distributed algorithms [7, 6], replaces the actual protocol with a trusted third party that implements the protocol's *functionality*. This allows us to separate our result from any particular protocol while also making it clear what it is about the protocol that we actually require. Protocol designers, therefore, need only show that their protocol successfully achieves the functionality encoded in our automaton.

Our automaton, given in Figures 4 and 5, encapsulates all runs of the attestation protocol where P_j (responder) measures P_i (initiator). This automaton simply waits for P_i to initiate the measurement and provide a *session identifier* for the initiated session. (This identifier allows the automaton to distinguish the various sessions of the protocol.) The automaton then informs P_j of the measurement and waits for P_j to agree to be measured. (These two steps represent the need for P_j to be aware of the protocol and to participate in it.) The behavior of attestation functionality is as follows:

- If $j \notin Byz$ (that is, the responder is not Byzantine), then the protocol can succeed, fail, or return an error.

  - At any time, the attestation automaton can transition to an error state. This might model, for example, timeouts in the underlying protocol or that the underlying channel is down. Once in an error state, the automaton will send an error event to each participant that has participated in the protocol so far.
  - If P_i is executing the same automaton as P_j and the protocol does not fall into an error state, then the protocol succeeds. The functionality executes two 'success' events: one for P_i and one for P_j. These events will contain cryptographic keys which the participants can use in later events.
  - If, on the other hand, P_i is not executing the same automata as P_j and the protocol does not return and error, then the protocol will fail. The functionality executes two 'failure' events: one for P_i and one for P_j.

- If, on the other hand, $j \in Byz$ (the responder is Byzantine), the protocol could proceed as above. However, the responder also has the option of choosing the ultimate resolution of the protocol itself by a special 'resolve' action. Using this action, the responder can have the automaton return success, failure, or error to the initiator.

We note a number of things about our attestation automaton:

- The responder can control the key distributed by the protocol to the following extent: it can either choose the key itself, or allow a fresh random key to be chosen.
- No matter how the key is chosen, the attestation automaton distributed the key to the responder before it distributes it to the initiator.
- Lastly, we note that a particular session identifier can only be used once. After a protocol completes, that is, the attestation automaton will mark the session identifier as 'used'.

3.3 Dispatchers

In this section, we introduce a new type of network module: the *dispatcher*. In our setting, dispatchers use attestation protocols to detect Byzantine faults in other participants, and use that ability to keep their participant from 'seeing' the activity of Byzantine participants. That is, our architecture positions dispatchers so that all network activity must pass 'through' a participant's dispatcher before it reaches the participant. However, the dispatcher will only allow traffic to pass through it if the traffic comes from a non-Byzantine participant. To enforce this requirement, dispatchers use an attestation protocol to measure other participants and create a shared key to authenticate subsequent traffic with the non-Byzantine participants.

More specifically, each dispatcher maintains a number of message-queues: one for all incoming messages, and one queue per participant for outgoing mes-

Signature:

Input:	Output
$start\_measure(id)_{i,j}$	$inform\_measure(id)_{i,j}$
$yes\_measure(id; x)_{i,j}$	$inform\_init\_success(id, k)_{i,j}$
	$inform\_resp\_success(id, k)_{i,j}$
Internal:	$resolve\_protocol(id, r, x)_{i,j}$
$encounter\_error(id)_{i,j}$	$inform\_init\_error(id)_{i,j}$
	$inform\_resp\_error(id)_{i,j}$

where $id \in \mathcal{N}$, k is a cryptographic key, x is either a cryptographic key or $\perp$, and r is in {`succeed`, `fail`, `error`}.

State Variables:

- *state*, a mapping from values of *id* to a state in the set {`no_prot`, `started`, `waiting`, `success1`, `success2`, `fail1`, `fail2`, `error2`, `error1`, `done`}.
- *key*, a mapping from values of *id* to either a cryptographic key or $\perp$, where initially $key(id) = \perp$ for all *id*.

Figure 4: The *attestation* I/O automaton $T_{i,j}$, Part I

sages. To process these queues, the dispatcher maintains an additional three data structures:

- For outgoing messages, the dispatcher maintains a mapping *out_key* which maps recipients to cryptographic keys. Recipients can also be mapped to $\perp$, meaning that no key has been established for that recipient.
- For incoming messages, the dispatcher maintains a mapping *in_key* which maps senders to cryptographic keys. Senders can also be mapped to $\perp$, meaning that no keys has been established for that sender.
- A set *in_error* of possible recipients. A recipient will be placed in this set when an attestation protocol returns an error.

When the dispatcher processes an outgoing message, it will look up the recipient in the mapping *out_key*. If a key is returned, the dispatcher will use it to apply a message authentication code (MAC) to the outgoing message. This cryptographic primitive acts as a symmetric signature: anyone with the key can create the MAC or use the MAC to verify that the message was not altered in transit. Because no one *without* the key can create a valid-seeming MAC, furthermore, the MAC can prove that the message was sent by a party that knows the key. In this work, the actual message sent will be the message, the MAC, and the sender's identity all together.

If, on the other hand, the dispatcher cannot find a key for the recipient in *out_key*, it will hold on to the message while it initiates the attestation protocol with the recipient. The outgoing message will remain in the relevant outgoing-message queue until

- The protocol completes, at which time it is processed as above, or

Transitions:

start_measure$(id)_{i,j}$:
Effect:
If *state*(id) = `no_prot` then set *state*(id) to `started`.

yes_measure$(id; x)_{i,j}$:
Effect:
if *state*(id) = `waiting` and P_i is the same automaton as P_j, then
- set *state*(id) to `success1`, and
- if $x \neq \perp$, set *key*$(id) = x$. Else, set *key*(id) to k, a randomly-chosen cryptographic key.

if *state*(id) = `waiting` and P_i is not the same automaton as P_j, then set *state*(id) to `fail1`.

inform_resp_success$(id, k)_{i,j}$:
Precondition:
state(id) = `success1`
key$(id) = k$
Effect:
set *state*(id) to `success2`

inform_resp_fail$(id)_{i,j}$:
Precondition:
state(id) = `fail1`
Effect:
set *state*(id) to `fail2`

encounter_error$(id)_{i,j}$:
Precondition:
state$(id) \neq$ `done`
Effect:
If *state*$(id) \in$ {`started` `success2`, `fail2`}, set *state*(id) to `error2`. If *state*$(id) \in$ {`waiting`, `success1`, `fail1`}. Else set *state*(id) to `error1`.

inform_resp_error$(id)_{i,j}$:
Precondition:
state(id) = `error1`
Effect:
set *state*(id) to `error2`

inform_init_error$(id)_{i,j}$:
Precondition:
state(id) = `error2`
Effect:
set *state*(id) to `done`

inform_measure$(id)_{i,j}$:
Precondition:
state(id) = `started`
Effect:
set *state*(id) to `waiting`

inform_init_success$(id, k)_{i,j}$:
Precondition:
state(id) = `success2`
key$(id) = k$
Effect:
set *state*(id) to `done`

inform_init_fail$(id)_{i,j}$:
Precondition:
state(id) = `fail2`
Effect:
set *state*(id) to `done`

resolve_protocol$(id, r, x)_{i,j}$:
Precondition:
state(id) = `started`
Effect:
If r = `succeed`, then set *state*(id) to `success1`. Also, if $x \neq \perp$, set *key*$(id) = x$. If $x = \perp$, set *key*(id) to a fresh random key k.

If r = `fail`, set *state*(id) = `fail2`.

If r = `fail`, set *state*(id) = `error2`.

Tasks:
Arbitrary.

Figure 5: The *attestation* I/O automaton $T_{i,j}$, Part II

- The protocol returns an error, at which time the dispatcher records the recipient in the set *in_error*. The dispatcher is also free to drop the message in this case.

Correspondingly, a dispatcher will engage in the attestation protocol with other dispatchers. When doing so, we note one quirk: if the dispatcher D_j already has a key in $in\_key(i)$ when P_i initiates a protocol, then D_j will instruct the attestation module to re-distribute that same key. (This fact will be used in the proof.) If D_j does not already have a key in $in\_key(i)$, on the other hand, it will instruct the attestation module to distribute a new, random key. In either case, if such a protocol succeeds then D_j will set $in\_key(i)$ to be the key distributed by the protocol.

Independently, the dispatcher will process incoming messages by verifying that they are authenticated with a key from an attestation protocol. That is, an incoming 'message' should actually be a triple containing message, MAC, and sender-identity. The dispatcher verifies that the sender-identity is on the second list, and then looks up the relevant key for messages from that sender. It then verifies the message's MAC using that key, and forwards the message on to its client-participant. Should the message's MAC not verify with the relevant key, the dispatcher will drop the message and initiate a new attestation protocol with that sender. (It may then tell the sender to initiate a new run of the attestation protocol, but we do not include that activity in this work.)

The messages between dispatchers and attesters in the case of successful attestations are summarized in Figure 6.

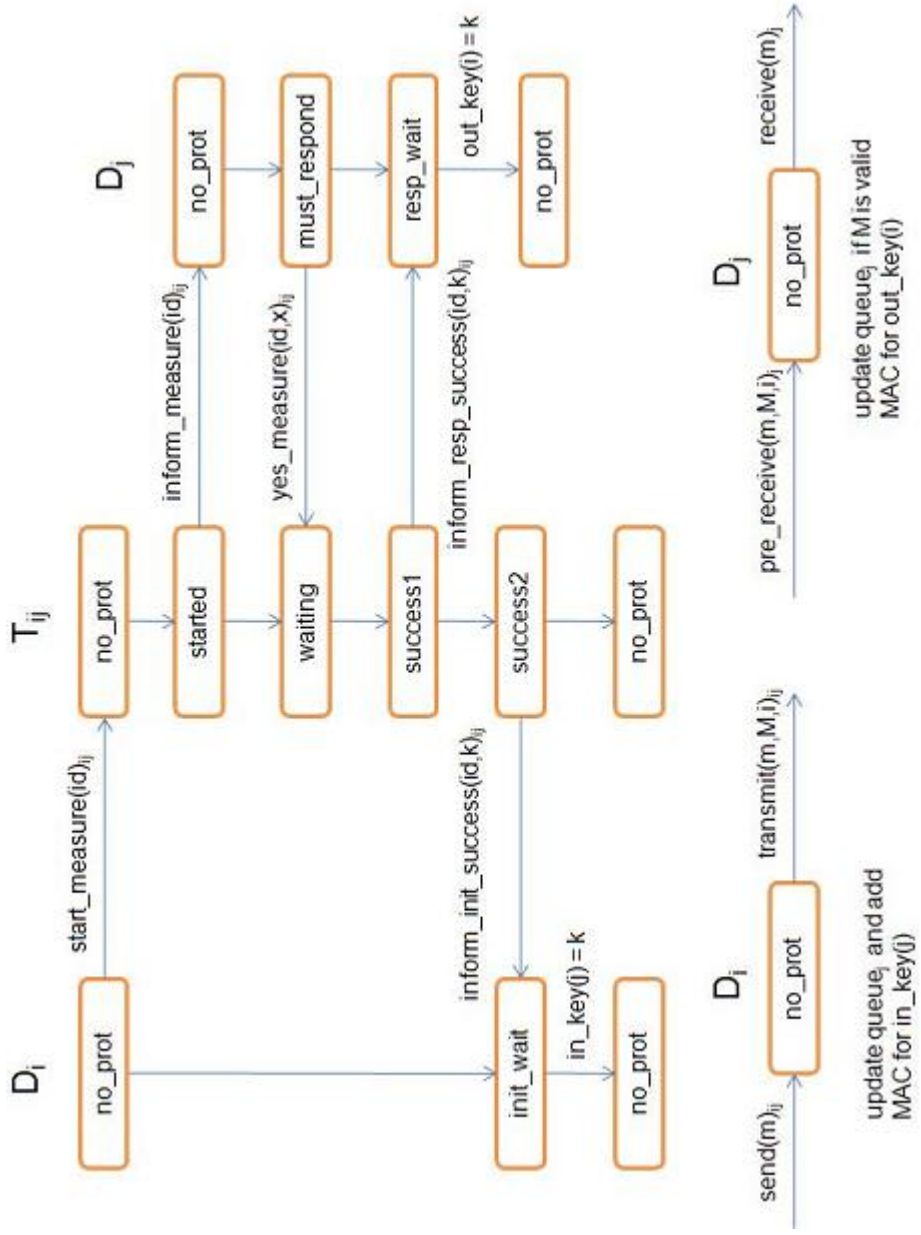


Figure 6: Dispatcher Protocol

Dispatchers are assumed to crash and recover with their clients. When a dispatcher crashes, it loses all state: all messages in all queues, and all information about prior attestation protocols.[5]

We formalize the dispatcher in Figures 7, 8, and 9. We note that dispatchers have two events not previously described:

- The output event $\mathit{transmit}(m)_{i,j}$ represents the transmission of the message by the dispatcher, meaning that the message is transmitted from the dispatcher to the channel.
- The input event $\mathit{pre\_receive}(m)_j$ represents the reception of the message by the dispatcher, meaning that the message is transmitted from the relevant anonymizer to the dispatcher.

Both of these additions are motivated by the same underlying reason: participants no longer directly communicate with channels or anonymizers. Thus, message transmission and reception each need an additional event to model the additional step imposed by the dispatcher. These additional events could 'take place' between the dispatcher and the network (channel and anonymizer) or between the dispatcher and the participant. We choose, in this work, to use the first of these two options for the following reason. Our theorem, below, states (roughly) that our two network models will 'look' the same to all algorithms. Rephrased, every trace possible for a given algorithm in the all-honest setting is a possible trace in the Byzantine setting and vice-versa. (This requires, of course, that participants be protected by dispatchers in the Byzantine setting.) For this correspondence result to be possible, participants must 'see' the same communications interface in both models. Thus, we will require that participants use $\mathit{send}(m)_{i,j}$ and $\mathit{receive}(m)_j$ events in both of our network models and that the new events of the Byzantine model be hidden from participants.

This requires, of course, that we redefine channels and anonymizer in terms of our new events. We do so by way of the $C^{re}_{i,j}$ and A^{re}_j I/O automata in Appendix B. We note, however, that the only difference between these automata and those of $C^{id}_{m,i}j$ and A^{id}_j is that the events $\mathit{send}(m)_{i,j}$ and $\mathit{receive}(m)_j$ have been replaced by $\mathit{transmit}(m)_{i,j}$ and $\mathit{pre\_receive}(m)_j$.

3.4 Our 'Real' Network

Before we provide our 'real' network, we must consider one more technical issue. As said in Section 3.1, Byzantine participants are allowed to be any I/O automata with the same external interface as honest participants. This implies that Byzantine participants send messages with the event $\mathit{send}(m)_{i,j}$ and receive messages with the event $\mathit{receive}(m)_i$. Now that honest participants have dispatchers, however, channels and anonymizers are now defined in terms of $\mathit{transmit}(m)_{i,j}$ and $\mathit{pre\_receive}(m)_i$. Therefore, there is a 'disconnect' between

[5] We implicitly assume that dispatchers to not execute any transitions between stopping and starting. Technically, this violates a central assumption of the I/O automata model—that of being input-enabled—but it is also in keeping with the I/O automata treatment of participant crashes.

Signature:

Input:	Output:
$send(m)_{i,j}$	$receive(m)_i$
$pre\_receive(m)_i$	$transmit(m)_{i,j}$
$inform\_measure(id)_{i,j}$	$start\_measure(id)_{i,j}$
$inform\_init\_success(id,k)_{i,j}$	$yes\_measure(id;x)_{i,j}$
$inform\_resp\_success(id,k)_{i,j}$	
$inform\_init\_fail(id)_{i,j}$	
$inform\_resp\_fail(id)_{i,j}$	Internal:
$stop_j$	$MAC\_drop(m)_i$
$recover_j$	$error\_drop(m)_{i,j}$

where $id \in \mathcal{N}$, k is a cryptographic key, and x is either a cryptographic key or $\perp$.

State Variables:

- *stopped*, a boolean value, initially set to false
- $queue_{in}$, a FIFO queue of messages, initially empty
- For every participant P_j: $queue_j$, a FIFO queue of messages, initially empty
- *in_key* and *out_key*, mappings from participant-identifiers to either cryptographic keys or $\perp$
- *in_error*, a set of participant-identifiers, initially empty
- For every participant P_j, $state_j$, a mapping from *id*-values to one of {`no_prot`, `init_wait`, `must_respond`, `resp_wait`}, initially set to `no_prot` for all values of *id*.

Figure 7: The dispatcher I/O Automaton D_i, Part I

Transitions:

$send(m)_{i,j}$:
Effect:
Add m to end of $queue_j$

$receive(m)_i$:
Precondition:
m' is first on $queue_{in}$,
$m' = (m, M, x)$,
$MAC\_ver\,(m; M; in\_key(x))$, and
stopped is false.
Effect:
remove first element of $queue_{in}$

$pre\_receive(m)_i$:
Effect:
Add m to end of $queue_{in}$

$transmit(m)_{i,j}$:
Precondition:
$m = (m', M, j)$
m' is first on $queue_i$,
$out\_key(j) \neq \perp$,
$M = MAC\_sign\,(m'; out\_key(j))$,
stopped is false
Effect:
remove first element of $queue_j$

$inform\_measure(id)_{j,i}$
Effect:
If $state_j(id) =$ `no_prot`, set $state_j(id)$ to `must_respond`

$start\_measure(id)_{j,i}$:
Precondition:
id is a randomly chosen identifier
$state_j(id) =$ `no_prot`
Effect:
set $state_j(id) =$ `init_wait`.

$inform\_init\_success(id, k)_{i,j}$
Effect:
if $state_j(id) =$ `init_wait`, then
- set $state_j(id) =$ `no_prot`
- if $j \in in\_error$, remove j from in_error
- set $in\_key(i) = k$.

$yes\_measure(id; x)_{j,i}$
Precondition:
$state_j(id) =$ `must_respond`
$x = in\_key(j)$,
Effect:
set $state_j(id) =$ `resp_wait`

Figure 8: The dispatcher I/O Automaton D_i, Part II

Transitions, II:

inform_resp_success$(id, k)_{i,j}$
Effect:
if $state_j(id) =$ `resp_wait` then
•set $state_j(id) =$ `no_prot`
•set $out\_key(i) = k$.

inform_init_fail$(id)_{i,j}$
Effect:
if $state_j(id) =$ `init_wait`, then set $state_j(id) =$ `no_prot`
Set $out\_key(j) = \perp$

stop$_j$
Effect:
If $j = i$, then
•set *stopped* to true,
•empty all queues,
•set *in_error* to the empty set,
•set $in\_key(i)$ and $out\_key(i)$ to $\perp$ for all i, and
•set $state_j(id) =$ `no_prot` for all id and j.

recover$_j$
Effect:
set *stopped* to false

inform_resp_error$(id)_{j,i}$
Effect:
None

error_drop$(m)_{i,j}$:
Precondition:
m is first on $queue_j$, and
$j \in in\_error$
Effect:
remove first element of $queue_j$

inform_resp_fail$(id)_{j,i}$
Effect:
if $state(id) =$ `resp_wait`, set $state_j(id) =$ `no_prot`.
Set $in\_key(j) = \perp$

MAC_drop$(m)_i$:
Precondition:
m is first on $queue_{in}$, and
either $m \neq (m', M, x)$, or $m = (m', M, x)$ and not $MAC\_ver\,(m'; M; in\_key(x))$
Effect:
remove first element of $queue_{in}$

inform_init_error$(id)_{i,j}$
Effect:
add j to *in_error*

Tasks:
Arbitrary.

Figure 9: The dispatcher I/O Automaton D_i, Part III

Byzantine participants and the new network model. We can resolve this issue in multiple ways. One could, for example, add 'dummy' dispatchers to the model that simply proxy for Byzantine participants and transform $send(m)_{i,j}$ into $transmit(m)_{i,j}$ and $pre\_receive(m)_i$ into $receive(m)_i$. This adds an unnecessary level of complexity, however, and so we avail ourselves of a simpler solution: we will require that Byzantine participants use events $transmit(m)_{i,j}$ and $pre\_receive(m)_i$ instead of $send(m)_{i,j}$ and $receive(m)_i$. Also, Byzantine participants can participant in attestation protocols directly, and so have the following events in their interfaces:

- $start\_measure(id)_{i,j}$
- $inform\_measure(id)_{i,j}$
- $yes\_measure(id; x)_{i,j}$
- $inform\_init\_success(id, k)_{i,j}$
- $inform\_resp\_success(id, k)_{i,j}$
- $inform\_init\_error(id)_{i,j}$
- $inform\_resp\_error(id)_{i,j}$
- $resolve\_protocol(id, r, x)_{i,j}$

Lastly, we need to resolve one technical issue. The ideal network model is defined over a message alphabet M. We assume such messages to remain valid in the ideal model, but must also allow authenticated messages such as those produced by dispatchers. Thus, our real network model will be defined over a message alphabet M^{re} defined to include both M and (m, M, x) triples where $m \in M$, M is a message authentication code, and x is a participant identifier.

Given this, we can define our 'real' network model:

Definition 2 (Real network) *Given a graph $G = (V, E)$ where $Byz \subseteq V$, let R_G be the I/O automaton created by composing the following components:*

- *For each node $i \in V \setminus Byz$, the participant automaton for honest participants.*
- *For each $i \in Byz$, some other automaton.*
- *For each honest participant P_i, a dispatcher D_i.*
- *An anonymizer A_i^{re} for every P_i, and*
- *A channel process $C_{i,j}^{re}$ for every edge (i, j) in E.*

We will denote the automaton for node i as P_i whether it is honest or Byzantine.

The relationship between the real and ideal network models is summarized in Figure 10.

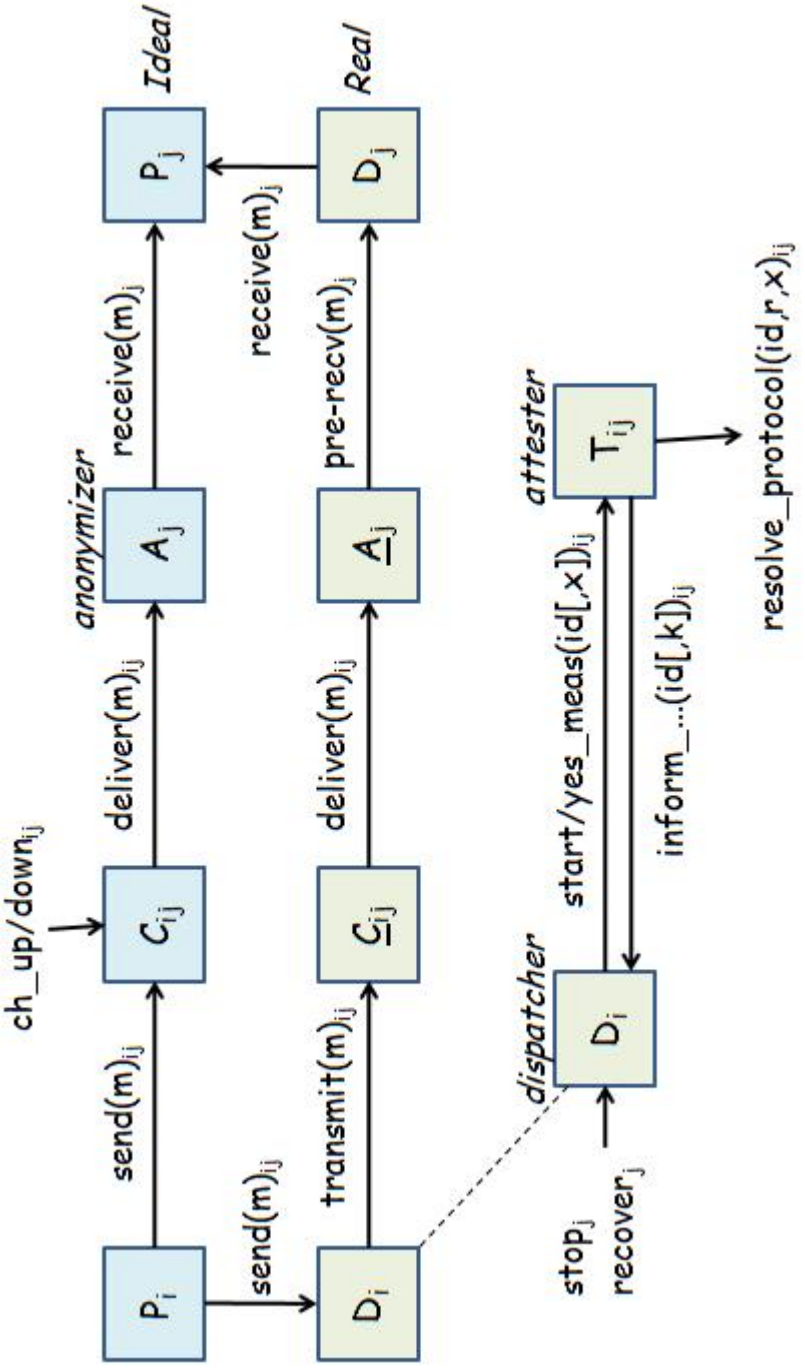


Figure 10: Real and Ideal Models

4 The Trace Equivalence Theorem

In this section, we state a general theorem: if an algorithm is resilient against crash failures in the network model of Definition 1, and attestation protocols are possible, then it is resilient against Byzantine failures in the network model of Definition 2. This result could be expressed as:

Attestation + crash-resilience = Byzantine-resilience

In particular, we show that two machines have the same set of traces:

- One machine for the real network model in which some participants are Byzantine but the honest participants are protected by dispatchers, and
- One machine for the ideal network model in which all channels from Byzantine participants to non-Byzantine participants are permanently down.

Theorem 3 *Let $G = (V, E)$ be a graph and $Byz \subseteq V$ be the set of Byzantine participants.*

- *Let the I/O automaton $\mathcal{A}_I$ be the automaton I_G after executing the sequence of events*
  - *$channel\_down_{i_1,j_1}$, $channel\_down_{i_2,j_2}$ ... $channel\_down_{i_m,j_m}$ for every $(i_x, j_x) \in E$*

  (and only those events).
- *Let the I/O automaton $\mathcal{A}_R$ be the automaton R_G after executing the sequence of events $channel\_down_{i_1,j_1}$, $channel\_down_{i_2,j_2}$ ... $channel\_down_{i_m,j_m}$ for every $(i_x, j_x) \in E$.*

Furthermore, we hide in $\mathcal{A}_I$ and $\mathcal{A}_R$ all events except those visible to the honest participants: those of the form

- *$send(m)_{i,j}$,*
- *$receive(m)_i$,*
- *$stop_i$, and*
- *$recover_i$*

where $P_i \notin Byz$.

If there are no events of the form $channel\_up_{i,j}$ for any $P_i \in Byz$, then

$$traces(\mathcal{A}_R) = traces(\mathcal{A}_I)$$

We prove this theorem in Appendix A. Before we do, however, we derive a quick corollary and make some general comments:

Corollary 4 *If an algorithm is implemented into participants P_1, P_2,... P_n. Then any trace-based property $\mathcal{P}$ that it satisfies in $\mathcal{A}_I$ it also satisfies in $\mathcal{A}_R$.*

Thus, Byzantine-resilience is no harder than resilience against crash failures, channel failures and message losses—if the attestation module of Figures 4 and 5 can be implemented.

We now make some general comments about this result.

Our network models are very low-level, allowing a wide variety of failures. We note, however, that this low level could be undesirable to algorithm designers who might prefer stronger guarantees such as reliable channels. We believe, however, that such algorithm designers can still make use of our result by implementing, in our network models, protocols to, for example, provide reliable message delivery over unreliable channels. Similarly, algorithm designers that wish for failure detection are free to implement failure-detection algorithms in our network model. (They may need to add further assumptions, such as clocks and timeouts.)

We note that the 'real' network model is inherently randomized. The attestation module uses randomness to generate cryptographic keys, the dispatchers use randomness to generate session identifiers, and the MAC algorithms might be randomized. Thus, our result only applies in settings where randomness is available.

This theorem has several interesting variants. One variant, for example, could reinstate the assumption of authenticated channels by removing the anonymizers. This would have the nice side-effect of simplifying both the real and ideal models, and would make the proof in Appendix A simpler. It would also make the theorem less general, however, and so we leave the anonymizers in our result.

Another simplification might be to remove any or all of: crash failures, channel failures, or message drops. However, we note that not all of these combinations are possible. Most importantly, the core of our theorem maps attestation failures to channel failures, and so any variation will need to contain channel failures. Also, we note that currently if the network models of the variation contain crash failures, they must contain message loss as well. To see why, note that we assume dispatchers must crash when participants do. We also assume that dispatchers lose all state when they crash. If a dispatcher had any outgoing messages in a queue when it crashed, therefore, those messages will be lost. But said messages had already been sent by the participant, and so they were in the 'moral equivalent' of a channel. Therefore, those same messages must be 'dropped' in the ideal network, and the ideal network needs to model message loss.

The above assumes, of course, that one keeps the assumptions and automata of this work. One might be able to retain crash-failures while also eliminating message-loss by assuming the dispatchers don't crash, or don't lose state when they do.

One interesting variant is to show a mapping not between to machines on the same network graph G, but between the real network on G and the ideal network on a sub-graph G' consisting only of those edges which start at a non-Byzantine node. This variant might prove challenging to prove, however: without knowing how participants are affected by the graph (or put another

way, how the distributed algorithm in question uses the network topology in its operation) such a variant might actually be false.

One last interesting variant might be to show a similar result for unordered channels: channels which might re-order messages. We believe such a result would be simple to show, perhaps best approached by changing the queues of our channel automata to sets.

5 An Erlang implementation

In this section, we describe how we instantiated our results in the programming language Erlang [1]. In many ways, Erlang seems nearly ideal for this instantiation. It is designed for distributed computing, and message-passing is a core primitive of the language. Furthermore, the semantics of Erlang (as formalized by Svensson and Fredlund [23]) match the network models of Sections 2 and 3 quite closely:

- Erlang's built-in message-passing mechanisms guarantee in-order message-delivery in all circumstances. Message-passing is also guaranteed to be reliable (that is, messages are never dropped) unless the receiving process has died.
- Furthermore, Erlang processes have the ability to `monitor` other processes, which means that the monitoring process is informed when the monitored process terminates.

Thus, Erlang is an almost-ideal language in which to implement distributed algorithms appropriate for our 'ideal' network model. However, this does not make the application of our theorem trivial. To apply the theorem, the dispatcher must implemented in Erlang and algorithms must be adapted so as to use it. We describe these in reverse order. In Section 5.1, we describe our *parse transform*: an automated technique for adapting Erlang code so as to use a dispatcher. In Section 5.2, we describe our implementation of the dispatcher in Erlang.

5.1 Our Parse Transform

Erlang provides simple, high-level primitives for message-passing and failure-detection as part of the language's kernel. This is a mixed blessing. On the one hand, the use of these primitives (and the use of Erlang in general) will make a given implementation very clear and easy to understand. On the other hand, it 'hard-codes' the implementation into the the direct-communication model of Section 2. For example, consider the Erlang expression for message-transmission:

```
E_PID ! E_V
```

With this expression, the executing process evaluates the expression `E_PID` into the process-identifier `PID`, evaluates the expression `E_V` into a value `V`, and sends the value `V` to the process-identifier `PID`. The process `PID` can be any Erlang process on any node (or any C process, for that matter). This unbounded flexibility with regard to communication is appropriate in the standard network model, where every participant can send messages directly to every other participant. In the dispatcher-model, however, all communication must go to the recipient's dispatcher. Erlang provides no way for the dispatcher to automatically 'intercept' messages sent to its client, however, meaning that every send-expression in the code must be rewritten.

As another example, consider the Erlang expression for monitoring:

```
erlang:monitor(process, PID)
```

This command is one of Erlang's built-in functions (BIFs) and, like many of them, appears to the programmer to be in the virtual `erlang` modude. With this command, the executing process invokes Erlang's monitoring mechanism on the process identified by the process-identifier `PID`. (The first argument, `process`, is simply a required keyword.) However, Erlang's built-in monitoring facilities do not use attestation protocols and will not recognize Byzantine failures. It is for this reason that our model assigns failure-detection to the dispatcher, who can make Byzantine failures 'look' like crash-failures to the algorithm. For this to work, however, the Erlang implementation must send `monitor` requests to the dispatcher and not the underlying Erlang mechanism.

Both of these expressions are built into the Erlang language's kernel, making it very hard to simply redefine their semantics. Furthermore, 23 other BIFs can fail if the some (remote) node or process has crashed. For Byzantine processes to 'look' crashed, therefore, these 23 BIFs must fail in exactly the same way when invoked on Byzantine nodes. But as with the expressions above, they are built into the language's kernel and cannot be simply redefined. Therefore, all of these BIFs and expressions must be *rewritten* before compilation into dispatcher-using code.

The easiest and most direct way to do this is to use a mechanism of the Erlang compiler. In particular, the compiler can be configured to apply a user-supplied *parse transformation*: a function from Erlang parse-trees to Erlang parse-trees. This function is applied to the target source-code after parsing but before error-checking. It is the resulting code, not the original code, that is ultimately compiled. Therefore, we can use this mechanism to capture and rewrite the above expressions (and the 23 other BIFs in question). As a result, we can morph Erlang code into a dispatcher-based version in a completely automated way and at the semantic level.

Our parse transform consists of a number of relatively local rewrite rules, shown in Tables 1, 2, and 3. The first table contains the rewrite rules for message transmission and reception, the second table contains rules for failure-detection, and the third contains all other BIFs rewritten by the parse transform.

Our parse transform is conservative in that it attempts to rewrite as little as possible. For the most part, it maps BIFs to functions by the same name

`erlang:send(Expr1, Expr2)`	`dispatcher:send(Expr1, Expr2)`
`erlang:send(Expr1, Expr2, OptList)`	`dispatcher:send(Expr1, Expr2, OptList)`
`erlang:send_after(Time, Expr1, Expr2)`	`dispatcher:send_after(Time, Expr1, Expr2)`
`erlang:send_nosuspend(Expr1, Expr2)`	`dispatcher:send_nosuspend(Expr1, Expr2)`
`erlang:send_nosuspend(Ex1, Ex2, Opts)`	`dispatcher:send_nosuspend(Ex1, Ex2, Opts)`
`Expr1 ! Expr2`	`dispatcher:send(Expr1, Expr2)`

`receive`
`  Pattern1 [when GuardSeq1] ->`
`  Body1;`
`  ...;`
`  PatternN [when GuardSeqN] ->`
`  BodyN`
`end`
	`GENSYM = dispatcher:get_nonce(),`
	`receive`
	`  {from_dispatcher, GENSYM, Pattern1}`
	`  [when GuardSeq1] ->`
	`  Body1;`
	`  ...;`
	`  {from_dispatcher, GENSYM, PatternN}`
	`  [when GuardSeqN] ->`
	`  BodyN`
	`end`

`receive`
`  Pattern1 [when GuardSeq1] ->`
`  Body1;`
`  ...;`
`  PatternN [when GuardSeqN] ->`
`  BodyN`
`after`
`  ExprT ->`
`  BodyT`
`end`
	`GENSYM = dispatcher:get_nonce(),`
	`receive`
	`  {from_dispatcher, GENSYM, Pattern1}`
	`  [when GuardSeq1] ->`
	`  Body1;`
	`  ...;`
	`  {from_dispatcher, GENSYM, PatternN}`
	`  [when GuardSeqN] ->`
	`  BodyN`
	`after`
	`  ExprT ->`
	`  BodyT`
	`end`

Table 1: Our parse transform: send and receive

`monitor_node(Node, Bool)` `erlang:monitor_node(Node, Bool)`	`dispatcher:monitor_node(Node, Bool)`
`erlang:monitor(Type, Item)`	`dispatcher:monitor(Type, Item)`
`erlang:monitor_node(Node, Flag, Opts)`	`dispatcher:monitor_node(Node, Flag, Opts)`
`erlang:demonitor(MonitorRef)`	`dispatcher:demonitor(MonitorRef)`
`erlang:demonitor(MonitorRef, OptList)`	`dispatcher:demonitor(MonitorRef, OptList)`
`link(PID)`	`dispatcher:link(PID)`
`unlink(PID)`	`dispatcher:unlink(PID)`
`spawn_link(Node, Fun)`	`dispatcher:spawn_link(Node, Fun)`
`spawn_link(Node, Modl, Func, Args)`	`dispatcher:spawn_link(Node,Modl,Func,Args)`
`spawn_opt(Node, Fun, [Option])`	`dispatcher:spawn_opt(Node, Fun, [Option])`
`spawn_opt(Node, Modl, Func, Args, [Opt])`	`dispatcher:spawn_opt(Node, ..., [Opt])`

Table 2: Our parse transform: failure-detection

`check_process_code(Pid, Module)`	`dispatcher:check_process_code(Pid, Module)`
`garbage_collect(Pid)`	`dispatcher:garbage_collect(Pid)`
`erlang:suspend_process(SPid, OptList)`	`dispatcher:suspend_process(SPid, OptList)`
`erlang:suspend_process(Suspendee)`	`dispatcher:suspend_process(Suspendee)`
`erlang:resume_process(Suspendee)`	`dispatcher:resume_process(Suspendee)`
`spawn(Node, Fun)`	`dispatcher:spawn(Node, Fun)`
`spawn(Node, Modl, Func, Args)`	`dispatcher:spawn(Node, Modl, Func, Args)`

Table 3: Our parse transform: miscellaneous

in the `dispatcher` module. By and large, a `dispatcher` function will produce the same effect as the corresponding BIF except that Byzantine nodes are made to 'look' as if they have crashed. For example, the function `spawn(Node, Fun)` should return a "useless PID" if the node `Node` is down. The function `dispatcher:spawn(Node, Fun)` should do the same if `Node` is down or Byzantine. (We will discuss these functions further in Section 5.2.)

Unfortunately, the last three rewrite rules in Table 1 must be more invasive. The first of these simply rewrites the Erlang expression for message-transmission into a call to the dispatcher. The original expression is not a function call, however, and so cannot be simply mapped to a same-named function. The last two rewrite-rules of that table, however, are much more invasive and deserve further discussion.

These two rules concern the two closely-related forms of Erlang's message-reception expression. To explain the rewrite rule, we first need to explain the semantics of these two expressions. Internally, every Erlang process has a *message queue* which holds all messages received by that process. The queue itself is managed by the underlying run-time system, however, and does not affect the process itself until the process executes a `receive` expression. This expression can contain a number of branches, each of which is guarded by a *pattern* and optional boolean tests called the *guard sequence*. When a `receive` expression is executed, the patterns of the expression are matched, in order, against the first message in the queue. If any of the patterns match the first message (and the associated guard-sequence returns `true`), then the message is removed from the queue and that pattern's branch is executed under the variable-binding produced by the pattern-matching unification.

More generally, the first message in the queue that matches any pattern of the `receive` expression (and satisfies the associated guard sequence) is removed from the queue and unified with the first such pattern. Execution then proceeds along that branch of the `receive` expression under the resulting variable-binding. All other messages in the queue, including those that preceded the matching message, are left in the queue—unchanged and in their original order.

If no message in the queue satisfies any of the pattern/guard-sequence pairs, then the default behavior is to block and wait for some message to enter its queue which does satisfy a pattern/guard-sequence pair. If the `receive` expression contains a *timeout* on the other hand, the process will only block for the stated amount of time before executing the timeout-branch. (The last row of Table 1

shows a `receive` expression with a timeout branch.) If the timeout branch is guarded by a time-out value of 0, the `receive` expression will execute that branch immediately upon exhausting the message queue.

Because of this non-trivial semantics for the `receive` expression, our parse transform strives to preserve the original expression as much as possible. However, the parse transform must also prevent the resulting code from ever successfully pattern-matching against a message not approved by the dispatcher. However, the semantics of Erlang do not make this straightforward. Because Erlang processes can send messages to any other Erlang process, an Erlang process can *receive* messages from any other process. Furthermore, messages are unauthenticated; the Erlang language does not provide any reliable mechanism for determining the sender of a given message.

To preserve the semantics of the `receive` expression while simultaneously limiting it to dispatcher-approved messages, our parse transform rewrites the patterns of the expression. In general, the general idea is that all messages from the dispatcher should be of a certain form, and that this form is limited to such messages. The form we use is:

```
{from_dispatcher, NONCE, MSG}
```

This Erlang triple contains a fixed type-identifying constant `from_dispatcher` as the first component,[6] a nonce (large random number) as the second component, and an arbitrary Erlang value as the third component. The third component is the message as received by the dispatcher. A message will match this pattern if and only if

- The original message, as received by the dispatcher, matches the original pattern, and
- The nonce authenticates the message as having come from the dispatcher.

The security of this new pattern comes from the second component. The nonce found there is matched against a secret value shared between the dispatcher and the participant. If it matches, it authenticates the messages as coming from the dispatcher. We acknowledge that this authentication mechanism is quite weak, but we are constrained by the Erlang language itself. Because guard-sequences must not have side-effects, the Erlang specification limits those functions which can be in guard-sequences to a small, predefined set. Therefore, we cannot use more complicated (and secure) cryptographic mechanisms in guard-sequences, making it difficult for our parse transform to introduce more secure authentication mechanisms without changing the semantics of the original `receive` expression.

5.2 The Dispatcher

In this section, we discuss the actual implementation of the dispatcher in Erlang. When the code is modified with the parse transform during compilation,

[6] A common Erlang idiom.

calls to message sending and receiving commands, as well as other node maintenance commands in the Erlang standard library, become calls to functions in our dispatchers, which is an Erlang implementation of our network model. The dispatcher has several services which implement this functionality - a main control thread, facilities for key and nonce management and storage, and auxiliary functions to simulate the functionality of the main Erlang library.

In our model, the dispatcher mediates communication between different nodes. In Erlang, this is implemented as a *named process* which, if it is not already running, is launched at each node upon receipt or sending of a message. This thread then maintains state about which node keys and message nonces it knows, and loops to handle service requests that come from the attestation manager and normal application threads both within a node, and outside of it.

There are several services that the dispatcher performs. The first is handling of messages destined for local processes. If the remote sender is known to be good (meaning that there is a key shared between the nodes), a new nonce is generated for the local recipient, and then the message is queued to be sent. This is done in order to ensure that message ordering is maintained between the endpoints. Messages from local threads automatically have nonces generated and are then forwarded. If the sender is known to be bad, the message is dropped, and the recipient will act as if the remote node had crashed.

The second service is handling messages to be delivered to the remote node. Local sends are intercepted and added to the local queue for processing. When the dispatcher comes to a message to be sent to a remote node, it checks to see if a key is available for that node. If not, then an attestation is performed, and either a new key is generated in the event of an attestation success, or the remote node is added to the local list of bad nodes. In the former case, the message is encrypted and sent to the dispatcher on the remote node, to be handled there. In the latter case, the message is dropped locally, and the sender is treated as if the recipient had crashed.

5.3 Interface to the Trusted Platform Module

The standard mechanism for using the Trusted Platform Module (TPM) today is to go through one of the libraries implementing the TCG Software Stack (TSS) interface[25].[7] In order to access the TPM's functionality in our distributed Erlang code, we created a C program that acts as an Erlang node (called `ei_tpm`), and an Erlang TPM interface that connects to the C node and provides a convenient set of functions usable by Erlang programs, called the `tpm_if` module. The C program utilizes the TSS internally. This had the additional advantage of allowing us to abstract away much of the complexity of the TSS for our Erlang use.

Our goal in creating this interface was to give Erlang programmers an easy-to-use interface to the TPM functions essential for remote attestation. (Future versions may include additional functionality providing access to other TPM

[7] We used the Linux implementation, TrouSerS.

features.)

The Erlang interface provides access to the following functions essential to remote attestation:

init : Launches the Erlang half of the TPM interface, which will detect the running C node. Must be run before any other commands in the tpm_if module, and after the C node (`ei_tpm`) has been launched.

take_ownership : Allows the user to take ownership of the TPM. The TPM must be activated and cleared. Only the TPM owner can perform certain commands, such as creating identity keys (see below).

create_ek : Allows the user to create an Endorsement Key (EK). The EK is the uniquely identifying key of the TPM, and is used in cryptographic protocols to prove that other keys, particularly AIKs, are associated with a genuine TPM. Some TPMs (currently, those manufactured by Infineon) come with pre-generated EKs, but this command can be used on initially blank TPMs or those whose pre-generated EK has been erased.

create_identity : Allows the user to create a new Attestation Identity Key (AIK). AIKs are signature keys held by the creating TPM, and can be used to prove that a quote (discussed later in this section) was produced by a genuine TPM.

get_pubkey : Retrieves the public half of a TPM key.

store_key_blob : Takes a TPM key blob provided by another command, such as `create_identity`, and writes it to disk. We refer to the key data structures returned by the the TPM as "blobs" because they are not meant to be interpreted by outside software. The secret key data in a key blob is encrypted.

read_key_blob : Reads a TPM key blob from disk, for use in other commands.

get_quote : Produces a record of the current values of selected Platform Configuration Registers (PCRs) in the TPM, signed by an AIK. This is called a TPM quote. The PCR values being reported are usually measurements taken during system boot, if a trusted boot loader is used, or by certain secure software if a dynamic measurement command such as Intel's SINIT or AMD's SKINIT is called. These measurements are normally hashes of the loaded code. Quotes allow remote parties to gain verifiable, recent information about the boot state of a machine.

`verify_quote` : Used to verify that a quote produced by a remote party was signed by the expected AIK and used the expected nonce. The caller is responsible for determining whether the included PCR values are acceptable or not, since there is no universal definition of "good" PCR values.[8]

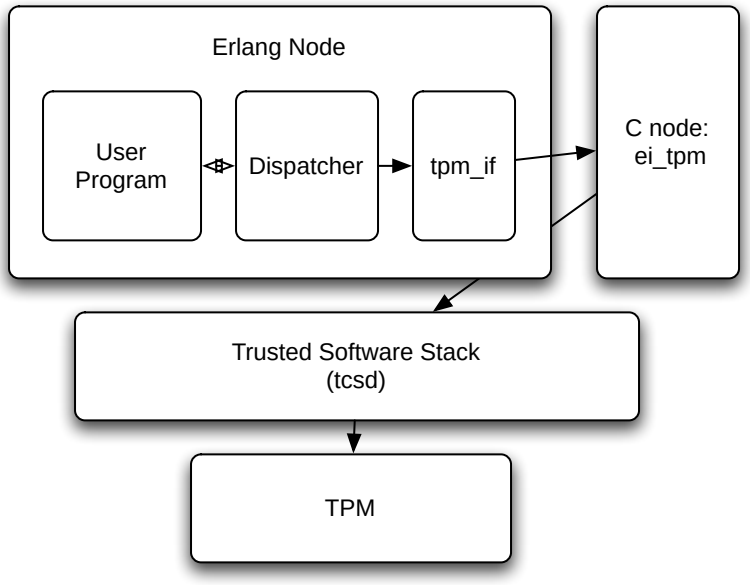
[8] PCR values usually include hashes of the booted image of the BIOS, boot loader, hypervisor, and other system software.

When designing our interface, we sought to reduce one of the user interface difficulties present in the TSS: ambiguous types and ambiguous relationships between functions. Many functions in the TSS use byte arrays as input or output, without any clear instructions for creation, interpretation, or passing of these arrays. For our Erlang interface, we tried to provide intuitive and self-explanatory function and argument names. We also strove to make the relationships between data types used in different functions clear, so the user did not need to understand the composition of our data structures in order to make use of our module.

5.3.1 Use of the TPM Interface by Dispatchers

In our demonstration, the TPM interface is used by the dispatchers. We use TPM quotes as a simple form of Byzantine failure detector. Nodes that can produce a fresh TPM quote describing an expected software configuration are considered safe; all other nodes are treated as Byzantine.

The user program's external communications are intercepted by the dispatcher, which may then perform a remote attestation or have a remote party request a local attestation. When an attestation requires TPM functionality, or the interpretation of TPM output, the `tpm_if` module is used. The `tpm_if:init` command creates a new erlang process running in the same node, as pictured below. This `tpm_if` process communicates with the separate C node, running the `ei_tpm` program, whenever the dispatcher requests a TPM-related function. `ei_tpm`, in turn, calls the Trusted Software Stack library (in this case, using the `tcsd` daemon) which actually communicates with the TPM.



The dispatchers in our demonstration use the *quote* functionality provided

by our interface in order to determine whether another node is Byzantine or not. We assume that all dispatchers can recognize the identities of good TPMs. In our demonstration, we provide a list of known public AIKs to each dispatcher; in a real-world scenario, a certificate authority would provide Attestation Identity Certificates, or AICs, testifying to the legitimacy of a given AIK, and dispatchers would only require a list of trusted certificate authorities.

An appraising dispatcher (appraiser) requests a quote from the unknown node, providing a set of desired registers and a fresh random number (nonce) to ensure that the quote is recent. Upon receipt of the quote, the appraiser verifies the quote against the expected public AIK for that node. A signature failure indicates that the quote was not produced by the expected node, or that the node did not have access to the TPM; in either case, the communicating party can be assumed to be byzantine, and no session key is provided. A nonce failure indicates that the quote is not fresh, and a replay attack may be occurring; again, no session key is provided in this case. The appraiser then evaluates the composite PCR values provided, to determine whether the software executing on the machine is acceptable. If it is, a session key is provided; if not, the communicating party is assumed to be byzantine. [9]

The dispatcher who receives an attestation request is referred to as the attester. Upon receiving a request, the attester calls the `get_quote` function using the nonce and PCR request provided by the appraiser, as well as an AIK to sign with. The resulting quote data and signature are returned to the appraiser for evaluation.

The exact mechanisms for transferring the TPM quote request and data from one dispatcher to another depend on the cryptographic protocol in use, as described in section 5.4.

5.4 EVA: A Simple Attestation Protocol

For this demonstration, we created an attestation protocol called EVA, inspired by the more complex CAVES protocol[19]. Only two parties participate in EVA: an Attester, who initiates the attestation protocol, receives attestation requests and interacts with the TPM, and a Verifier, who determines whether to issue a shared key to the Attester. The E is for our external Expert, who has previously provided trusted information about known TPM keys and their association with expected communication partners.

We assume for this protocol that the parties are known to each other, and share any public keys necessary for confirming the identity of other participants. In particular, the Verifier and Attester both know a public key for the other, and the Verifier has a pre-established association between the Attester's identity, their public key, and their TPM identity key. At the end of the protocol, the Verifier and Attester should share a session key.

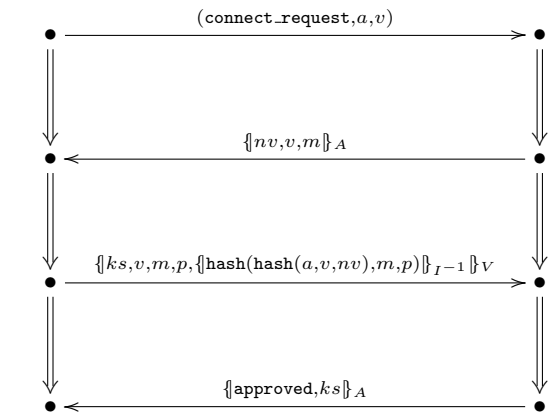
Our security goals for this protocol were that the Verifier should know that the PCR values included in the quote hash were reported by a legitimate TPM

[9] For the demonstration prototype, any valid PCR composite is considered acceptable.

(owner of I^{-1}), that those PCR values were recent, and that the shared session key ks is known only to legitimate participants. We check these goals by verifying the following properties:

1. The Attester and Verifier should agree on the current PCR values (p), the PCR mask[10] (m), the nonce (nv), and the session key ks. They should also agree on the principal identifiers a and v.
2. The Verifier should freshly generate nv. nv should be cryptographically linked to p.
3. The Attester should freshly generate ks. ks should not be retrievable by an adversary without the cooperation of the Attester or Verifier.

For purposes of our analysis, the Verifier assumes that his own private key and the TPM's private key remain uncompromised. The Attester assumes that the his own key and the Verifier's private key remain uncompromised.



This protocol uses both the names for participants and unique keys for those participants. In all cases, the lower-case letter indicates the name (v for the verifier) and the upper-case letter indicates that participant's public-private key pair (V for the verifier's public key, V^{-1} for its private key.)

We have verified this protocol using the tool CPSA[19] and demonstrated that all of our desired security properties have been met.

5.4.1 EVA implemented

In our demonstration, both the Attester and Verifier role are the responsibility of the dispatcher. When a message is received from an unrecognized party, the

[10] A set of requested PCR indices is called a mask.

dispatcher launches a process which will run the Verifier role of the protocol. After receiving the third message, the Verifier will evaluate the PCR contents contained in the hash to determine whether the reporting machine is in an acceptable state. If so, the Verifier will store the enclosed session key ks for future use and confirm its use of ks by responding to the Attester.

All certificates and known public keys are passed to the dispatchers, both Verifier and Attester, through preloaded configuration files.

5.4.2 On the Use of Keys

For analysis purposes, we have assumed that each participating party has a well-known public key in addition to their TPM identity key. This brings up two questions: why are both keys needed, and how does the Verifier reliably associate the Attester's public key with the TPM identity key?

The primary reason we need two keys is that TPM identity keys can only be used for signing messages. It is not possible to encrypt messages using an identity key. A separate TPM storage key can be used to encrypt and decrypt messages as described here, or a non-TPM-resident key can be used. It is worth mentioning that the TPM is also not designed to be a fast cryptographic processor; any cryptographic operations that can be safely performed with non-TPM-resident keys will be significantly faster.

The association of the Attester's public key with the TPM identity can be made in several ways. Some examples include:

- If TPM keys are used for both, then the TPM_CertifyKey command can be used to create a certificate showing that they are both resident in the same TPM.
- If a software-held key is used, the association can be made and certified by the system administrator, probably during system provisioning. The certificate is then provided to the Verifier, in advance or in response to a query.
- The identity key can sign a locally-created certificate containing the software-held key information. The locally created certificate does not prove anything about the security properties of the key– many parties could have access to the secret half, for example– but does prove that someone local to the system approved the key.

We used the second option, for simplicity. When provisioning our demonstration nodes, each node was provided with a list of the expected TPM identity keys, and an associated public key to be used by the associated system's dispatcher. A mismatch between the two keys would cause an error.

5.4.3 Attestation Variants

This protocol was designed for *single-party attestation*, in which one participant is seeking a privilege that another party controls and must prove their fitness

and identity. (In our implementation, the Attester dispatcher is requesting that the Verifying dispatcher forward messages from the Attester's node to the running software.) However, there are many scenarios in which other forms of attestation are required.

For example, we assume in our implementation that the threat model is byzantine nodes sending false messages to well-behaved nodes. If the well-behaved nodes are handling sensitive information, however, we may wish to implement *mutual attestation*, in which both the sending and receiving nodes undergo attestation. One way of implementing mutual attestation would be to execute the EVA protocol twice, establishing separate keys for each direction of communication. More efficient mutual attestation protocols, in which no shared key is agreed upon until both parties have a chance to assess the other, also exist.

6 Leader Election

In this section, we instantiate our result on a particular class of algorithms known as *leader-election algorithms*. We first define these algorithms in terms of a trace property, showing that Corollary 4 applies. We then turn to a particular leadership-election algorithm and discuss how we have applied our parse transform of Section 5.1 to a particular implementation of it for demonstration purposes.

6.1 Leader-Election Algorithms

In this section, we instantiate Corollary 4 on a particular class of algorithms: *leadership election* algorithms. A leadership-election algorithm is a distributed algorithm in which the participants simply attempt to agree on which single participant will become a distinguished 'leader'. Very often, these algorithms are used by some higher-level task to choose some single point of coordination. There are many formulations of the leadership-election problem, and we use the formalism by Stoller [22], which builds on the formalism by Garcia-Molina [10]. In this formalism, participants have unique *process identifiers* and are required to be at all times in exactly one of the following states:

- `down`, meaning that the participant has crashed,
- `election`, meaning that it is participating in an election,
- `reorganization`, meaning that it believes that the election has finished but it is not yet ready to engage in the higher-level task, and
- `normal`, meaning that it believes it knows the identity of the elected leader and is engaging in the higher-level task.[11]

In the last two states, each participant P_i has its own internal understanding of the leader's identity, which we will denote as $P_i.leader$. Likewise, the state

[11] Note that unlike Garcia-Molina, we regard the higher-level task as fixed and constant. Also, we keep the higher-level task explicit while Stoller chooses to omit it for simplicity.

of P_i will be denoted $P_i.state$. Given this notation, Garcia-Molina defines a leadership-election algorithm as a distributed algorithm that satisfies the following properties:[12]

1. At any point in time, for any participants i and j, if $P_i.state$ is `normal` or `reorganization`, and if $P_j.state$ is `normal` or `reorganization`, then $P_i.leader = P_j.leader$.
2. For a given system, there exists a constant c such that if no events $stop_i$ or $recover_i$ occur for any i in a period of at least c, then by the end of that period:
   - There is some participant P_i such that $P_i.state = \texttt{normal}$ and $P_i.leader = i$, and
   - For all other participants P_j, $j \neq i$,

     $$(P_i.state = \texttt{down}) \vee (P_i.state = \texttt{normal} \wedge P_i.leader = i)$$

   We do not consider explicit time in this work, so we replace this goal with a weaker unbounded liveness goal: eventually either a $stop_i$ or $recover_i$ occurs for some i, or there exists an agreed-upon leader P_i as above.

With our modification, leadership election is a trace property. Thus, Corollary 4 applies.

Also, note that despite the word 'election', the leader does not need to be elected by majority vote. There are, in fact, no fairness properties regarding the selection of the leader at all, and algorithms are free to choose leaders in a wide variety of ways. The algorithm we consider in this paper, for example, chooses leaders based on process identifiers and not popularity. We describe this algorithm in Section 6.2.

6.2 The Bully Algorithm

In this section, we present the Bully algorithm for leadership election. This algorithm was first proposed by Garcia-Molina [10], who used explicit timeouts for failure detection. Subsequently, it has been re-cast in terms of failure-detectors by Stoller [21], and it is his description that we reproduce here.

This algorithm regards the participants' process-identifiers as *priorities* and assumes that these priorities can be sorted. To keep consistent with the literature, lower priority-values are assumed to correspond with more urgent priorities. For a given priority i, we can regard the priorities in $lesser(i)$ as those priorities of lesser value and the priorities in $greater(i)$ as those priorities of higher value. Put another way, participants with priorities in $lesser(i)$ have priority over P_i, and P_i has priority over those participants with priorities in $greater(i)$. Furthermore, all participants are assumed to know the identifiers of all other participants.

[12] Property 1 is part of 'Assertion 1' in Garcia-Molina [10] and property 2 is 'Assertion 2.' That work also contains other properties concerning the higher-level task and channel failures. We will not consider those additional properties here.

To quote Stoller [21] (mapping his notation into that of this work) the Bully algorithm works as follows:

> Each node i has a status, initially `normal`. If node i detects the failure of its leader, then it sets its status to $\texttt{election}_1$, indicating that it is in stage 1 of organizing an election. In stage 1, node i checks whether nodes in $\mathit{lesser}(i)$ are operational. If some of them are operational, node i waits, giving those higher-priority nodes a chance to become leader. If none of them are operational (*i.e.*, if node i receives $\mathit{inform\_stopped}(j)_i$ for all $j \in \mathit{lesser}(i)$), then node i sets its status to $\texttt{election}_2$, indicating that it is in stage 2 of organizing an election. In stage 2, node i prepares nodes in $\mathit{greater}(i)$ for a new leader by sending them `Halt` messages. When a node receives a `Halt` message, it sends an `Ack` message and sets its status to `Wait`, indicating that it is waiting for the outcome of an election. If a node with status `Wait` detects the failure of the node that halted it, then it starts an election itself.
>
> When a node i organizing stage 2 of an election has received an acknowledgment from or failure notification for each node in $\mathit{greater}(i)$, then it becomes the leader, setting its status to `normal` and sending a `Leader` message to each node in $\mathit{greater}(i)$ from which it received an acknowledgment. When those nodes receive `Leader` messages from node i, they accept node i as the new leader and set their status to `normal`.

The full description is given in Stoller's technical report [21]. We pause to note two things about this algorithm:

- First, Stoller's version of the algorithm makes use of crash failure detectors, with properties characterized with two Linear Temporal Logic (LTL) formulæ. To examine these formulæ, we must first define some predicates on states. Given an execution fragment

$$s_0, \pi_1, s_1, \pi_2, s_2, \ldots \pi_r, s_r, \ldots :$$

  - up_i holds when P_i is operational. That is, it holds for all states *except* for those after a stop_i event and before the next $\mathit{recover}_i$ event.
  - $\mathit{monitor}_i(j)$ holds for those states that come immediately after a $\mathit{monitor}(j)_i$ event,
  - $\mathit{demonitor}_i(j)$ holds for those states that come immediately after a $\mathit{demonitor}(j)_i$ event, and
  - $\mathit{downsig}_i(j)$ holds for those states immediately after a $\mathit{inform\_stopped}(j)_i$ event.

  With these predicates, Stoller specifies the following properties of failure detectors:[13]

[13] Technically, Stoller does not allow the possibility that a failure-detector could crash, or how failure-detectors should act if they remain alive while their client has stopped. This is an oversight (personal communication, 13 April 2009) which we correct here.

– Completeness: If participant P_j is being monitored by failure detector FD_i and stops, then eventually P_j recovers, or P_i is notified, or P_i de-monitors P_i, or P_i'resets' the monitor request by re-issuing it, or FD_i and P_i crash:

$$\Box(monitor_i(j) \implies \Box(\neg up_j \implies \Diamond(up_j \vee downsig_i(j) \vee monitor_i(j) \vee demonitor_i(j) \vee \neg up_i)))$$

– Accuracy: Participant P_i receives $inform\_stopped(j)_i$ only if P_j is actually down after the most recent $monitor(j)_i$:

$$\Box(monitor_i(j) \implies \neg downsig_i(j) \mathcal{W} \neg up_j) \wedge \Box(demonitor_i(j) \implies \neg downsig_i(j) \mathcal{W} monitor_i(j))$$

(Stoller also imposes a time-constraint, requiring that FD_i inform P_i of crashes within some fixed and constant amount of time. We do not consider the issue of time in this work, however, and so do not impose a similar requirement here.)

- Secondly, we note that our implementation does not follow Stoller's description directly, but is a variation of this algorithm implemented in Erlang by Svensson and Arts [24]. Svensson and Arts note, correctly, that Stoller's version of the algorithm could result in unnecessary elections: elections that occur despite the fact that the leader has not failed. In the algorithm described by both Garcia-Molina and Stoller, for example, every stopped participant immediately calls for a new election upon recovery. Thus, a new election will be held even though it will select the same leader as before (who may never have crashed).

  Svensson and Arts modify the algorithm so that elections will only be held when the leader crashes. From their description [24]:

  > We make this change in two steps, first we changed the algorithm such that no new election would be started if a process with lower priority[14] than the leader was activated. This change is fairly straightforward, and just requires a small modification to the behavior when a newly activated process is polled by the elected leader. Instead of restarting the election process, the newly activated process is informed of who the leader is...
  >
  > In addition we wanted to do something similar when a node with higher priority[15] than the present leader is activated. This however turned out to be much more complicated. The reason for the complexity is the fact that a node with high priority is likely to conclude that there are no processes active with a higher priority and therefore initiates a new election. (Note

[14]'Lower priority' in the sense of less urgent; a higher ID value.
[15]Again, 'higher priority' in the sense of more urgent; a lower ID value.

> however that this behavior is required, otherwise an election would never be initiated in the first place.) The basic trick here is to make sure that a process that knows who the leader is will not surrender to the newly activated process, instead it sends a reply saying who (he thinks) is the leader...
>
> Unfortunately, Svensson and Arts do not give any formal, high-level description of their new algorithm or prove its correctness. Instead, they tested their implementation of their algorithm in Erlang extensively (with both QuickCheck [3] and abstract-trace verification [2]). Taking a cue from Knuth,[16] we chose to build on the tested code rather than to implement ourselves a proved-correct algorithm.

The Svensson-Arts implementation is described in [24]. We apply our parse transform (described in Section 5.1) to this implementation and execute it in conjunction with our Erlang-implemented dispatcher.

Acknowledgements

The authors are grateful for our stimulating discussions with Doug Jensen and Leonard Monk, both of whom helped us adjust our perspective.

[16] "Beware of bugs in the above code; I have only proved it correct, not tried it." [12]

References

[1] Joe Armstrong. *Making reliable dstributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, November 2003.

[2] T. Arts and L.-Å Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[3] T. Arts and J. Hughes. Erlang/quickcheck. In *Proceedings, Ninth International Erlang/OTP User Conference*, November 2003.

[4] Boris Balacheff, Liqun Chen, Siani Pearson (ed.), David Plaquin, and Graeme Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, 2003.

[5] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings, 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*, pages 136–145. IEEE Computer Society, October 2001.

[6] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[7] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[8] George Coker, Joshua D. Guttman, Peter Loscocco, Justin Sheehy, and Brian T. Sniffen. Attestation: Evidence and trust. In Liqun Chen, Mark Dermot Ryan, and Guilin Wang, editors, *Proceedings, the 10th International Conference on Information and Communications Security (ICICS)*, volume 5308 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[9] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. In *Ada-Europe 2002*, volume 2361 of *LNCS*, pages 24–50. Springer, 2002.

[10] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[11] Kai Han, Guanhong Pei, Binoy Ravindran, and E. Douglas Jensen. Real-time, byzantine-tolerant information dissemination in unreliable and untrustworthy distributed systems. In *ICC*, pages 1727–1731, 2008.

[12] Donald Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Private correspondence, March 1977.

[13] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, pages 21–29, New York, NY, USA, 2007. ACM.

[14] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[15] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correcness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, April 1987. Technical Report MIT/LCS/TM-387.

[16] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137 – 151, August 1987.

[17] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-quarterly*, 2(3):219 – 246, September 1989.

[18] D. Malkhi and M. Reiter. Unreliable intrusion detection in ditributed computations. In *10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1997.

[19] John D. Ramsdell, Joshua Guttman, Jonathan Millen, and Brian O'Hanlon. An analysis of the caves attestation protocol using CPSA. July 2009. MTR090213.

[20] H. Sayeed, M. Abu-Amara, and H. Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distributed Computing*, 9(3):147–156, 1995.

[21] Scott D. Stoller. *Leader Election in Distributed Systems with Crash Failures*. Indiana University, Computer Science Dept., May 1997. Revised July 1997.

[22] Scott D. Stoller. Leader election in asynchronous distributed systems. *IEEE Transactions on Computers*, 49(3):283–284, March 2000.

[23] H. Svensson and Lars-Åke Fredlund. A more accurate semantics for distributed Erlang. In *Proceedings of the ACM SIPGLAN 2007 Erlang Workshop*, October 2007.

[24] Hans Svensson and Thomas Arts. A new leader election implementation. In *Proceedings of the ACM SIGPLAN 2005 Erlang Workshop*, pages 35 – 39, September 2005.

[25] Trusted Computing Group. *TCG Software Stack (TSS) Specification*, version 1.2 level 1 errata a edition, 2007. http://www.trustedcomputinggroup.org/files/resource_files/6479CD77-1D09-3519-AD89EAD1BC8C97F0/TSS_1_2_Errata_A-final.pdf.

A Proof of Theorem 3

We prove Theorem 3 in two steps: we first build a simulation relation $\mathcal{R}_{IR}$ from $\mathcal{A}_I$ to $\mathcal{A}_R$ to show that $traces(\mathcal{A}_I) \subseteq traces(\mathcal{A}_R)$, and then we build a simulation relation $\mathcal{R}_{RI}$ from $\mathcal{A}_R$ to $\mathcal{A}_I$ to show that $traces(\mathcal{A}_R) \subseteq traces(\mathcal{A}_I)$.

Before we can do this, however, we must introduce some notation:

Definition 5 *If X is a participant, channel, anonymizer, or dispatcher I/O automaton, and s is a state of $\mathcal{A}_R$, then by $s.X$ we mean the state of the X process in s.*

For example, $s.P_3$ is the state of P_3 in s. Likewise, we will use $s.X.v$ to mean the variable v in the state of $s.X$. Thus, $s.D_3.down$ is the state variable *down* in $s.Dispatcher_3$.

Definition 6 *If $\mathcal{R}$ is a relation, let $\mathcal{R}(y) = \{x : (x, y) \in \mathcal{R}\}$.*

We also need to define an operation on the queues of $\mathcal{A}_R$:

Definition 7 *Let q be a queue of messages and D be a dispatcher automaton. Then $valid(q, D)$ is the sub-queue of q consisting of those m in q where $m = (m', M, x)$ such that $MAC\_ver\,(m'; M; D.in\_key(x))$.*

Here, $MAC\_ver\,(m'; M; k)$ is true if the MAC M can be verified on m' with the key k.

Lemma 8 $traces(\mathcal{A}_I) \subseteq traces(\mathcal{A}_R)$

We prove this by constructing a simulation relation $\mathcal{R}_{IR}$ as follows: If s_I is a state of $\mathcal{A}_I$ and s_R is a state of $\mathcal{A}_R$, then $(s_I, s_R) \in \mathcal{R}_{IR}$ iff:

- For all participants $P_i \notin Byz$,
  - $s_R.P_i$ and $s_I.P_i$ are in the same state.
  - $s_R.D_i.queue_{in}$ is the empty queue, and
  - $s_R.D_i.queue_j$ is the empty queue for all j.
- For all $P_i \notin Byz$, $P_j \notin Byz$, $s_I.C^{id}_{i,j}.stopped$ is false if and only if
  - $s_R.C^{re}_{i,j}.stopped$ is false, and
  - $s_R.D_i.out\_key(j) = s_R.D_j.in\_key(i)$ or $s_R.D_i.out\_key(j) = \perp$.
- For all $P_i \notin Byz$, $P_j \notin Byz$, $s_I.C^{id}_{i,j}.queue = valid\big(s_R.C^{re}_{i,j}.queue, s_R.D_j\big)$.
- For all $P_j \notin Byz$, $s_I.A^{id}_j.queue = valid\big(s_R.A^{re}_j.queue, s_R.D_j\big)$.

To show that this is a simulation relation, we need to show two things:

1. If $s_I \in start(\mathcal{A}_I)$, then $\mathcal{R}_{IR}(s_I) \cap start(\mathcal{A}_R) \neq \emptyset$.
   jkm: has "start" been defined?

2. If s_I is a reachable state of $\mathcal{A}_I$, $s_R \in \mathcal{R}_{IR}(s_I)$ is a reachable state of $\mathcal{A}_R$, and (s_I, π, s'_I) is a transition of $\mathcal{A}_I$ with event π, then there is an execution fragment α of $\mathcal{A}_R$ starting with s_R and ending with $s'_R \in \mathcal{R}_{IR}(s'_I)$ such that $traces(\pi) = traces(\alpha)$. *jkm: why "traces"? Don't we just want to pick out the non-hidden events from α?*

To show the first of these, let $s_I \in start(I)$. Then for all $P_i \notin Byz$,

- $s_I.P_i$ is in a start-state for participant P_i,
- $s_I.A_i^{id}.queue$ is the empty queue.
- $s_I.C_{i,j}^{id}.queue$ is the empty queue for all j.

Furthermore, each channel $C_{i,j}^{id}$ is stopped. Thus, let $s_R \in \mathcal{R}_{IR}(s_I)$ where

- $s_R.P_i = s_I.P_i$,
- $s_R.D_i.queue_{in}$ is the empty queue.
- $s_R.D_i.queue_j$ is the empty queue for all j.
- $s_R.C_{i,j}^{re}.queue$ is the empty queue for all j.
- $s_R.A_i^{re}.queue$ is the empty queue for all j.
- Every channel $C_{i,j}^{re}$ is stopped.

Thus, $s_R \in \mathcal{R}_{IR}(s_I)$. Because $s_R \in start(\mathcal{A}_R)$, furthermore, $\mathcal{R}_{IR}(s_I) \cap start(\mathcal{A}_R) \neq \emptyset$.

Next, we demonstrate the second property by case-analysis. Let s_I be a reachable state of $\mathcal{A}_I$, let $s_R \in \mathcal{R}_{IR}(s_I)$ is a reachable state of $\mathcal{A}_R$, and suppose that (s_I, π, s'_I) is a transition of $\mathcal{A}_I$. Consider all possible transitions, categorized by the event π:

- $send(m)_{i,j}$: We begin by noting that it cannot be the case that $i \in Byz$: those participants begin the execution of $\mathcal{A}_I$ stopped and they never recover. We also note that this event changes the state of P_i and possibly $C_{i,j}^{id}$. In particular, message m is added to the end of $C_{i,j}^{id}.queue$ if and only if the channel is not stopped. Let us consider, then, two cases:
  1. Suppose that channel $C_{i,j}^{id}$ is not stopped. Then this event adds m to the tail of $C_{i,j}^{id}.queue$. We also note that two things must be true about s_R:
     - The channel $C_{i,j}^{re}$ is not stopped, and
     - The key $s_R.D_i.out\_key(j)$ is either $\perp$ or $s_R.D_i.out\_key(j)$.

     Therefore, consider the state s'_R which is the same as s_R except that
     - $s'_R.P_i$ is in the same state as $s'_I.P_i$,
     - if $s_R.D_i.out\_key(j) = \perp$ then $s'_R.D_i.out\_key(j) = s'_R.D_j.in\_key(i) = k$ for some new cryptographic keys k, and
     - The message (m, M, i) has been added the end of $s'_R.C_{i,j}^{re}.queue$ where $M = MAC\_sign\,(m; s'_R.D_i.out\_key(j))$.

     Then s_R can transition to s'_R by the following sequence of events:

- The event $send(m)_{i,j}$, which adds m to the end of the previously-empty queue $D_i.queue_j$.
- If it is the case that $D_i.out\_key(j) = \perp$, then the sequence continues with the events
  - $start\_measure(id)_{i,j}$, starting a measurement protocol,
  - $inform\_measure(id)_{i,j}$,
  - $yes\_measure(id; x)_{i,j}$,
  - $inform\_resp\_success(id, k)_{i,j}$, whereupon $D_j.in\_key(i) = k$, and
  - $inform\_init\_success(id, k)_{i,j}$, whereupon $D_i.out\_key(j) = k$.
- Lastly, the sequence ends with the second event: $transmit(m)_{i,j}$.

We note that all of these events except the first are hidden in $\mathcal{A}_R$, and so $\mathcal{A}_R$ transitions from s_R to s'_R with trace $send(m)_{i,j}$.

2. Suppose, on the other hand, that channel $C^{id}_{i,j}$ is stopped. Then the event $send(m)_{i,j}$ does not change the queue $C^{id}_{i,j}.queue$. Also, this implies that in s_R, at least one of the things are true:
   - The channel $C^{re}_{i,j}$ is stopped, or
   - The key $s_R.D_i.out\_key(j)$ is neither $\perp$ nor $s_R.D_j.in\_key(j)$.

   Based on these alternatives, we consider a number of sub-cases. In each sub-case, we will find a sequence of events by which s_R can transition to a $s'_R \in \mathcal{R}_{IR}(s'_I)$ with trace $send(m)_{i,j}$.

   (a) Suppose that the second statement is true, and the key

   $$s_R.D_i.out\_key(j)$$

   is neither $\perp$ nor $s_R.D_j.in\_key(j)$. Then consider the sequence of events:
   - $send(m)_{i,j}$,
   - $transmit((m, M, i))_{i,j}$ where

   $$M = MAC\_sign\,(m; D_i.out\_key(j))$$

   .

   The first event will place m on the queue $D_i.queue_j$. The second event will take this message off the queue. If $C^{re}_{i,j}$ is down, then $C^{re}_{i,j}.queue$ will not be changed. But even if the channel is up, it will not be the case that

   $$MAC\_ver\,(m; M; D_j.in\_key(i))\,.$$

   Therefore, it will not be the case that the new message m will be on

   $$valid\big(C^{re}_{i,j}.queue, D_j\big)$$

   . Thus, the machine $\mathcal{A}_R$ can transition by the above sequence of events into a state s'_R where $s'_R \in \mathcal{R}_{IR}s'_I$. Furthermore, the second event in the sequence is hidden, so both $\mathcal{A}_R$ and $\mathcal{A}_I$ produce the trace $send(m)_{i,j}$.

(b) If the second statement is false, on the other hand, then the first statement must be true. Let us consider two subcases:

- Suppose $s_R.D_i.out\_keyj \neq \perp$. Then again consider the sequence of events:
  - $send(m)_{i,j}$,
  - $transmit((m, M, i))_{i,j}$ where $M = MAC\_sign\,(m; D_i.out\_key(j))$.

  The first event will place m on the queue $D_i.queue_j$. The second event will take this message off the queue. Because $C^{re}_{i,j}$ is down, however, the queue $C^{re}_{i,j}.queue$ will not be changed. Thus, the machine $\mathcal{A}_R$ can transition by the above sequence of events into a state s'_R where $s'_R \in \mathcal{R}_{IR}s'_I$. Furthermore, the second event in the sequence is hidden, so both $\mathcal{A}_R$ and $\mathcal{A}_I$ produce the trace $send(m)_{i,j}$.
- Suppose that $s_R.D_i.out\_key(j) = \perp$. Then consider the sequence of events:
  - $send(m)_{i,j}$,
  - $start\_measure(id)_{i,j}$,
  - $inform\_init\_error(id)_{i,j}$,
  - $error\_drop(m)_{i,j}$

  Again, the queue $C^{re}_{i,j}.queue$ will not be changed. Thus, the machine $\mathcal{A}_R$ can transition by the above sequence of events into a state s'_R where $s'_R \in \mathcal{R}_{IR}s'_I$. Furthermore, the second event in the sequence is hidden, so both $\mathcal{A}_R$ and $\mathcal{A}_I$ produce the trace $send(m)_{i,j}$.

- $receive(m)_j$: This event changes the state of P_j and A^{id}_j. In particular, it removes message m from the head of $A^{id}_j.queue$. By the definition of our relation, therefore, m must also be the head of $valid\big(s_R.A^{re}_j.queue, s_R.D_j\big)$. In this case, consider the state s'_R which is the same as s_R except that $s'_R.P_i$ is in the same state as $s'_I.P_i$ and $s'_R.A^{id}_j.queue$ is the same as $s_R.A^{id}_j.queue$ except that all messages at the head of $s_R.A^{id}_j.queue$ are removed up to and including the first message $m' = (m, M, i)$ where

$$MAC\_ver\,(m; M; D_j.in\_key(i))\,.$$

  In this case, s'_I and s'_R agree on the state of all non-Byzantine participants, and $s'_I.A^{id}_j.queue = valid\big(s'_R.A^{id}_j.queue, s'_R.D_j\big)$. So, $s'_R \in \mathcal{R}_{IR}(s'_I)$, and $\mathcal{A}_R$ can also transition from s_R to s'_R by performing the sequence of events:

  - For every message x at the head of $s_R.A^{id}_j.queue$ which is not the first $m' = (m, M, i)$ where

$$MAC\_ver\,(m; M; D_j.in\_key(i))\,,$$

    the following sequence of events:
    - One event $pre\_receive(x)_j$, delivering the message x to D_j, making it the only message on $D_j.queue_{in}$, and

  - One event $MAC\_drop(x)_i$, removing it from $D_j.queue_{in}$ as being malformed or incorrectly authenticated.
- For the first message $m' = (m, M, i)$ where

$$MAC\_ver\,(m; M; D_j.in\_key(i),)$$

the sequence of messages

  - One event $pre\_receive(m')_j$, delivering the message m' to D_j, making it the only message on $D_j.queue_{in}$, and
  - One event $receive(m)_j$, removing m' from $D_j.queue_{in}$ and delivering m to P_j.

The only event in this sequence which is not hidden in $\mathcal{A}_R$ is the last event, meaning that $\mathcal{A}_R$ can transition from s_R to s'_R with trace $receive(m)_j$.

- $deliver(m)_{i,j}$: This event changes the state of $C^{id}_{i,j}$ and A^{id}_j. In particular, it removes m from the head of $C^{id}_{i,j}.queue$ and adds it to the tail of $A^{id}_j.queue$. Note that due to the definition of our relation, it must be the case that m is also the head of $valid\big(s_R.C^{re}_{i,j}, s_R.D_j\big)$. Let $m' = (m, M, i)$ be the first message in $s_R.C^{id}_{i,j}.queue$ where

$$MAC\_ver\,(m; M; D_j.in\_key(i))\,.$$

Given this, consider the state s'_R which is the same as s_R except that all message on the head of $s_R.C^{id}_{i,j}.queue$ up to and including m' have been removed and added (in order) to the tail of $s_R.A^{re}_j.queue$. Thus, $s'_R \in \mathcal{R}_{IR}(s'_I)$, and $\mathcal{A}_R$ can also transition from s_R to s'_R by performing a single $deliver(x)_{i,j}$ event for every message x on the head of $s_R.C^{id}_{i,j}.queue$ up to and including m'. We note that the event $deliver(x)_{i,j}$ is hidden in both $\mathcal{A}_R$ and $\mathcal{A}_I$, and so both automata make their transitions with an empty trace.

- $stop_i$: This event affects only P_i, and only if it is not stopped. If P_i is not stopped in s_I, it will be stopped in s'_I. Consider, then, the effect that the event $stop_i$ has on the state s_R. This event will affect D_i in many ways, and will stop P_i if it is not stopped already. Thus, s_R will transition to a state s'_R such that $s'_R \in \mathcal{R}_{IR}(s'_I)$ and also produce a trace with the single event $stop_i$.
- $recover_i$: Analogous to $stop_i$, above.
- $channel\_down_{i,j}$: This event affects only $C^{id}_{i,j}$, and only if it is not stopped. If this is the case, $C^{id}_{i,j}$ is stopped in s'_I. Consider, then, the effect that the event $channel\_down_{i,j}$ has on the state s_R. This event will stop $C^{id}_{i,j}$ if it is not stopped already. Thus, s_R will transition to a state s'_R such that $s'_R \in \mathcal{R}_{IR}(s'_I)$ and also produce a trace with the single event $channel\_down_{i,j}$.
- $channel\_up_{i,j}$: We note that this event is hidden in $\mathcal{A}_I$, and so produces the empty trace. Secondly we note that this event only matters if $i \notin Byz$; otherwise, it is irrelevant to our relation. We then consider two cases:

1. If $C_{i,j}^{id}$ is not stopped in s_I, then this event has no effect on $\mathcal{A}_I$ and $s'_I = s_I$. Therefore, it is already the case that $s_R \in \mathcal{R}_{IR}(s'_I)$, and so $\mathcal{A}_R$ can transition to a state in $\mathcal{R}_{IR}(s'_I)$ via the empty trace.
2. If, on the other hand, $C_{i,j}^{id}$ is stopped in s_I, then it will be started in s'_I. But consider the states s_R and s'_R. Since $C_{i,j}^{id}$ is stopped in s_I, it must be the case that
   (a) $C_{i,j}^{re}$ is stopped, or
   (b) $D_i.out\_key(j)$ is neither $\perp$ nor $D_j.in\_key(i)$.

   If the first of these statements is true and the second is false, then consider the effect of the event $channel\_up_{i,j}$ on automata $\mathcal{A}_R$ in s_R. This will cause $\mathcal{A}_R$ to transition to a state in which $C_{i,j}^{re}$ is up, and D_i and D_j agree on an actual key for communication. Therefore, $\mathcal{A}_R$ is in a state $s'_R \in \mathcal{R}_{IR}(s'_I)$ and transitioned there via the empty trace.
   Now suppose, on the other hand, that the second statement is true. (The first statement may or may not also be true.) Then, consider the effect of the following sequence of events on $\mathcal{A}_R$:
   - $channel\_up_{i,j}$,
   - $start\_measure(id)_{i,j}$,
   - $inform\_measure(id)_{i,j}$,
   - $yes\_measure(id; x)_{i,j}$,
   - $inform\_resp\_success(id, k)_{i,j}$,
   - $inform\_init\_success(id, k)_{i,j}$,

   where id is some new identifier and $x = s_R.D_j.out\_key(i)$. We note that this sequence of events is possible due to (1) the definition of the attestation and dispatcher automata and (2) the assumption that the event $channel\_up_{i,j}$ only occurs in $\mathcal{A}_I$ when $i \notin Byz$. If this is the case then the protocol must be able to succeed when $i \notin Byz$ (because P_i must both then be running the honest-participant automaton) and can succeed when $j \in Byz$.
   Let s'_R be the state of $\mathcal{A}_R$ after this sequence of events. We note that in s'_R, $C_{i,j}^{re}$ cannot be stopped. (If it was stopped before the sequence, the first event brings it back up. Otherwise, the first event has no effect.) Furthermore, it will be the case that in s'_R, $D_i.out\_key(j) \neq \perp$ and $D_i.out\_key(j) = D_j.in\_key(i)$. Thus, s'_R satisfies the portion of the relation $\mathcal{R}_{IR}$ concerning the channel being stopped.
   However, we must also consider that part of the relation concerning the channel's message queue. And for that, we need to consider two cases:
   (a) If $s_R.D_j.in\_key(i) = \perp$, then $valid\big(s_R.C_{i,j}^{re}.queue, s_R.D_j\big)$ is the empty queue. Thus, $C_{i,j}^{id}.queue$ is empty as well in s_I, and is in s'_I as well. But because $s'_R.D_j.in\_key(j)$ is a new cryptographic key not used by any messages in $C_{i,j}^{re}.queue$, it will be that

   $$valid\big(stateReal'.C_{i,j}^{re}.queue, s'_R.D_j\big)$$

   will be empty as well.

(b) If $s_R.D_j.in\_key(i) \neq \perp$, on the other hand, then $s'_R.D_j.in\_key(i) = s_R.D_j.in\_key(i)$. (Recall from Section 3.3 that if the responder already has a key when an attestation protocol is initiated, then the key will re-distribute that key.) Thus,

$$valid\big(stateReal'.C^{re}_{i,j}.queue, s'_R.D_j\big) = valid\big(stateReal.C^{re}_{i,j}.queue, s_R.D_j\big),$$

and so

$$valid\big(stateReal'.C^{re}_{i,j}.queue, s'_R.D_j\big) = s'_I.C^{id}_{i,j}.queue.$$

In either case, $stateReal'$ and $stateIdeal'$ also satisfy the queue-correspondence portion of $\mathcal{R}_{RI}$, and $s'_R \in \mathcal{R}_{RI}(s'_I)$. Furthermore, the above sequence of events are all hidden in $\mathcal{A}_R$, and so both $\mathcal{A}_R$ and $\mathcal{A}_I$ produce the same (empty) trace.

- $anon\_drop(n)_i$: This event changes the state of A^{id}_j. In particular, the nth element of queue $A^{id}_j.queue$, is removed from that queue.
  Due to the definition of the relation $\mathcal{R}_{IR}$, it must be the case that in s_R, the queue $valid\big(s_R.A^{re}_j.queue, s_R.D_j\big)$ that at least n elements. Thus, let $m' = (m, M, i)$ be the nth message in $s_R.A^{re}_j.queue$ where

  $$MAC\_ver\,(m; M; D_j.in\_key(i))\,.$$

  Given this, consider the state s'_R which is the same as s_R except that the message m' has been removed from $A^{re}_j.queue$. Thus, $s'_R \in \mathcal{R}_{IR}(s'_I)$, and $\mathcal{A}_R$ can also transition from s_R to s'_R by performing a single $anon\_drop(x)_i$ event where x is the index of m' in $s_R.A^{re}_j.queue$.
  We note that the event $anon\_drop(x)_i$ is hidden in both $\mathcal{A}_R$ and $\mathcal{A}_I$, and so both automata make their transitions with an empty trace.
- $channel\_drop(n)_{i,j}$: This case is analogous to $anon\_drop(n)_i\, j$, above. In particular, this event changes the state of $C^{id}_{i,j}$ by removing the nth element from the queue $C^{id}_{i,j}.queue_i$. Due to the definition of the relation $\mathcal{R}_{IR}$, then, it must be the case that in s_R, the queue $valid\big(s_R.C^{id}_{i,j}.queue, s_R.D_j\big)$ must also have at least n elements. Thus, let $m' = (m, M, i)$ be the nth message in $s_R.C^{id}_{i,j}.queue$ where $MAC\_ver\,(m; M; D_j.in\_key(i))$. Given this, consider the state s'_R which is the same as s_R except that the message m' has been removed from $s_R.C^{id}_{i,j}.queue$. Thus, $s'_R \in \mathcal{R}_{IR}(s'_I)$, and $\mathcal{A}_R$ can also transition from s_R to s'_R by performing a single $channel\_drop(x)_{i,j}$ event where x is the index of m' in $s_R.C^{id}_{i,j}.queue$.
  We note that the event $channel\_drop(x)_{i,j}$ is hidden in both $\mathcal{A}_R$ and $\mathcal{A}_I$, and so both automata make their transitions with an empty trace.

Before we an define the simulation relation from $\mathcal{A}_R$ to $\mathcal{A}_I$, we need one operation on queues:

Definition 9 *We use the notation $\|$ to mean queue-concatenation. If $queue_1$ and $queue_2$ are queues, then $queue_1 \| queue_2$ is the queue where the head of $queue_1$ is appended to the tail of $queue_2$.*

Lemma 10 $traces(\mathcal{A}_R) \subseteq traces(\mathcal{A}_I)$

We prove this by defining a simulation relation $\mathcal{R}_{RI}$ as follows: If s_R is a state of $\mathcal{A}_R$ and s_I is a state of $\mathcal{A}_I$, then $(s_R, s_I) \in \mathcal{R}_{RI}$ iff:

- For all participants $P_i \notin Byz$, $s_R.P_i$ and $s_I.P_i$ are in the same state.
- For all $P_i \notin Byz$, $P_j \notin Byz$, $s_I.C^{id}_{i,j}.stopped$ is false if and only if:
  - $s_R.C^{re}_{i,j}.stopped$ is false, and
  - $s_R.D_i.out\_key(j) = s_R.D_j.in\_key(j)$ or $s_R.D_i.out\_key(j) = \perp$.
- For all $P_i \notin Byz$ and $P_j \notin Byz$,

$$s_I.C^{id}_{i,j}.queue = s_R.D_i.queue_j \| valid\big(s_R.C^{re}_{i,j}.queue, D_j\big)$$

- For all $P_j \notin Byz$,

$$s_I.A^{id}_j.queue = valid\big(s_R.A^{re}_j.queue \| s_R.D_j.queue_{in}, D_j\big)$$

To show that this is a simulation relation from $\mathcal{A}_R$ to $\mathcal{A}_I$, we must demonstrate the following two properties:

1. If $s_R \in start(\mathcal{A}_R)$, then $\mathcal{R}_{RI}(s_R) \cap start(\mathcal{A}_I) \neq \emptyset$.
2. If s_R is a reachable state of $\mathcal{A}_R$, $s_I \in \mathcal{R}_{RI}(s_R)$ is a reachable state of $\mathcal{A}_I$, and (s_R, π, s'_R), then there is an execution fragment α of $\mathcal{A}_I$ starting with s_I and ending with $s'_I \in \mathcal{R}(s'_R)$ such that $traces(\pi) = traces(\alpha)$.

To show the first two of these properties, let $s_R \in start(\mathcal{A}_R)$. Then we note that for all $P_i \notin Byz$:

- $s_R.P_i$ is in a start-state for participant-automaton P_i,
- $s_R.D_i.queue_{in}$ is empty,
- $s_R.D_i.queue_j$ is empty (for all j), and
- $s_R.A^{re}_i.queue$ is empty.

Furthermore, $C^{re}_{i,j}$ is stopped and $s_R.C^{id}_{i,j}.queue$ is empty for all i and j. Thus, $s_I \mathcal{R}_{RI}(s_R)$ where for all $i \notin Byz$,

- $s_I.P_i = s_R.P_i$,
- $s_I.A^{id}_i.queue$ is empty,
- $s_I.C^{id}_{i,j}.queue$ is empty,
- $s_I.C^{id}_{i,j}$ is stopped.

Furthermore, $s_I \in start(\mathcal{A}_I)$.

Next, we demonstrate the second property by case-analysis. Let s_R be a reachable state of $\mathcal{A}_R$, let $s_I \in \mathcal{R}_{RI}(s_R)$ is a reachable state of $\mathcal{A}_I$, and suppose that (s_R, π, s'_R) is a transition of $\mathcal{A}_R$. Consider all possible transitions, categorized by the event π:

- $send(m)_{i,j}$: This event will change the state of P_i and D_i. In particular, P_i will transition to some other state, and m will be added to the end of $D_i.queue_j$. Consider, then the state s'_I which is exactly like s_I except that $s'_I.P_i$ is the same as $s'_R.P_i$ and $s'_I.C^{id}_{i,j}.queue$ is the same as $s'_I.C^{id}_{i,j}.queue$ with m added to the end. Then $\mathcal{A}_I$ can transition from s_I to s'_I by a $send(m)_{i,j}$ event, producing the trace $send(m)_{i,j}$. Furthermore, $s'_I \in \mathcal{R}(s'_R)$.
- $transmit(m)_{i,j}$: We consider two cases:
  - If $i \in Byz$, then this event changes the state of P_i and $C^{re}_{i,j}$. However, both of these automata are irrelevant to the relation $\mathcal{R}_{RI}$. Therefore, it is already the case that $s_I \in \mathcal{R}_{RI}(s'_R)$. And because the event $transmit(m)_{i,j}$ is hidden in $\mathcal{A}_R$, both $\mathcal{A}_R$ and $\mathcal{A}_I$ produce the empty trace.
  - If $i \notin Byz$, on the other hand, this event will affect D_i and possibly $C^{re}_{i,j}$. We note that it will be the case that $m = (m', M, i)$ where $MAC\_sign\,(m; D_i.out\_key(j))$ and m' is at the head of $s_R.D_i.queue_j$. This event removes message m' from $D_i.queue_j$. The effect it has on $C^{re}_{i,j}$, on the other hand, depends on two factors: whether $s_R.C^{re}_{i,j}$ is stopped, and whether $s_R.D_i.out\_key(j) = s_R.D_j.in\_key(i)$.
    - If $s_R.C^{re}_{i,j}$ is stopped, then the message m is not added to the tail of $C^{re}_{i,j}.queue$. In this case, consider the state s'_I which is exactly like s'_I except that the message m' has been removed from $s_I.C^{id}_{i,j}.queue$. Then $\mathcal{A}_I$ can transition from s_I to s'_I by the event $channel\_drop(a - b + 1)_{i,j}$ where a is the number of elements in $s_I.C^{id}_{i,j}.queue$ and b is the number of elements in $s_R.D_i.queue_j$.
    - If $stateReal.C^{re}_{i,j}$ is not stopped, however, but $s_R.D_i.out\_key(j) \neq s_R.D_j.in\_key(i)$, then the message m is added to the tail of $C^{re}_{i,j}$. However, it will not be the case that

      $$MAC\_ver\,(m'; M; D_j.in\_key(i))\,,$$

      and so m' will not be part of the queue $valid\big(C^{re}_{i,j}.queue, D_j\big)$. Thus $\mathcal{A}_I$ can again transition from s_I to s'_I by the event

      $$channel\_drop(a - b + 1)_{i,j}$$

      where a is the number of elements in $s_I.C^{id}_{i,j}.queue$ and b is the number of elements in $s_R.D_i.queue_j$.
    - If $stateReal.C^{re}_{i,j}$ is not stopped and but $s_R.D_i.out\_key(j) = s_R.D_j.in\_key(i)$, then the message m is added to the tail of $C^{re}_{i,j}$. Furthermore, it will now be the case that

      $$MAC\_ver\,(m'; M; D_j.in\_key(i))\,,$$

      and so m' will be part of the queue $valid\big(C^{re}_{i,j}.queue, D_j\big)$. Thus, $s_I = s'_I$.

In all three of these cases, s_I can transition to a state s_I' such that $s_I' \in \mathcal{R}_{RI}(s_R')$. Furthermore, the transitions will produce the empty trace, as does the original event $transmit(m)_{i,j}$.

- $deliver(m)_{i,j}$: This event will change the state of $C_{i,j}^{re}$ and A_j^{re}. In particular, m will be removed from the front of $C_{i,j}^{re}.queue$ and added to the end of A_j^{re}. There are two cases:
  1. $m = (m', M, x)$ where $MAC\_ver\,(m; M; D_j.in\_key(j))$. In this case, we note that m' must be at the head of the queue $valid\big(C_{i,j}^{re}.queue, D_j\big)$ and thus at the head of $stateIdeal.C_{i,j}^{id}.queue$. Consider the state s_I' which is identical to s_I except that m' has been removed from the head of $s_I.C_{i,j}^{id}.queue$ and added to the tail of $s_I.A_j^{id}.queue$. Then $\mathcal{A}_I$ can transition from s_I' to s_I' by the event $deliver(m')_{i,j}$. But because the 'deliver' event is hidden in both $\mathcal{A}_R$ and $\mathcal{A}_I$, both machines produce only the empty trace when they make their transitions.
  2. $m \neq (m', M, x)$ or not $MAC\_ver\,(m; M; D_j.in\_keyj)$. In this case, we note that neither m nor m' will be at the head of the queue $valid\big(C_{i,j}^{re}.queue, D_j\big)$. Thus, this event is irrelevant to the relation $\mathcal{R}_{RI}$. Thus, s_I is already in $\mathcal{R}_{RI}(s_R')$, and $\mathcal{A}_I$ does not need to make a transition at all. But because the 'deliver' event is hidden in both $\mathcal{A}_R$, the machine $\mathcal{A}_R$ also produces the empty trace when it makes its transition.
- $pre\_receive(m)_i$: This event changes both A_j^{re} and D_j. In particular, the message m is removed from the head of A_j^{re} and added to the tail of $D_j.queue_{in}$. This change is 'invisible' to our relation, and so $s_I \in \mathcal{R}_{RI}(s_R')$. Furthermore, this event is hidden in $\mathcal{A}_R$, and both it an the non-transition of $\mathcal{A}_I$ produce the empty trace.
- $receive(m)_i$: Again, this event will change the state of P_j and D_j. In particular, P_j will transition to some other state, and (m, M, i) (for some i, and where $MAC\_ver\,(m; M; D_j.in\_key(i))$) will be removed from the front of $D_j.queue$. But in this case, m will be the head of $valid(D_j.queue_{in}, D_j)$, and so will be the head of $A_j^{id}.queue$ in s_I. Consider, then the state s_I' which is exactly like s_I except that $s_I'.P_i$ is the same as $s_R'.P_i$ and $s_I'.A_j^{id}.queue$ is the same as $s_I'.A_j^{id}.queue$ with m removed from the head. Then $\mathcal{A}_I$ can transition from s_I to s_I' by a $receive(m)_i$ event, producing the trace $receive(m)_i\, j$, and $s_I' \in \mathcal{R}(s_R')$.
- $stop_j$: If $j \in Byz$, then this event affects only P_j. However, the state of P_j is irrelevant to the relation $\mathcal{R}_{RI}$, and therefore s_I is already in $\mathcal{R}_{RI}(s_R')$. Again, $\mathcal{A}_I$ does not need to make a transition and so produces the empty trace. But the event $stop_j$ is hidden in $\mathcal{A}_R$, and so $\mathcal{A}_R$ produces the empty trace as well.

  If $j \notin Byz$, on the other hand, the effects of this event are
  - To stop P_j and D_j,
  - To empty $D_j.queue_{in}$ and $D_j.queue_i$ for all i,
  - To empty all sets in D_j, and

– To set $D_j.in\_key(i)$ and $D.out\_key(i)$ to $\perp$ for all i.

Thus, consider the state s'_I which is the same as s_I except that

– Participant P_j is stopped,
– The queue $A_j^{id}.queue$ is empty,
– For all i:
  1. The queue $C_{i,j}^{id}.queue$ is empty,
  2. The channel $C_{i,j}^{id}$ is stopped, and
  3. $C_{j,i}^{id}.queue = valid\big(C_{j,i}^{re}.queue, D_i\big)$.

In this case, $s'_I \in \mathcal{R}_{RI}(s'_R)$, and $\mathcal{A}_I$ can transition from s_I to s'_I by the sequence of events:

1. $stop_j$,
2. One event of the form $anon\_drop(1)_j$ for each element of $s_I.A_j^{id}.queue$,
3. For all i:
   (a) One event of the form $channel\_drop(1)_{i,j}$ for each element of $s_I.C_{i,j}^{id}.queue$,
   (b) One event of the form $channel\_down_{i,j}$,
   (c) One event of the form $channel\_down_{j,i}$,
   (d) One event of the form $channel\_drop(a-b+1)_{j,i}$ for each element of $s_R.D_j.queue_i$ (where a is the number of elements in $s_I.C_{j,i}^{id}.queue$ and b is the number of elements in $s_R.D_j.queue_i$).

Because all events other than the first one are hidden in $\mathcal{A}_I$, $\mathcal{A}_I$ can also transition from s_I to $stateIdeal'$ by way of trace $stop_j$.

- $recover_j$: This event starts (or re-starts) P_j and D_j. Therefore, consider the state s'_I which is identical to s_I except that $s'_I.P_j = stateReal'.P_j$. Then $\mathcal{A}_I$ can transition from s_I to s'_I by way of the event $recover_j$.
- $channel\_down_{i,j}$: This event affects only $C_{i,j}^{re}$, and only by stopping it if it not already stopped. Consider the state s'_I which is exactly like s'_I except that $C_{i,j}^{id}$ is stopped if not already. Then $\mathcal{A}_I$ can transition from s_I to s'_I by way of the event $channel\_down_{i,j}$, and $s'_I \in \mathcal{R}_{RI}(s'_R)$.
- $channel\_up_{i,j}$: This event affects only $C_{i,j}^{re}$, and only by starting it if it is stopped. To find the state s'_I of interest, we consider two cases:
  1. If $D_i.out\_key(j) \neq D_j.in\_key(i)$ and $D_i.out\_key(j) \neq \perp$, then let $s'_I = s'_I$.
  2. If, on the other hand, $D_i.out\_key(j) = D_j.in\_key(i)$ or $D_i.out\_key(j) = \perp$, then let s'_I be the state of $\mathcal{A}_I$ after executing event $channel\_up_{i,j}$.

  In both of these cases, $s'_I \in \mathcal{R}_{RI}(s'_R)$. Also, we note that the event $channel\_up_{i,j}$ is hidden in both $\mathcal{A}_I$ and $\mathcal{A}_R$. Thus, $\mathcal{A}_I$ can transition from s_I to s'_I by way of the empty trace, the same trace by which $\mathcal{A}_R$ transitions from s_R to s'_R.
- $channel\_drop(n)_{i,j}$: This event removes the n element of $C_{i,j}^{re}.queue$. We first note that this event is hidden in $\mathcal{A}_R$, and so produces no trace. We next consider two cases:

  - If the nth element of $C^{re}_{i,j}$ is $m = (m', M, x)$ and

$$MAC\_ver\left(m'; M; D_j.in\_key(i)\right),$$

    then consider the state s'_I which is exactly like s_I except that the message m' is removed from $C^{id}_{i,j}.queue$. (If there is more than one instance of m' on that queue, we remove the instance associated to m by the mapping $valid\big(C^{re}_{i,j}.queue, D_j\big)$.) Then $s'_I \in \mathcal{R}_{RI}(s'_R)$ and $\mathcal{A}_I$ can transition from s_I to s'_I by way of the event $channel\_drop(n)_{i,j}$ (where n is the index of m' in $s_I.C^{id}_{i,j}.queue$) producing the empty trace.
  - Otherwise, we note that $s_I \in \mathcal{R}_{RI}(s'_R)$ already, and so $\mathcal{A}_I$ can transition from s_I to s_I by the empty trace.

- $anon\_drop(n)_j$: similar to $channel\_drop(n)_{i,j}$ above, except considering the queue $A^{re}_j.queue$ instead of $C^{re}_{i,j}.queue$.
- $MAC\_drop(m)_j$: This event affects only D_j, and only by deleting the head of $D_j.queue_{in}$ because either $m \neq (m', M, x)$ or not

$$MAC\_ver\left(m'; M; D.in\_key(x)\right).$$

  But in this case, the message m' is not at the head of $s_I.A^{id}_j.queue$, and so deleting this message from $D_j.queue_{in}$ is ignored by the relation $\mathcal{R}_{RI}$. Thus, it is the case that $s_I \in \mathcal{R}_{RI}(s'_R)$ already. Because the event $MAC\_drop(m)_j$ is hidden in $\mathcal{A}_R$, furthermore, the empty trace of $\mathcal{A}_I$ as it transitions from s_I to s_I is the same as the empty trace produced by $\mathcal{A}_R$ as it transitions from s_R to s'_R.
- $error\_drop(m)_{i,j}$: This event affects only D_i, and only be removing m from the head of $D_i.queue_j$. Thus, consider the state s'_I which is just like s_I except that m has been removed from $C^{id}_{i,j}$. Then $\mathcal{A}_I$ can transition from s_I to s'_I by way of the event $channel\_drop(a-b+1)_{i,j}$ (where a is the number of messages in $s_I.C^{id}_{i,j}.queue$ and b is the number of messages in $s_R.D_i.queue_j$). Because the event $error\_drop(m)_{i,j}$ is hidden in $\mathcal{A}_R$ and the event $channel\_drop(a-b+1)_{i,j}$ is hidden in $\mathcal{A}_I$, both automata produce the empty trace when they make their transitions.
- $start\_measure(id)_{i,j}$, $inform\_measure(id)_{i,j}$, $yes\_measure(id; x)_{i,j}$, $inform\_init\_fail(id)_{i,j}$, $inform\_resp\_fail(id)_{i,j}$, $inform\_init\_error(id)_{i,j}$, $inform\_resp\_error(id)_{i,j}$: These events change only the states of D_i, D_j and the attestation automaton. However, none of these changes are relevant to the relation $\mathcal{R}_{RI}$, and all of these events are hidden in $\mathcal{A}_R$. Thus, s_I is already in $\mathcal{R}_{RI}(s'_R)$, and $\mathcal{A}_I$ does not need to make any transitions at all to produce the same trace as $\mathcal{A}_R$.
- $inform\_init\_success(id, k)_{i,j}$: This event changes the state of D_i, and in particular sets $D_i.out\_key(j)$ to be k, which cannot be $\perp$. We consider two cases:
  1. If $s'_R.D_i.out\_key(j) = s'_R.D_j.in\_key(i)$ and $C^{re}_{i,j}$ is not stopped, then consider the state s'_I which is just like s_I except that if $C^{id}_{i,j}$ is stopped

in s_I it is not stopped in s'_I. Then $s'_I \in \mathcal{R}_{RI}(s'_R)$, and $\mathcal{A}_I$ can transition from s_I to s'_I via the event $channel\_up_{i,j}$. Because both the event $inform\_init\_success(id,k)_{i,j}$ and the event $channel\_up_{i,j}$ are hidden, both $\mathcal{A}_I$ and $\mathcal{A}_R$ produce the empty trace as they make their transitions.

2. Otherwise, we note that s_I is already in $\mathcal{R}_{RI}(s'_R)$ and the event $inform\_init\_success(id,k)_{i,j}$ is hidden in $\mathcal{A}_R$. Thus, $\mathcal{A}_R$ produces the empty trace as it makes its transition, and $\mathcal{A}_I$ can produce the same trace as it remains in s_I.

- $inform\_resp\_success(id,k)_{i,j}$: This event changes the state of D_j, and in particular sets $D_j.in\_key(i)$ to be k, which cannot be $\perp$. Before we continue, we make an observation: If $s'_R.C^{re}_{i,j}.queue$ or $s'_R.A^{id}_j$ contains a message $m = (m', M, x)$ such that $MAC\_ver\,(m'; M; k)$, then it must also be the case that $MAC\_ver\,(m'; M; D_j.in\_key(i))$ as well. That is, if a message's MAC can be verified with the 'new' key, then it can be verified with the 'old' key as well. This can be seen by examining two cases:
  - If $s'_R.D_j.in\_key(i) = s_R.D_j.in\_key(i)$, then the above observation trivially follows.
  - If $s'_R.D_j.in\_key(i) \neq s_R.D_j.in\_key(i)$, on the other hand, then we note that in the attestation protocol, the 'responder' (j) receives the key before the 'initiator' (i). Thus, there cannot be any messages in the channel or anonymizer that can be verified with the new key. The above observation trivially follows.

  We note that the converse does not necessarily follow: there could exist messages that can be verified with the old key but not the new key. Consider two cases:

  1. If $s'_R.D_j.in\_key(i) = s_R.D_j.in\_key(i)$, then we note that s_I is already in $\mathcal{R}_{RI}(s'_R)$. Therefore, $\mathcal{A}_I$ does not need to make a transition, and does not produce any trace. But the event $inform\_resp\_success(id,k)_{i,j}$ is already hidden in $\mathcal{A}_R$, and so $\mathcal{A}_R$ also produces the empty trace.
  2. If $s'_R.D_j.in\_key(i) \neq s_R.D_j.in\_key(i)$ on the other hand, consider the sequence of events for $\mathcal{A}_I$:
     - One $channel\_drop(n)_{i,j}$ for each message m in $C^{re}_{i,j}.queue$ such that $m = (m', M, x)$ and $MAC\_ver\,(m'; M; s_R.D_j.in\_key(i))$ (from the highest value of n to the lowest), and
     - One $anon\_drop(n)_i$ for each message m in $A^{re}_j.queue$ such that $m = (m', M, x)$ and $MAC\_ver\,(m'; M; s_R.D_j.in\_key(i))$, and
     - One $anon\_drop(n)_i$ for each message m in $D_j.queue_{in}$ such that $m = (m', M, x)$ and $MAC\_ver\,(m'; M; s_R.D_j.in\_key(i))$.

     Then $s'_I \in \mathcal{R}_{RI}(s'_R)$, and $\mathcal{A}_I$ can transition from s_I to s'_I via the sequence above. Because both the event $inform\_resp\_success(id,k)_{i,j}$ and the events above are hidden, both $\mathcal{A}_I$ and $\mathcal{A}_R$ produce the empty trace as they make their transitions.

B I/O Automata for $C_{i,j}^{re}$ and A_i^{re}

Signature:

Input:
$transmit(m)_{i,j}$, $m \in M$
$channel\_down_{i,j}$
$channel\_up_{i,j}$

Output
$deliver(m)_{i,j}$, $m \in M$

Internal
$channel\_drop(m)_{i,j}$

State Variables:
stopped, a boolean value initially set to 'false'
queue, a FIFO queue of messages in M, initially empty.

Transitions:

$transmit(m)_{i,j}$:
Effect:
if *stopped* is false, add m to the *queue*. Otherwise, no effect.

$channel\_down_{i,j}$:
Effect:
set *stopped* to true.

$channel\_up_{i,j}$:
Effect:
set *stopped* to false.

$deliver(m)_{i,j}$:
Precondition:
m is first on *queue*.
Effect:
remove first element of *queue*.

$channel\_drop(m)_{i,j}$:
Precondition:
m is last on *queue*.
Effect:
remove last element of *queue*

Tasks:
Arbitrary.

Figure 11: The lossy channel I/O automaton $C_{i,j}^{re}$

Signature:

Input:
$deliver(m)_{i,j}$, $m \in M$

Output
$pre\_receive(m)_j$, $m \in M$

Internal
$anon\_drop(m)_j$

State Variables:
queue, a FIFO queue of messages, initially empty.

Transitions:

$deliver(m)_{i,j}$:
Effect:
add m to *queue*.

$pre\_receive(m)_j$:
Precondition:
m is the first element of *queue*.
Effect:
remove the first element of *queue*

$anon\_drop(n)_j$:
Precondition:
queue has at least n elements.
Effect:
remove the nth element of *queue*.

Tasks:
Arbitrary.

Figure 12: The *anonymizer* I/O automaton A_j^{re}

C TPM Interface Module Documentation

The Erlang tpm_if module is designed to run in conjunction with the C-to-Erlang TPM interface module ei_tpm.

C.1 Module Description

The Erlang tpm_if module is designed to run in conjunction with the C-to-Erlang TPM interface module ei_tpm. The tpm_if module should be run from an erlang process with the short name e1 and the shared cookie "secret". (For a shell, $ erl -sname e1 -setcookie secret) The ei_tpm program should be run first, on machine Hostname, and provided with an identification Number. (By default, the identification number is 99.) Initialize the communication between the C node and the erlang shell by calling tpm_if:init(Nodename), where Nodename is cNumber@Hostname. (The default nodename on host foobar would be c99@foobar.) If the interface is successfully initialized, other TPM commands can then be run.

In order for tpm_if and ei_tpm to function properly, both the Trousers daemon `tcsd` and the Erlang port mapper daemon `epmd` must be running.

C.2 Function Index

take_ownership/1 This function is used to take ownership of the TPM.

create_ek/0 This function creates an endorsement key.

create_identity/2 This function creates an identity key with minimal user input.

create_identity/4 This function creates an identity key with maximal user flexibility.

get_pubkey/1 This function retrieves the public half of a TPM RSA key.

get_quote/3

init/1 This function initializes the TPM interface, connecting the Erlang interface to the C back end.

read_key_blob/1 Utility function for easily reading key blobs from files.

store_key_blob/2 Utility function for easily storing key blobs.

verify_quote/5 This function determines whether a TPM Quote is legitimate (contains the correct PCR information and nonce, signed by the correct key).

C.3 Function Details

C.3.1 take_ownership/1

```
take_ownership(Authstring::string()) -> Result::int()
```

This function is used to take ownership of the TPM.

Input An string that will be used as the TPM's owner authorization.

Output An integer representing success or failure. Positive values are success, negative are failure.

C.3.2 create_ek/0

```
create_ek() -> Result::int()
```

This function creates an endorsement key.

Input None.

Output Result is an integer. Result value of 1 indicates successful creation of an Endorsement Key. A negative Result value indicates failure.

C.3.3 create_identity/2

```
create_identity(IdentLabel::binary(), Auth::atom()) -> Result
```

This function creates an identity key with minimal user input.

Input IdentLabel is a binary label that will identify the created identity in identity certificates and certificate requests. Authstring is the owner authorization atom.

Output Result contains a key handle for this Identity, or an error.

C.3.4 create_identity/4

```
create_identity(Flaglist::list(), Algorithm::atom(), Identlabel::binary(),
                Authstring::atom()) -> Result
```

This function creates an identity key with maximal user flexibility.

Input Flaglist is a list of key flags from the following set:

```
tss_key_no_authorization, tss_key_authorization,
tss_key_authorization_priv_use_only, tss_key_non_volatile,
tss_key_volatile, tss_key_not_migratable, tss_key_migratable,
tss_key_type_default, tss_key_type_signing, tss_key_type_storage,
tss_key_type_identity, tss_key_type_authchange, tss_key_type_bind,
```

```
tss_key_type_legacy, tss_key_type_migrate, tss_key_size_default,
tss_key_size_512, tss_key_size_1024, tss_key_size_2048,
tss_key_size_4096, tss_key_size_8192, tss_key_size_16384,
tss_key_not_certified_migratable, tss_key_certified_migratable,
tss_key_struct_default, tss_key_struct_key, tss_key_struct_key12,
tss_key_empty_key, tss_key_tsp_srk
```

Including a tss_key_type flag of a type other than identity will produce an error. Algorithm is an atom matching one of the following:

```
TSS_ALG_RSA, TSS_ALG_DES, TSS_ALG_3DES, TSS_ALG_SHA,
TSS_ALG_SHA256, TSS_ALG_HMAC, TSS_ALG_AES128, TSS_ALG_MGF1
TSS_ALG_AES192, TSS_ALG_AES256, TSS_ALG_XOR, TSS_ALG_AES
```

IdentLabel is a binary label that will identify the created identity. Authstring is the owner authorization atom.

Output Result contains a key handle for this Identity, or an error.

C.3.5 get_pubkey/1

```
get_pubkey(KeyBlob::binary()) -> {Exponent::binary(),Modulus::binary()}
```

This function retrieves the public half of a TPM RSA key.

Input An integer handle of a loaded TPM key or a binary TPM wrap key blob.

Output Two binaries representing the modulus and exponent of the key, or a string with an error message.

C.3.6 get_quote/3

```
get_quote(AIKBlob::binary(), PCRMask::list(), Nonce::binary())
          -> tuple()
```

This function generates a fresh TPM Quote, as well as the current values of individual PCRs in order to make later verification more useful.

Input PCRMask is a list of integers representing the registers to be quoted. Nonce is data used for freshness checking; it should be 20 bytes long. AIKBlob is the binary key blob of an identity key created on this TPM.

Output A tuple QuoteInfoData, SignatureData, PCRValues of the TPM quote, its signature, and the current PCR values from the selected registers.

C.3.7 init/1

```
init(Nodename::atom()) -> none()
```

This function initializes the TPM interface, connecting the Erlang interface to the C back end. This process should be spawned, not run, as it creates a long-lived server for accessing the TPM that other tpm_if commands will use.

Input A nodename of the form 'Nodename@Hostname', where Nodename is the nodename of the ei_tpm process (by default, c99) and Hostname is the short or long name of the machine on which the ei_tpm program is running.

Output None.

C.3.8 read_key_blob/1

```
read_key_blob(Filename::string()) -> Result
```

Utility function for easily reading key blobs from files.

Input Filename is the name of a file containing a stored key blob (see store_key_blob command).

Output Result is either the binary key blob, usable in all commands calling for a KeyBlob, or an error.

C.3.9 store_key_blob/2

```
store_key_blob(KeyBlob::binary(), Filename::string()) -> none()
```

Utility function for easily storing key blobs.

Input KeyBlob is a binary key blob, such as that produced by the create_identity command. Filename is the name of the file in which to store the key blob.

Output None

C.3.10 verify_quote/5

```
verify_quote(QuoteData::binary(), QuoteSig::binary(), Nonce::binary(),
             AIKPubMod::binary(), AIKPubExp::binary()) -> atom()
```

This function determines whether a TPM quote is legitimate; in other words, whether it contains the correct PCR information and nonce, signed by the correct key.

Input A TPM Quote blob. A list of expected PCR values. The expected input nonce. The public key of an Identity key that is presumed to have signed the nonce.

Output True if the Quote contains the input PCR vector and nonce, and is signed by the private key associated with AIKPub. False otherwise.