UNIVERSITY OF KANSAS

# Remote Attestation Protocol Synthesis and Verification with a Privacy Emphasis

by

Paul Kline

A thesis submitted in partial fulfillment for the
degree of Master in Computer Science

in the
Dr. Perry Alexander
Department of Electrical Engineering and Computer Science

June 2018

# Declaration of Authorship

I, Paul Kline, declare that this thesis titled, 'Remote Attestation Protocol Synthesis and Verification with a Privacy Emphasis' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ________________________________________

Date: ________________________________________

*"If you formulate a question properly, mathematics gives you the answer. It's like having a servant that is far more capable than you are. So you tell it 'do this,' and if you say it nicely, then it will do it."*

Savas Dimopoulos, Stanford University

UNIVERSITY OF KANSAS

# *Abstract*

Dr. Perry Alexander
Department of Electrical Engineering and Computer Science

Master of Computer Science

by Paul Kline

Remote attestation is innately challenging and wrought with auxiliary challenges. Even determining *which* information to request can be a challenge. In cases that a presumptuous request is denied, perhaps mutual trust can be built incrementally to achieve the same result. All the while, we must 1. Respect our own privacy policy (do not reveal more than we must about ourselves) 2. Respond to counter-attestation requests (build trust slowly) 3. Avoid "Measurement Deadlock" situations (avoid or handle cycles) In addition to these guidelines, there are basic properties of a remote attestation procedure we would like verified. For example, both sides of the remote attestation process should line up– i.e. the appraising party receives when the attesting party sends and vice versa. Using the theorem prover Coq we explore designing, modeling, and verifying a bipartisan remote attestation procedure via an imperative protocol language which supports dynamically generating execution steps to perform a mutually agreeable attestation protocol from nothing other than a party's initial privacy policy.

# *Acknowledgements*

I would like to thank Dr. Perry Alexander for his guidance throughout this research and Adam Petz for his perspectives, troubleshooting, and continued presence throughout a number of challenges.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **TPM** | **T**rusted **P**latform **M**odule |
| **PCR** | **P**latform **C**onfiguration **R**egister |

# Symbols

| | | |
|---|---|---|
| $\Rightarrow$ | one-sided protocol | single-step eval one |
| $\Rightarrow^*$ | one-sided protocol | multi-step eval |
| $\Rrightarrow$ | dual-sided protocol | single-step eval |
| $\Rrightarrow^*$ | dual-sided protocol | multi-step eval |

*For/Dedicated to/To my…*

# Chapter 1

# Preface

## 1.1 Introduction

In a world increasingly dominated by computers, we are forced to place more and more trust in them regardless of whether or not that trust is well-founded. Often, in fact, this trust has been misplaced or at best exaggerated. One reason for this that the supply of trust simply cannot keep up with the demand we have for computing and therefore has historically remained an afterthought. The trust problem is exacerbated by the trend of shifting operations from one's own machine to the cloud. The highly inter-machine nature of many operations complicates trust further. To address the need to establish trust in a remote machine, much research has been focused on how to execute *Remote Attestation*. In simple terms, it is the process of learning information about the state of a remote machine with high assurance that the information gathered is true.

Remote attestation can more formally be defined as "the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network" [1]. In this paper, we propose a remote attestation schema that dynamically generates remote attestation protocols from an appraiser's desired measurements and privacy policy and reason about this schema's correctness. This schema provides the following benefits.

- In traditional remote attestation, one party is the appraiser and the other is the attester who attests to the state of the target. Note that we will often use the terms *target* and *attester* interchangeably since one often attests to the state of one's own system. Static role assignment does not allow the attesting party to gain trust in the appraiser, only the inverse. The appraiser does not get the opportunity to gain the trust of an apprehensive attester, if necessary, to obtain all desired measurements.

This schema provides a way for *both* parties to incrementally build trust in each other.

- Other formal models of remote attestation require entire protocols to be defined statically in order to verify their properties. While enumerating all possible paths of execution for protocols which build trust incrementally is theoretically possible if the cardinality of all possible measurement requests is finite, this computation is an unreasonable prerequisite. For example, if there are 10 possible properties to request and an appraiser desires to know 3 of them, there are well over 100 million unique message sequences which must be accounted for (see appendix A for this calculation). To compute this type and store it statically is a significant burden. Such static protocols certainly have their use in this model, however. Specifically, we abstract over most of the detail contained within a single property request and response. Each property request can be viewed as a small, static remote attestation with a rigid type.

Instead of defining an entire protocol statically, we abstractly define the meaning of taking one step in a remote attestation instance. We can then prove properties of this single step which hold for every occurrence. Parties can repeat this step as many times as necessary. We then go on to prove various correctness properties about a single step as well as answering bigger verification questions such as, "will we ever stop taking steps?" and "will sides stepping individually create aligned sends and receives?"

## 1.2 Background

A practical use case for remote attestation occurs when a banking customer wishes to view their information remotely. It is in both the bank's and customer's best interests that this information remain confidential. Therefore prior to revealing sensitive information, an institution would prefer to be assured that the customer's computer has not been compromised in some way. For example, the bank may want assurance that the remote system is running an approved and updated anti-virus program on an approved version of an approved operating system.

Those unfamiliar with remote attestation protocols may immediately think, "What stops one from lying about the anti-virus to the bank? Or a virus taking over and attesting that everything is fine?" This is a real concern. But researchers have devised a way for the appraiser to determine with a very high degree of confidence that the values it receives are (a) recent and (b) really are the result of performing the requested measurement (modulo hardware tampering). This process requires the target system to have a Trusted

Platform Module (TPM) hardware chip. The properties of a TPM are numerous and will not be discussed in detail here (the specification is ~2,000 pages [2]). However, it is foundational for the ideas presented below to know of its existence and the basic services a TPM provides us when performing remote attestation.

### 1.2.1 TPM

The TPM contains Platform Configuration Registers (PCRs) that are housed within the TPM itself. These registers are not your typical read/write registers and can only be read/written to when certain access control requirements (localities) are met (determined by the TPM). The TPM is meant to be intimately involved in the boot process storing measurements of each action taken and process started in a special PCR that can only be modified during boot (see Intel's Trusted Execution Technology [3]). The TPM contains a secret hardware key that certifies child keys used to sign the contents of PCR registers upon request. Therefore after boot in software, an appraiser can request a signed data packet containing the requested PCR value(s) that can be compared to known "golden" hashes to gain assurance that the system is not running rouge software and only what we have pre-defined as acceptable. The target receives this request and forwards it to the TPM which returns the values in a signed message for the target system to report. Because the target does not have access to the private keys used by the TPM to sign the values, the target cannot modify the contents of the message without detection. We can continue chaining trust outward from the boot process by also storing the hash of a *measurer* program which will be executing the measurement requests received from an appraiser. The evidence sent to the appraiser includes the quote from the TPM regarding the measured values as well as the measured hash of the measurer program itself. With this information, the appraiser can examine the evidence, see that it came from a real TPM, and be confident the measured values are true. A more in-depth view of TPMs can be found in The ten-page introduction to Trusted Computing [4].

## 1.3 Remote Attestation

The procedure discussed in this paper requires the presence of a TPM in both the attesting and appraising parties. Particular measurement and evidence evaluation methods are abstracted over in this paper since they are their own subjects of research. This includes the process of starting the trusted measurer program.

Encapsulating these ideas, whenever we refer to the value of a measurement, know that this is, in reality, a complex value which contains much more information that the primitive values used in this abstraction (containing hashes, signatures, etc). Similarly, the evaluation of evidence is much more complicated and involves verifying keys; establishing, storing, and trusting a database of golden hashes; and establishing trust in one's own evidence evaluator program.

To (ironically) give more detail on this abstraction, we encapsulate the process by which an appraising party evaluates evidence provided in response to a measurement request by representing such an evaluation as a comparison of primitive values e.g. $x? = 4$. We will also be assuming that all communication occurs in an encrypted session e.g. SSL. With these abstractions in mind, we can focus this research on the higher level protocols which implement the listed abstractions. We focus on what kinds of messages are sent back and forth and the procedure to arrive at a mutual state of trust.

As a general rule, one does not reveal more about oneself than necessary during the process of remote attestation. Therefore, a predetermined sequence of messages is insufficient to perform *mutual* remote attestation simply because one cannot know the privacy policy of the other party to know which order to request which measurements. For example, say appraiser $A$ is a wireless access point which requires a singular measurement $x$ of attesting party $B$, a client wishing to connect. Many clients acting in the role of $B$ may be willing to reveal $x$ without any counter-appraisal of entity $A$ and readily perform and provide the measurement resulting in a grand total of two messages. However, there may be an occasional apprehensive client who only reveals property $x$ when property $y$ is obtained and scrutinized from the requesting party. This in turn may result in a counter-counter-appraiser from entity $A$ if $A$'s privacy policy dictates property $y$ is reveal-able only if property $z$ is measured and scrutinized. Obtaining the initially desired measurement $x$ can be possible, though the exact sequence and quantity of messages must be prescribed dynamically from the privacy policies of the involved parties. We will refer to protocol *satisfiability* as an appraiser receiving measurement values for all initially desired measurements while both parties abide by their privacy policies. A satisfiable protocol may be unsatisfiable in a static context. Instead of attempting one large step, trust can be gained incrementally with an end result equivalent to the result of a static protocol. Note that sharing one's privacy policy with the requesting party may be possible (given the privacy policy allows its own sharing) to short-circuit the initial banter of requests by computing a resolving protocol. However this does not solve the general case since revealing one's privacy policy is considered a breach of privacy. Therefore we do not discuss this optimization further. Our procedure computes the same resolution dynamically.

Let us examine another use case involving a customer and bank. In figure 1.1, a bank is

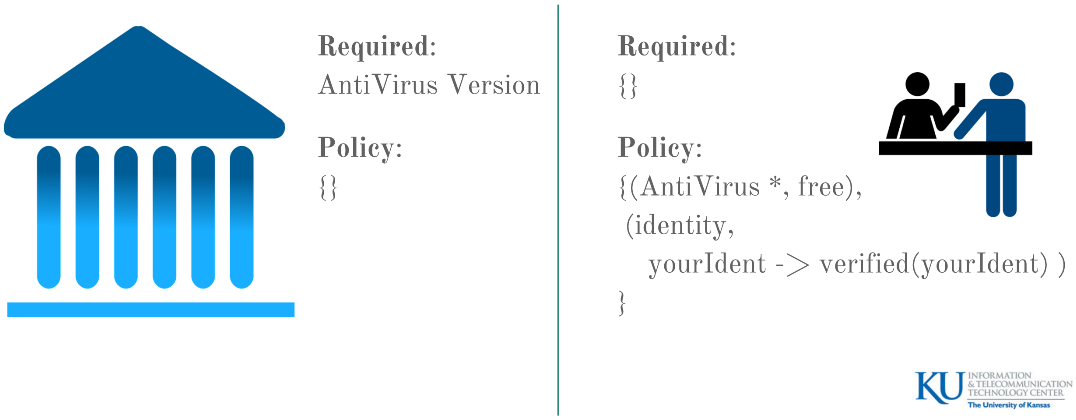



FIGURE 1.1: Example 1 Privacy

requesting the anti-virus version of its customers when they wish to interact with their accounts. The empty privacy policy of the Bank indicates that under no circumstances will it reveal measurements about itself. If no rule is listed for a measurement the default behaviour is to deny the request. The customer has no initial measurement requests to ask of the appraising party. In the customer's privacy policy, we see that any information regarding anti-virus software will freely be given without verifying anything about the requesting system. The identity of this customer is only revealed once the identity of the requester is known and passes the 'verified' test. In figure 1.2, the customer Bob begins

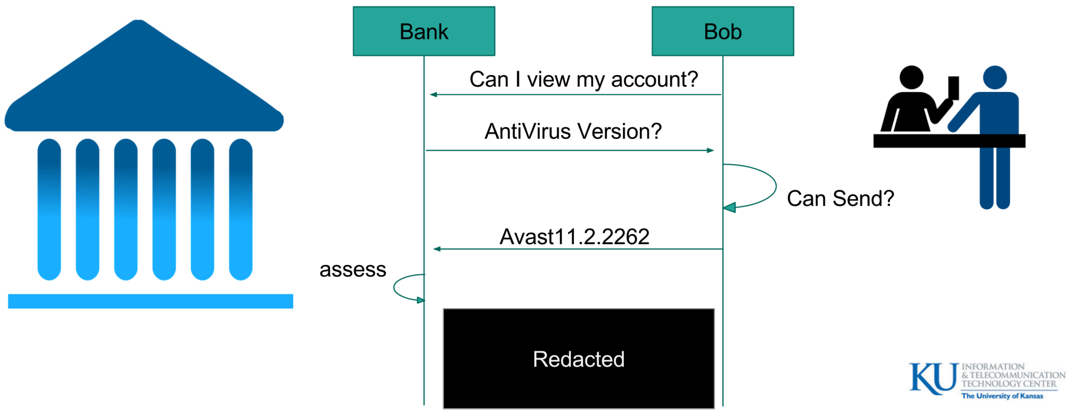



FIGURE 1.2: Example 1 Protocol

the conversation. In response, the bank requests to know the customer's anti-virus version. Bob checks his privacy policy to confirm he is able to reveal this information, performs the measurement, and responds. The bank assess the evidence provided, is satisfied with the values, and begins the process of sharing the sensitive information with a higher degree of certainty of the information's safety than without performing remote attestation.

We can make the example slightly more interesting if Bob first wants to confirm the identity of the requester before revealing his information as we see in figures 1.3 and 1.4.

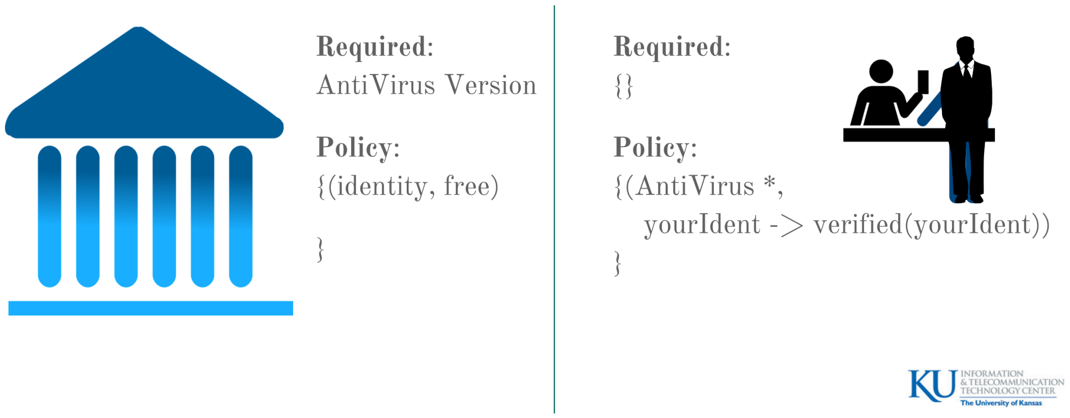



FIGURE 1.3: Example 2 Privacy

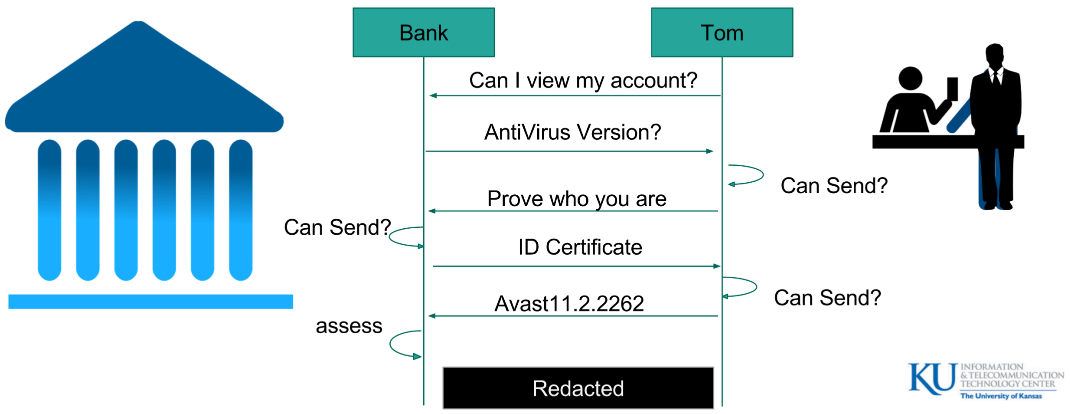



FIGURE 1.4: Example 2 Protocol

A circumspect reader may have noticed a fatal flaw in what we have seen so far. What if we have the following slightly modified scenario in figure 1.5?

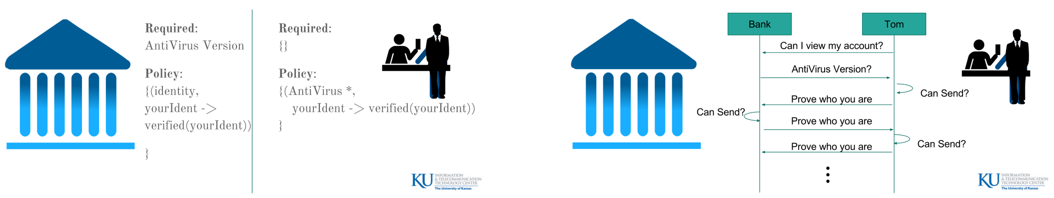



FIGURE 1.5: Infinite Regression

In figure 1.5, we have the notion of gaining incremental trust by countering requests with other requests, but as of yet we have no means of preventing such a regression. We refer to this situation as *measurement deadlock*. We will address this situation later.

In this section we have explored some use cases exemplifying the scenarios for which we wish to create a procedure handle and model using Coq. In summary, our remote attestation protocol must perform the following functions:

- Accomplish the initial attestation goal,
- Respond to counterattestation requests,
- Avoid *measurement deadlock* scenarios as the one in the previous example,
- “Line up" ie. sends and receives.

In the next chapter, we will explore a model addressing these requirements.

# Chapter 2

# Remote Attestation Procedure Representation in Coq

We have chosen Coq as our medium to model our protocol and procedure. We will first define a protocol language to represent all possible valid actions an attester or appraiser may take. The chaining together of these actions will constitute a party's *protocol*. We will then examine how each of these statements will be evaluated in what we call *one-sided* evaluation as opposed to the later defined *Dual evaluation* to refer to modeling the evaluation of an Attester and Appraiser's protocols simultaneously. In both cases, evaluation will be modeled via an inductive relation. The choice to use an induction relation for evaluation as opposed to a computational model is not made lightly as there are a number of benefits and drawbacks to either computational or relational models as discussed in detail by Pierce [5]. Ultimately, the benefits of a computational representation do not overcome its Achilles heel–its inability to examine individual terms for theorem proving. Most notably, pattern matching is only possible on inductive types and not possible for blocks of computation. Additionally, experience has shown that a computational representation tends to create incoherent and complex statements that are difficult to work with. Therefore an inductive representation of evaluation is well worth the cost of having to manually define evaluation. Once a protocol language is defined and one-sided evaluation explored, we can use our protocol language to construct a Statement we will refer to as a *single step*. The *single-step* protocol encapsulates the proper action for a party to take at each step of the remote attestation process. A party's entire protocol can then simply be defined as repeating the single-step protocol as many times as necessary. The combination of an Attester and Appraiser each evaluating a sequence of single steps while sharing a network constitutes the entire model of the remote attestation protocol procedure.

## 2.1 The Protocol Language Definition

The language we have constructed to model a single protocol is defined as an inductive data type below.

`Inductive` **Statement** :=
| SendStatement : **Term** → **Participant** → **Participant** → **Statement**
| ReceiveStatement : **VarID** → **Statement**
| EffectStatement : **Effect** → **Statement**
| Compute : **VarID** → **Computation** → **Statement**
| Assignment : **VarID** → **Term** → **Statement**
| Choose : **Condition** → **Statement** → **Statement** → **Statement**
| Chain : **Statement** → **Statement** → **Statement**
| StopStatement : **Statement**
| EndStatement : **Statement**
| Skip : **Statement**
| Wait : **Statement**.

This abstract syntax encompasses all possible actions that can be performed by one party during a remote attestation. Note that since this language is defined as `Statement` and it will be used to define protocols, we will sometimes use the terms 'Statement' and 'protocol' interchangeably. We will now explain the purpose of each `Statement` constructor.

### 2.1.1 Network Actions

The semantics of '`SendStatement`' and '`ReceiveStatement`' are relatively straightforward from their nomenclature. These are the only statements whose execution can modify the network state. Construction of a '`SendStatement`' requires three values: a `Term` to send (discussed later), the sender's identity, and the receiver's identity. '`ReceiveStatements`' require nothing more than a variable identifier which will be used to refer to the content of the received message.

### 2.1.2 Effectful Statements

An '`EffectStatement`' is the only statement form allowed to modify the execution environment. The effect type contains the following constructors:

`Inductive` **Effect** :=

| effect_StoreRequest : **Term** → **Effect**
| effect_ReduceStatewithMeasurement : **Term** → **Effect**
| effect_ReducePrivacyWithRequest : **Term** → **Effect**
| effect_MvFirstDesire : **Effect**
| effect_rmFstQueued : **Effect**
| effect_cp_ppUnresolved : **Term** → **Effect**
| effect_setAllGood : **AllGood** → **Effect**.

A ‘`StoreRequest`‘ modifies the executor’s state, preppending the measurement request to the current request queue. Storing multiple requests in the state at the same time is necessary to support counter-attestations for incremental trust. For example, when an attester responds to a measurement request with a counter-measurement request and receives another measurement request as the response, the attester must remember both of these requests and fulfill them once the prerequisites have been met. The length of this queue is bounded by the total number of possible unique requests. This is due to the fact that we will always reject a measurement request if a measurement request is received twice within one attestation instance. We will discuss this useful design choice in more detail later.

The ’`ReduceStatewithMeasurement`‘ constructor represents the effectful computation of modifying the executor’s state when the received message is a measurement value. We can then ‘reduce’ our execution environment in the following ways:

- The current list of unresolved requests is simplified with the result. i.e. the request for that value is no longer pending. One of two things can happen after receiving a measurement: Either the measurement meets our requirement imposed upon it or it does not. The result of scrutinizing the received value is reduced to setting a Boolean flag in the execution state appropriately named ‘`AllGood`’ which can be set to ‘`Yes`’ or ‘`No`’ depending on if the requirement was met respectively. We will later prove that modification of this flag is not possible by any ‘unauthorized’ `Statements`. This flag is checked by the ‘single-step’ protocol, and an `AllGood` state of `No` will trigger the termination of the remote attestation procedure.

- The privacy policy is reduced. Each remote attestation that an attester endures begins with a ‘fresh’ copy of its privacy policy. Throughout the attestation process, the attester may become more willing to share information about itself when more knowledge is gained about the requesting party. Therefore, any privacy policy rules which request the received value for some value’s release are lessened or severed depending on if that measurement meets each specified requirement. If the value does not meet the Privacy Rule's Requirement, we indicate that the measurement behind that rule should *never* be revealed should it by chance subsequently be requested. If the requirement is met, we remove that requirement from the rule and replace it with `Free`, easing in its possible subsequent release.

- The list of measurement requests yet to be requested must also be reduced for the received measurement. It is possible that the value received was in response to a counter attestation which also happens to be an initially desired measurement. For example, an appraiser has the following initial requests $\{A, B, C\}$. The appraiser first sends request $A$. In response to $A$, the appraiser does not receive the measurement, but a counter-attestation for measurement $X$. The appraiser's privacy policy dictates measurement $C$ must be known before releasing $X$ so a counter-*counter* attestation is sent to the attester asking for measurement $C$. If the attester obliges and sends the value $c$ for request $C$, not only must we reduce the privacy policy requirement to possibly release $X$ which we must do for the appraiser to receive the initial goal measurement $C$, but we have received a value that by chance we were going to request anyway. Therefore the requirement in the initial request list imposed by $C$ is also applied to measurement $c$ and $C$ is removed from our list of measurements to request.

  Not only does the above save time, but is required by our schema to maintain satisfiable protocols. A simple way to avoid Measurement Deadlock is for each party to always deny multiple requests for the same measurement should it occur more than once in a remote attestation session. Therefore, each party should ensure that they only request measurements as few times as possible.

The '`ReducePrivacyWithRequest`' statement will amend the executor's privacy policy such that should the received request be received again later, it will immediately be denied. The single step protocol executes this statement when fulfilling a request. This prevents 'Measurement Deadlock' cycles from occurring. Note that when a party receives a measurement value, the *entire* execution state is reduced including any potential privacy policy requirements in addition to pending requests of the other party. This ensures that a party will not request measurements more times than necessary.

'`MvFirstDesire`' is called after a measurement request is sent. The effect is simply to move the request from the `toRequest` list into the `unresolved` requests list.

'`cp_ppUnresolved`' represents the effect when a received measurement request cannot be fulfilled without the success of a counter measurement request as indicated by the privacy policy. In this case, the needed measurement request is simply added to the `toRequest` list.

'**setAllGood**' is used throughout the one-step protocol to indicate success or failure of the current branch.

### 2.1.3 Computations

Computations are encapsulated within this Statement type. Note that they do not effect the state other than bringing a variable into scope. The possible computations are defined as follows:

`Inductive` **Computation** :=
| compGetMessageToSend
| compGetNextRequest
| compGetfstQueue.

'`compGetMessageToSend`' will perform the requested measurement. '`compGetNextRequest`' can simply be thought of as a getter which returns the next measurement to request from the other party. Similarly, '`compGetfstQueue`' is simply a getter for the first stored measurement request received in the executor's state.

### 2.1.4 Assignment

Occasionally, a 'vanilla' assignment is needed which this statement provides.

### 2.1.5 Choose

This is our one and only branching construct. Upon evaluation of the `Condition`, either the first `Statement` is executed or the second.

### 2.1.6 Chain

A `Statement` can be multiple `Statements`.

### 2.1.7 Termination

Termination is indicated by both the `StopStatement` and the **EndStatement**. `StopStatement` indicates the early termination of the protocol due to a measurement request being denied or a requirement not being met. The `EndStatement` indicates that there is simply no more protocol to run. The distinction is very important when defining how to append two protocols into a single protocol. Branches that end in `StopStatement` are not appended to, only `EndStatements`.

### 2.1.8 Skip

`Skip` has no clear purpose until we talk about evaluation of protocols. We use Skip to indicate successful execution of a statement. This is necessary since our evaluation relation will be from $Statement * State \rightarrow Statement * State$ which differs from the more common evaluation of $Statement \rightarrow State$. The reason for this difference should become clear later and is due to our necessity to include some sort of network in our model.

### 2.1.9 Wait

It is possible that a Receive statement will not succeed (that is, evaluate to a `Skip`) if no message is present in the network. It is for this reason that evaluation leads to another statement rather than solely a new state. When a `Receive` is encountered with no corresponding message present in the network, a `Wait` Statement is prepended onto our statement chain which can only be removed by an evaluation rule which ensures the presence of a message. We will provide two evaluation relations: a one-sided evaluation relation which will simplify a `Statement` chain as far as it can, i.e. until an `End`, `Stop`, or `Wait`. We will also need a "larger" evaluation relation to simplify an entire protocol execution which includes both parties `Statement` Chains along with a network. We will call the "large" evaluation relation `DualEval`. `DualEval` will mimic parallel, isolated execution by alternating execution between sides of the protocol when a `Stop`, `End`, or `Wait` is encountered. We will later prove that our method of generating Statements for our protocol will never end in a party waiting for a message. We will always end with both parties at one of `StopStatement` or `EndStatement`.

Now we we have discussed the purpose of each statement, we will now view in more detail how these statements are evaluated in the one-sided evaluation.

## 2.2 One sided Evaluation

For brevity, we will refer to the statements `SendStatement`, `ReceiveStatement`, `StopStatement`, and `EndStatement` as `Send`, `Receive`, `Stop`, and `End` respectively.

### 2.2.1 Chaining

Perhaps most importantly we need to define how a chain is evaluated.

$$\frac{(stm1, st, n) \Rightarrow (Skip, st', n')}{(stm1 \gg stm2, st, n) \Rightarrow (stm2, st', n')} \qquad \text{(EvalChain)}$$

The above notation to states, "Given $stm1$ under state $st$ and network $n$ evaluates ($\Rightarrow$) to $Skip$ with state $st'$ and network $n'$, then the chain of statements $stm1 \gg stm2$ under $st$ and $n$ evaluates to $stm2, st', n'$. We ensure that previous command succeeds before evaluating the next statement and propagate the new executor state and network. It is possible that evaluation of $Stm1$ does not evaluate so nicely. In this case we circumvent execution of the remainder of the chain with the following rule.

$$\frac{(stm1, st, n) \Rightarrow (Stop, st', n')}{(stm1 \gg stm2, st, n) \Rightarrow (Stop, st', n')} \qquad \text{(EvalChainBad)}$$

If $stm1$ evaluates to a bad state indicated by **Stop**, any statements that were to follow $stm1$ are removed and not evaluated. **Stop** has no further evaluation rules and evaluation ends.

### 2.2.2 Send

We have two rules for evaluating a send. The following applies to sending any message other than a `StopMessage` abbreviated to `Stop`:

$$\frac{st[t \mapsto c] \wedge c \neq Stop}{(Send\ t, st, n) \Rightarrow (Skip, st, (c :: n))} \qquad \text{(EvalSend)}$$

Sending a **Stop** is handled in the following rule:

$$\frac{st[t \mapsto c] \wedge c = Stop}{(Send\ t, st, n) \Rightarrow (Stop, st, (c :: n))} \qquad \text{(EvalSendStop)}$$

With *EvalSendStop*, we prevent anyone from sending a `Stop` unless they plan on immediately stopping. In this way we can easily guarantee that the other side has stopped if one side receives a `Stop`. By choosing a relational model and encoding this requirement in the evaluation, we essentially get this proof for free since it is impossible to send a `Stop` without stopping oneself. In both evaluation rules for `Send`, the state remains unmodified.

### 2.2.3 Receive

`Receive` has three evaluation rules. We use the function $\rho(n, st)$ to denote the message for participant with state $st$ on network $n$.

Like the `Send` evaluation rules, we have a special case for receiving a `Stop`. If the network provides a `Stop` message, receiving evaluates to `Stop` statement such that the state is only modified with the special variable $R$, the most recently received message, set to `Stop`. The network is untouched other than the message being removed.

$$\frac{\rho(n, st) = Stop}{(Receive\ v, st, n) \Rightarrow (Stop, st[R \mapsto Stop], n - \rho(n, st))} \qquad \text{(EvalReceiveStop)}$$

If a **Receive** is encountered before a message exists for the participant, a **Wait** is inserted.

$$\frac{\rho(n, st) = \bot}{(Receive\ v, st, n) \Rightarrow (Wait >> Receive\ v, st[R \mapsto Stop], n)} \qquad \text{(EvalReceiveWait)}$$

This is yet another example for why a relational definition is preferred to the computational. Coq requires proof of termination for every functional definition. A functional definition of *eval* described thus far would be a significant burden on the designer since in the `Receive` case, the fix-point argument can actually *grow*. Such a definition is not impossible, but requires a lengthy manual proof to establish termination. This option was explored when modeling computationally, but was never completed before the switch to a relational model was made. Perhaps in the future, Coq will be able to deduce proof of termination from non-trivially reducing arguments. In the meantime, it is only for the truly desperate.

Finally in all other cases, we receive as normal.

$$\frac{\rho(n, st) = m}{(Receive\ v, st, n) \Rightarrow (Skip, st[R \mapsto m], n - \rho(n, st))} \qquad \text{(EvalReceive)}$$

### 2.2.4 Wait

As stated, a `Wait` can be removed once a message is ready to be received. The state and network are preserved.

$$\frac{\rho(n, st) = m}{(Wait, st, n) \Rightarrow (Skip, st, n)} \qquad \text{(EvalWait)}$$

Since evaluation is defined manually, we also need a case for propagating a `Wait` in a `Chain`.

$$\frac{(Stm1, st, n) \Rightarrow (Wait \gg Stm1, st', n')}{(Stm1 \gg Stm2, st, n) \Rightarrow (Wait \gg Stm1 \gg Stm2, st', n')} \quad \text{(EvalWait)}$$

There are 15 evaluation rules in total. The above listed are the most relevant for understanding our attestation protocol. Not listed are the evaluation rules for **Effect**, **Compute**, **Assignment**, and **Choose**–none of which contain anything enexpected.

## 2.3 Attestation Protocol

Our protocol will be constructed by repeating a sequence of one-step protocols. We will define our attestation protocol by first defining what it means for one party to take one step.

### 2.3.1 One Step

A single (one) step denotes a single network action of `Send` or `Receive` in addition to any computation or effects needed. By composing our protocols this way, we can prove that adjacent steps always alternate between sending and receiving unless sending or receiving a **Stop** in which case we terminate. Therefore, if both participants' protocols are composed of nothing other than single steps and one party initially sends and the other receives, all possible protocols will line up.

The first `Statement` in a one-step is a `Choose` which examines the state to see if the intention of the current step is to send or to receive. Initially, the appraiser will send the first message containing the first measurement request. The target's first step is to receive. This can be accomplished by every party running a server which listens for attestation requests. For a party to move from one step to the next, it is required that the action to perform in the state (send or receive) is flipped except in the cases of sending or receiving a **Stop** in which case no further actions are taken.

**Sending State** If the state tells us we must send, we will send *something*. The first action is to check if we have any current measurement requests queued up that we have yet to fulfill. If there is at least one, we check if we can answer the measurement request under our current privacy policy. If we can, we will preform the measurement and send it, completing our single step. There are two cases for which a privacy policy would not allow a measurement to be taken: reconcilable or irreconcilable. The measurement request is irreconcilable if there is no level of trust which would allow the desired measurement to be taken. This could have been set initially as a very strict privacy policy, or could have been the result from simplifying a privacy policy with a received measurement value which does not meet the requirement. If we are irreconcilable, we send a `Stop` since the request in unfulfillable. Otherwise, we respond with a measurement request that when resolved could soften our privacy policy to reveal the initially requested value.

We have now answered what we will send under all subcases of having a measurement request queued in the state. If there are no pending requests, we check our own list of desired measurements to see if there is anything more we would like to request of the other party. If there are no more, we send `Stop` to indicate that we are done. Otherwise, we make the request and remove it from our list.

The above case of sending a `Stop` may at first seem troubling since it appears that the target could short-circuit the attestation session by sending a **Stop** when it is its turn to send and there are no more requests to fulfill or make. Though we can be sure that this will never happen. The only difference we speficy between an Attester and Appraiser is that (a) the Appraiser initiates the conversation by sending the first measurement request, and (b) the Attester has no initial measurement desires. The only requests made by the target are spawned in response to requests from the appraiser. Therefore, a target is forever responding with measurement values or measurement requests in effort to release requested measurements from the privacy policy. Therefore, there is always a pending request from the appraiser otherwise the attestation session would have been ended by the appraiser by now.

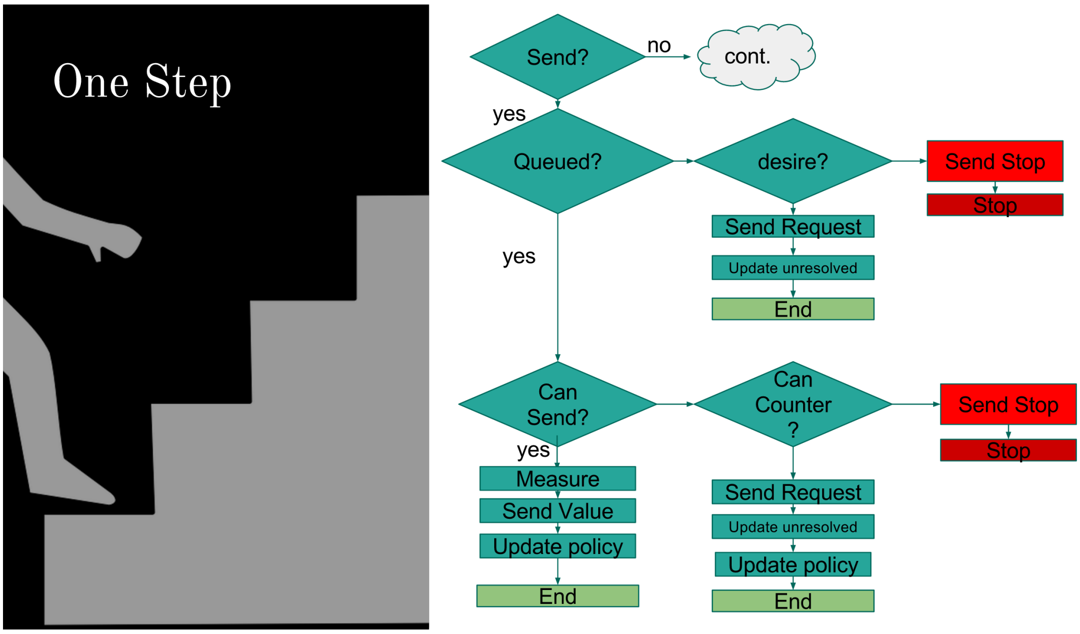



Figure 2.1: Control flow when a one-step protocol sends

**Receiving State** The execution path taken if we are to receive depend on the contents of the message. There are only three possibilities: a measurement value, a measurement request, or a stop message was received. If we receive a measurement value we reduce our state accordingly by relaxing our privacy policy to release a pending measurement request and/or reduce our own pending requests of the other party. If the measurement does not meet our requirement, we modify our state to trigger a `Stop` message in the next step to end the attestation session. The justification for this is as follows. If we receive a measurement value, it is in response to requesting it. The two ways we could have requested it are (a) it was part of our initial desires as an appraiser or (b) as a counter request in response to some number of steps ago an initial request. In either case if the measurement value does not satisfy our condition attached to it, the desire of some party is unsatisfiable and attestation terminates.

We should address the possibility that an Appraiser is flexible and wants to know just one of several measurements which is not handled by the above short circuiting action which would end in mutual termination. An 'or' may very well be needed, and the solution is simple. Each set of measurements desired by the appraiser logically separated by an 'or' are each split up into their own sub-attestation protocol session instances. Then at a higher level, we can evaluate the satisfaction of one of them. Therefore, it is useful that this attestation protocol indicates the satisfiablilty of the initial requests as a whole.

If the message we receive is a measurement request, we store it in the state for the subsequent `Send` step to handle. In the last case that we have received a `StopMessage`, we stop– an indication that the appraiser has all the information desired or an indication that the other party has determined this session is unsatisfiable either due to (a) a request is unsatisfiable, or (b) a measurement value was unsatisfactory.

The entire unfolded definition of one step is the following.

```
Definition OneProtocolStep (st : State) : Statement :=
IFS IsMyTurntoSend THEN
  IFS IsAllGood THEN
    EffectStatement (effect_setAllGood Unset) >>
    IFS QueuedRequestsExist THEN
      IFS CanSend THEN
        Compute toSendMESSAGE compGetMessageToSend >>
        SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) >>
        Compute (variden 1) compGetfstQueue >>
        EffectStatement (effect_ReducePrivacyWithRequest (variable (variden 1))) >>
        EffectStatement effect_rmFstQueued >>
        EffectStatement (effect_setAllGood Yes) >> (*all good here! *)
        EndStatement
      ELSE (*Can't send and queued request exists *)
        EffectStatement (effect_setAllGood No) >> (* no, this is bad! *)
        SendStatement (const constStop) (getMe st) (notMe (getMe st)) >>
        StopStatement (*Give up!*)
```

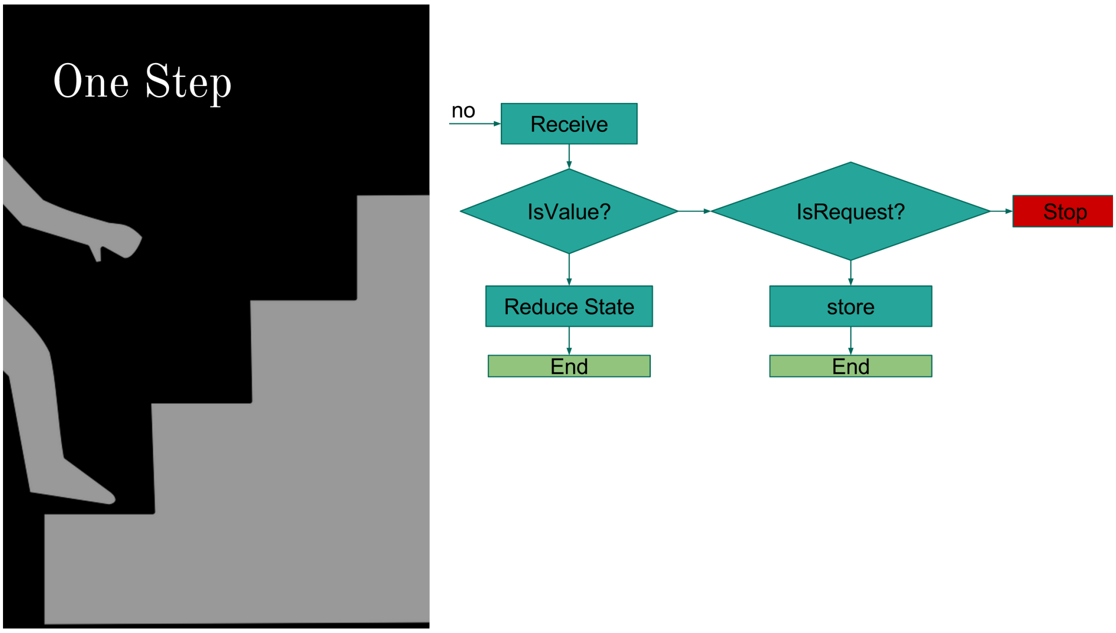



FIGURE 2.2: Receiving step continued from previous

```
    ELSE (*No queued up things for me. So I can continue down my list of things I want. *)
      IFS ExistsNextDesire THEN
        Compute toSendMESSAGE compGetNextRequest >>
        EffectStatement effect_MvFirstDesire >>
        SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) >>
        EffectStatement (effect_setAllGood Yes) >>
        EndStatement
      ELSE (* I must send, nothin queued, nothin left I want, quit! *)
        EffectStatement (effect_setAllGood Yes) >> (*all is well, just out! *)
        SendStatement (const constStop) (getMe st) (notMe (getMe st)) >>
        StopStatement
  ELSE
    SendStatement (const constStop) (getMe st) (notMe (getMe st)) >>
    StopStatement
ELSE
  ReceiveStatement (receivedMESSAGE) >>
  IFS (IsMeasurement (variable (receivedMESSAGE))) THEN
    EffectStatement (effect_ReduceStatewithMeasurement (variable (receivedMESSAGE)) ) >>
    EndStatement
  ELSE
    IFS (IsRequest (variable (receivedMESSAGE))) THEN
      EffectStatement (effect_StoreRequest (variable (receivedMESSAGE))) >> EndStatement
    ELSE (*we must have received a stop *)
      StopStatement.
```

Note that a single-step protocol is constructed from a state argument. In this way, a participant constructs a new step from their old state each time the previous single-step evaluated in an `End`. The following properties ensure that step repeating is always valid. ‘Valid’ in this context means that no party sends a message out of turn or attempts to receive more messages than are or will be sent.

- A *OneProtocolStep* will always perform exactly one send or exactly one receive. A formal proof of this can be found in the next section.
- For a remote attestation, the appraiser sends the first message. That is to say, the appraiser's first single-step is to send, while the attester's is to receive. The first step of both parties line up.
- A new step is only created if the previous step ended in a good state. Recall `End` is a good terminator and `Stop` is our bad/abrupt terminator.
- The only way for *OneProtocolStep* to end with a `Stop` instead of an `End` is either
  - a `Stop` was sent
  - a `Stop` was received.
- Every time a party creates a new step, the network action to take is flipped from the previous step's state.
- If a `Stop` is sent, the sender does not create a new step because `Send Stop` evaluates to a `Stop` as enforced by the evaluation rule *EvalSendStop*.
- If a `Stop` is received, the receiver does not create a new step because `Receive Stop` evaluates to a `Stop` as enforced by the evaluation rule *EvalReceiveStop*.
- It is impossible for the two parties to get out of step since each step is not computed until the previous completes and with the previous' result. That is to say, if a party sends any message other than `Stop`, it will always create a next step which receives. It cannot work ahead because all further action depends on the value of the message received. If a party receives any message other than `Stop`, it will always switch to send. The farthest two parties could get out of sync is an attempt to receive before a send from the other party. This scenario is safely captured by the $Wait$ evaluation rules.

### 2.3.2 Correctness of a Single Step

We will now discuss in more detail the correctness of the *OneProtocolStep*. Absolute correctness is often too high a standard to reach or even define–what does it mean to be correct? But there are particular properties about our model for which we can prove correctness and gain a higher confidence in the model's overall correctness.

#### 2.3.2.1 Language Properties

Before we examine the correctness of *OneProtocolStep*, we will first examine more general properties about the language and evaluation rules used to define it.

We can formally prove that **Effect** Statements are the only statements which modify the execution state. More formally, we state this as

```
Theorem thm_onlyEffect_effects : ∀ (stm stm': Statement) (st st': State) (n n' : Network),
 (stm,st,n) ⇒ (stm',st',n') →
 getProState st = getProState st' ∨ ∃ e, (headStatement stm) = EffectStatement e.
```

*getProState* is an accessor so that we ignore any particular execution variable differences. In all cases, either the state remains unchanged, or the very first statement was an `Effect`. The full proof can be viewed in the git repository [6].

In a similar fashion, we can prove that `Send` and `Receive` Statements are the only Statements which affect the network.

```
Theorem thm_onlySendOrReceiveChangesNetwork : ∀ (stm stm': Statement) (st st':
State) (n n' : Network),
 (stm,st,n) ⇒ (stm',st',n') →
 n = n' ∨
 (∃ t p1 p2, headStatement stm = SendStatement t p1 p2) ∨
 (∃ vid, headStatement stm = ReceiveStatement vid)
 .
```

Either the networks are identical or a **Send** or **Receive** occurred.

We will not list all proofs here and now shift our attention to verifying properties about the single-step protocol.

#### 2.3.2.2 Single Step Properties

We have claimed that each side of our one-step will always line up. We now prove this formally. We begin by defining methods for counting the maximum and minimum number of network actions taken by a statement in our language.

```
Fixpoint countMaxNetworkActions (stm : Statement) : nat :=
```

```
match stm with
 | SendStatement x x0 x1 ⇒ 1
 | ReceiveStatement x ⇒ 1
 | Choose x x0 x1 ⇒ max (countMaxNetworkActions x0) (countMaxNetworkActions x1)
 | Chain x x0 ⇒ (countMaxNetworkActions x) + (countMaxNetworkActions x0)
 | _ ⇒ 0
end.
```

$countMinNetworksActions$ is defined similarly.

We then show that the maximum actions for $OneProtocolStep$ is equal to the minimum is equal to 1.

```
Lemma onestepProtocolmaxAction_eq_minAction : ∀ st,
countMinNetworkActions (OneProtocolStep st) = (countMaxNetworkActions (OneProtocolStep st)).
```

```
Theorem onestepProtocolmaxAction_eq_1 : ∀ st,
(countMaxNetworkActions (OneProtocolStep st)) = 1.
```

We would like to speak more specifically about the network action being taken. For example that the action is `Send`. We can define functions that count only sends and only receives. Using these, we can define an inductive type which certifies a particular statement to perform precisely one network action of either send or receive throughout all possible branches.

```
Inductive SingularNetworkAction :Statement → Action → Prop :=
 | s_Send (stm : Statement): (countMaxReceives stm) = 0 ∧
                             (countMinSends stm) = 1 ∧
                             (countMaxSends stm) = 1 → SingularNetworkAction
stm ASend
 | s_Receive (stm : Statement): (countMaxSends stm) = 0 ∧
                             (countMinReceives stm) = 1 ∧
                             (countMaxReceives stm) = 1 → SingularNetworkAction
stm AReceive.
```

This new inductive proposition allows us to state our proofs clearly.

```
Theorem oneStepSend : ∀ st stm' n,
 evalChoose IsMyTurntoSend st = true →
```

$(OneProtocolStep\ st,\ st,\ n) \Rightarrow (stm',st,n) \to$
$SingularNetworkAction\ stm'\ ASend.$

`Theorem` $oneStepReceives : \forall\ st\ stm'\ n,$
$evalChoose\ IsMyTurntoSend\ st = false \to$
$(OneProtocolStep\ st,\ st,\ n) \Rightarrow (stm',st,n) \to$
$SingularNetworkAction\ stm'\ AReceive.$

We would like to formally prove that a single-step which receives will always end in a good state so long as the message was not a `Stop`. We introduce the notation $\Rightarrow^*$ to denote multistep evaluation.

`Theorem` thm_receiveAlwaysFinishes : $\forall$ *vars prst n m n'*, evalChoose IsMyTurntoSend (state *vars prst*) = false $\to$
receiveN *n* (getMe (state *vars prst*)) = Some (*m* , *n'*) $\to$
*m* $\neq$ constStop $\to \exists$ *prst'* ,
(OneProtocolStep (state *vars prst*), (state *vars prst*) , *n*) $\Rightarrow^*$ (EndStatement , (state ((receivedMESSAGE , *m*) : : *vars*) *prst'*) , *n'*).

In all cases that we receive a `Stop`, we will stop.

`Theorem` thm_oneStepProtoStopsWhenTold :$\forall$ *v p n n'*, evalChoose IsMyTurntoSend (state *v p*) = false $\to$
receiveN *n* (getMe (state *v p*)) = Some (constStop , *n'*) $\to$
((OneProtocolStep (state *v p*) , (state *v p*) , *n*) $\Rightarrow^*$ (StopStatement , assign receivedMESSAGE constStop (state *v p*), *n'*)) .

We have mentioned that a simple solution to avoid measurement deadlock situations is to simply remove items from one's privacy policy as they are requested. If there is no rule for a measurement, it is always denied. Removing an item, if present, from the privacy policy after it is requested ensures that the second time a measurement is requested, the request is denied. $handleRequestST$ is the function encapsulating this functionality. It modifies the privacy policy in the returned state to no longer contain an entry for the requested item description $d$. The fact that $findAndMeasureItem$ returns `None` is indicative of the fact that once a request has been handled, additional measuring is not allowed by the privacy policy.
`Theorem` thm_isRemovedFromPrivacyhandleST : $\forall$ *st d*,
findandMeasureItem (getPrivacy (handleRequestST *st d*)) *d* = None.

Another property we would like to formally prove is that we never send a measurement value without permission from the privacy policy. An exact formal proof of this is elusive. The difficulty stems from the fact the privacy policy check must occur in a Choose Statement which is “far away” from the actual send. Some simple proofs are difficult in proof assistants. Though we make the correctness argument below.

- Recall within the definition of $OneProtocolStep$:

```
IFS CanSend THEN
        Compute toSendMESSAGE compGetMessageToSend >>
        SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) >>
        Compute (variden 1) compGetfstQueue >>
        EffectStatement (effect_ReducePrivacyWithRequest (variable (variden 1))) >>
        EffectStatement effect_rmFstQueued >>
        EffectStatement (effect_setAllGood Yes) >> (*all good here! *)
        EndStatement
```

- $CanSend$'s evaluation performs the privacy policy check which we wish to associate with sending the associated measurement.
- Within this if branch is the only occurrence of $compGetMessageToSend$ which performs the measurement.
- Therefore, the only way to send a measurement value is if the privacy policy has first allowed it to be measured.

We have so far discussed only one-sided evaluation. In the next section we examine how both sides of the protocol procedure can be evaluated.

## 2.4 Dual Evaluation

The Dual Evaluation rules dictate how both sides of the protocol can be executed in tandem. In brief, we model the simultaneous execution of protocols by alternating a one-step evaluation as far as possible and then switching, repeating until both sides arrive at the `Stop` state. We will use the notation $\Big(\big[stmL, stL\big], \big[stmR, stR\big], n\Big) \Rrightarrow \Big(\big[stmL', stL'\big], \big[stmR', stR'\big], n'\Big)$ to denote that the left participant's statement chain $stmL$, state $stL$; right participant's statement chain $stmR$, state $stR$; both under network $n$ evaluate to left participant's statement chain $stmL'$, state $stL'$; right participants statement chain $stmR'$, state $stR'$; and new network $n'$.

### 2.4.1 Stepping

$$\frac{(stmL, stL, n) \Rightarrow^* (End, stL', n')}{\Big(\big[stmL, stL\big], \big[stmR, stR\big], n\Big) \Rrightarrow \Big(\big[oneStep(r(stL')), r(stL')\big], \big[stmR, stR\big], n'\Big)} \quad \text{(DualLeft)}$$

If the left side of the protocol multi-steps to an `End` then the pair evaluates to the left side starting a new step with the network action reversed and resultant network. We have a symmetric rule DualRight for evaluating the right side.

### 2.4.2 Stopping

The only way for two valid protocols to complete is if both sides do so simultaneously. That is to say that the total network actions taken by each side are the same and the final network is empty. We have already established that one party sending a **Stop** will result in both parties stopping.

$$\frac{(stmL, stL, n) \Rightarrow^* (Stop, stL', n') \land (stmR, stR, n') \Rightarrow^* (Stop, stR', n'')}{\Big(\big[stmL, stL\big], \big[stmR, stR\big], n\Big) \Rrightarrow \Big(\big[Stop, stL'\big], \big[Stop, stR'\big], n''\Big)} \tag{DualStopLeft}$$

Again, we have a dual for this rule where the right side finishes first and passes the network for the left side to finish. Every Remote Attestation/Dual Evaluation will end by a party sending a `Stop` message. Recall that a `Stop` is sent in response to one of the following state conditions:

- A measurement value $m_{v1}$ has been received that does not meet its associated requirement, $req\_m_{v1}$. Said requirement arrived in our state in one of two ways:
  - $req\_m_{v1}$ came from the initial list of measurements to request of the target. Therefore, the entity is the Appraiser and has learned enough negative information about the target to end Remote Attestation.
  - A measurement request for $m_{v0}$ was initially made, but the entity's privacy policy dictated that the value could not be released until $m_{v1}$ is known from the opposite party and meets requirement $req\_m_{v1}$. Note that it makes no difference if $m_{v0}$ was requested initially or as a counter request itself. In either case, the failure to meet the requirement propagates into the failure of a requirement on an initial request or the failure of a requirement which would eventually lead to the release of an initially requested measurement value.

  In either case, the Remote Attestation session terminates.

- The current pending request in the execution environment would irreconcilably violate the privacy policy.

- Finally, if the single-step's action is to send, yet there are no pending measurement requests to fulfill nor measurement requests to send, a `Stop` is sent to indicate termination (see One-step figure above). This could only be the Appraiser. We have already discussed that the Attesting party will always have something to send, otherwise attestation would have ended before that point.

Therefore, simultaneous **Stop**s is the only valid termination of Dual Evaluation.

### 2.4.3 Waiting

The rules `DualLeft` and `DualRight` do not address potentially valid evaluations that result in prepending a `Wait` to the protocol. Therefore we must have additional rules to eliminate `Wait`s. The following rule states that if the multi-step evaluation of the left side protocol results in network $n'$ and some Statement prepended with `Wait`, and the right side protocol, under the resultant $n'$ network, multi-steps to an `End` resulting in network $n''$, then our Dual evaluates to the result from the left side sans `Wait` and the right takes another step while reversing its network action. A symmetric rule exists for the right side.

$$\frac{(stmL, stL, n) \Rightarrow^* (Wait >> stL', stL', n') \land (stmR, stR, n') \Rightarrow^* (End, stR', n'')}{\Big([stmL, stL], [stmR, stR], n\Big) \Rrightarrow \Big([stmL', stL'], [oneStep(r(stmR')), stR'], n''\Big)} \tag{DualWaitLeft}$$

At first examination, these wait rules may appear to let the protocols become out of sync since the right side takes another step before the left side has finished the previous. Before we discuss the rule's validity, we must first introduce the two other Dual Wait rules we have so we can reference them.

$$\frac{(stmL, stL, n) \Rightarrow^* (Wait >> stL', stL', n') \land (stmR, stR, n') \Rightarrow^* (Stop, stR', n'')}{\Big([stmL, stL], [stmR, stR], n\Big) \Rrightarrow \Big([stmL', stL'], [Stop, stR'], n''\Big)} \tag{DualWaitLeftStopRight}$$

As always, there is a right version of this rule as well. We may wish to add another Dual Stop Evaluation rule to avoiding having to make the following progress-less evaluation valid: $(Stop, st, n) \Rightarrow^* (Stop, st, n)$

We know that the DualWaitLeft/Right rules are valid in creating another single step from the following argument.

- We can assume that the very first statements created were done so using the *oneStep* constructor.
- Additionally, one protocol's first action will be to send and the other will receive.
- We assume that our protocols are in sync up to this point (inductive argument).
- A `Wait` is only inserted on a receiving action. Therefore since the protocols are in sync, the right side, once evaluated, will send a message on the network. Recall that we have proved one steps will always perform exactly one network action.

- Since we know that the right side has ended in a good state by evaluating to `End` and not `Stop`, we know that the right side has not sent a `Stop` message. Sending a `Stop` would trigger the other side to stop and take no more steps.
- Therefore, we know the left side will at least send one more message since worst case scenario will send a `Stop` message, and the right side is safe to take another step which will receive.

Though this doesn't seem to address the out-of-stepness issue in full. We have only proved that these `Wait` evaluation rules will not end up creating two protocol sides with differing numbers of network actions. The rest of the proof is straightforward: After the application of the rule $DualWaitLeft$(or Right) we have two choices: we can choose to evaluate the left side as far as it can go or the right side. If the right side is chosen, it will soon be forced to wait on the evaluation of the left side since we have yet to evaluate the left side to send a message the right is expecting, or even evaluate the left side far enough to receive the last message we were waiting for. If we are now forced to evaluate the left side or chose to evaluate the left side first and avoid the right side `Wait` insertion makes little difference. The left side will soon evaluate to the end of its single-step. Again, the two possible final statements are `End` and `Stop`. If `End` is encountered, $DualLeft$ applies, and another step is created for the left side. The only way for a sending single-step to arrive at a **Stop** is to send a `Stop`, thus consuming the remaining step on the right side. Interestingly, it can be argued that these wait rules are not needed since carefully choosing DualLeft or DualRight can avoid all insertions of `Wait`. In practice, however, these waiting elimination rules prevent the need to backtrack in proofs. No matter which side is evaluated first, a valid evaluation path to completion exists.

## 2.5 Overall Correctness

Provable overall correctness is not achievable for any system. How could you prove that your definition of correctness is correct? What we can do, however, is prove individual properties about our system which creates a hostile environment for incorrectness to survive, thereby boosting our confidence in the system's overall correctness.

We have discussed a number of proofs already as they were relevant. We also get a number of smaller proofs for free by the expressive nature of Coq. For example, let us look at how we defined measurement values. Recall that a measurement value is the measurement taken in response to a measurement request. In Coq we use `Description` to refer to a valid property description.

```
Inductive Const :=
 | constValue (d: Description) : (measurementDenote d) → Const
 | constRequest : Description → Const
```

| constStop : **Const**.

The term `Const` is used to hold constants (values) as opposed to variables. Indeed the the three constructors listed above are the only three values needed in this schema since these are the three valid message types. A `constValue` actually holds two values: a `Description` and a value. The value is meant to be the result of measuring what was described. We take advantage of the expressiveness of Coq's specification language and create the type level function `measurementDenote` to *compute* the correct type that this measurement must be. For example, if `Description` $A$ is known to have a measurement type of `String`, it is impossible to create a `constValue` $A$ that holds a `Boolean`. We are guaranteed to contain a measurement value of the correct type. A note to Coq programmers: By placing the dependent type within a constructor rather than at the type level, we are able to list the simple type `Const` wherever it is used rather than dealing with the headache of not even being able to ask if two `Const`s are the same because they are of different types. Avoiding dependent types by wrapping them in a constructor is a common practice used throughout our Coq code to achieve some of the benefits of dependent types without most of the headache.

## 2.6 Step Termination

Since our schema involves repeatedly taking single steps, it is worthwhile to prove that in all cases we eventually stop taking the next step. Having a possibility for an infinite loop is no use to anyone. The proof is as follows:

- If any party receives a `Stop`, both parties cease taking steps.
- Therefore, we need only prove that eventually some party will send a **Stop**.
- What is sent is dependent on:
  - the list of currently requested measurements $Q_s$. If empty, we will send the first from
  - the list of desired property measurements $D_s$. If empty, we will send `Stop`
- We want to prove that each send step makes progress towards emptying these lists.
- Recall that we first check if $|Q_s| >= 1$. If so there are three possibilities:
  - The desired measurement is unobtainable. We send a **Stop**, finishing the proof for this case.
  - The desired measurement is obtainable and we take and send the measurement, shrinking $|Q_s|$ by 1 and finishing the proof for this case.

  - A counter-request is sent which must be fulfilled before the measurement is revealed.
  - In response, the counter-request can be:(i) denied, a `Stop` received;(ii) answered, a measurement value received, in which case either a `Stop` is sent for an unsatisfactory value or the desired measurement is taken and $|Q_s|$ shrinks by 1; or (iii) more interestingly, a counter-request could also be countered. Therefore we must prove that the depth of counter requests is finite. This would indicate that the initial request will eventually be answered with a stop or a value shrinking $|Q_s|$ by 1.
    * Let $P_p$ be the set of all properties in the requirements section of the executor's privacy policy.
    * Let $S_p = P_p \cup Q_s$
    * $|S_p|$ is finite since it is the union of two finite sets.
    * Then the maximum depth of counter-requests possible which hold back a single property measurement is $|S_p| - 1$. Recall that if a property is ever requested twice, the second request is always denied and a `Stop` is sent. Therefore the deepest possible chain of counter requests is $|S_p| - 1$. This will either result in a `Stop` being sent or the initial property request being measured and sent, shrinking $|Q_s|$ by 1.
  - In all cases, a `Stop` is sent or $|Q_s|$ shrinks by 1.

- If $|Q_s| = 0$, the first element from $D_s$ is sent, shrinking $|D_s|$ by 1.
- If both are empty, a `Stop` is sent.
- In all cases, a Stop is eventually sent, ending remote attestation.

# Chapter 3

# Future Work

## 3.1 Design amendments

This model is not yet perfect which is why some proofs remain elusive. It is not that they are not true under the current model–I believe that they are. However, perhaps some structure tweaking could get rid of some of the difficulties that have arose in theorem proving under this model. For example, The Dual Evaluation rules listed in the previous chapter are ambiguous: more than one rule could be chosen for evaluation for a given configuration. Not only that, but attempting to apply a Dual Evaluation rule that is doomed to fail is not immediately obvious in such a large proof state as this model creates. Attempting its application can lead to hours of fruitless time. Proof automation can help with this fruitless checking, but still needs to know which rules to apply. However, even with ambiguity removed, knowing which rules to apply is still non-trivial due to the heavy single-step evaluation needed in the preconditions. Once a better solution is found, proof automation could be used even more so than it is.

## 3.2 Extensions

This work could be modified into a 'negotiation' protocol which finds a mutually satisfiable remote attestation session *type*. The main difference would be a simplification in that measurements are not taken or sent upon request, but rather a placeholder indicating that, given all previously requested measurements in the protocol thus far meet their requirements, this particular measurement is releasable at this time. This could be desirable for two reasons.

Once we have a protocol type, simplification can occur to lessen network traffic. For instance if it is known that there are many targets with roughly the same privacy policy and 'baby stepping' turns out not to be necessary, all requests could be sent at once instead of request, measurement; request, measurement; etc.

The second reason has many of the prerequisites of the first. If no 'baby stepping' is necessary so request aggregation can occur, we can remove the short-circuiting functionality from an instance of a remote attestation session type. This may be desirable for a paranoid appraiser that considers it too revealing that just after receiving a measurement value it indicates to stop. If an attester were to remember the order of all messages exchanged it would be possible for the attester to deduce the precise measurement value the appraiser did not like–though it is still a mystery to the attester why the value was rejected. Without short-circuiting, the appraiser quietly makes a judgment call of the attester the result of which remains a complete mystery to the attester. Then the 'sensitive' information can begin with false values if it is undesirable that the attester be aware of the dissatisfied appraiser. Though it should be noted that this behavior is also exhibitable in the protocol's current form. An attester does not know if a stop message was received because the appraiser has no further requests or if the appraiser was dissatisfied with the most recently received measurement value.

Many defined inductive evaluation steps include an *exists* statement due to the multiplicative growth of evaluation proofs necessary to account for all evaluation cases. At the time of creation this was not seen as an issue, but proved (pun) to be one when attempting larger proofs which depend on the evaluation proofs. Ideally, the use of 'exists' would be stricken from all evaluation proofs and perhaps more significant properties could be verified.

As mentioned early on, many details are overlooked because they are out of scope of our focus (TMP quotes, encryption, etc). An obvious extension would be to incrementally add detail to this representation until fully instantiated.

The privacy policy is currently implemented with a one-for-one mentality. Meaning at most, you can only require one other measurement before releasing what was requested of you. Initially, we had an AND requirement and OR requirement. However, implementing this evaluation added undesirable complexity for this initial model. In reality, privacy policies must have these options.

# Chapter 4

# Conclusion

Here we have designed a protocol procedure for Remote Attestation which supports the integration of a dynamic privacy policy of both parties which adapt to the current level of trust in the opposing party. Additionally, the protocol procedure allows for trust to be built incrementally over a dynamically determined sequence of messages which allows for any number of counter attestations instead of attempting bulk requests. This system, properties, and evaluation were modelled within the theorem prover Coq. Specifically, we built the protocol language, its evaluation, and modeling valid execution of the protocols of both parties simultaneously. We also successfully proved a number of properties about this model. Some larger proofs we would have liked to have seen formally unfortunately remain informal verbal arguments based on smaller formal and informal proofs. I believe the main cause of this incomplete formality can be attributed to the following observations:

- A perceived correct though non-optimum model creates undesirable complexity in proof states.
- A non-expert Coq Programmer (me)
- Immature proving tools. This divisive opinion is my own. Though Coq has been around for over 30 years and is considered an industry standard in the field, if automated theorem proving is to the evolution of programming languages, I would say Coq is somewhere between C and C++. It has a long way to go before we see something as equivalently powerful as Haskell. For example, one of Coq's most powerful features, dependent types, is also its worst. In fact, some professional Coq programmers actively avoid and advice avoiding using them due to.. well their inability to be used. As a result, many elegant definitions feel like hacks that are in opposition to the design of the whole system rather than in sync. A proving system should aid in proving after all, not hinder it. Elegant definitions should live

on the surface instead of having to dig so deeply for them. Though in all fairness, could I design a better system? Most certainly not.

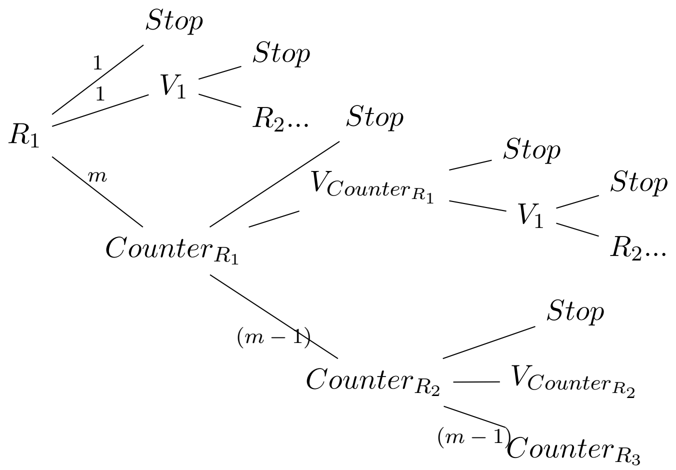
# Appendix A

# Cardinality of possible message paths

The formula for computing all possible communication paths for an appraiser and attester is as follows. Let $m$ be the cardinality of requestable properties and $n$ be the number of requests an appraiser would like to make. In response to a request, the appraiser may receive:

- a **stop** ending that path
- a **value** of the measurement, in which case we stop (if the value is bad) or continue on to the next property request.
- a **counter-request** which must be fulfilled before the desired property measurement is released.

The following tree provides a nice visual for the possible cases in response to $Request_1$.




We will not expand this and further. We have explored the tree far enough to approximate the cardinality in the following function. We can approximate the size of this tree in the following way:

```
size :: Measurements -> Requests -> Integer
size 0 _ = 2
size _ 0 = 0
size m n =  1 + value + counterRequest where
     value = 1 + (size (m-1) (n-1))
     counterRequest = m * (size (m-1) n)
```

If we recursively reach $m = 0$, our base case is 2: A **stop** or a **value**; no further measurements can be requested. If we reach $n = 0$, there are no more requests to be made. Otherwise, we could receive a **stop**, a **value**, or a **counterRequest**. As we saw in the tree, a value can be responded to with a **stop**, or continuing the attestation: thereby decreasing $m$ and $n$ by one. Remember that we remove the possibility of requesting a property twice to avoid Measurement Deadlock hence the $m - 1$. There are $m$ counterRequests we could receive. Note that the recursive call to *size* does not decrement $n$ since we have not yet received the value we desire and have not moved on. Example cardinalities include:

- $size(2, 1) = 14$
- $size(3, 1) = 44$
- $size(5, 2) = 1,930$
- $size(10, 3) = 114,190,992$

Hence it is certainly unfeasible to enumerate all possible protocol paths statically to arrive at well-typed protocol (Session Types).

# Appendix B

# Proof list

## B.1 Formal Proofs

### B.1.1 Network Related Proofs

---

**Description:** If findandMeasureItem returns a constValue, it is, in fact, the result of the initial request (d=d0).

**Formal Statement:** `Theorem` thm_findAndMeasureItemL : ∀ *pp d d0 val x*, findandMeasureItem *pp d* = Some (constValue *d0 val* , *x*) → *d* = *d0*.

---

**Description:** If an item has been removed from the privacy policy, findAndMeasureItemL will never succeed.

**Formal Statement:** `Theorem` thm_youCantFindit : ∀ *pp d*, findandMeasureItem (rmAllFromPolicy *pp d*) *d* = None.

---

**Description:** After handling a request, subsequent requests of that description will fail.

**Formal Statement:** `Theorem` thm_removedFromPrivacy : ∀ *pp d pp' c ri*,
handleRequest' *pp d* = (*pp'* , *c* , *ri*) →
findandMeasureItem *pp' d* = None.

---

**Description:** Ensures that we are, in fact, returning the head of the request list, if we canSend.

**Formal Statement:** `Theorem` thm_canSendL : ∀ *pp ls d*, (canSend *ls pp* = Some *d* )->
(head *ls*) =Some *d*.

---

**Description:** If a message exists on the network for the participant, then removing that message from the network results in shrinking the network in size by exactly 1.
**Formal Statement:** Lemma thm_rmMessSmallerL : ∀ $n$ $p$, existsMessageForMe $n$ $p$ → S (length (rmMess $n$ $p$))= length $n$.

---

**Description:** Calling `receiveMess` will shrink the network by exactly 1.
**Formal Statement:** Theorem thm_receivingShrinks' : ∀ $c$ $n$ $p$, receiveMess $n$ $p$ = Some $c$ →
length $n$ = length (rmMess $n$ $p$) + 1.

---

**Description:** The resultant network from calling `receiveN` is the same network returned by removing a message by calling `rmMess`.
**Formal Statement:** Theorem thm_receiveN_NewNetworkrmMessage : ∀ $c$ $n$ $n'$ $p$, receiveN $n$ $p$ = Some ($c$, $n'$) → $n'$ = rmMess $n$ $p$.

---

**Description:** The resulting network from calling `receiveN` shrinks the network by exactly 1.
**Formal Statement:**
Theorem thm_receivingShrinks : ∀ $n$ $c$ $p$ $n'$, receiveN $n$ $p$ = Some ($c$,$n'$) →
length $n$ = S (length $n'$) .

---

**Description:** The `Send` statement increases the number of messages in the network by 1. **Formal Statement:** Theorem thm_sendOnNetworkAppends : ∀ $f$ $t$ $c$ $n$, length (sendOnNetwork $f$ $t$ $c$ $n$) = length $n$ + 1.

### B.1.2 One Step Related Proofs

---

**Description:** The maximum number of network actions taken by any branch of `OneProtocolStep` is equal to the minimum number of network actions taken by any branch. i.e. All execution paths through `OneProtocolStep` perform exactly the same number of network actions.
**Formal Statement:** Theorem thm_onestepProtocolmaxAction_eq_minAction : ∀ $st$,

countMinNetworkActions (OneProtocolStep $st$) = (countMaxNetworkActions (OneProtocol-Step $st$)).

---

**Description:** The maximum number of actions taken by `OneProtocolStep` is 1. With the previous proof, we have that every `OneProtocolStep` will always perform exactly 1 network action.

**Formal Statement:** Theorem thm_onestepProtocolmaxAction_eq_1 : ∀ $st$, (countMaxNetworkActions (OneProtocolStep $st$)) = 1.

---

**Description:** If a `OneProtocolStep` is instructed to Send, the network action performed will be Send. With the previous proof, we know that if the step is to send, exactly one send will occur and no receives. Refer to 2.3.2.2 for the Propositional definition of `SingularNetworkAction`.

**Formal Statement:** Theorem thm_oneStepSend : ∀ $st$ $stm'$ $n$,
evalChoose IsMyTurntoSend $st$ = true →
(OneProtocolStep $st$, $st$, $n$) ⇒ ($stm'$, $st$, $n$) →
**SingularNetworkAction** $stm'$ ASend.

---

**Description:** Similar to the above proof, we prove that if the action is to Receive, exactly one Receive will occur and no Sends.

**Formal Statement:** Theorem thm_oneStepReceives : ∀ $st$ $stm'$ $n$,
evalChoose IsMyTurntoSend $st$ = false →
(OneProtocolStep $st$, $st$, $n$) âĞŠ ($stm'$, $st$, $n$) →
**SingularNetworkAction** $stm'$ AReceive.

---

**Description:** If `CanSend` $st$, which checks the privacy policy if the desired measurement can be taken and sent, evaluates to `true`, then computing the message to send will always succeed.

**Formal Statement:** Theorem thm_canSendST_implies_handleExists : ∀ $st$, evalChoose CanSend $st$ = true → ∃ $c$, handleCompute compGetMessageToSend $st$ = Some $c$.

---

**Description:** If there is still at least 1 property to request of the other party, then computing the message to send will always succeed.

**Formal Statement:** `Theorem` thm_ifwillthenway : $\forall$ *st*, evalChoose ExistsNextDesire *st* = true $\rightarrow\exists$ *c*, handleCompute compGetNextRequest *st* = Some *c* .

---

**Description:** As discussed in 2.3.2.1, this proof states that the `effect` statements are the only ones which modify the execution state.

**Formal Statement:** `Theorem` thm_onlyEffect_effects : $\forall$ (*stm stm'*: **Statement**) (*st st'*: **State**) (*n n'* : Network),

(*stm*, *st*, *n*) $\Rightarrow$ (*stm'*, *st'*, *n'*) $\rightarrow$

getProState *st* = getProState *st'* $\vee$ $\exists$ *e*, (headStatement *stm*) = EffectStatement *e*.

---

**Description:** Only the `Send` or `Receive` statements modify the network.

**Formal Statement:** `Theorem` thm_onlySendOrReceiveChangesNetwork : $\forall$ (*stm stm'*: **Statement**) (*st st'*: **State**) (*n n'* : Network),

(*stm*, *st*, *n*) âĞŠ (*stm'*, *st'*, *n'*) $\rightarrow$

*n* = *n'* $\vee$

($\exists$ *t p1 p2*, headStatement *stm* = SendStatement *t p1 p2*) $\vee$

($\exists$ *vid*, headStatement *stm* = ReceiveStatement *vid*)

.

---

**Description:** The `Action` to take, `Send` or `Receive`, is not modified by any statement in the language.

**Formal Statement:** `Theorem` thm_noOneTouchesAction : $\forall$ *stm stm' n n' st st'*, (*stm*, *st*, *n*) $\Rightarrow$ (*stm'*, *st'*, *n'*) $\rightarrow$ getAction *st* = getAction *st'*.

---

**Description:** From the previous proof, it follows that any multi-step also does not modify the `Action`

**Formal Statement:** `Theorem` thm_noOneTouchesAction_m : $\forall$ *stm stm' n n' st st'*, (*stm*, *st*, *n*) $\Rightarrow^*$ (*stm'*, *st'*, *n'*) $\rightarrow$ getAction *st* = getAction *st'*.

---

**Description:** When a `OneProtocolStep`'s action is to Receive and a `Stop` is not the received message, the `OneProtocolStep` always evaluates to `End`.

**Formal Statement:** `Theorem` thm_receiveAlwaysFinishes : $\forall$ *vars prst n m n'*, evalChoose IsMyTurntoSend (state *vars prst*) = false $\rightarrow$

receiveN *n* (getMe (state *vars prst*)) = Some (*m*, *n'*) $\rightarrow$

$m \neq$ constStop $\rightarrow \exists$ *prst'*,
(OneProtocolStep (state *vars prst*), (state *vars prst*), $n$) $\Rightarrow^*$ (EndStatement, (state ((receivedMESSAGE, $m$)::*vars*) *prst'*), *n'*).

# Bibliography


[1] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10 (2):63–81, 2011. ISSN 1615-5270. doi: 10.1007/s10207-011-0124-7. URL http://dx.doi.org/10.1007/s10207-011-0124-7.

[2] Trusted Computing Group. Tpm library specification, 2014. URL https://trustedcomputinggroup.org/tpm-library-specification/.

[3] Intel. *Intel Trusted Execution Technology Software Development Guide: Measured Launched Environment Developer's Guide.* Intel. URL http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[4] Andrew Martin et al. The ten-page introduction to trusted computing. *Computing Laboratory, Oxford University Oxford*, 2008.

[5] Benjamin Pierce, Chris Casinghino, Michael Greenberg, V Sjöberg, and B Yorgey. Software foundations. 2010. *Available in:< http://www. cis. upenn. edu/% 7ebcpierce/sf/current/>. Accessed on*, 30, 2015.

[6] Paul Kline. Remote attestation protocol synthesis. https://github.com/paul-kline/protosynth, 2017.

[7] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. In *Proceedings of the Third Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[8] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004.

[9] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[10] Zhidong Shen and Qiang Tong. The security of cloud computing system enabled by trusted computing technology. In *Signal Processing Systems (ICSPS), 2010 2nd International Conference on*, volume 2, pages V2–11. IEEE, 2010.

[11] C. J. Hawthorn, K. P. Weber, and R. E. Scholten. Littrow configuration tunable external cavity diode laser with fixed direction output beam. *Review of Scientific Instruments*, 72(12):4477–4479, December 2001. URL http://link.aip.org/link/?RSI/72/4477/1.

[12] A. S. Arnold, J. S. Wilson, and M. G. Boshier. A simple extended-cavity diode laser. *Review of Scientific Instruments*, 69(3):1236–1239, March 1998. URL http://link.aip.org/link/?RSI/69/1236/1.

[13] Carl E. Wieman and Leo Hollberg. Using diode lasers for atomic physics. *Review of Scientific Instruments*, 62(1):1–20, January 1991. URL http://link.aip.org/link/?RSI/62/1/1.