Implementing TPM Commands in the Copland Remote Attestation Language

Josiah Gray

Submitted to the Department of Electrical Engineering & Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Thesis Committee:

Perry Alexander: Chairperson

Dr. Andrew Gill

Dr. Bo Luo

Dr. Fengjun Li

Dr. Matthew Moore

Date Defended: 12/15/2020

The Thesis Committee for Josiah Gray certifies
That this is the approved version of the following thesis:

Implementing TPM Commands in the Copland Remote Attestation Language

Committee:

Chairperson: Dr. Perry Alexander

Dr. Andrew Gill

Dr. Bo Luo

Dr. Fengjun Li

Dr. Matthew Moore

Date Approved: 12/16/2020

Abstract

So much of what we do on a daily basis is dependent on computers: email, social media, online gaming, banking, online shopping, virtual conference calls, and general web browsing to name a few. Most devices we depend on for these services are computers or servers that we do not own, nor do we have direct physical access to. We trust the underlying network to provide access to these devices remotely. But how do we know which computers/servers are safe to access, or verify that they are who they claim to be? How do we know that a distant server has not been hacked and compromised in some way?

Remote attestation is a method for establishing trust between remote systems. An *appraiser* can request information from a *target* system. The target responds with *evidence* consisting of run-time measurements, configuration information, and/or cryptographic information (i.e. hashes, keys, nonces, or other shared secrets). The appraiser can then evaluate the returned evidence to confirm the identity of the remote target, as well as determine some information about the operational state of the target, to decide whether or not the target is trustworthy.

A tool that may prove useful in remote attestation is the TPM, or "Trusted Platform Module". The TPM is a dedicated microcontroller that comes built-in to nearly all PC and laptop systems produced today. The TPM is used as a root of trust for storage and reporting, primarily through integrated cryptographic keys. This root of trust can then be used to assure the integrity of stored data or the state of the system itself. In this work, I will explore the various functions of the TPM and how they may be utilized in the development of the remote attestation language, "Copland".

Contents

List of Figures

Chapter 1

Introduction

So much of what we do on a daily basis is dependent on computers: email, social media, online gaming, banking, online shopping, virtual conference calls, and general web browsing to name a few. Most devices we depend on for these services are computers or servers that we do not own, nor do we have direct physical access to. We trust the underlying network to provide access to these devices remotely. But how do we know which computers/servers are safe to access, or verify that they are who they claim to be? How do we know that a distant server has not been hacked and compromised in some way?

Remote attestation is a method for establishing trust between remote systems [4]. An *appraiser* can request information from a *target* system. The target responds with *evidence* consisting of run-time measurements, configuration information, and/or cryptographic information (i.e. hashes, keys, nonces, or other shared secrets). The appraiser can then evaluate the returned evidence to confirm the identity of the remote target, as well as determine some information about the operational state of the target, to decide whether or not the target is trustworthy.

The University of Kansas and the MITRE Corporation are collaborating on research to develop a language for specifying remote attestation, called Copland. Copland is a language designed to be a unified language that diverse platforms can use to perform attestation, while also being formally verified to provide certain security guarantees [7].

A tool that will prove essential in the development of Copland as a method for establishing trust is the TPM, or "Trusted Platform Module". The TPM is a dedicated microcontroller that comes built-in to nearly all PC and laptop systems produced today [1]. The TPM is used as a root of trust for storage and reporting, primarily through integrated cryptographic keys. In this context, a root of trust is a hardware or software component with a known behavior, that is certified by an authority, like the manufacturer, via a signed certificate [4]. A root of trust for storage ensures that data is stored in a way that will preserve secrecy; the TPM primarily does this by sealing data and keys, as we will discuss later in the "TPM Functionality" section, and by encrypting keys that encrypt data, forming a key hierarchy. A root of trust for reporting reliably attests to the results of measurements by storing and reporting information about the platform; the TPM accomplishes this by extending PCRs with measurements taken during the boot process, which we will also discuss in the "TPM Functionality" section. Acting as a root of trust, the TPM can then be used to assure the integrity of stored data or the state of the system itself. Another important component of remote attestation is a root of trust for measurement that can reliably prepare measurements of the software state of a device. The TPM is meant to securely store and report measurements, not measure the software itself; so the root of trust for measurement must be provided by some other component.

In this work, I will explore the various functions of the TPM and how they may be utilized in the development of Copland. I will also provide a proof-of-concept by presenting sample code I wrote for an interpreter that converts Copland request terms into evidence utilizing a TPM.

Chapter 2

Background

2.1 Remote Attestation

As stated in the work “Principles of Remote Attestation” [4], remote attestation is the process of making claims about properties of a target system by supplying evidence to a remote appraiser over a network. It is not sufficient to base trust in a system solely on verification of the system’s identity, we must also verify properties of the system to ensure that it has not been compromised in some way. The remote appraiser will want to know as much information as possible about the operational state of the target in order to determine if it is trustworthy. At the same time, the target will want to limit the amount of information it reveals about itself so as not to make itself vulnerable to attacks; the more information that is known about a system, the greater the “surface of attack”.

We will begin by defining the terminology used in discussing remote attestation:

An *appraiser* is a party that must make a decision whether or not to trust some other party. A *target* is a party about which an appraiser must make a decision.

Attestation is the act of making a claim to an appraiser about the properties of a target by providing evidence that supports the claim; the party that performs this act is called the *attester*.

The appraiser's decision-making process based on attested evidence is called *appraisal*. To *measure* a target is to collect evidence about it via direct and local observation; measurements can consist of something as simple as a SHA hash of software code present on the target system, a list of the target's configuration settings, or more complex observations of the target system's behavior.

An *attestation protocol* is a cryptographic protocol involving a target, an attester, and an appraiser, whose purpose is to supply evidence that will be considered authoritative by the appraiser, while also respecting the privacy goals of the target; the attester may supply evidence that it has directly observed, reduced evidence (e.g. a hash of evidence), or credentials granted by a third party based on evaluation of evidence. The result of an attestation protocol may depend on a combination of facts that the appraiser can check directly, such as cryptographic signatures, and those they cannot. An appraiser must decide whether to trust a target based on uncheckable facts.

The term *trust* has many meanings depending on context; we will focus on two definitions from (1) the TCG (Trusted Computing Group), who designed the TPM, and (2) Coker et al. as presented in "Principles of Remote Attestation" [4]. TCG base trust on the predictable behavior of a target system based on the evidence provided. Coker et al. define trust as follows: "Principal B *trusts* principal A with regard to statement ϕ if and only if, from the fact that A has said

ϕ, B infers that ϕ was true at a given time." This means that an appraiser should be able to infer that the evidence a target has provided is truthful based solely on the evidence provided and assumptions about how that evidence was collected. This is where the TPM becomes an important root of trust. We will discuss the specifics of the TPM in the following sections, but for now we will simply state that a TPM provides cryptographically signed evidence about the state of the system it is attached to. As an example, measurements of a target's state may be placed into the PCRs (Platform Configuration Registers) of the target's TPM. The TPM can produce a quote, which is a cryptographically signed report of specific PCR contents. If an appraiser finds that a TPM quote is a valid signature over certain PCR values, it may trust the target concerning statements about the evidence that was placed into the PCRs. In this way, the TPM acts as a root of trust regarding claims about a target that an appraiser cannot directly check. Note that this definition of trust differs from that of the TCG, in that predicting the behavior of a system is the purpose of measurement, rather than the basis of trust itself. Measurements collect evidence that supports predictions about the behavior of the target, and trust is based on the verification of these measurements.

A *root of trust* is a hardware or software component with known behavior, that a certificate asserts to be present on a system. Technically, hardware components can be emulated with software; however, hardware roots of trust are generally considered to be more secure, as hardware is immune to malware attacks. A *root of trust for measurement* is a hardware device that can reliably prepare measurements of the software state of a system. A *root of trust for reporting* is a hardware device that can reliably attest to the result of a measurement. A *root of trust for* *storage* is a hardware device that ensures certain data will be stored in a way that will preserve their secrecy. The TPM serves as a root of trust for reporting since its hardware architecture ensures that only the roots of trust for measurement can write to certain PCRs (during the boot process). Specific PCRs are reserved for storing measurements of critical software, such as the BIOS and the platform's operating system, and these PCRs can only be extended by programs that have been measured and verified earlier in the boot process. The TPM also serves as a root of trust for storage by using encryption to store cryptographic keys, including private Attestation Identity Keys (AIKs) used to sign attestations provided by the TPM. Every TPM has an Endorsement Key (EK), which is a unique asymmetric key pair embedded during manufacturing. The TPM can generate multiple AIKs, which are encrypted by the EK, acting like aliases for the EK. Having multiple AIKs allows a user to maintain anonymity between different services.

Finally, a system provides *measured boot* if hashes of the code controlling successive stages of the boot process are stored in PCRs on the TPM. During a reboot, the platform begins with a core root of trust measurement (CRTM), which measures the next software to run, extending a PCR with that measurement. That software in turn measures the next software to run and extends a PCR with that measurement, and so on. As an example, the CRTM measures the BIOS and its configuration data, extending the measurements to PCRs 0 and 1, respectively; the BIOS then measures and extends the master boot record and its configuration data, using PCRs 4 and 5; this process continues until the boot is complete, measuring each software before it runs. Upon boot completion, the PCR values represent the history of all measurements made during the boot process, in the order they were completed. A system provides *secure late launch* if the CPU has an instruction that automatically constructs a hash of the contents of a region of memory, stores the hash in a PCR, and then transfers control to the instruction at the start of the region of memory.

“Principles of Remote Attestation” [4] outlines five principles meant to guide the development of remote attestation systems: (1) attestation must be able to deliver temporally fresh evidence; (2) comprehensive information about the target system should be available; (3) the target system (or its owner) should be able to constrain the disclosure of information about the target; (4) attestation claims should have explicit semantics to allow decisions to be derived from several claims; and (5) the underlying attestation mechanism must be trustworthy. This work is primarily concerned with addressing the fifth principle by posing the TPM as a possible root of trust for remote attestation using the Copland language.

2.2 Copland

Copland is a language that is the product of research by the University of Kansas and the MITRE Corporation. Its purpose is to provide a basis for specifying layered remote attestations by bridging the gap between formal analysis of attestation security guarantees and concrete implementations [7]. As the diversity and complexity of computer systems evolve, it becomes apparent that no single set of simple measurements will be sufficient for remote attestation; the quantity and type of evidence required for attestation is dependent on the structure of each system involved as well as the risk posed to each system. The evidence generated by a Windows machine may look different from evidence produced by a Linux machine. Some simple interactions may only require a high-level integrity check of the target system, like checking that software versions running on the target are up-to-date, or the results of a general virus scan. More sensitive interactions, such as those involving bank accounts or patient medical records, may require more extensive evidence to confirm that neither party is compromised by malware.

Given the broad diversity of ways for a target system to attest its operational state to an appraiser, we need a way to negotiate a mutually agreeable orchestration of measurements. We require a single language to express these orchestrations, with an unambiguous semantics so that there will be a common understanding of what measurements will be made and what guarantees the resulting evidence provides. This is the purpose of Copland, to provide a semantically explicit means for systems to negotiate what evidence should be collected, by which parties, and in what order.

In Copland, measurements are referred to as *USMs* (Userspace Measurements). These measurements occur at specified *places*, where a place is an identifier that corresponds to an *attestation manager* that is capable of responding to an attestation request. The attestation managers and measurers are programs that run on a system, and thus have access to the operating system (OS), user applications, and hardware of the system; these components are, in fact, what we are interested in measuring to establish trust. USMs produce evidence as output, and take as input certain parameters related to the measurement being performed or even other evidence produced by earlier measurements; this forms the basis for "layered attestation," where evidence is nested by several different measurements. As an example, consider the following Copland request:

$$@_p \ USM \ \bar{a}$$

where the measurement USM is made at place p, taking as input the list of parameters $\bar{a}$. This request produces evidence of the form:

$$U_p(\xi)$$

which indicates that a measurement of type U took place at target place p and took input evidence ξ (note that ξ is used to represent the empty evidence).

For a more advanced example, let us consider a scenario where a machine is configured to have two layered privilege levels: the above userspace location p, and a more protected location labeled place q. We want to measure the OS kernel from this more protected location q before performing the above userspace measurement to be sure the system is not corrupted by a rootkit in the OS kernel. Measurements of the kernel are referred to as Kernel Integrity Measurements (KIMs). Additionally, we also want each piece of evidence to be signed to form a chain of evidence. We can sequence these two measurements using the following single request:

$$@_q((KIM\ \ p\ \ \bar{a}_2 \rightarrow SIG) \prec @_p(USM\ \ \bar{a}_1 \rightarrow SIG))$$

The $\prec$ symbol indicates that the left request should be completed before the right request is executed. The $\rightarrow$ symbol routes data from the left term to the right term; in this case, the evidence from the KIM and USM measurements are routed to two instances of a digital signature primitive, corresponding to the two different locations which each have their own signing keys. This request will produce evidence of the form:

$$[\![K^p_q(\xi)]\!]_q \;\; ;; \;\; [\![U_p(\xi)]\!]_p$$

The ;; indicates the sequenced order in which the evidence was generated. The square brackets $[\![\;]\!]$ represent signatures using the private key associated with each place. The KIM term indicates that a kernel measurement was performed from the vantage point of q on target location p.

2.3 The TPM

A TPM is a cryptographic coprocessor present on nearly all commercial PCs and servers produced today [1]. Despite being nearly omnipresent, there exist relatively few software applications that utilize them, so they go relatively unnoticed by most users.

In the 1990's, computer engineers from the Trusted Computing Group (TCG) began developing the TPM as a solution to the lack of hardware and software-based security for personal computers as the Internet grew in popularity and began to become a place of commerce. Cost pressures dictated that the TPM had to be cheap to produce, resulting in a hardware chip that was physically attached to a PC's motherboard. In 2003, TPM version 1.1b was the first widely deployed TPM to be released; the most recent version is TPM 2.0, released in 2016. Its basic functions included RSA key generation, limited secure storage (mainly for storing keys), secure authorization, and device-health attestation. These functions were intended to guarantee user privacy by creating anonymous identity keys (requiring owner authorization) based on certificates provided with the TPM. The TPM also contains dynamic memory Platform Configuration Registers (PCRs) that are reserved for the system's boot-sequence to ensure the integrity of measurements made as the system boots.

Developed between 2005 and 2009, TPM 1.2 improved upon 1.1b by including a standard software interface and standardizing the hardware interface. After the release of TPM 1.2, the TrouSerS project was released. TrouSerS is an open-source TCG Software Stack that is compliant with the TPM 1.1b and 1.2 specifications [2,8]; it provides a standard API for accessing the functions of the TPM. TPM 1.2 also added a small amount of nonvolatile RAM (NVRAM), primarily as a place to store the certificate for the TPM's endorsement key, so the certificate was not stored on a hard disk where it could be erased. Being able to generate and store keys to identify users and encrypt data, the TPM was able to replace other security coprocessors (such as smart cards), and provide additional functionality. The TPM also provides device identification for the system it is on since it is integrated directly onto the motherboard.

In 2005, due to concerns regarding successful attacks on the SHA-1 algorithm, that was heavily used in the TPM 1.2 architecture, work immediately began on the specification for TPM 2.0. The main improvement for TPM 2.0 was to make it more flexible in terms of which cryptographic algorithms it should use. Algorithms would no longer be hard-coded into the TPM specification; rather, TPM 2.0 would incorporate an algorithm identifier to permit the use of several different algorithms without the need to change the TPM specification itself. All TPM structures were changed to for allow algorithm independence.

Chapter 3

TPM Functionality

Having covered the basics of remote attestation and the origins of the TPM, we will now discuss the primary functions of the TPM and how they will be useful in the design of the Copland language. Every function performed by a TPM is related to mitigating some kind of cryptographic attack. The main functions we are concerned with in this work are random number generation, identification, encryption, key storage, NVRAM storage, and the PCRs.

The simplest function provided by the TPM is random number generation. The TPM contains a hardware random number generator (RNG), which is an essential component of cryptography. This RNG has many uses, including seeding the OS random number generator or other pseudo-random number generators, generating nonces for cryptographic protocols, generating one-time use symmetric keys for quick file encryption, and generating long-term storage keys. Copland can take advantage of this RNG by generating nonces for attestation requests to ensure freshness of evidence, and generating keys for identifying systems or encrypting data/evidence for later retrieval.

The next function provided my the TPM is device identification. Because a

TPM can store private keys and is physically embedded on the motherboard of a system, that system can be uniquely identified by signatures using the keys stored on the TPM. Additionally, since the use of TPM keys requires user authorization, both the user and the system can be authenticated when such a key is used; TPM generated keys are encrypted so that they can only be used by the TPM that created them, and only someone with knowledge of the owner authorization code can authorize the use of the key. Issues may arise in cases where a single user must be authenticated on multiple machines, requiring their identification key to be present on multiple TPMs, or if a user wants to upgrade to a new machine but keep their old key. Fortunately, the TPM provides solutions to these issues.

As stated previously, every TPM contains a unique embedded Endorsement Key (EK); this EK uniquely identifies the specific device being used, as it cannot be moved. The TPM can also generate "migratable" keys that can be moved to a different TPM; these keys could be used to identify a specific user on different machines. Identification has many diverse uses, such as a VPN verifying user or machine identities before granting access to the network; signing or decrypting emails using asymmetric keys; a user identifying themselves to a bank or other sensitive service; remote login; etc. Copland may take advantage of TPM keys to identify target systems or even identify specific users operating on several different systems.

In addition to generating keys for device/user identification, the TPM also performs encryption. In order to support cryptographic signing, the TPM includes public/private key encryption functionality. However, the TPM does not include large enough storage to handle encrypting entire files. Instead, the TPM can encrypt and store symmetric keys that can be used to encrypt files outside of the TPM. This enables several useful functions, including general file encryption/decryption, on both local and removable drives; full disk encryption; and password encryption for a password manager.

The TPM itself has limited internal storage for keys or other data; this poses a problem for users who require many different keys for many different uses. Fortunately, because the TPM is embedded on a PC's motherboard, we can utilize the disk storage of the PC to store as many keys as necessary. With access to a self-generated private key, the TPM can encrypt keys with the corresponding public key and store the resulting encrypted key on the system's hard drive. These keys stored on the hard drive can be backed up or erased according to the user's needs, but because they are encrypted by a TPM public key, they can only be decrypted by the private key stored on the TPM. This may be utilized in Copland by allowing an appraiser to encrypt storage keys so that they cannot be used until a target provides evidence that it is in a safe state. This is also useful in the case where a single computer is shared by multiple users who want to store their keys on the shared machine. A TPM can encrypt each user's identification keys so that they can only be decrypted when the correct user presents their corresponding credentials (usually an authorization code). For remote attestation, this can be useful in determining if a specific user is operating a remote system.

The TPM comes with a small amount of nonvolatile RAM (NVRAM) that has several uses. NVRAM provides a place to store keys that should not be available while the machine is turned off; for example, if you don't want keys to be read from a local or removable drive when the system is off. The TPM's NVRAM can also control read and write capabilities separately, so a user can store a small amount of data (e.g. a key) for later use without worrying about it being deleted by an attacker or by accident. This NVRAM is also important for storing keys that can be used when the system does not have access to its main storage; for example, during the boot cycle or before a self-encrypting drive has been decrypted with a password. This is also where root keys for the TPM certificate chains are stored. Another use case that is relevant to attestation is using NVRAM as a place to store some representation of what the state of the machine is supposed to be. Since remote attestation is concerned with determining if a target is in a safe state to trust before communication, the TPM's NVRAM may be useful in allowing a system to monitor and store representations of its own state to be sure it is not compromised.

The final function we will examine is the TPM's Platform Configuration Registers (PCRs). Because the TPM is attached to the motherboard, and therefore available before boot, it can be used as a place to store measurements taken during the boot process. This is exactly what the PCRs are designed for. PCRs are registers that store hashes of measurements so that the TPM can later report those measurements by signing them with a private signing key. Including the TPM on the motherboard also gives the CPU direct access to the SENTER and SINIT commands that initialize the TPM during boot up and perform the first measurement. SENTER is a hardware "command" that only the CPU can execute, resetting all PCR values to 0 at the beginning of the boot process. SENTER then measures the SINIT policy and gives control to the SINIT command. SINIT measures the Measured Launch Environment (MLE) and returns control to SENTER. SENTER then invokes the MLE to continue the boot process.

PCRs cannot be directly overwritten, except when they are reset when the machine is shut off (there are a few registers that can be reset for use by a user or software application, but boot measurements are only stored in registers that cannot be manually reset so as to preserve their integrity). PCRs are not written to directly, they undergo an operation called *extension*. When a register is extended, the current hash value is concatenated with the new measurement, and then this concatenated data is hashed and stored in the register. This can be symbolically represented as follows:

$$V_{new} = hash(V_{old} :: measurement)$$

where V_{new} and V_{old} represent hashed values stored in a PCR and "::" represents concatenation. These hashed values cannot be spoofed since they require the exact set of measurements to be extended in the exact same order due to the one-way nature of hash functions.

These registers can also be used for authentication. For example, a user can create a key or other TPM object that cannot be used unless a certain PCR or set of PCRs are in a specific state, resulting from trusted measurements during boot. This is the basis for a function called *sealing*, where a TPM encrypts data using a specified key and set of PCRs and will not decrypt the data when asked unless the set of PCRs are in the exact same configuration they were in at the time of "sealing." Using one of the few resetable PCRs, a user could reset and extend the PCR using specific values, seal a key to that PCR's state, and then either reset or extend the PCR again so that the key cannot be decrypted and used until the PCR is extended to the same value it had before. This idea of sealing data or keys to specific PCR states is useful for attestation. A specific key or shared secret can be sealed to PCRs representing a safe state after boot, so that they cannot be unsealed if the machine is compromised; a change in the underlying state of the machine will change the boot measurements and PCR values. An appraiser could also verify some information about the state of a target by requesting that the target take hashes of specific programs (like anti-virus) that the appraiser expects the target to have. The target can then extend a PCR with these hashes and have their TPM produce a signed quote of the value of that PCR to verify that the target is running safe and up-to-date software.

Chapter 4

Copland Interpreter

Now that we have covered the basics functionality of the TPM and the purpose of the Copland language, we will examine a proof-of-concept interpreter that integrates the TPM into Copland terms. This interpreter is part of the Git repo CAP-Tools [5], and requires the TSS (TPM Software Stack) from Microsoft [6]. The TSS acts as an interface for communicating with the TPM, allowing a user to issue TPM commands by writing code in C++, Java, JavaScript, Python, and .Net languages (e.g. F# and C#). This TSS from Microsoft serves as an updated replacement for TrouSerS as a way to interface with TPM 2.0. The interpreter is written in F#, a functional-first language designed for functional programming on .Net. Although it is beyond the scope of this work, this implies that it would be possible for properties of the interpreter to be formally verified using a proof language, like Coq. For convenience while testing, I used a TPM simulator (available from `https://www.microsoft.com/en-us/download/details.aspx?id=52507`), but the interpreter works with a hardware TPM as well.

```
let rec evalAPDT t place ev tpm =
    match t with
    | KIM (p, mId, args) ->  K (mId, args, place, p,
        toBstring (getKIM mId args p  place), ev)
    | USM (mId, args)  -> U (mId, args, place,
        toBstring (getMeasure mId args tpm), ev)
    | SIG -> G (place, ev, toBstring (getSignature ev  place))
    | NONCE ->  N (place, toBstring (generateNonce ()), ev)
    | HSH -> H (place, toBstring (getHash (encodeEv ev)))
    | AT (p1, t1)  -> evalAPDT t1 p1 ev tpm
    | LN (t1, t2)  -> evalAPDT t2 place (evalAPDT t1 place ev tpm) tpm
    | BRS ((sp1, sp2), t1, t2)  ->
        let ev1 =
            match sp1 with
            | SP.ALL -> ev
            | SP.NONE -> Mt
            | value -> failwithf "Unexpected enum member: %A: %A"
                typeof<SP> value
        let ev2 =
            match sp2 with
            | SP.ALL -> ev
            | SP.NONE -> Mt
            | value -> failwithf "Unexpected enum member: %A: %A"
                typeof<SP> value
        SS ((evalAPDT t1 place ev1 tpm), (evalAPDT t2 place ev2 tpm))
    | BRP ((sp1, sp2), t1, t2)  ->
        let ev1 =
            match sp1 with
            | SP.ALL -> ev
            | SP.NONE -> Mt
            | value -> failwithf "Unexpected enum member: %A: %A"
                typeof<SP> value
        let ev2 =
            match sp2 with
            | SP.ALL -> ev
            | SP.NONE -> Mt
            | value -> failwithf "Unexpected enum member: %A: %A"
                typeof<SP> value
        PP ((evalAPDT t1 place ev1 tpm), (evalAPDT t2 place ev2 tpm))

let getEvidence t tpm = evalAPDT t 0 Mt tpm
```

Figure 4.1. APDT grammar from `Evaluator.fs`

The grammar presented in figure 4.1 is derived from the APDT (Attestation Protocol Description Term) grammar given in "Orchestrating Layered Attestation" [7]. When we want to evaluate a Copland term (an APDT) to generate evidence, we call **getEvidence t tpm**, that takes parameters **t**, a term to be evaluated, and **tpm**, a tpm connection object defined by the TSS. This will call the function **evalAPDT** to recursively evaluate nested Copland requests. **evalAPDT** takes parameters **t** and **tpm**, the same parameters as before, in addition to a place identifier and initial evidence, in this case place **0**, indicating the local machine the evaluator is running on, and **Mt**, indicating empty initial evidence.

The evaluator includes match cases for cryptographic primatives, like signing evidence (**SIG**), generating nonces (**NONCE**), and hashing data (**HSH**). Nested Copland terms for layered attestation are covered by the match cases **AT**, which directs a term to be evaluated at a specified location; **LN**, which orders terms to be evaluated linearly; and **BRS** and **BRP**, which allows terms to be evaluated in parallel branches. We are most interested in the match case **USM**, which evaluates terms to execute measurements that generate evidence, some of which invoke TPM functions.

The **USM** match case calls the function **getMeasure mId args tpm** to execute a measurement and generate evidence. **getMeasure** takes three parameters: **mId** identifies what type of measurement will be made from a list of possible measurements, each labeled with an index number; **args** is a list of arguments that the measurement corresponding to index **mId** will require to execute; and **tpm** is the same TPM connection object as mentioned before. The structure of this **getMeasure** function is presented in figure 4.2.

```
let getMeasure mId args tpm =
    match mId with
    | 1 -> getFileHash (List.head args)
    | 2 -> filesInDir (List.head args)
    | 3 -> filesInDirHash (List.head args)
    | 4 -> numFilesAtRoot ()
    | 5 -> TpmGetRandom (Convert.ToUInt16 (args.Head), tpm)
    | 6 -> TpmReadPcr (Convert.ToUInt32 (args.Head), tpm)
    | 7 -> TpmExtendPcr (Convert.ToUInt32 (args.Item(0)),
        System.Text.Encoding.ASCII.GetBytes(args.Item(1)), tpm)
    | 8 -> TpmResetPcr (Convert.ToUInt32 (args.Head), tpm)
    | 9 -> TpmQuote (args.Item(0), args.Item(1), tpm)
    | _ ->  Array.zeroCreate 1  // "***Unknown USM: "
```

Figure 4.2. getMeasure function from `MakeMeasurements.fs`

Notice that some of these measurements, like **getFileHash**, do not require the use of a TPM to produce evidence; while others, like **TpmReadPcr** require a TPM connection object as one of their parameters. Measurements can be almost anything that produces data that can be used as evidence for establishing trust; this list of simple measurements is meant to demonstrate how Copland terms are evaluated to produce evidence, including terms that utilize TPM functions.

Using the TSS library [6], I have written several measurement functions that issue commands to the TPM to generate evidence. These are the functions **TpmGetRandom**, **TpmReadPcr**, **TpmExtendPcr**, **TpmResetPcr**, and **TpmQuote**, which we will discuss in a bit more detail (the definitions for these functions can be found in the file `TPMMeasurements.fs` of the `FSharpInterpreter`).

TpmGetRandom (numberOfBytes, tpm) takes as input a 16-bit integer, specifying the number of random bytes to generate, and the TPM connection object that the TSS uses to execute the instruction on a specific TPM. This function uses the TPM's random number generator to generate random bytes of data that can be used as nonces.

TpmReadPcr (index, tpm) takes as input the index of a PCR who's value we want to read out, and the TPM connection object. The TPM has 24 PCRs, indexed 0-23. This function returns the PCR value at the specified index as evidence; often a 20-byte SHA hash value, but the TPM can be configured to use different hash functions.

TpmExtendPcr (index, dataToExtend, tpm) takes as input the index of the PCR we want to extend, a byte array of data to extend the PCR with, and the TPM connection object. This function extends the specified PCR with the data provided and then reads and returns the PCR's new value after extension as evidence.

TpmResetPcr (index, tpm) takes as input the index of the PCR we want to reset and the TPM connection object. After resetting the specified PCR, the function reads and returns the value of the PCR as evidence, which should be all 0's after reset. Most PCRs cannot be reset after boot, so as to preserve the integrity of measurements taken and stored during the boot process. PCR 16 is allocated for debugging purposes and can be reset by the user; most of the examples I wrote for the interpreter use PCR 16 for this reason.

The final USM measurement is **TpmQuote (data, pcrIndicesString, tpm)**, which takes as input some extra data to include in the quote (for example, a nonce to ensure the quote is fresh), a list of PCR indices to be quoted, and the TPM connection object. The function returns a TPM quote signed by a key previously loaded onto the TPM as evidence. The quote is a hash of the data provided and the specified PCR values.

The file `Evaluator.fs` contains a demonstration in which we form Copland requests meant to interact with the TPM and return the TPM's response as evidence, and then evaluate these terms to produce evidence. The following is an example of one Copland term and its evaluation:

```
let copReadPcr16 = USM (6, ["16"])
let evReadPcr16 = getEvidence copReadPcr16 tpm
```

This creates a Copland term **USM (6, ["16"])**, meant to read the value of PCR 16 from the TPM, and then evaluates the term by calling the **getEvidence** function. Note that in the interpreter, the USM measurement for reading a PCR value is identified by the index 6 and given the argument list containing only the value "16", the index of the PCR to read from; all the USMs in this interpreter are identified by a unique index number, since a measurement can be almost anything that the appraiser might find useful, so long as the appraiser and target agree on a common list of USMs and their identifiers. The above Copland term then evaluates to the hash value stored in PCR 16, which is returned as the evidence.

Copland terms can be nested in order to execute multiple measurements with a single request.

```
let copExtendRight = LN (USM (7, ["16"; fileHashString]),
    USM (7, ["16"; "abc"]))
let evExtendRight = getEvidence copExtendRight tpm
```

This example extends PCR 16 with the hash of a file, then extends PCR 16 with the string "abc" (which are converted to byte arrays by the **getMeasure** function before calling **TpmExtend**). Note in this example, USM 7 is the measurement that extends PCRs, and it takes as input the index of a PCR to extend and a string value to extend the PCR with. These two instances of USM 7 are combined using the linear operator "LN," which simply executes the first measurement, immediately followed my the second measurement. This Copland term produces as evidence the final value of PCR 16 after extending with a file hash and then "abc."

As a follow-up example, consider the following Copland term:

```
let copExtendWrong = LN (USM (7, ["16"; "abc"]),
    USM (7, ["16"; fileHashString]))
let evExtendWrong = getEvidence copExtendWrong tpm
```

This example is almost identical to the previous one, except the USM terms requesting to extend PCR 16 have been switched. Evaluating this term extends PCR 16 with "abc" and then the file hash, returning the resulting PCR value. Note that since extension concatenates and hashes the PCR value, the order of extensions is important. The resulting evidence from this term will be completely different from the previous example.

Finally, we will consider an example using the TPM quote function. Consider the following example:

```
let dataToQuote = "1234"
let pcrsToQuote = "15,16,17"
let copQuote = USM (9, [dataToQuote; pcrsToQuote])
let evQuote = getEvidence copQuote tpm
```

In this example, we use USM 9, corresponding to the function **TpmQuote**. To request the quote, we provide "1234" as the extra data to be included in the quote, along with the list of PCR indices 15, 16, and 17. Evaluating this term returns a byte array representing the hashed quote from the TPM that includes the extra data and the contents read from PCRs 15, 16, and 17.

These examples demonstrate how to generate Copland terms involving the TPM and evaluate the terms using the interpreter to produce useful evidence. Currently, all evidence generated by the interpreter comes in the form of a byte array, as it is a simple data type that is easy to interpret, transmit between systems, and write to files (for example, using Json [3]). More advanced TPM functions, like sealing data to PCR states, return more complicated data objects defined by the TSS, making it difficult to write them to files and transfer between different systems. Further work is required to realize the more powerful TPM functions like sealing and unsealing data to a TPM using Copland terms, but as a basic proof-of-concept, this experiment demonstrates that it is both useful and possible to incorporate TPM commands for use in the Copland language for remote attestation.

Chapter 5

Conclusion

In this work, we have covered the basics of remote attestation, the Copland language, and described the functionality of the TPM. As a root of trust for storage and reporting, the TPM is a critical tool in collecting, storing, and reporting evidence for remote attestation. Given how powerful the TPM is as a root of cryptographic security and system attestation, it is clear that it will serve as a fundamental component in the development of the Copland language.

I have also presented an interpreter capable of evaluating basic Copland terms to produce meaningful evidence using TPM functions, along with several examples of how these Copland terms are formed and evaluated by the interpreter. The interpreter and these examples serve as a concrete implementation of some of the concepts presented in "Principles of Remote Attestation" and "Orchestrating Layered Attestation" [4, 7]. Further work will be required to implement the more advanced TPM functions, like sealing and key generation/loading, but the interpreter presented serves as an important first step in implementing TPM functionality in the Copland language.

References

[1] W. Arthur, D. Challener, and K. Goldman. *A Practical Guide to Tpm 2.0: Using the Trusted Platform Module in the New Age of Security.* Apress, 2015.

[2] Ashley, Debora, G. Wilson, H. C. Lo, K. Yoder, F. Gunter, R. Maciel, J. Schopp, K. H. Kiwi, M. Halcrow, R. Andrade, E. Ratliff, and T. Lendacky. Trousers. `https://sourceforge.net/projects/trousers`, Sep 2014.

[3] T. Bray. The javascript object notation (json) data interchange format. 2014.

[4] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

[5] ku sldg. Captools. `https://github.com/ku-sldg/CAPTools`, 2018.

[6] Microsoft. Tss.msr. `https://github.com/microsoft/TSS.MSR`, 2015.

[7] J. Ramsdell, P. D. Rowe, P. Alexander, S. Helble, P. Loscocco, J. A. Pendergrass, and A. Petz. Orchestrating layered attestations. In *Principles of Security and Trust (POST'19)*, Prague, Czech Republic, April 8-11 2019.

[8] srajiv. trousers. `https://github.com/srajiv/trousers`, 2010.