Securing ARP From the Bottom Up

Jing (Dave) Tian[1], Patrick McDaniel[2],
Padma Krishnaswamy[3], and Kevin Butler[1]

[1] University of Oregon, Eugene OR 97405 USA
[2] The Pennsylvania State University, University Park PA 16802 USA
[3] Federal Communications Commission, Washington DC USA

Abstract. The basis for all network communication is the Address Resolution Protocol, which maps IP addresses to a device's MAC identifier. ARP resolution has long been vulnerable to spoofing and other attacks, and past proposals to secure the protocol have focused on key ownership rather than the identity of the machine itself. This paper introduces *arpsec*, a secure ARP protocol that is based on host attestations of their integrity state. In combination with bottom-up host measurement, we define a formal ARP binding logic that bases additions of new ARP responses into a host's ARP cache on a set of operational rules and properties, implemented as a Prolog engine within the *arpsec* daemon. Our proof of concept implementation is designed within the Linux 3.2 kernel environment and we show that using commodity TPMs as our attestation base, *arpsec* incurs an overhead ranging from 7% to 15.4% over the standard Linux ARP implementation. This formally-defined protocol based on bottom-up trust provides a first step towards a formally secure and trustworthy networking stack.

1 Introduction

The Address Resolution Protocol (ARP)[15] is a fundamental component of network connectivity. Working below the network layer, ARP binds IP addresses to the Media Access Control (MAC) identifier of a network device, e.g., an Ethernet card or a WiFi adapter, and thus underpins all local network activity, which in turn informs how wide-area routing occurs. ARP is subject to a variety of attacks including spoofing and cache poisoning as originally described by Bellovin [1]; tools such as *dsniff* [19] and *nemesis* [10] can be used to easily launch these respective attacks. One could use simple scripts and tools like nemesis to launch ARP cache poisoning attacks. An attack on ARP can subsequently enable more sophisticated denial-of-service and man-in-the-middle [11] attacks.

Different methods have been proposed to make the ARP secure, including the static ARP configuration[25], the ARP spoofing detection[8], the policy-based ARP security[14,12] and the ARP extension using the Public Key Infrastructure (PKI) system[2,9,5]. Yet none of these methods is widely accepted by the community or deployed in a large scale network; we thus have to reconsider the challenges to securing ARP. First, the basic ARP protocol itself must remain unchanged. There is no "flag day" by which time all ARP implementations embedded into the large variety of Internet-connected IPv4 devices will change. Second, the overhead of the implementation should be as small as possible in order to optimize performance. Any enhancement for the ARP security with the overhead far beyond the original implementation is not acceptable. Third, the ARP security mechanism should be flexible and reliable. Hard-coded security policies may not satisfy the variant network environments. Last, we want to know if the remote machine could be trusted. Trust here applies to both the true identity and the system integrity state of the remote. Even if the MAC/IP binding belongs to the remote machine, we may not want to add the binding into our ARP cache (and further leak the secret data) if the remote does not install up-to-date operating system patches or certain application software is modified. In other words, the remote with identity impersonation or bad system integrity state is not trusted.

In this paper, we propose *arpsec* - the ARP Security Based on the Logic and Trusted Platform Module (TPM)[24]. *arpsec* does not change or extend the ARP itself. Instead of hard-coded security policies, *arpsec* formalizes the security requirement of the ARP using the logic. A logic prover then could reason about the validness of an ARP/RARP reply from the remote based on the logic rules and the previously stored binding history of the local system. A TPM attestation protocol is also implemented to challenge the remote machine if the logic layer fails to determine the trustiness of the remote. Using the TPM hardware, we could figure out the true identity of the remote and if the remote machine is in a good integrity state (thought to be not compromised). Besides the defense from common ARP attacks, *arpsec* also rises the bar of security to the level of trusted computing and contributes to the secure operating systems. We have implemented *arpsec* in the Linux operating system. Our experiments show a small system overhead no more than 15.4% comparing with the original system and the other two popular methods. We also explore the way to deploy *arpsec* in the wild and discuss some potential issues and challenges, as well as possible solutions.

Note that the TPM hardware attack[20] is not considered in this paper. We treat the TPM hardware (and the BIOS) as the Root of Trust and trust its results. Eventually, one can use the TPM (attestation) as a heavy weapon dealing with the identification and trustiness of the remote machine. A lot of research has been done by leveraging the power TPM for software in different layers of the software stack[17,6,7,13]. However, here comes the tradeoff between the level of trust and the system performance. The speed of the TPM attestation is limited by the speed of the TPM hardware, which is usually slower than the current CPU[18]. Comparing with the TPM attestation, the logic layer in *arpsec* is a light-weighted framework which provides the logic reasoning for the ARP security requirement examination. In the *arpsec* daemon (arpsecd) implementation, the logic layer defends the ARP security once an ARP/RARP message arrives. The TPM layer would not be invoked until the logic layer fails to determine the trustworthiness of the message. However, for some attacks, the TPM attestation may be the only reliable countermeasure. As we will see later, *arpsec* provides the flexibility to tune its component to fit different security requirements.

The remainder of this paper is structured as follows. Section 2 outlines the background on ARP security issues and Trusted Computing and discusses common vulnerabilities based on the current ARP design and implementation. Section 3 details the design and architecture of *arpsec*, including the logic layer and the TPM layer in the arpsecd. Section 4 shows details and tradeoffs during the implementation. Section 5 describes a general procedure to deploy *arpsec* in the wild. Section 6 provides the performance evaluation settings and results. Section 7 discusses some potential issues within *arpsec* and possible solutions. Section 8 recalls the past efforts on ARP security. Section 9 concludes.

2 Background

We first talk about ARP security issues based on the current ARP design and implementations in general. Then we explain common attacks against the ARP and high-level attacks based on ARP attacks. A brief review of Trusted Computing and TPM is provided at the end of this section.

2.1 Address Resolution Protocol (ARP)

The ARP/RARP[15,4] is the glue between layer 3 and layer 2 in the IPv4 networks and usually implemented within the operating system for performance and security considerations. Every time before an IP packet is sent out from the Ethernet card, the ARP cache (table) will be queried to find the MAC address given the target IP address of the packet. If the MAC/IP binding is not found, an ARP request will be broadcasted to the whole network. Only the remote with the target IP address should send back an ARP reply containing its MAC address. In reality, however, every machine in the network could send an ARP reply claiming that it has the requested MAC address. As there is no ARP reply authentication mechanism, most operating systems would either accept the first reply or the latest one if multiple replies respond to the same request. Even worse, for better performance, most operating systems would also process ARP requests from other machines and add the MAC/IP bindings for future use. Though all the MAC/IP bindings in the ARP cache have some Time-To-Live (TTL) control, the timer is usually large and designed for performance considerations instead of security reasons. Take the Linux operating system as an example. It always accepts the first ARP reply to the request and ignores others. It also rejects the ARP reply without the request while processing ARP requests from other machines. The TTL for each entry in the Linux ARP cache is around 20 minutes[2]. Behaving a little bit different from the Linux operating system, the Solaris and Windows operating systems suffer from the same ARP security problems.

One basic ARP attack is the ARP message spoofing. The adversary could inject a new MAC/IP binding into the victim's ARP cache simply by sending a forged ARP request or reply to the victim. The MAC address in the ARP message usually belongs to the adversary and the IP address usually refers to some other potential victim. The other basic ARP attack is the ARP cache poisoning, where the adversary generates the ARP reply using certain MAC address given the request from the victim. Actually, there is no big difference between the ARP spoofing and ARP (cache) poisoning[25]. Both of them try to have a malicious MAC/IP binding inserted in the victim's ARP cache. We explicitly distinguish the two attacks in order to help explain the design details of *arpsec* in later sections.

Based on the ARP spoofing/poisoning, high-level attacks are enabled, including MITM[11] attacks and DoS[25] attacks. For the DoS attack, the adversary could inject the victim's MAC address into a certain machine. Then all the IP traffic from that machine targeting a certain IP address will be redirected to the victim. When multiple machines have the victim's MAC address in their ARP caches, Distributed Denial of Service (DDoS) attack happens. Comparing with DoS/DDoS, the MITM attack seems to be more serious from the security perspective. With the help of ARP spoofing/poisoning, the MITM attack is trivial with a simple script no more than 100 lines. Yet once happens, it is often hard to detect. Timing of the IP packet may help find this kind of attack if the packet delay is obvious. For most cases, the MITM attack goes undetected. Figure 1 displays the ARP attack tree with the ARP spoofing/poisoning centered. While leafs in the tree are the basic attacking means to fulfill the ARP attack, the roots are the high-level attacks supported by the ARP attack mentioned before. Note that, high-level attacks may also be enabled by other attacks, like DNS Pharming attack. Even without MITM attacks, the identity theft (impersonation) or secret leakage may happen via ARP attacks directly.

2.2 Trusted Platform Module (TPM)

A Trusted Platform Module (TPM) is a cryptographic chip embedded in the motherboard. Though implemented by various vendors, all the TPM chips follow the TPM specification[24] designed by the Trusted Computing Group (TCG). As TPM is the hardware designed for security, together with the BIOS, it could be used as the Root of Security and to build the security chain for the software along the software stack, including boot loaders, operating systems and applications[17,6]. One key facility that TPMs provide is to securely store data, such as private keys, digital signatures, passwords or even biometrics, like fingerprints in Apple iPhone 5S.

With the help of TPM, we could know the true identity of the remote via the Attestation Identity Key (AIK) verification during the TPM attestation. To create an AIK pair, the TPM hardware has to talk to the Privacy Certification Authority (PCA) or Attestation Certification Authority (ACA) using the information embedded in itself to prove its identity and get the AIK credentials. We could also know if the remote machine is in a good integrity state via the Platform Configuration Registers (PCRs). Given the good value of PCRs, as long as the PCR values from the remote during the TPM attestation are different, we

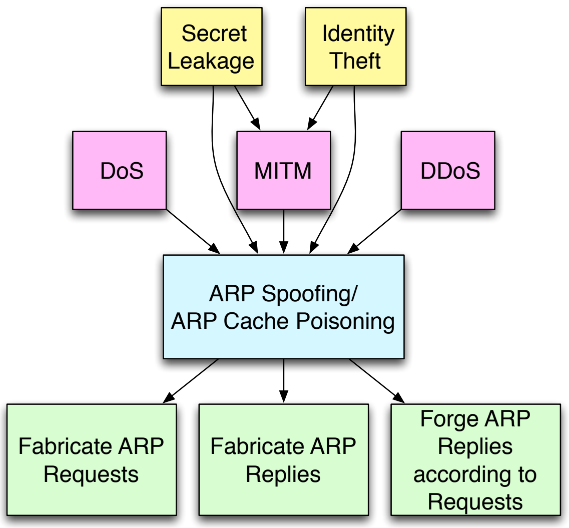

Fig. 1. The ARP attack tree

assume the remote may be or have already been compromised and thus do not trust it. Note that the AIK private key and the measurement of PCRs are all stored in the TPM Non-volatile ROM (NVROM). Unless the TPM hardware is compromised, currently from software, there is no way to hack into the TPM and change the values in it.

3 Design of *arpsec*

Different with the latest work on the ARP security, like S-ARP and TARP, which take the advantage of the PKI system and extend the original ARP, *arpsec* formalizes the ARP security requirement using the logic and examines the ARP/RARP message using a logic prover and the TPM attestation, without changing the original ARP. *arpsec* is designed to respond to attacks below:

– ARP/RARP message spoofing: the adversary fabricates an ARP/RARP request/reply to inject a new MAC/IP binding into the victim's ARP cache.
– ARP cache poisoning: the adversary fabricates an ARP/RARP reply to substitute/distorts the original MAC/IP binding in the victim's ARP cache.
– identity theft/impersonation: the adversary uses the victim's MAC/IP binding while the victim is offline.
– data leakage via the compromised remote: the adversary gets the (secret/privacy) data of the victim machine from the connected and compromised remote machine.

The first two attacks are the basic ARP attacks on which many other high-level attacks rely, like MITM attacks and DoS attacks. Note that both ARP and RARP (requests and replies) are covered in our security scope. In the third attack, the PKI system may help. However, as all these secret keys are saved somewhere in the hard disk and once the key is stolen or broken, there is no way to know the real identity of the remote. The last attack is common when the target machine is hard to compromise. Instead, the 'trusted' and connected remote machine may be compromised and leak the sensitive data from from the target machine if, for example, certain important security patch is missed in the operating system or a buggy version of application is used. We address all these vulnerabilities via *arspec* - ARP Security Based on the Logic and TPM. The architecture of *arspec* is illustrated in Figure 2 as below.

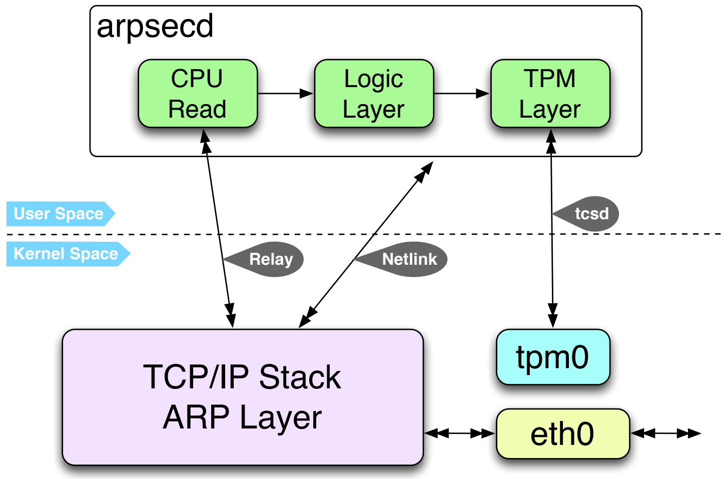

Fig. 2. The *arspec* Architecture

In the user space, arpsecd is the daemon process of *arpsec* running in the local machine and taking the control of processing of all the ARP/RARP messages from the kernel. There are three major components in the arpsecd - the CPU read, the logic layer and the TPM layer. The CPU read component retrieves all ARP/RARP request/reply messages from the kernel space and passes the preprocessed, logic-friendly messages to the logic layer component. The logic layer component then tries to handle these messages based on the message type, system state and the logic rules. We will detail the logic layer in the following section. For the ARP/RARP reply, if the logic layer is unable to validate the message, the TPM layer will then challenge the remote machine using the TPM attestation. Only the MAC/IP bindings (in the ARP/RARP reply) validated by the logic layer or succeeded in the TPM attestation could be added into the local ARP cache. The pseudo code of arpsecd ARP/RARP processing is listed in Algorithm 1 as below.

```
while there is an ARP/RARP msg from the kernel do
    check the msg type;
    if msg.type == ARP request then
        if msg is for us then
            reply the request;
        else
            drop the request;
        end
    else if msg.type == ARP reply then
        if msg is for us then
            if msg passes the logic layer then
                add the MAC/IP binding into the ARP cache;
            else
                if msg passes the TPM layer then
                    add the MAC/IP binding into the ARP cache;
                else
                    drop the reply;
                end
            end
        else
            if msg passes the logic layer then
                add the MAC/IP binding into the ARP cache;
            else
                drop the reply;
            end
        end
    else
        The same handling here for the RARP msg;
    end
end
```

Algorithm 1: The *arpsec* ARP/RARP Message Processing

3.1 Logic Formulation

The logic layer in *arpsec* is the first filter used to testify the trustiness of an ARP/RARP reply message. Though light-weighted comparing with the TPM layer, the logic layer uses a logic prover which is built on solid logic rules. To leverage the power of the logic reasoning, firstly, we introduce an ARP system binding logic formulation.

An instance of an ARP binding system is defined as $\mathcal{A} = \{\mathcal{N}, \mathcal{M}, \mathcal{T}, \mathcal{S}, \bar{\mathcal{S}}, \bar{\mathcal{R}}\}$, where

$$\begin{aligned}
\mathcal{T} &= P \\
\mathcal{N} &= (\epsilon, n_0, \ldots, n_a) \\
\mathcal{M} &= (\epsilon, m_0, \ldots, m_b) \\
\mathcal{S} &= (s_0, \ldots, s_c) \\
\bar{\mathcal{S}} &= \mathcal{S} \times \mathcal{T} \\
\bar{\mathcal{R}} &= \mathcal{S} \times \mathcal{N} \times \mathcal{M} \times \mathcal{T}
\end{aligned}$$

Intuitively, $\mathcal{T}$ is a set of all positive integers representing an infinite and totally ordered set of time epochs. $\mathcal{N}$ is the collection of network addresses and $\mathcal{M}$ is the collection of media addresses. For convenience and as described below, both address sets contain a special address ϵ representing the lack of binding assignment. $\mathcal{S}$ is the set of systems that making assertions about the address bindings within the network. $\bar{\mathcal{S}}$ represents the timing of system trust validations (e.g., system attestations); $\bar{s}_{i,j} \in \bar{\mathcal{S}}$ where system s_i was successfully vetted at time t_j. $\bar{\mathcal{R}}$ are the binding assertions made in the course of operation of the ARP protocol, where $\bar{R}_{i,j,k,l} \in \bar{\mathcal{R}}$ if system s_i asserts the binding (n_j, m_k) at time t_l. Lastly, for ease of exposition, we introduce the following derived binding and trust time-state elements within the system:

$$\begin{aligned}
\mathcal{A} &= (A_0, \ldots, A_{|P|}) \\
\mathcal{B} &= (B_0, \ldots, B_{|P|})
\end{aligned}$$

Trust state : The trust state $\mathcal{A}$ of the system is a totally set of subsets of $\mathcal{S}$ representing the instantaneous set of systems that have been determined to be in trusted state in each epoch (e.g., have been vetted through system attestations). The trust state of the A at time t_k, A_k is:

$$A_k = \bigcup s_i \in \mathcal{S} \quad | \quad \exists \bar{s}_{i,j} = 1, (k-h) \leq j < k$$

Or simply, A_k is the set of all systems $s_i \in \mathcal{S}$ that have been vetted as trustworthy within the last h epochs. The security parameter h represents the durability of a system trust state. In the initial state of the system all systems are untrusted, e.g., $A_o = \{\emptyset\}$.

Binding state : We refer to the B_k as the binding state at time $\mathcal{T}_k$. The states of the binding system $\mathcal{B}$ are a totally ordered sequence of relations over $\mathcal{N}$ and $\mathcal{M}$ representing the instantaneous binding of network to media addresses, where:

$$\forall B_k \in \mathcal{B} : B_k = \mathcal{N} \times \mathcal{M}$$

Note further that each B_k is constrained by a set of *coherency* properties that define correct operation of the binding protocol. Namely, $\forall B_k \in \mathcal{B}$:

$$\begin{aligned}
&(1)\ \forall\ n_l \in \mathcal{N} &&: \exists (n_l, m_o), m_o \in \mathcal{M} \\
&(2)\ \forall\ m_o \in \mathcal{M} &&: \exists (n_l, m_o), n_l \in \mathcal{N} \\
&(3)\ \nexists\ (n_l, m_o), (n_p, m_q) &&: n_l = n_p \neq \epsilon \\
&(4)\ \nexists\ (n_l, m_o), (n_p, m_q) &&: m_o = m_q \neq \epsilon
\end{aligned}$$

That is, all network addresses (constraint 1) and media addresses (2) must have an assignment at each epoch. Further, the network address not bound to the unassigned element ϵ must be bound to exactly one media address (3), and the media address not bound to the unassigned element ϵ must be bound to exactly one network address (4).

We also define the set of rules with operational properties for the binding set. We state that $(n_j, m_k) \in B_l$ if and only if:

$$\begin{aligned}(5)\ &\exists\ \bar{R}_{i,j,k,x} \in \bar{\mathcal{R}},\ x \leq l,,\ s_i \in A_x,\\ &\nexists\ \bar{R}_{v,j,p,y} \in \bar{\mathcal{R}}, p \neq k, y > x, s_v \in A_y,\\ &\nexists\ \bar{R}_{v,q,k,y} \in \bar{\mathcal{R}}, j \neq q, y > x, s_v \in A_y\end{aligned}$$

Constraint (5) indicates that any binding in B_l was asserted at or prior to time t_l by a trusted system, and no later assertion for that network or media address was subsequently received at or before t_l was asserted.

Finally, by definition, all network and media addresses are unassigned in the initial state B_0:

$$B_0 = \forall n_l \in \mathcal{N}, (n_l, \epsilon) \ \bigcup\ \forall m_o \in \mathcal{M}, (\epsilon, m_o)$$

In general, constraint (5) is the core property used by the logic prover to implement the ARP security. The logic layer stores all the verified bindings with the remote system identifiers and the time epochs. For any given MAC/IP binding in the ARP/RARP reply message from the remote, If there exits a binding record from the same (trusted) remote in the past and the time epochs of that record is no more than the pre-defined number epochs (security parameter h) before the current time epochs, the logic layer would trust this binding, add the binding the to the local ARP cache and add this binding record into the logic prover for future reasoning. The security parameter is also a tradeoff between the reliability and the performance, as it determines the time range of the past we would trust to validate the current event. Note that though the logic formulation is used for the reasoning initially, it also provides the ARP cache data provenance, thanks to its ability to store the history records.

3.2 TPM Attestation

The TPM layer in *arpsec* is a heavy weapon we could count on to prove the trustworthiness of the remote once the logic layer fails. To get the trust to the remote, the TPM attestation[24] is used. It is based on the Attestation Identification Key (AIK) authentication and the Platform Configuration Registers (PCRs) verification. The AIK is used by the attestation to sign the data generated by the TPM hardware. To create an AIK pair, the TPM within the system has to provide its hardware information to a Privacy Certification Authority (PCA) following the procedure of AIK certificate enrollment scheme. PCRs are the measurement of the boot loader, the operating system and even the application software within the system. Unlike other PKI system, the AIK private key and the PCRs values are are all saved in the TPM itself. Currently there is no other way to change these values except the hardware attack. Because of these merits of TPM, the AIK becomes the natural way for identification validation and PCRs give the trustworthy information about the system integrity state. We design the ARPSEC Attestation (AT) protocol for communication between the local machine (the challenger) and the remote machine (the attester). The AT request and reply look like below in Figure 3.

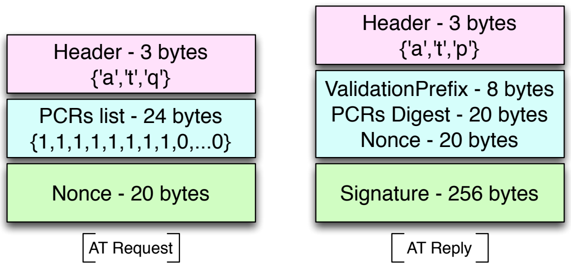

Fig. 3. The AT Request/Reply

The whole TPM attestation procedure starts with an AT request from the challenger - a local machine, where the arpsecd is running. The AT request contains a 3-byte header, a 24-byte PCR list and a 20-byte nonce. The header indicates that the message is an AT request. The PCR list is a byte-based bitmap. It tells the attester the PCR indices the challenger is interested in. In Figure 3, the challenger is interested in PCR `0-7`. The nonce is generated by the TPM for anti-replay. Once the attester receives this AT request, it checks for the header at first to filter other network garbage. It then reads the PCRs based on the PCR list in the request. When the PCR values and the nonce are ready, the attester executes the TPM Quote command to generate the 256-byte signature. This signature is actually a 48-byte validation data signed by the AIK private key. The data is consisted of a 8-byte prefix, a 20-byte digest and the same 20-byte nonce. The prefix is eventually the PCR bit masks and length information. The digest is for the PCR values. The nonce here is exactly the one from the AT request.

Upon receiving the AT reply from the attester, the challenger first examines the header and then compares the locally generated nonce with the one in the AT reply to prevent the replay attack. Moving forward, the challenger uses the SHA1 algorithm to compute the reference digest using the known good PCR values of the remote. By comparing the PCR digest computed locally with the one in the AT reply, the challenger could know if the remote is in a good integrity state since the remote boots up. To further determine if the remote is the one it is talking to, the challenger uses the AIK public key of the remote and the validation prefix from the AT reply to generate the reference signature. By comparing this signature with the one in the AT reply, the challenger then could know if the identity of the remote is trustworthy. Once all these testings pass, the challenger can make a reasonable argument that the remote could be trusted. The challenger can then add the MAC/IP binding into its ARP cache. Note that all these cryptographic operations use the TPM hardware instead of software, like OpenSSL.

4 Implementation

We have implemented *arpsec* in the Linux operating system, with the kernel version 3.2.0.55, using C and Prolog. The implementation details of the arpsecd is shown in Figure 4. Note that *arpsec* is free and open-source. All the code is available for download at OSIRIS lab code repository[21]. We have also developed other versions using different architecture for reference, including the IPC version (Prolog running as a standalone process) and libpcap version (using a kernel module without direct kernel changes). The primary goal of the implementation is the high performance. The overhead of *arpsec* should be as small as possible comparing with the original ARP and other methods. We also consider the case where *arpsec* and the original ARP coexist in the same network.

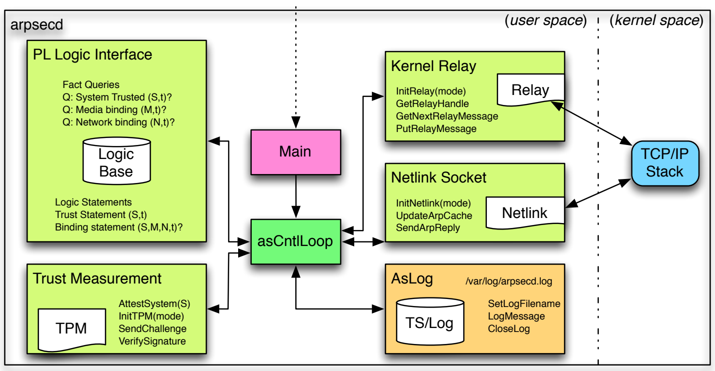

Fig. 4. The Implementation of *arpsec* Daemon (arpsecd)

Like Figure 4 depict, in the kernel space, the kernel relay[26] is used to transport all ARP/RARP messages from the kernel space to the user space. Comparing with the netlink socket and the /proc file system, the kernel relay is specially designed for the heavy traffic communication between the kernel space and the user space. A netlink socket is also used in the arpsecd to manipulate the ARP cache. It provides similar functionalities of the system call `ioctl()` for the

ARP cache management but uses the low-level kernel APIs to get rid of the extra locking in `ioctl()`. By adding this new netlink socket into the kernel, we could also trigger the kernel to send out the ARP/RARP reply given any request from the user space. In a word, the kernel does not handle any ARP/RARP message but relays them to the user space for arpsecd processing efficiently.

In the user space, the logic formulation of the ARP binding system is implemented in GNU Prolog (GProlog)[3]. We integrate the GProlog-based logic prover into our C-based arpsecd using the GProlog-C interfaces. Comparing with the IPC architecture between the arpsecd and the GProlog interpreter, this implementation improves the performance more than 50 times. The security parameter in the logic formulation is 5 now, which means we only trust the ARP binding history within the past 5 *seconds*. We have also implemented a white list and two black lists in the front of the logic layer within the arpsecd. The white list contains the MAC/IP bindings we trust under no conditions. The two black lists contain potentially malicious MAC addresses or IP addresses respectively. Currently, only the MAC/IP binding failed in the TPM attestation will be added into the black list accordingly. All the entries in the black lists have the same TTL - 200 *seconds*. A timer thread deletes the entry from the black lists once its TTL expires. The arpsecd TPM component is built on the top of Trousers API[23] following the TPM specification 1.2[24]. To store the TPM information (PCRs and AIK public keys) of remotes, an internal database is also implemented using a standard CSV configuration file. Note that this configuration file is only for the testing and not for the real deployment.

A TPM daemon (tpmd) is also implemented for the remote machines to process the TPM attestation from the arpsecd. The same as the arpsecd TPM component, the tpmd is also built upon the Trousers API following the TPM specification 1.2. To create the AIK pair and get the AIK credential for TPMs, PrivacyCA[16] is used instead of the real PCA. In our current implementation and testing environment, we are interested in the PCR `0-7`. More PCRs could be covered and extended if the measurement of applications is also cared.

5 Deployment

Besides the logic prover, *arpsec* relies on the knowledge of AIK and PCRs, which are built upon the TPM hardware. Different with S-ARP and TARP networks, where PKI system is bound to the MAC/IP binding in the ARP reply directly, *arpsec* does not care the IP address and thus does not distinguish the static IP network from the DHCP network. Instead, *arpsec* uses the MAC/AIK/PCRs binding to validate the trust to the remote. We introduce the idea of the TPM Information Management Server (TIMS) into the network. When a new machine tries to join the network, it has to create the AIK pair and get the AIK credential from the TIMS. In this step, the TIMS is acting as a Privacy CA or an Attestation CA (ACA). The procedure follows the AIK certificate enrollment scheme defined by the TCG Infrastructure Working Group. As one TPM could generate multiple AIKs, we require each new machine to create a new AIK pair using the TIMS when firstly joining the network. Once the AIK pair is created and certified, the new machine has to send its MAC address and all the PCR values (thought to be a good system integrity state) via a secure channel, like SSH, to TIMS. The TIMS then creates a new entry for this new MAC/AIK/PCRs binding in its database for this new machine and distributes this new TPM Information Entry (TIE) to all arpsecd running on the machines within the network via a secure channel again. Upon receiving the new TIE from the TIMS, the arpsecd will insert it into its own internal TPM information database. Note that the AIK/PCRs, bound with the MAC address, is also used for the TPM attestation by the TIMS for the TIE management down below.

There are some cases we need to care in the *arpsec* network. If the MAC address is changed for one machine, the machine has to let the TIMS be aware about this. In this case, the machine could send a TIE Update message containing the old and new MAC addresses to the TIMS. The TIMS should first launch a TPM attestation (just like the arpsecd) towards the requesting machine. Only upon the success of the attestation will the TIMS update the existing TIE with the new MAC address and distribute the TIE Update to other arpsecds. If a new MAC address is added into the machine, different actions will be taken based on whether the existing AIK is reused or not. If the new MAC address is bound with an existing AIK, a TIE Add message containing the new MAC address and the existing AIK should be sent to the TIMS. Again the TIMS will do the TPM attestation before creating a new TIE in its database and distributing this TIE Add to other arpsecds. If the new MAC address tries to use a new AIK, then it is the same procedure as a new machine joins into the network except an extra TPM attestation by the TIMS before the new AIK pair generation, making sure the machine is in a good integrity state. Anytime one machine wants to discard its existing AIKs or change the TPM hardware or upgrade the BIOS/operating system (and thus changing the PCRs), it has to send a TIE Remove with all the registered MAC addresses before it starts a new registration. The TIMS will then clear all the entries with those MAC addresses and notify all arpsecds to remove those entries too. Though a hybrid network is not desired, the arpsecd has implemented a white list for the machines which may not have TPM hardwares but have to be trusted anyhow, like gateway routers or DNS servers.

6 Performance Evaluation

To fully understand the overhead of *arpsec*, we compare our implementation with the original ARP and other two important methods, S-ARP and TARP. We follow the experiment settings of TARP, providing two types of measurement: the macro-benchmark and the micro-benchmark. The macro-benchmark is the overall system overhead one can see from the application layer. The micro-benchmark provides performance profiles of some important operations within S-ARP, TARP and *arpsec*, including key generations and TPM interactions.

Our testing environment involves 4 Dell Optiplex 7010 desktop PCs. All these machines have Quad-Core Intel i5-3470 3.20 GHz CPU, 8GB memory, Intel Pro/1000 ethernet card (1000 Mbps full duplex) and are running on Ubuntu LTS 12.04 (x86-64) with the Linux kernel version 3.2.0.55. All these machines are equipped with the TPM hardware from STM (version 1.2 and firmware 13.12) and using the Trousers API 1.2 rev 0.3. To eliminate the impact from the Internet (DNS, routers, gateways), all the 4 machines are connected with a 1000-Mbps HP ProCurve layer-2 switch, constructing a LAN network. Finally, as S-ARP and TARP were written on old Linux kernel 2.6, we have ported the S-ARP and TARP implementations to our testing environment without changing the core functionalities. All the experiments down below share the same hardware and software configurations.

6.1 Macro-benchmark

The macro-benchmark is based on the round-trip-time (RTT) from the `ping` command. Instead of using certain profiler or direct measurement in the kernel, we use the common `ping` command to evaluate the general overhead from the view of applications. Though indirect measurement, the RTT from the `ping` command actually shows the whole system overhead in the wild accurately. The macro-benchmark we used is also consistent with the ones used by S-ARP and TARP. Then can we have a better comparison among these different methods.

Like TARP, we also implemented a custom `ping` command: `ncping` (no-cache ping), which clears the local ARP cache before each ICMP echo request is sent. With the `ncping` command, we can get the performance evaluation in the worst case and reveal the true overhead of different methods. We have performed three groups of experiments including `ping` with the target MAC/IP binding in the ARP cache, `ping` without the target MAC/IP binding in the ARP cache and `ncping`. Each group has testings for the original ARP, S-ARP, TARP and *arpsec* respectively. Each testing is consisted of 1000 ICMP echo requests or 10×1000 requests for the `ping` without caching. Figure 5, 6 and 7 show the evaluation for all these testings.

Figure 5 shows the RTT average (mean), min ($mean - 2\sigma^2$) and max ($mean + 2\sigma^2$) from the `ping` command with the target binding in the ARP cache. All these measurements are in millisecond. Note that in all our experiments, the internal caching of S-ARP and TARP is always enabled to get the best performance. As shown, once the target binding is in the ARP cache, RTT average values of all these methods look similar ranging from 0.210 ms to 0.240 ms. The max and min values among these methods are also comparable. This makes sense as no ARP/RARP message processing happens in this case. However, S-ARP introduces the biggest overhead among all these methods. Interestingly, *arpsec* has the smallest `ping` RTT (0.018 ms faster than the original ARP). On one hand, we attributes this to our kernel modification, where all ARP messages are relayed to the user space rather than further processing in the kernel space. On the other hand, this reveals the high efficiency of kernel relay.

Figure 6 demonstrates the most common scenario where the target binding is not in the ARP cache at first. In this case, the first `ping` takes much more time comparing with the followings as an ARP request will be broadcasted and

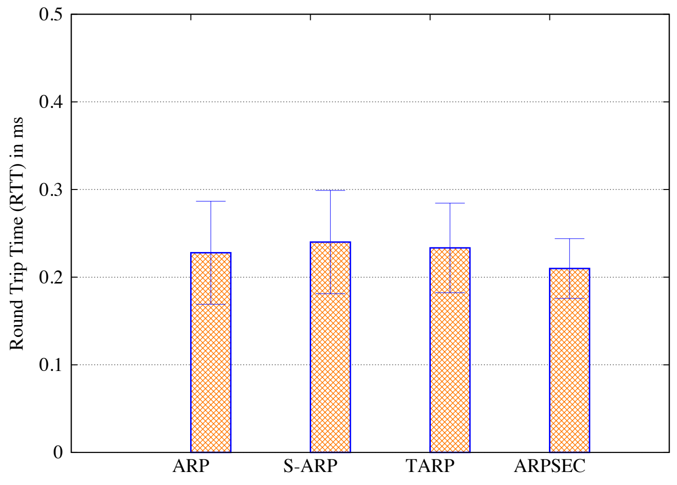

Fig. 5. The RTT (ms) of `ping` command with the target binding in the ARP cache for 1000 ICMP echo requests

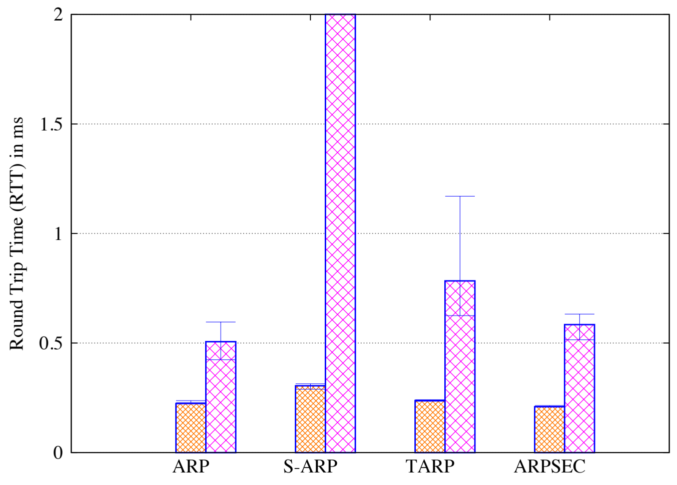

Fig. 6. The RTT (ms) of `ping` command without the target binding in the ARP cache for 10×1000 ICMP echo requests

the corresponding reply will be handled before the MAC/IP binding could be added into the ARP cache. Once the reply is processed and the binding is added into the ARP cache, the whole system run into the same case in Figure 5 and the RTT average values of all these methods converge to the similar level of the original ARP. To show the average time of the first-ARP-Reply processing, we repeated the 1000-run `ping` for 10 times. S-ARP, TARP and *arpsec* daemons were restarted each time to show the real processing time without the help of caching. As shown in the figure, for each method, the left bar is the average of 10-time averages for 1000 runs and the right bar is the average of 10-time first `ping`s. The max and min values here are the real ones rather than the confidence intervals. The left bars basically show the case in Figure 5. From the right bars, one can see that comparing with the original ARP, *arpsec* has the smallest overhead by 15.4%; TARP has a medium overhead by 54.7%. S-ARP, without the help of caching, introduces the biggest overhead, taking average 64 *ms* for the new MAC/IP binding (and the value is not shown in the figure).

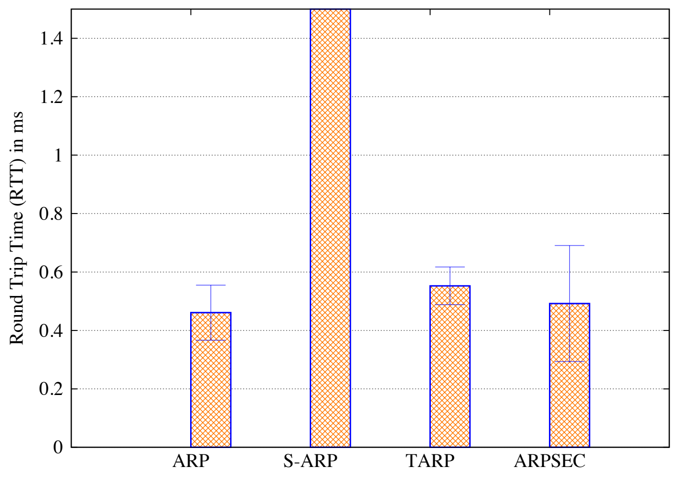

Fig. 7. The RTT (*ms*) of `ncping` command for 1000 ICMP echo requests

Figure 7 displays the worst case using the `ncping` command, where the ARP cache will be flushed before each ICMP echo request is sent. The max and min values here are the confidence intervals like Figure 5. With the help of internal caching and one-setup signature validation, TARP introduces a small overhead by 19.9% comparing with the original ARP. Even with caching, S-ARP still shows the biggest overhead with the RTT average value 8.4 *ms* (not shown in the figure) because of the time synchronization and communication with the

AKD. Comparing with S-ARP and TARP, *arpsec* performs the best introducing a 7% overhead. Note that the TPM attestation would not be triggered until the logic prover fails. The security parameter used by the logic prover is 5 seconds in our testing. Also, the RTT value of *arpsec* does not mean that the TPM operation is fast. Moreover, because of the asynchronous TPM operation design, the RTT value of *arpsec* is free from the degradation caused by the TPM attestation, but only limited by the user vs. kernel space communication and the logic prover. We will detail this in the later section.

6.2 Micro-benchmark

Using the GProlog-C interfaces, the logic prover could run as a pure C component without dragging down the overall performance of the arpsecd. The prominent bottle neck in *arpsec* then is the TPM hardware. In general, we know that the TPM hardware is slow comparing with the normal CPU. But how slow? In the micro-benchmark, we first compare the key generation time among all these methods and then dive into different TPM operations used by *arpsec*. Table 1 and 2 give the details.

Protocol	Min	Avg	Max	Mdev
S-ARP	36.155	90.364	330.722	34.794
TARP	5.17	31.005	69.476	10.826
TARP*	0.47	1.007	1.068	0.022
arpsec	3879	12841	46759	6062

Table 1. The Time (ms) of 100 key generations with the key length 1024 bits

TPM	Min	Avg	Max	Mdev
AIKgen	864	9385	43716	5932
Rand	10.915	11.399	11.468	0.035
Quote	324.467	336.109	336.541	0.698
SigVerify	0.120	0.199	0.213	0.006
AttVerify	0.208	0.307	0.344	0.009

Table 2. The Time (ms) of different TPM operations used by *arpsec* with 100 repetitions

Table 1 shows the key generation time of different methods. Note that TARP* stands for the ticket generation instead of the public/private key pair generation. *arpsec* means the TPM AIK pair generation. As one can tell, S-ARP does well with the mean time 90.364 ms. TARP is the fastest with the mean time 32.012 ms (public/private key pair + ticket). Comparing with local key generations of

S-ARP and TARP, *arpsec* is the slowest with the mean time 12.841 seconds, because we use the PrivacyCA as the PCA to generate the AIK pair, following the complicate AIK certificate enrollment scheme mentioned before. Fortunately, the AIK generation is one-time effort. After this, the AIK private key is stored in the TPM and could be used in a secure manner.

Table 2 profiles some TPM operations used by *arpsec*. AIKgen refers to Trousers API `Tspi_TPM_CollateIdentityRequest()`. Rand refers to `Tspi_TPM_GetRandom()`. Quote refers to `Tspi_TPM_Quote()`. SigVerify refers to `Tspi_Hash_VerifySignature()`. Note that AttVerify includes all the function calls related with the TPM attestation verification for the arpsecd. Again, AIKgen is time consuming, like we saw in Table 1. Besides it, the TPM Quote may be the slowest with the mean time 336.109 *ms* comparing with other TPM operations in this table. Be aware that all these performance evaluations only apply to our testing environment using TPM hardwares from STM (version 1.2 and firmware 13.12).

Combining the macro-benchmark and micro-benchmark, as well as design and implementation details mentioned before, we summarize the general characteristic comparison among S-ARP, TARP and *arpsec*, as shown in Figure 3.

Protocol	Mechanism	Formal Prove	Remote Integrity Measurement	ARP Change	Kernel Change	Overhead
S-ARP	PKI	N	N	Y	Y	Huge
TARP	Ticket-based PKI	N	N	Y	N	Small
arpsec	Logic+TPM	Y	Y	N	Y/N	Small

Table 3. General characteristic summary

7 Discussion

As we talked before, *arpsec* relies on the logic prover and the TPM attestation. The logic formulation for the ARP binding system we have created is simple, straightforward and intuitive. This is partially because the original design of ARP is simple. Even with the simple logic rules, we are able to record all the ARP cache update events, which enables the ARP cache data provenance 'accidentally'. On the one hand, with the ability to 'remember', the logic prover, besides reasoning, could potentially serve other applications, like forensics or machine learning. On the other hand, the logic system itself could be extended based on the complexity of the protocol we are trying to formalize.

To enable the TPM attestation, we introduce a TPM daemon (tpmd) and require each remote machine to install it. Note that an extra TPM daemon is necessary based on the TPM specification and the Trousers API. The Trousers API provides a tcsd daemon process with RPC interfaces. But these interfaces are for the remote TPM management rather than the TPM attestation from a challenger. Our tpmd follows the routine talking with the tcsd instead of calling the TPM directly like libtpm. One may argue what if the tpmd is compromised. In this case, the tpmd could forge the PCR values using the previous good values (if they are saved in the disk). But it is unable to forge the results of the TPM Quote command, which is executed by the TPM hardware using the PCRs and the AIK private key only known and seen to the TPM hardware itself.

Even using the TPM attestation, *arpsec* still introduces a security hole in the whole attestation procedure. Once the logic layer fails, the TPM layer tries to send a challenge to the remote by adding the to-be-validated MAC/IP binding to the local ARP cache temporarily. If the binding turns out to be trusted, that is fine. Otherwise, before the binding is removed from the cache, we will trust the adversary for around 300-500 ms because of TPM operations (Table 2). This situation gets even worse when TCP transmission delay happens. Currently, we set the TCP socket timeout to be 2 *seconds*. To eliminate this security hole, the ARP request could carry the challenge and the ARP reply should include the AT reply as well. However, by doing this, we had changed/extended the ARP like S-ARP and TARP did.

The limitation of the TPM attestation is that it only attests to what was loaded into the system during the boot time. It does not tell if the running code has been compromised or not. To attest to the current integrity state of a running system, a dynamic measurement of the whole Trusted Computing Base (TCB) and target applications should be needed, which means we have to measure the boot loader, the operating system, the libraries and applications - almost everything in the memory. Besides the TCB size, secure storing and fast execution of the measurement are the other two critical challenges of the runtime integrity checking. Advanced TPM chips or cryptographic coprocessors may provide elegant solutions without hurting the overall system performance.

Though not designed for DoS attacks, *arpsec* could handle certain DoS attack in the ARP layer. As mentioned before, once the TPM attestation fails, the malicious MAC address or IP address will be added into the corresponding black list. When the same MAC address or IP address is contained in the following ARP/RARP reply, the reply will be dropped without processing. However, if the malicious MAC address or IP address keeps changing, *arpsec* has to examine each message as the black list does not help in this case. Moreover, if the DoS attack is triggered from the higher layer, like ICMP or even HTTP, *arpsec* is not able to prevent that and those attacks are outside the scope of this paper.

The MITM attack is another one worth attention. As mentioned in the deployment, each machine has to prove its identity using the TPM hardware and provide the good PCRs for future reference before joining the network. As long as the PCRs stay unchanged, we trust this machine. However, machines with good PCRs could still play MITM attacks by injecting a new TIE with a new MAC address into the TIMS. Making the TIE include the IP address could be a solution but the IP address inclines to change in the wild. This solution would also burden the management of the TIE and complicate the design of the TIMS.

Like other TPM related protocols, *arpsec* is not able to detect the TPM cuckoo attack[13] either.

As shown in Table 2, *arpsec* performance is limited by the TPM hardware. The TPM chip is designed to be cheap - only few dollars. While the low price helps embed a TPM chip into each machine even in mobile phones, it limits the scope of TPM usages. The ARP may (or maybe not) bear the 336 ms TPM Quote command delay, the IP packet processing within the kernel would never allow that happen. As the TPM 2.0 library specification is published for review now, we are hoping that the new specification could promote and enhance the TPM hardware performance and thus extend the usage scope.

8 Related Work

A lot of researches have been done trying to conquer the ARP security issues. The easiest way with zero overhead is the static ARP cache configuration[25]. Predefined MAC/IP bindings could be added into the ARP cache manually. While this method sounds like a perfect solution, one can imagine how complicated it would be to manage and update these bindings in a dynamic IP world. This method works for DoS attacks but not for MITM attacks. Some other methods focus on the ARP spoofing detection. ARPWatch[8] monitors the ARP network traffic. By recording all the ARP cache updates, it could email the administrator if certain MAC/IP binding is changed. While ARPWatch notifies some changes happened in the local ARP cache, it leaves the judgement to the user and the changes have already happened anyway.

Other solutions use security policies to prevent ARP attacks. ArpON[14] defines different ARP binding policies for different networks, including static networks, dynamic networks or hybrid networks. Instead of using centralized server for management, it requires each host within the network to run the ArpON daemon and respect the same policies. The cooperative authentication could then be achieved and prevent ARP attacks. ArpON claims to countermeasure the MITM attack. The only problem here is the complexity of defining and updating the policies for different network environments besides the foreseeable overhead - the cooperative authentication.

A middleware approach is also proposed to provide both the backward compatibility and the flexibility the same time[22]. It does not need the operating system changes by taking the advantage of Streams paradigms of the Solaris operating system. It is claimed to have both preventions and detections. However, as no cryptograph is involved, there is no solid ARP message authentication mechanism except the ARP layer hardening in the Solaris. Moreover, the whole design is based on the traditional wireline telecom networks - COIPP. The unified maintainence and alarm interfaces are usually not available in the Internet.

Recent solutions apply Public Key Infrastructure (PKI) to the ARP security enhancement. S-ARP[2] uses self-generated public/private key pairs and a Authoritative Key Distributor (AKD) for the ARP reply authentication. Each host generates its own key pair and the public key is registered in the AKD.

The AKD then is responsible for creating the MAC/public-key binding in its internal database and distributing the binding to each S-ARP host within the network. Upon receiving the ARP reply signed by the remote's private key, the host may have to ask the AKD for the public key if it is not cached or the key is expired. To prevent replay attacks, time synchronization is needed between the host and the AKD. Besides the complexity of the key management and time synchronization, S-ARP introduces a big system overhead comparing with the original ARP implementation[9].

Similar with S-ARP, TARP[9] uses the PKI system but in a different way. Instead of self-created public/private key pair for each host, only the Local Ticket Agent (LTA) generates the key pair. The LTA signs the valid MAC/IP binding using its own private key as a ticket. All the ARP replies have to include tickets from the LTA for attestation. Upon receiving the reply, the host uses the LTA's public key to validate the MAC/IP binding. By transferring the workload from each host to the LTA, TARP introduces a fairly small system overhead for the host. The same as S-ARP, TARP also changes/extends the original ARP to carry the extra information used for authentication. However, the question still remains - can we trust the remote even if the ticket verification succeeds?

In this paper, we are looking for a possible solution for the ARP security, which provides a flexible yet reliable security mechanism rather than hard-coded policies, tackles all the known ARP attacks, mitigate high-level attacks, keeps backward compatibility and gives a provable trust of the remote by only introducing $7\% - 15.4\%$ system overhead.

9 Conclusion

The ARP is simple but essential to the whole IPv4 network. A lot of efforts have been done trying to provide the ARP with security features. Recent works take the advantage of the PKI system to prevent the ARP spoofing and ARP cache poisoning. But we are still not able to answer the questions, like what if the private key is broken or how could we know if the remote is trustworthy or not. Different with all the past methods on the ARP security, *arpsec* provides a logic prover to reason about the validness of ARP/RARP replies and uses the TPM attestation to guarantee the trust of the remote. Comparing with the original ARP, *arpsec* introduces only $7\% - 15.4\%$ system overhead.

The ARP security will still remain a hot topic in the future research unless some security mechanism is accepted, adopted by the community and widely deployed in the wild. *arpsec* shows a new solution here. Currently, besides the code optimization, we are looking for places where we could apply the framework - logic formulation framework plus the TPM attestation, through which we could achieve a flexible, reliable yet trustworthy security mechanism.

References

1. Bellovin, S. M. Security problems in the tcp/ip protocol suite. *Computer Communications Review 2* (April 1989), 32–48.

2. Bruschi, D., Ornaghi, A., and Rosti, E. S-ARP: a Secure Address Resolution Protocol. *Computer Security Applications Conference 2003 Proceedings. 19th Annual* (2003), 66–74.
3. Diaz, D., et al. The GNU Prolog web site. http://gprolog.org/.
4. Finlayson, R., Mann, T., Mogul, J., and Theimer, M. A Reverse Address Resolution Protocol. http://tools.ietf.org/rfc/rfc903.txt, June 1984.
5. Issac, B. Secure AP and Secure DHCP Protocols to Mitigate Security Attacks. *International Journal of Network Security 8* (March 2009), 107–118.
6. Jaeger, T., Sailer, R., and Shankar, U. PRIMA: policy-reduced integrity measurement architecture. *Proceeding SACMAT '06 Proceedings of the eleventh ACM symposium on Access control models and technologies* (2006), 19–28.
7. Kauer, B. OSLO: Improving the security of Trusted Computing. *Proceeding SS'07 Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (2007).
8. Lawrence Berkeley National Laboratory Network Research Group. arpwatch: the ethernet monitor program. http://ee.lbl.gov/, 2006.
9. Lootah, W., Enck, W., and McDaniel, P. TARP: Ticket-based Address Resolution Protocol. *ACSAC 2005* (2005).
10. Nathan, J. Nemesis. http://nemesis.sourceforge.net/, 2004.
11. Ornaghi, A., and Valleri, M. Man in the middle attacks Demos. http://www.blackhat.com/presentations/bh-europe-03/bh-europe-03-valleri.pdf, 2003. Blackhat 2003.
12. Ortega, A. P., Marcos, X. E., Chiang, L. D., and Abad, C. L. Preventing ARP Cache Poisoning Attacks: A Proof of Concept using OpenWrt. *Network Operations and Management Symposium 2009* (2009), 1–9.
13. Parno, B. Bootstrapping trust in a "trusted" platform. *Proceeding HOTSEC'08 Proceedings of the 3rd conference on Hot topics in security* (2008).
14. Pasquale, A. D. ArpOn: ARP Handler Inspection. http://arpon.sourceforge.net/index.html, 2008.
15. Plummer, D. C. An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. http://tools.ietf.org/search/rfc826, November 1982.
16. PrivacyCA. Privacy CA. http://privacyca.com/.
17. Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. *Proceedings of the 13th USENIX Security Symposium* (2004).
18. Schmitz, J., Loew, J., Elwell, J., Ponomarev, D., and Abu-Ghazaleh, N. A Framework for Performance Evaluation of Trusted Platform Modules. *Design Automation Conference (DAC) 48th ACM/EDAC/IEEE* (2011), 236–241.
19. Song, D. dsniff. http://monkey.org/~dugsong/dsniff/, 2000.
20. Tarnovsky, C. Deconstructing a 'Secure' processor. *Black Hat DC* (2010).
21. Tian, J., Butler, K., and McDaniel, P. ARPSEC source code. https://github.com/daveti/arpsec_gpc, 2014.
22. Tripunitara, M. V., and Dutta, P. A middleware approach to asynchronous and backward compatible detection and prevention of ARP cache poisoning. *Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual* (1999), 303–309.
23. TrouSerS. The open-source TCG Software Stack. http://trousers.sourceforge.net/.
24. Trusted Computing Group. TPM Main Specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

25. Whalen, S. An Introduction to ARP Spoofing. `http://rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf`, 2001.
26. Zanussi, T., et al. relay (formerly relayfs). `http://relayfs.sourceforge.net/`.