# Securing ARP From the Ground Up

Jing (Dave) Tian
University of Florida
Gainesville, FL, USA
daveti@ufl.edu

Kevin R. B. Butler
University of Florida
Gainesville, FL, USA
butler@ufl.edu

Patrick D. McDaniel
The Pennsylvania State University
University Park, PA, USA
mcdaniel@cse.psu.edu

Padma Krishnaswamy
Federal Communications Commission
Washington, DC, USA
padma.krishnaswamy@gmail.com

## ABSTRACT

The basis for all IPv4 network communication is the Address Resolution Protocol (ARP), which maps an IP address to a device's Media Access Control (MAC) identifier. ARP has long been recognized as vulnerable to spoofing and other attacks, and past proposals to secure the protocol have often involved modifying the basic protocol.

This paper introduces *arpsec*, a secure ARP/RARP protocol suite which a) does not require protocol modification, b) enables continual verification of the identity of the target (respondent) machine by introducing an address binding repository derived using a formal logic that bases additions to a host's ARP cache on a set of operational rules and properties, c) utilizes the TPM, a commodity component now present in the vast majority of modern computers, to augment the logic-prover-derived assurance when needed, with TPM-facilitated attestations of system state achieved at viably low processing cost. Using commodity TPMs as our attestation base, we show that *arpsec* incurs an overhead ranging from 7% to 15.4% over the standard Linux ARP implementation and provides a first step towards a formally secure and trustworthy networking stack.

## 1. INTRODUCTION

The Address Resolution Protocol (ARP) [20] is a fundamental part of IPv4 network connectivity. Operating below the network layer, ARP binds an IP address to the Media Access Control (MAC) identifier of a network device, e.g., an Ethernet card or a Wi-Fi adapter, which in turn completes the process of routing the packet to its intended destination. Such communication relies on the last hop for correct delivery. ARP is subject to a variety of attacks including spoofing and cache poisoning, as originally described by Bellovin [1]. Tools such as *dsniff* [26] and *nemesis* [14] can be used respectively to easily launch such attacks. An attack on ARP can subsequently enable more sophisticated denial-of-service (DoS) and man-in-the-middle (MitM) [15] attacks.

While numerous methods have been proposed to secure ARP [35, 9, 18, 16, 30], they fall short of offering a comprehensive solution to these problems. First, a successful security solution must ensure that the basic ARP protocol itself remains unchanged. There is no "flag day" on which 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 system performance. Third, the ARP security mechanism should be flexible and reliable. Hard-coded security policies may not be applicable to varying network environments. Last, we need to know if the remote machine can be trusted. Trust here applies to both the authentication and the system integrity state of the remote machine, e.g., even if a binding is correct, we may not wish to add a remote host that cannot attest to the correctness of its operation. While past proposals have ranged from localized solutions to those involving public key infrastructures [2, 10, 5], they have not been widely deployed, either due to requiring specific network configurations, creating large system overheads, or requiring fundamental changes to ARP.

In this paper, we propose *arpsec*, an ARP security approach based on logic and the use of the Trusted Platform Module (TPM) [32], to implement security guarantees. *arpsec* does not change or extend the ARP itself. Instead of hard-coded security policies, *arpsec* formalizes the ARP system binding using logic. A logic prover then reasons about the validity of an ARP reply from the remote machine based on the codified logic rules and the previously stored binding history on the local system. A TPM attestation protocol is also implemented to challenge the remote machine if the logic layer fails to determine the trustworthiness of the remote machine. Using TPM hardware, we can authenticate (establish the identity) of the remote and discover whether the remote machine is in a good integrity state (i.e., not compromised). *arpsec* defends from most categories of ARP attacks by tethering address bindings to trusted hardware, establishing the basis for a trustworthy networking stack.

We have implemented *arpsec* in the Linux 3.2 kernel, using commodity TPMs and a Prolog engine. Our experiments show that *arpsec* only introduces a small system overhead, ranging from 7% to 15.4% compared to the original ARP and incurs the lowest overhead when compared to the two PKI-based ARP security proposals, S-ARP [2] and TARP [10].


*CODASPY'15* March 02 - 04 2015, San Antonio, TX, USA

ACM 978-1-4503-3191-3/15/03 ...$15.00.
http://dx.doi.org/10.1145/2699026.2699123.

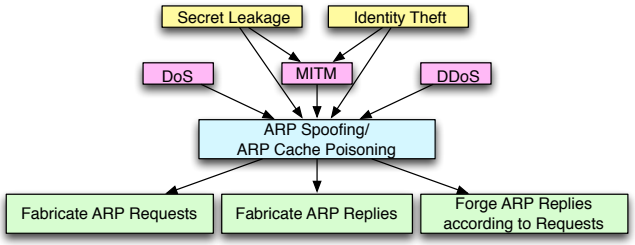



Figure 1: An attack tree for ARP.

The remainder of this paper is structured as follows. Section 2 outlines the background on ARP security issues and trusted computing. Section 3 details the design and architecture of *arpsec*. Section 4 shows details and tradeoffs during the implementation. Section 5 provides the performance evaluation. Section 6 discusses potential issues with *arpsec* and possible solutions. Section 7 reviews the past efforts on ARP security, and Section 8 concludes.

# 2. BACKGROUND

We first discuss ARP security issues based on the current ARP design and implementations, before explaining common attacks against ARP. As *arpsec* uses the TPM, a brief review of trusted computing is provided.

## 2.1 ARP Security Issues

ARP [20] is the glue between Layer 3 and Layer 2 in IPv4 networks, allowing for a binding between IP addresses and medium access control (MAC) addresses unique to a particular network interface card (NIC). [1] Before an IP packet is sent out from a NIC (e.g., an Ethernet card), the host's ARP cache is queried to find the MAC address assigned to the target IP address of the packet. If the MAC/IP binding is not found, an ARP *request* will be broadcast to the entire network segment (the *broadcast domain*). Only the host with the target IP address should send back an ARP *reply* containing its MAC address.

In reality, 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. In this case, most operating systems either accept the first reply or the latest one if multiple replies respond to the same request. They further optimize performance by processing ARP requests from other machines and adding MAC/IP bindings for future use. Though all bindings in the ARP cache have some Time-To-Live (TTL) control, the timer is usually large and designed for performance rather than security. As an example, Linux always accepts the first ARP reply to the request and ignores others. It also rejects ARP replies without a request while processing ARP requests from other machines. The TTL for each entry in the Linux ARP cache is around 20 minutes [2]. Solaris and Windows have similar optimizations and hence, similar security issues [27, 12].

One basic attack against ARP is 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 other basic ARP attack is cache poisoning [35], where the adversary generates the ARP reply using certain MAC address given the request from the victim. Both spoofing and poisoning attacks attempt to insert a malicious MAC/IP binding in the victim's ARP cache.

As shown in Figure 1, the attacks described above act as enablers for other adversary actions, such as man-in-the-middle (MitM) attacks [15] and denial of service (DoS) [35] attacks. For a DoS attack, the adversary can inject the victim's MAC address into a particular machine or substitute the victim's MAC address with another one. In the former case, all the IP traffic from that machine targeting a certain address will be redirected to the victim, while in the latter case, the victim would never receive the messages intended for it to provide the service. MitM attacks are particularly serious, since with the help of ARP spoofing/poisoning, the adversary can interpolate himself into the traffic between victims by injecting his MAC/IP binding into both victims' ARP cache. Both attacks are also quite simple to implement, with small usable scripts widely available, and these in turn can lead to attacks compromising user identity or allowing the leakage of secret information.

## 2.2 Trusted Platform Module (TPM)

A Trusted Platform Module (TPM) is a cryptographic chip embedded in motherboards. Though implemented by various vendors, all TPM chips follow the TPM specification [32] designed by the Trusted Computing Group (TCG). In conjunction with the system BIOS, TPMs can be used to form a root of trust in a system and to build the trust chain for the software along the software stack, including boot loaders, operating systems and applications [21, 6, 8, 17].

TPMs can help to determine the true identity of a remote host via the Attestation Identity Key (AIK) verification during the TPM attestation. After creating an AIK pair, the TPM hardware communicates with a Privacy Certification Authority (PCA) or Attestation Certification Authority (ACA) using the information embedded in itself to prove its identity and get the AIK credentials. A remote machine proves its integrity state by reporting the values of its Platform Configuration Registers (PCRs). If the *measurement* of PCR values during a TPM attestation different from what is expected, the remote may be compromised and thus not trustworthy. It is important to realize that the AIK private key and the measurement of PCRs are all stored in the TPM itself. Unless the TPM hardware is compromised [28], there is no disclosed method of hacking into the TPM through software and changing PCR values.

# 3. DESIGN

## 3.1 Threat Model

The hardware, BIOS, boot loader, operating system and the corresponding system libraries, as well as the *arpsecd* daemon, are trusted components in our local host. However, except for the TPM hardware in the remote machine, we do not trust anything generated by the remote machine. Moreover, the adversary may have compromised the remote machine and gained root permission, through which any ARP attacks can be launched, including ARP message spoofing and ARP cache poisoning. The adversary may also leak secret information he has gotten from the victim and use this information to impersonate the victim on another machine while taking the victim machine offline. In short, for the

[1] Reverse ARP [4] is generally obsolete in favor of other bootstrapping protocols such as DHCP and BOOTP.

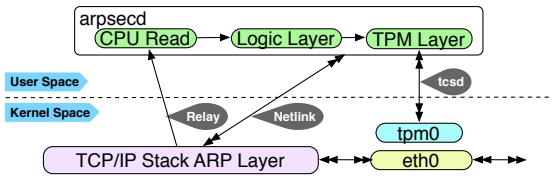



Figure 2: The architecture of *arpsec*.

```
while there is an ARP 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
end
```

**Algorithm 1:** ARP message processing within *arpsec*

local *arpsec* host machine, the TCB includes all hardware and system software required to start *arpsecd*; the local host machine should also only trust the TPM hardware within the remote machine during the ARP processing.

It is important to note that the TPM hardware attacks, like TPM deconstructing [28] and TPM reset attack [8] are not considered in this paper, nor are potential TPM relay attacks, whose preferred solution is a special-purpose hardware interface [17]. Also, *arpsec* is not designed for DoS/DDoS attacks, though it has the ability to defend against simple DoS attacks, which we discuss further in Section 6.

## 3.2 System Design

Compared to proposals such as S-ARP and TARP, which take advantage of the PKI system to extend ARP, *arpsec* formalizes ARP address binding and validates ARP messages using both a logic prover and TPM attestations, all without requiring any changes to the original protocol. The architecture of *arpsec* is illustrated in Figure 2.

In the user space, *arpsecd* is the daemon process of *arpsec* that runs in the local machine and takes control of processing of all the ARP messages from the kernel. There are three major components in the *arpsecd* daemon: the CPU read, the logic layer and the TPM layer. The CPU read component retrieves all ARP 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 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 reply) validated by the logic layer or TPM attestation could be added into the local ARP cache. The pseudo code of *arpsecd* ARP processing is listed in Algorithm 1. Note that *arpsec* ignores ARP requests not for itself. These are usually processed by ARP implementations for performance but leave the ARP cache vulnerable.

## 3.3 Logic Formulation

The logic layer in *arpsec* is the first filter used to testify the trustiness of an ARP/RARP reply message. The logic layer imposes minimal performance costs when compared to the TPM layer, using a logic prover and a list of ARP logic rules. To leverage the power of the logic reasoning, firstly, we introduce an ARP system binding logic formulation.

**Intuition**: *The logic layer tracks statements (attestations) by systems that particular media addressed are mapped to network addresses. The timing of these statements are tracked such that the logic can "prove" exactly which binding is the most authoritative at a given time. The logic judges a binding to be authoritative if it is the most recent one received from a trusted system. At runtime, the system generates, if possible, the proof of a binding before using it for network communication.*

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

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

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, both address sets contain a special address $\epsilon$ representing the lack of binding assignment, described below. $\mathcal{S}$ is the set of systems that makes 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 $t_j$. $\bar{\mathcal{R}}$ is 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:

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

$\mathcal{R}$ is the key conceptual element here; each element of $\mathcal{R}$ captures the fact that system $s_i$ stated (through an attestation) a binding of network address $n_j$ to media address $m_k$ at time $t_l$. The remainder of the logic simply reasons from the set of statements which binding should be considered authoritative at a given time.

**Trust state** : The trust state $\mathcal{A}$ of the system is a totally ordered 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:

$$A_k^h = \{s \mid \exists j, (k-h) \leq j < k : (s, j) \in \bar{\mathcal{S}}\}$$

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 $B_k$, which is a relation over $\mathcal{N}$ and $\mathcal{M}$ representing the instantaneous binding of network to media addresses, where:

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

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

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

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 $\epsilon$ must be bound to exactly one media address (3), and the media address not bound to the unassigned element $\epsilon$ must be bound to exactly one network address (4).

We 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:

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

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 exists a binding record from the same (trusted) remote in the past that is no older than a pre-defined number of epochs (security parameter $h$) when compared to the current epoch. 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. Security parameter $h$ represents a tradeoff between reliability and performance, as it determines the time range of the past we would trust to validate the current event.

The Prolog engine continuously consumes assertions received at an end host and infers $B_k$ at each time epoch using the above constraints. That generation thus provides a proof of authority; if a binding $(n_i, m_j) \in B_k$, then it is authoritative and can be used for communication at time $t_i$.

## 3.4 TPM Attestation

If an incoming ARP/RARP request or reply cannot be validated by the logic layer, *arpsec* turns to a TPM layer as a second line of defense. To establish trust in the remote host, we use a TPM attestation [32], whose general operation is described in Section 2. A *measurement* is taken based on the current state of the underlying hardware, BIOS, boot loader, and operating system, with each value stored in a PCR. The TPM is tamper-resistant and access to PCRs is not possible except through expensive hardware attacks. The Attestation Identification Key (AIK) thus provides identity while the PCRs determine system integrity state.

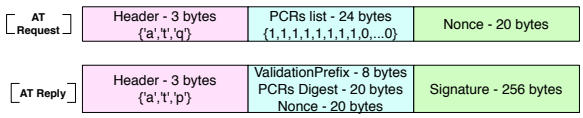



Figure 3: The AT Request/Reply

We design the *arpsec* Attestation (AT) protocol for communication between the local machine (also known as the challenger) and the remote machine (also known as the attester), as shown in Figure 3. The request contains a header, a list of PCRs, and a nonce to prevent replay. The PCR list indicates which registers are of interest - for our purposes, these are registers 0 through 7. [2] When the host receives this challenge, it responds with a TPM *Quote*, which includes the nonce, PCR values and their corresponding digest, signed by the AIK private key, in the AT reply. If at this point any of these values fail and the signature cannot be validated, the address binding is purged from the ARP cache. Note that we do not put any MAC/IP binding into the AT reply. Comparing to the TPM, a MAC/IP binding is easy to fake and thus not trustworthy.

# 4. IMPLEMENTATION

We have implemented *arpsec* in Linux with the 3.2.0.55 kernel, using C and Prolog. The implementation details of the *arpsecd* daemon is shown in Figure 4. Our goals were high performance and *incremental deployment* to allow *arpsec* and standard ARP to coexist in the same network.

Depicted in Figure 4, the *relay* mechanism [36] transports ARP messages from kernel to user space, as it is designed to manage large amounts of asymmetric traffic. We also use a netlink socket to communicate from user to kernel space, in order to manipulate the ARP cache. This provides similar functionality to the `ioctl()` calls for cache management but uses the low-level kernel APIs to get rid of the extra locking in `ioctl()`. Using this netlink socket, we could also trigger the kernel to send the ARP reply given any request, at which point it is relayed to user space for efficient processing.

In 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, providing a 50X performance improvement compared to IPC between *arpsecd* and the GProlog interpreter. We set a 5-second security parameter, meaning that every 5 seconds we expect a new attestation of the ARP binding.

We also implemented a whitelist and two blacklists before logic processing occurs. The whitelist contains the MAC/IP bindings known to be good under all conditions. The two blacklists 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

[2]PCRs 0 through 7 cover the measurement for hardware, BIOS, boot loader and even OS [32] Other PCRs could be extended to cover system libraries and even applications.

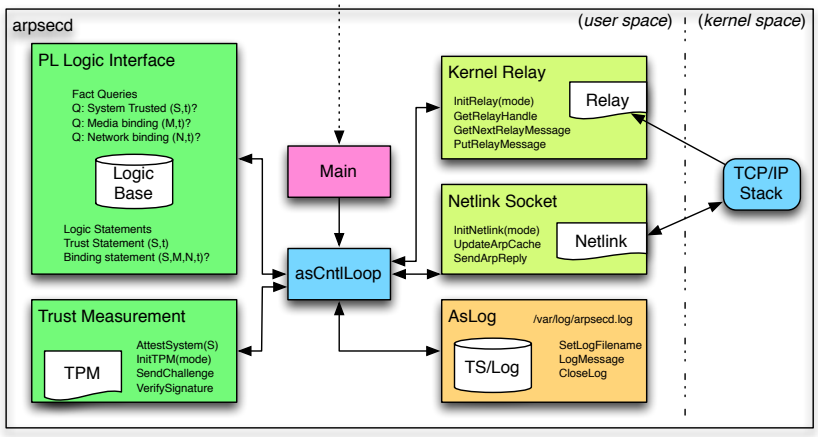



Figure 4: The implementation of *arpsec* daemon (*arpsecd*)

list . All entries in the blacklists have the same TTL of 200 seconds, at which point they are removed.

The *arpsec* TPM component is built on the top of the Trousers API [31] following the TPM 1.2 specification [32]. TPM information (PCRs and AIK public keys) of remote hosts is stored in an internal database.

*arpsec* relies on the knowledge of AIK and PCRs, which are built upon the TPM hardware. Compared to S-ARP and TARP, where keys are bound to a PKI, *arpsec* does not distinguish hosts through public keys. Instead, *arpsec* uses the MAC/AIK/PCRs binding to validate the trust to the remote host. To managing these bindings within an *arpsec* network, we introduce a *TPM Information Management Server* (TIMS) into the network. When a new machine wants to join the network, it creates the AIK pair to get the credential from the TIMS. The TIMS acts as a Privacy CA or an Attestation CA (ACA), and certifies the TPM within this new machine. The procedure follows the AIK certificate enrollment scheme defined by the TCG.

More information on the deployment of the TIMS and about the TPM Information Entry (TIE) bundles that we distribute from it can be found in our technical report [29].

# 5. PERFORMANCE EVALUATION

To fully understand the overhead of *arpsec*, we compare our implementation with standard ARP as well as the proposals that most closely mirror the security guarantees that we provide, S-ARP and TARP. We follow the experiment settings of TARP, providing macro- and microbenchmarks. Our testing environment involves 4 Dell Optiplex 7010 desktop PCs with quad-Core Intel i5-3470 3.20 GHz CPU, 8GB memory with Intel Pro/1000 full duplex Ethernet cards, running Ubuntu LTS 12.04 (x86-64) with Linux kernel version 3.2.0.55. All machines are equipped with TPM hardware from STM (version 1.2 and firmware 13.12), running Trousers API 1.2 rev 0.3. To eliminate the impact from exterior network traffic, all machines are isolated on a 1000-Mbps HP ProCurve switch. As S-ARP and TARP were written on Linux kernel 2.6, we have forward-ported the S-ARP and TARP implementations to our testing environment.

## 5.1 Macrobenchmark Testing

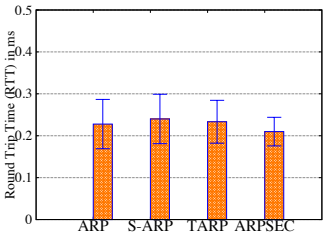
We benchmark performance based on the round-trip-time (RTT) using `ping` to provide overhead from an application's or user's perspective. This benchmark we used is also consistent with what was used by S-ARP and TARP. Like TARP, we also implemented a custom `ping` command: `ncping` (no-cache ping), which clears the local ARP cache before each ICMP echo request is sent. With `ncping`, 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: (*a*) `ping` with the target MAC/IP binding in the ARP cache, (*b*) `ping` without the target MAC/IP binding in the ARP cache, and (*c*) `ncping`. Each test consists of 1000 ICMP echo requests or $10 \times 1000$ requests for the `ping` without caching.

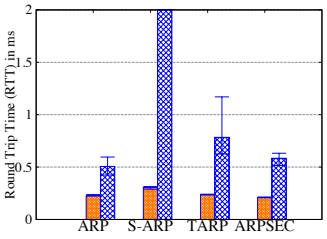
Figure 5a shows the RTT average (mean), min ($mean - 2\sigma^2$) and max ($mean + 2\sigma^2$) from the `ping` command with the target binding in the ARP cache. In all experiments, internal caching of S-ARP and TARP is enabled to maximize performance. Once the target binding is in the ARP cache, RTT average values of all these methods look similar ranging from 0.210 to 0.240 $ms$. The max and min values among these methods are also comparable, which is intuitive given no ARP processing is occurring. We attribute *arpsec*'s slightly faster processing time to efficiency of processing in user space and of the relay system.

Figure 5b demonstrates the most common scenario, where the target binding is initially in the ARP cache. The first `ping` now takes much more time, as the ARP request will be broadcast and the corresponding reply will be handled before the binding can be added to the cache. Once the reply is processed and the binding is added, the performance is the same as in Figure 5a and the average RTTs converge to be similar to standard ARP.

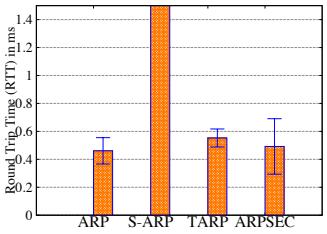
To show the average time of the first-ARP-Reply processing, we repeated the 1000-run `ping` for 10 times. S-ARP, TARP and *arpsec* daemons were restarted each time to show the real processing time without caching. As shown in the figure, the left bar is the average over all 1000 pings and the right bar is the average of 10 first-time `ping`s. The left bars show the amortized costs are close to cached processing. From the right bars, we see that after standard ARP,




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




(b) The average RTT ($ms$) of `ping` command without target binding in the ARP cache for 1000 ICMP echo requests over 10 runs. The right bar is the average RTT of the first `ping` during the 10 runs.




(c) The average RTT ($ms$) of `ncping` command for 1000 ICMP echo requests

Figure 5: The macro-benchmark of *arpsec*

*arpsec* has the smallest overhead by 15.4%. S-ARP, without the help of caching, introduces the biggest overhead, taking on average 64 $ms$ for the new MAC/IP binding.

Figure 5c displays worst-case performance using the `ncping` command, where the ARP cache will be flushed before each ICMP echo request is sent. With the help of internal caching and one-setup signature validation, TARP introduces a small overhead of 19.9% comparing with the original ARP. Even with caching, S-ARP still shows the largest 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 is not triggered until the logic prover fails. The security parameter used by the prover is 5 seconds in our testing. Also, the RTT value of *arpsec* does not mean that TPM operation is fast, only that a quote can be amortized in the overhead. Because of the asynchronous operation of the TPM, 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.

## 5.2 Micro-benchmark

Using the GProlog-C interfaces, the logic prover can run as a pure C component without affecting performance of *arpsecd*. The prominent bottleneck in *arpsec* is then the TPM hardware, which is slow compared to a CPU [22].

Table 1 shows the key generation time of different methods. Here TARP* shows the ticket generation instead of the public/private key pair generation. *arpsec* is the cost of TPM AIK pair generation and the AIK public key certification. S-ARP has a mean key generation time of 90.364 $ms$. TARP is the fastest with the mean time 32.012 $ms$ (public/private key pair + ticket). By contrast, *arpsec* is the slowest with the mean time 12.841 seconds, because we use the PrivacyCA to certify the AIK public key generated by the local TPM, following the complicated AIK certificate enrollment scheme described in the previous section. Fortunately, the AIK generation and certification is one-time effort. After this, the AIK private key is stored in the TPM and could be used in a secure manner. Moreover, both S-ARP and TARP have either the key expiration or the ticket expiration issue, which means after a certain time, either the key or the ticket has to be re-generated for each host within the network. In the long run, the time of *arpsec*'s one-time, offline key generation will be amortized by the key/ticket re-generation of S-ARP or TARP.

Table 2 profiles some TPM operations used by *arpsec*: AIK generation, cost of obtaining a random value, a TPM quote, signature verification, and TPM attestation verification, respectively. *AIKgen* is time consuming, as we saw from Table 1. Otherwise, the quote operation is the slowest with the mean time 336.109 $ms$. We summarize the comparison among S-ARP, TARP and *arpsec* in Table 3.

# 6. DISCUSSION

*Arpsec* is comprised of both a Logic Layer and a TPM Layer; we now discuss the implications of each of these layers. The logic formulation for the ARP binding system we have created is simple, straightforward and intuitive, due in part to the simple design of ARP. Even with these simple logic rules, we are able to record all ARP cache update events, which implicitly capture the provenance history of the ARP cache. This could potentially allow the logic prover to act as a forensic system identifying compromised hosts, and the logic system itself can be extended to formalize other network protocols that build on ARP.

To implement the TPM attestation protocol in *arpsec*, we introduce a TPM daemon (*tpmd*) and require each remote machine to install it. An extra TPM daemon is necessary based on the TPM specification and the TrouSerS API (the same reason why *arpsecd* is implemented in user space rather than a kernel module). TrouSerS provides a *tcsd* daemon process with RPC interfaces. These interfaces are for remote TPM management rather than TPM attestation from a challenger. Our *tpmd* communicates with the *tcsd* instead of calling the TPM directly as in `libtpm` [3]. Even if *tpmd* is compromised, the attacker would be unable to circumvent the TPM attestation protocol. This would require either a forged TPM Quote on known-good PCR values, or for the attacker to posses the AIK private key. An attacker would be unable to recover the AIK or edit the PCRs because they are stored on board the TPM.

By adding a host to the ARP cache and purging it if the attestation fails, we make a design trade-off. In the worst

[3]libtpm (http://ibmswtpm.sourceforge.net/) was developed before the TrouSerS API and the usage is deprecated.

| Protocol | Min | Avg | Max | Mdev |
|---|---|---|---|---|
| ARP | 36.16 | 90.36 | 330.7 | 34.79 |
| TARP | 5.17 | 31.01 | 69.48 | 10.83 |
| TARP* | 0.47 | 1.007 | 1.068 | 0.022 |
| *arpsec* | 3879 | 12841 | 46759 | 6062 |

Table 1: The key generation time ($ms$) with the key length 1024 bits averaged by 100 runs

| TPM | Min | Avg | Max | Mdev |
|---|---|---|---|---|
| AIKgen | 864 | 9385 | 43716 | 5932 |
| Rand | 10.92 | 11.40 | 11.47 | 0.035 |
| Quote | 324.5 | 336.1 | 336.5 | 0.698 |
| SigVerify | 0.120 | 0.199 | 0.213 | 0.006 |
| AttVerify | 0.208 | 0.307 | 0.344 | 0.009 |

Table 2: The TPM operation time ($ms$) averaged by 100 runs

| Protocol | Mechanism | Formal Proofs | Remote Integrity | Change to ARP? | Change to Kernel? | Overhead |
|---|---|---|---|---|---|---|
| S-ARP | PKI | N | N | Y | Y | Large |
| TARP | Ticket-based PKI | N | N | Y | N | Small |
| *arpsec* | Logic+TPM | Y | Y | N | Y/N | Small |

Table 3: General comparison among S-ARP, TARP and *arpsec*

case, we have made an incorrect binding for 300-500 $ms$ until a TPM quote fails and a binding is removed from the cache. This can be exacerbated by TCP transmission delays. Currently, we set the TCP socket timeout to be 2 $s$. To optimize for security of the binding, we can purge it immediately after the challenge is sent and wait for the attestation before we update the cache again. This creates considerable overhead, however. Alternately, the ARP request could carry the challenge and the ARP reply could encapsulate the AT reply, at the cost of creating a protocol change to ARP.

Another limitation of using TPM attestation is that it only attests to what was loaded into the system at boot time (or load-time using integrity measurement). Runtime integrity checking provides more guarantees at the cost of requiring extra processors or significant overhead. Integrity systems such as IMA [21] or PRIMA [6] could be integrated with *arpsec*. To further reduce the TCB size, we could potentially run *arpsec* in an isolated root of trust environment with a dynamic root of trust as offered by Flicker [11] or TrustVisor [7]. A trusted path [38, 37] to the NIC could ensure IP and MAC identifiers are correctly retrieved.

Though not designed to defend against DoS attacks, *arpsec* could handle certain attacks against ARP. 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 a higher-level network protocol, this would be out of the scope of *arpsec*. Such protection could be helpful against spanning tree attacks and VLAN hopping, however [24].

The performance of *arpsec* is limited by the TPM hardware. The TPM chip is designed to be cheap - only a 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. As shown in Table 2, TPM Quotes impose a 336 $ms$ delay when TPM attestation is required. As the TPM 2.0 library specification is published for review now, new TPM implementations based on it could further reduce the cost of Quote operations.

While TPMs have been widely deployed in servers, desktops, laptops and even mobile devices, many legacy machines lack them. For these machines, software TPMs could be used as a replacement, such as `libtpm` mentioned above, or vTPMs [19] in cloud environments. However, as the TPM Quote command occurs in software rather than hardware, a secure, trusted and isolated execution path [38, 37] is needed to guarantee trustworthiness.

Currently, *arpsec* supports both ARP and RARP. Future work will support Gratuitous ARP and IPv6.

# 7. RELATED WORK

The security of address binding operations in IPv4 contexts, particularly ARP, has received considerable attention, focused on the problems summarized in previous sections. Although the threats are in a LAN context, as they impact correct packet delivery to destinations, it is critical that countermeasures to them be successfully employed.

Alternatives range from early suggestions for static bindings [1], which at normal scale on any type of network with frequent host additions/removals is intractable; to ARP modifications in S-ARP [2] and TARP [10] which introduce signed attestations, in the form of addresses bound to a public key or a ticket. S-ARP participants self-generate key pairs and register the public key in the central Authoritative Key Distributor (AKD). The AKD maintains the public key/MAC bindings and distributes these to all S-ARP hosts. TARP relies on Kerberos-style tickets and a central ticket-granting service to provide authentication, and is hence faster due to the use of symmetric keys. These approaches all require modifications to ARP itself, which limits adoption.

Another solution implements a Cache Poisoning Checker [30] to intercept ARP requests and responses and inspect them for correctness. It does not modify the ARP protocol itself. A more wide-ranging approach [23] aims at preventing IP address spoofing by using the IP address as an identity based key. However, address-based assertion has limited use in environments where IP address assignment is dynamic, and does not address binding of IP to MAC identifiers.

Other solutions use security policies to prevent ARP attacks. ArpON [18] defines different ARP binding policies for different networks, including static, dynamic, or hybrid networks. Instead of a centralized management server, hosts within the network all run the ArpON daemon and respect the same policies, thus adding complexity to defining and updating policies for different network environments.

Few proposals have considered a more holistic perspec-

tive of overall protocol design. Wang et al. consider secure networking from proposal design to formal verification [34] but focus on BGP and meta-routing rather than address binding. There has been little emphasis on approaches tying together system attestations grounded in hardware with formally verifiable operation of Internet protocols. While the Trusted Computing Group defines the TPM standard for roots of trust and standardizes its use network connect (network access control), protocol operations supporting Internet infrastructure have not been considered [33, 25].

The use of IPV4, and thus ARP, is relevant for the foreseeable future, but IPv6 deployment is increasing. The IPv6 Neighbor Discovery Protocol (NDP) [13] has capabilities beyond ARP, relying on autoconfiguration based on address binding. The concepts used to secure ARP are applicable to NDP as well, as we discuss in our technical report [29].

# 8. CONCLUSION

This work has proposed *arpsec*, a secure ARP protocol that provides a logic prover to reason about the validity of ARP/RARP replies and uses TPM attestation to guarantee the trust in remote hosts. Compared to the original ARP, *arpsec* introduces only $7\% - 15.4\%$ system overhead. While providing a formally secure and trustworthy networking stack will will remain an issue into the future, *arpsec* points the way to a new solution in this space through use of a logic prover and TPM hardware and minimizing the system overhead without impacting current implementations.

## Acknowledgements

We wish to thank Stephen L. Squires and Jim Just for their helpful comments and discussion of the problem space. This work is funded in part by the US National Science Foundation under grant CNS-1254198. The views expressed in this paper are solely those of its authors and do not reflect the FCC position on the subject matter and related concepts.

# 9. REFERENCES

[1] S. M. Bellovin. Security problems in the ICP/IP protocol suite. *Comp. Comm. Review*, 2:32–48, April 1989.

[2] D. Bruschi, A. Ornaghi, and E. Rosti. S-ARP: a Secure Address Resolution Protocol. In *ACSAC*, 2003.

[3] D. Diaz et al. The GNU Prolog web site. `http://gprolog.org/`.

[4] R. Finlayson et al. A Reverse Address Resolution Protocol. `http://tools.ietf.org/rfc/rfc903.txt`, June 1984.

[5] B. Issac. Secure AP and Secure DHCP Protocols to Mitigate Security Attacks. *International Journal of Network Security*, 8:107–118, March 2009.

[6] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *ACM SACMAT*, 2006.

[7] J. M. McCune et al. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE S&P*, 2010.

[8] B. Kauer. OSLO: Improving the security of Trusted Computing. In *USENIX Security Symposium*, 2007.

[9] LBNL Network Research Group. arpwatch: the ethernet monitor program. `http://ee.lbl.gov/`, 2006.

[10] W. Lootah, W. Enck, and P. McDaniel. TARP: Ticket-based Address Resolution Protocol. *ACSAC*, 2005.

[11] J. M. McCune et al. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM EuroSys*, 2008.

[12] Microsoft Technet. Address Resolution Protocol. `http://technet.microsoft.com/en-us/library/cc940021.aspx`.

[13] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). `https://tools.ietf.org/html/rfc4861`, September 2007.

[14] J. Nathan. Nemesis. `http://nemesis.sourceforge.net/`.

[15] A. Ornaghi and M. Valleri. Man in the middle attacks Demos. `http://www.blackhat.com/presentations/bh-europe-03/bh-europe-03-valleri.pdf`, Blackhat 2003.

[16] A. P. Ortega et al. Preventing ARP Cache Poisoning Attacks: A Proof of Concept using OpenWrt. In *Net. Ops. & Mgmt. Symp.*, 2009.

[17] B. Parno. Bootstrapping trust in a "trusted" platform. In *USENIX HotSec*, 2008.

[18] A. D. Pasquale. ArpOn: ARP Handler Inspection. `http://arpon.sourceforge.net/index.html`, 2008.

[19] Perez, Ronald, Reiner Sailer, and Leendert van Doorn and others. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security Symposium*, 2006.

[20] D. C. Plummer. 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.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security*, 2004.

[22] J. Schmitz, J. Loew, J. Elwell, D. Ponomarev, and N. Abu-Ghazaleh. A Framework for Performance Evaluation of Trusted Platform Modules. In DAC, 2011.

[23] C. Schridde, M. Smith, and B. Freisleben. TureIP: Prevention of IP Spoofing Attacks Using Identity-Based Cryptography. In *SIN'09 Proc. 2nd Intl. Conf. on Security of information and networks*, pages 128–137, 2009.

[24] L. Senecal. Understanding and preventing attacks at layer 2 of the OSI reference model. In *Proc. 4th Comm. Networks & Services Research Conf*, 2006.

[25] S. Frankel, R. Graveman, J. Pearce, and Mark Rooks. Guidelines for the Secure Deployment of IPv6. `http://csrc.nist.gov/publications/nistpubs/800-119/sp800-119.pdf`, 2010. NIST.

[26] D. Song. dsniff. `http://monkey.org/~dugsong/dsniff/`, 2000.

[27] Symantec. Solaris Kernel Tuning for Security. `http://www.symantec.com/connect/articles/solaris-kernel-tuning-security`, Dec 20, 2000.

[28] C. Tarnovsky. Deconstructing a 'Secure' processor. *Black Hat DC*, 2010.

[29] J. Tian, K. Butler, P. McDaniel, and P. Krishnaswamy. Securing ARP From the Ground Up. Tech. Report REP-2015-573, Univ. of Florida, Jan. 2015.

[30] M. V. Tripunitara and P. Dutta. A middleware approach to asynchronous and backward compatible detection and prevention of ARP cache poisoning. In *ACSAC*, 1999.

[31] TrouSerS. The open-source TCG Software Stack. `http://trousers.sourceforge.net/`.

[32] Trusted Computing Group. TPM Main Specification. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[33] Trusted Computing Group. Glossary. `http://www.trustedcomputinggroup.org/developers/glossary`.

[34] A. Wang, L. Jia, C. Liu, B. T. Loo, et al. Formally verifiable networking. In *ACM HotNets*, 2009.

[35] S. Whalen. An Introduction to ARP Spoofing. `http://rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf`, 2001.

[36] T. Zanussi et al. relay (formerly relayfs). `http://relayfs.sourceforge.net/`.

[37] Z. Zhou, M. Yu, and V. Gligor. Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O. In *IEEE S&P*, 2014.

[38] Z. Zhou, V. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE S&P*, 2012.