A Formal Verification of ArpON – A Tool for Avoiding Man-in-the-Middle Attacks in Ethernet Networks

Danilo Bruschi, Andrea Di Pasquale, Silvio Ghilardi, Andrea Lanzi, and Elena Pagani

Abstract—Since the nineties, the Man-in-The-Middle (MITM) attack has been one of the most effective strategies adopted for compromising information security in network environments. In this article, we focus our attention on ARP cache poisoning, which is one of the most well-known and more adopted techniques for performing MITM attacks in Ethernet local area networks. More precisely, we will prove that, in network environments with at least one malicious host in the absence of cryptography, an ARP cache poisoning attack cannot be avoided. Subsequently, we advance ArpON, an efficient and effective solution to counteract ARP cache poisoning, and we use a model-checker for verifying its safety property. Our main finding, in accordance with the above impossibility result, is that the only event that compromises the safety of ArpON is a cache poisoning that nevertheless is removed by ArpON itself after a very short period, thus making it practically infeasible to perpetrate an ARP cache poisoning attack on network hosts where ArpON is installed.

Index Terms—Network protocols, ARP, man-in-the-middle attacks, secure ARP, formal verification

1 Introduction

Since the nineties, the Man-in-The-Middle (MITM) attack has been one of the most effective strategies adopted for compromising information security on Internet. MITM attacks target network traffic that flows between endpoints, compromising its confidentiality and integrity. It is a form of active wiretapping attack in which the attacker intercepts and selectively modifies communicated data to masquerade as one or more of the entities involved in a communication association [43].

MITM attacks exploit vulnerabilities at various levels of the OSI (Open System Interconnection) architecture: Man-in-the-browser attacks at application level [27], Secure Socket Layer (SSL) hijack at the transport level [19], IP spoofing at the network layer [33] and ARP Poisoning attacks at the data link layer [1]. Over the years these attacks have been adapted to work with new emerging network technologies such as GSM (Global System for Mobile communications) and UMTS (Universal Mobile Telecommunications System) [16], WiFi [5] and to IoT (Internet-of-Things) systems [18]. Today they still represent a major concern for security professionals. In this paper, we will focus our attention on ARP cache poisoning, which is one of the most well-known and commonly adopted techniques for performing MITM attacks.

ARP (Address Resolution Protocol) is a Data Link Layer [6] protocol whose main task is to establish a bind between the two addresses that characterize an Internet host, namely the IP (Internet Protocol) address and the MAC (Media Access Control) address. Since the Internet is a network of local area networks (LAN), the former is used to distinguish a host on the Internet,[1] while the latter is used for referring to a particular host inside a LAN. The MAC address, which is 48bits long, is fixed and hardwired in the network interface card by its manufacturer. By contrast, the IPv4 address, which is 32bits long, is assigned to a host by either a network administrator (static address), or a DHCP (Dynamic Host Configuration Protocol) server (dynamic address) [25] depending on the network the host is connected to. Thus, every host h connected to Internet is identified by a pair $\langle \text{MAC}_h, \text{IP}_h \rangle$, such that for every two hosts both their IP addresses and their MAC addresses are distinct. Furthermore, h maintains a dynamic table which contains the pair $\langle \text{MAC}, \text{IP} \rangle$ for any host it needs to communicate with. The construction and management of such a table, which is called ARP cache, is the main task of the ARP protocol. By poisoning the content of such a table an attacker can modify the communication flow inside a LAN.

Several solutions to the ARP cache poisoning attack problem have been proposed in the literature, that though require significant changes to the nodes, cannot coexist with

- *Danilo Bruschi, Silvio Ghilardi, Andrea Lanzi, and Elena Pagani are with the Università degli Studi di Milano, 20122 Milano, Italy. E-mail: {danilo.bruschi, silvio.ghilardi, andrea.lanzi, elena.pagani}@unimi.it.*
- *Andrea Di Pasquale is with the Infovista S.A., 41126 Milano, Italy. E-mail: spikey.it@gmail.com.*

Manuscript received 31 July 2020; revised 23 July 2021; accepted 26 September 2021. Date of publication 7 October 2021; date of current version 11 November 2022.
The work was conducted in the framework of SEED Project "Situational awareness in critical Infrastructure Environment (SENTINEL)" funded by the Università degli Studi di Milano.
(Corresponding author: Elena Pagani.)
Digital Object Identifier no. 10.1109/TDSC.2021.3118448

1. IPv4 is the version of the implementation of the IP protocol currently used. It will be substituted in a future by IPv6. Addresses will become 128 bits long in IPv6.

standard ARP implementations, and in some circumstances fail in preventing/solving poisoning. In Section 3, we perform a detailed analysis.

In this paper, the authors approach the problem from a formal perspective in order to provide a more definitive response to it.

More precisely, we formally define the problem of constructing and managing an ARP cache, namely the Address Translation Problem, and we provide a formal proof that in the presence of a malicious host, the Address Translation Problem is impossible to solve. Given such a result, we devised the ArpON protocol, a solution to the ARP cache poisoning problem that is based on the strategy of mitigating the effects of an ARP cache poisoning attack by returning a poisoned ARP cache to a "non-dangerous" state in the shortest span time possibile. Using a formal prover [30] we also verified the safety property of ArpON.[2] This is a very significant result in that it provides a further step toward the application of formal verification techniques for the analysis as well as synthesis of real network protocols. Recent research regarding infinite-state systems has provided very few working examples of specifications involving quantifiers, such as those required for modeling ArpON, since the border leading to undecidability phenomena is quite close, and very few methodologies are available.

We have formally proved that in static environments, i.e., LANs where the hosts have predefined IP addresses, ArpON is safe from ARP poisoning attacks. On the other hand, in dynamic environments in which the hosts' IP addresses are not known a priori and can change during protocol execution, in accordance with the impossibility result mentioned above, the only event which compromises the safety of the protocol is a cache poisoning which however is removed quickly – as we are able to prove – making practically infeasible to perpetrate such attacks.

Obviously, in order to properly work, ArpON requires some additional message exchanges, which however affect neither the approach scalability (the communication overhead is independent of the LAN size), nor message latency, as we show through simulations in Section 5.

ArpON is thus an efficient and effective solution to counteract ARP cache poisoning attacks in that it incurs in low operational costs, is backward compatible, transparent to the ARP protocol, and to our knowledge is the only protocol at the data link layer which exhibits a formal safety proof. ArpON is completely compliant with every version of the ARP protocol as specified in the relevant Request for Comments (RFCs) [20], [21], [42] and its source code has been made available and has been downloaded by more than 100,000 users since January 2016.[3]

The main contributions of this paper with respect to the state of the art can be summarized as follows:

- We provide the first formal definition of the Address Translation Problem addressed by the ARP protocol, and we formally prove its impossibility in the presence of a single malicious host in the more general network model, namely, the adoption of dynamic network addresses with no use of cryptography.
- We formalize the ArpON protocol using the array-based declarative approach for the modeling of infinite-state reactive parameterized systems.
- Using the *Model Checker Modulo Theories* (MCMT) model-checker, we prove the safety property of ArpON, in the particular case in which LAN hosts have a static, persistent, addressing.
- Using the *Model Checker Modulo Theories* (MCMT) model-checker, we prove that in the more general context the safety property of ArpON does not hold but cache poisoning is always removed after a very short period of time (ArpON *shaded area*).
- We evaluate - through simulations - the length of the ArpON shaded area. We show that it lasts 0.11 ms, an interval much shorter than that required by a packet to traverse the TCP/IP (Transmission Control Protocol/Internet Protocol) stack on an optimized platform, which is the first step required for performing a MITM attack. More precisely, such a value has been estimated in 0.53ms [31], which prevents the possibility of bringing a successful MITM attack when ArpON is in use, comparing it against real system measurements.

The paper is organized as follows. Section 2 provides background on the ARP protocol and on MITM attacks. Section 3 provides a brief overview of related research. Section 4 contains a general description of ArpON. Section 5 reports some experimental values we obtained by executing ArpON in a simulation environment. Section 6 formally defines the Address Translation Problem and proves the impossibility of deterministically solving it in the presence of a malicious host. Section 7 introduces some preliminaries on the formal prover that has been adopted, and reports the main results obtained by the formal verification of ArpON. Section 8 contains the concluding remarks.

2 Background

In this section we introduce the basic concepts for understanding the ARP protocol and the ARP cache poisoning attack.

2.1 Address Resolution Protocol

As mentioned earlier, the main task of ARP [20], [21], [42] is to learn hosts' MAC addresses corresponding to IPv4 addresses and write them into the ARP cache. Within an ARP cache, we distinguish between persistent entries and dynamic entries. Persistent entries contain mappings that are known a priori, are manually configured by the system administrator, and permanently remain inside the cache, unless explicitly removed. Dynamic entries are related to mappings which are not known beforehand and need to be learnt at runtime. A dynamic entry usually has a lifetime of approximately 10 minutes; after that period the entry is automatically removed from the ARP cache.

In simple terms, dynamic entries are built by ARP in the following way. Consider two hosts h and k that are on the

2. Where by safety property we mean that no "bad things" happen during any execution of the solution protocol [8].

3. https://arpon.sourceforge.io/

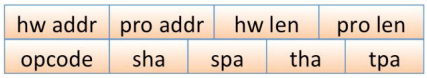

Fig. 1. Payload of the ARP messages.

same LAN; h needs to send a packet to k. h usually knows IP_k i.e. k's IP address,[4] but in order to reach k in its local network h has to know k's MAC address. First, h looks for a $\langle \text{MAC}_k, \text{IP}_k \rangle$ entry in its own ARP cache. If the entry is not found, h sends an *ARP request* message to all hosts in the LAN, asking for the MAC address of the owner of the IP_k address. Once k receives the message, it sends an *ARP reply* message to h supplying its own MAC address. Once h receives the ARP reply, it updates its own cache with the entry $\langle \text{MAC}_k, \text{IP}_k \rangle$. Symmetrically, k updates its own cache with the mapping $\langle \text{MAC}_h, \text{IP}_h \rangle$.

In Fig. 1, the payload of ARP messages is shown: the fields in the first line contain the type and length of both the IPv4 and MAC addresses. In the second line, the `opcode` specifies the type of message, the `sha` and `spa` fields are respectively the MAC and IPv4 addresses of the message source, while the `tha` and `tpa` fields are the MAC and IPv4 addresses of the message target.

ARP also provides the opportunity for a host to announce its IPv4 and MAC addresses, either at boot or upon changes. This is useful for example when a host joins a LAN. Such an announcement, also called a *gratuitous ARP* message, is usually broadcast as either an ARP request or an ARP reply. Gratuitous announces have both `sha` $=$ `tha` and `spa` $=$ `tpa` to report the address correspondence to be announced. A gratuitous ARP sent using an ARP request is not intended to solicit a reply; rather, it updates possible cached entries for the sending host in the ARP tables of the receivers of the packet. Such an update is performed because of the *implicit ARP assumption that all the hosts in a LAN are trustable*.

Furthermore, before beginning to use an IPv4 address (whether received from manual configuration, DHCP, or some other means), a host h must usually verify whether the address is already in use: to this end, it broadcasts an *ARP probe* message, i.e., a "fake" ARP request where the source mapping is empty (`spa`$= 0.0.0.0$) in order to not leave traces in other hosts' caches, while the target IP address is the one the host would use. If another host k exists in the LAN already using the IP address, k sends a unicast *ARP reply* message, signaling that the address is already in use.

2.2 MITM Attacks

The ARP protocol can easily be subverted by performing *Man-in-the-Middle* (MITM) attacks (see e.g., [14]), using a technique known as ARP poisoning or ARP spoofing, described in the following.

Let us consider three hosts in a LAN x, w, z and their corresponding MAC and IP addresses $\langle \text{MAC}_x, \text{IP}_x \rangle$, $\langle \text{MAC}_w, \text{IP}_w \rangle$, $\langle \text{MAC}_z, \text{IP}_z \rangle$. If w convinces z that x's MAC address is MAC_w, all messages that z wants to send to x will actually be addressed to MAC_w; consequently, they will be received by w. In this way, the attacker w will hijack all the communications between z and x, acting as it sits in the middle of the communication, hence the name. More precisely, this is the case of a half-duplex MITM as the attacker is able to intercept only one traffic flow (from z to x). We refer to full-duplex MITM when the attacker is able to monitor both the traffic flows; this would imply that w is also able to convince x that z's MAC address is MAC_w.

The goal of MITM attacks is to overtake a communication session between two hosts in order to intercept and view the information being exchanged between them. ARP poisoning is not difficult to obtain by leveraging some of the features of the ARP protocol. Some methods adopted to perform a MITM attack, which are mostly based on the fact that ARP assumes that all hosts in the network are trustable, are:

- a host h can craft a gratuitous ARP message where the pair $\langle$`sha`, `spa`$\rangle$ is set to $\langle \text{MAC}_h, \text{IP}_x \rangle$; in this way, roughly speaking, the ARP caches of the remaining hosts in the LAN are poisoned with a wrong value, and h will intercept all messages directed to x by all the hosts in the LAN;
- when receiving an ARP request/reply, a host immediately updates its own ARP cache with the information contained in the message. Again, a suitably crafted ARP reply can be used for ARP poisoning *also in case no previous ARP request was generated* (*unsolicited*), as ARP is stateless and hosts do not remember the messages they have sent.

Since the ARP cache entries are periodically refreshed, an attacker who is interested in maintaining a MITM attack for a long time has to continuously send ARP messages suitably crafted so that the ARP cache entries of interest stay poisoned.

3 Related Research

In this section, we briefly describe several defense mechanisms against ARP poisoning attacks, which have been proposed in the literature.

ArpWatch [36] is a user-space tool for monitoring ARP traffic on computer networks. It keeps track of MAC/IP address pairings. It generates Syslog activities and reports via e-mail certain changes of the observed pairings of IP addresses with MAC adresses, along with a timestamp when the pairing appeared on the network.

Anticap [10] is a kernel patch that does not update the ARP cache when an ARP reply carries a different MAC address for a given IP already in the cache and issues a kernel alert. In this case, ARP specification is no longer adhered to, as legal gratuitous packets are dropped. Antidote [44] is a different kernel patch that intercepts ARP replies announcing a change in a $\langle$MAC, IP$\rangle$ pair and tries to discover if the previous MAC address is still viable. In that case, the update is rejected and the new MAC address is added to a list of "banned" addresses. If Antidote is installed, a host can spoof the sender MAC address and force a host to ban another host. In [46], a solution that implements two distinct queues, one for requested addresses and one for received replies, is proposed. The system discards a reply if either the corresponding request was never sent, i.e., is not in the queue, or an IP address associated with a different Ethernet address is already present in the *received* queue.

4. E.g., through the resolution of k's symbolic name by the Domain Name System.

In all the above cases, the solutions contain the same vulnerability. That is, when an ARP request is broadcast and both the victim and the attacker receive the message, the first to reply will take over the other.

S-ARP [15] and TARP [37] use asymmetric cryptography and Authoritative Key Distributor to assert the authenticity of ARP messages. TARP [37] introduces a signed attestation in the form of addresses to a public key or ticket. Messages are digitally signed by the sender, thus preventing the injection of spoofed information. Unfortunately, cryptography and key management at the Data Link layer are not compatible with most existing LANs protocols and devices, and would require extensive changes. Furthermore, they have a significant impact on performance, and are not always affordable as in the case of Industrial LANs. Furthermore, the S-ARP solution is not compatible with the legacy code since the S-ARP packet format is different from the ARP-Packet format as defined in [42]. In [45] a further protocol, namely Arpsec, has been introduced, which uses TPM (Trusted Platform Module) attestation to guarantee the trust in remote hosts. Arpsec, however, requires TPM hardware on each host to work as well as a key management support.

In [38], the MR-ARP protocol is introduced which prevents ARP poisoning MITM attacks recurring to voting schema. When a host receives an ARP request/reply message that contains a MAC address for an IP address different from the one registered in the ARP cache, MR-ARP requests the neighbouring nodes to vote for the new IP address. This schema is based on the assumption that votes can be delivered almost instantaneously, but this condition may not be valid in some LAN environments such as wireless networks, where data rates can change on the basis of signal-to-noise ratio (SNR), i.e., auto rate fallback (ARF).

4 ARPON: ARP HANDLER INSPECTION

In this section, we describe the ArpON protocol by providing architectural as well as implementation details. Architecturally speaking, ArpON is divided into three modules, namely, *SARPI* for LANs where only static persistent addresses are used, *DARPI* for LANs where just dynamic addresses are used, and *HARPI* that merges SARPI and DARPI when both persistent and dynamic addresses are in use. These modules are described and a pseudo code is provided in the following sections.

4.1 Overview

ArpON is a daemon that works in parallel with the ARP protocol and is compatible with legacy implementations of ARP. As a consequence, in a LAN, hosts that use the traditional ARP can coexist with hosts that have installed the "ARP + ArpON" solution; the latter hosts having their ARP caches protected from MITM ARP poisoning attacks. The main task of ArpON is to supervise ARP cache management, relying on its own cache, which is different from that used by ARP.

More precisely, any ArpON instance manages two different caches: a SARPI cache and a DARPI cache. ArpON works in user space and cooperates with the ARP protocol in the kernel to manage the Ethernet interfaces present in a host; an instance of the ArpON daemon exists for each Ethernet interface. The general architecture is described in Fig. 2.

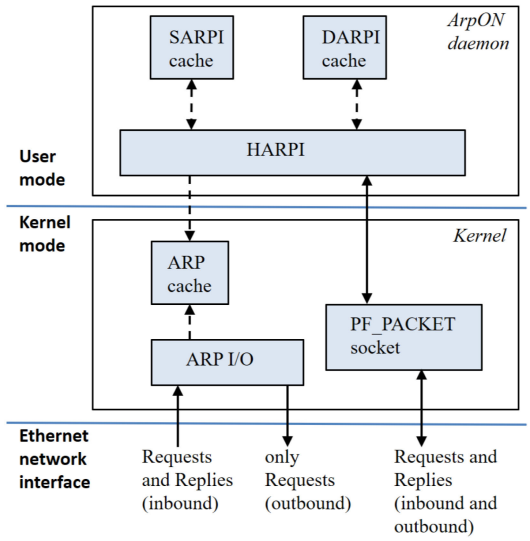

Fig. 2. ArpON general architecture.

Most of the management of the Ethernet interface still remains a kernel responsibility. Upon the reception of ARP messages, ArpON – on the basis of its policies – overwrites the ARP cache and decides whether to create, maintain or delete cache entries.[5]

The behavior of the SARPI and DARPI modules is driven by a set of policies that define – on the basis of the network packet received on the network interface – the operations to be performed on the ARP cache and the SARPI/DARPI caches, as described in the following. In the following, the term *basic* request denotes an ARP request unleashed by a process at the application level, in contrast with those generated autonomously by ARP such as probe and gratuitous requests. Similarly, *basic* replies differentiate from gratuitous replies.

4.2 SARPI: Static ARP Inspection

As mentioned above, SARPI works in a LAN environment with IP addressing completely static and persistent. We assume that every host has a configuration file that contains the trustable mappings of all the hosts in the network. The task of SARPI is to avoid the occurrence into the ARP cache of any persistent mapping different from those in the configuration file.

Algorithm 1 supplies the pseudo-code for SARPI. All messages generated by SARPI are broadcast.[6] At start up, SARPI executes the *Clean* policy (lines 18, 1-5), which consists of removing all the entries, both static and dynamic, from the ARP cache, and copying all the trustable mappings from the configuration file to the SARPI cache. Subsequently, all the mappings contained in the SARPI cache are copied as persistent entries in the ARP cache following the *Update* policy (lines 19, 7-9).

5. Such a functionality is obtained by ArpON by modifying in the `proc filesystem` the `arp_ignore` and the `arp_accept` parameters which are used by the sysadmin to configure the way ARP behaves when ARP requests and ARP gratuitous announces are received.

6. ff:ff:ff:ff:ff:ff by convention is the destination MAC address used for broadcast messages.

Algorithm 1. SARPI

```
1: Clean()
2:  Disable in the Operating System the generation of ARP replies in response to received ARP requests for all local addresses;
3:  Disable in the Operating System the creation of new IP entries in the ARP cache triggered by unsolicited and gratuitous ARP requests and replies;
4: ARP_cache ← {}; // empty
5: SARPI_cache ← config-file;
6:
7: Update()
8: ARP_cache ← SARPI_cache ∪ (ARP_cache - SARPI_cache);
9: start timer(10 min.):
10:
11: Allow(S)
12: ARP_cache(S) ← ⟨spa, sha⟩; // dynamic
13:
14: Refresh(S)
15: ARP_cache(S) ← SARPI_cache(S); // persistent
16:
17: Main()
18: Clean();
19: Update();
20: when timeout ∨ ARP packet do
21:   if timeout then
22:     Update();
23:   else if (ARP_pkt.opcode=Request ∨ ARP_pkt.opcode=Reply) ∧ ARP_pkt.sha=myMAC ∧ ARP_pkt.spa=myIP then
24:       skip; // outbound basic/probe/gratuitous ARP requests and replies
25:   else if ((ARP_pkt.opcode=Reply ∧ ARP_pkt.tpa=myIP) ∨ ARP_pkt.spa=ARP_pkt.tpa) ∧ ARP_pkt.spa= S ∧ S /= 0.0.0.0 ∧ S ≠ myIP ∧ S ∈ NETWORK(myIP) then
26:       if S ∈ SARPI_cache then
27:           Refresh(S);
28:       else
29:           Allow(S);
30:       end if
31:   else if ARP_pkt.opcode=Request ∧ ARP_pkt.tpa=myIP ∧ ARP_pkt.spa= S ∧ S ≠ 0.0.0.0 ∧ S ≠ myIP ∧ S ∈ NETWORK(myIP) then
32:       new ARP_pkt': ARP_pkt'.opcode ← Reply;
33:       ARP_pkt'.tpa ← S; ARP_pkt'.tha ← ff:ff:ff:ff:ff:ff;
34:       ARP_pkt'.spa ← myIP; ARP_pkt'.sha ← myMAC;
35:       broadcast ARP_pkt'; // eth_dest = ff:ff:ff:ff:ff:ff
36:       if S ∈ SARPI_cache then
37:           Refresh(S);
38:       else
39:           Allow(S);
40:       end if
41:   else if ARP_pkt.opcode=Request ∧ ARP_pkt.tpa=myIP ∧ ARP_pkt.spa= 0.0.0.0 then
42:       new ARP_pkt': ARP_pkt'.opcode ← Reply;
43:       ARP_pkt'.tpa ← 0.0.0.0; ARP_pkt'.tha ← ff:ff:ff:ff:ff:ff;
44:       ARP_pkt'.spa ← myIP; ARP_pkt'.sha ← myMAC;
45:       broadcast ARP_pkt'; // eth_dest = ff:ff:ff:ff:ff:ff
46:   else
47:       skip; // inbound ARP probe reply
48:   end if
49: end do
```

However, persistent entries may be removed or modified by the system administrator. Hence, every 10 minutes, ArpON executes the Update procedure to refresh the persistent entries in the ARP cache with the SARPI cache (lines 21-22). If, for some reason, a permanent entry in the ARP cache of a host h is removed, its value will remain undefined until the next Update is performed. In the meantime, h can be exposed to an ARP cache poisoning attack. In order to avoid this risk, a *Refresh* policy (lines 14-15) has been introduced that works as follows. When a host h receives either a basic ARP reply or a gratuitous announce (lines 25-30), SARPI verifies whether the spa field is present in the SARPI cache. In this case, the corresponding (safe) persistent mapping is immediately copied into the ARP cache by the Refresh policy. If by contrast the spa value is not present in the SARPI cache, then this is a dynamic mapping, which is not an issue for SARPI, and the *Allow* policy (lines 11-12) is applied.

When a host h receives a basic ARP request from a host u (lines 31-40), h sends (broadcast) an ARP reply message as usual, providing its own MAC address to u. At the same time, h updates its own ARP cache with either:

1) the mapping related to u contained in the ARP request, if IP_u is not contained in its SARPI cache (Allow policy);
2) or the corresponding mapping contained in the SARPI cache (Refresh policy).

The reception of an ARP probe message (lines 41-48) raises the generation of an ARP reply only in the case the receiving host owns the probed IP address, in order to avoid duplicate addresses.

Algorithm 2. DARPI - Policies

```
1: Clean()
2:  Disable in the Operating System the generation of ARP replies in response to received ARP requests for all local addresses;
3:  Disable in the Operating System the creation of new IP entries in the ARP cache triggered by unsolicited and gratuitous ARP requests and replies;
4: ARP_cache ← {}; // empty
5: DARPI_cache ← {};
6:
7: Allow(S)
8: ARP_cache(S) ← ⟨spa, sha⟩;
9:
10: Deny(S)
11: ARP_cache(S) ← ⟨⊥, ⊥⟩;
12:
13: Verify(S)
14: new ARP_pkt: ARP_pkt.opcode ← Request;
15: ARP_pkt.spa ← myIP; ARP_pkt.sha ← myMAC;
16: ARP_pkt.tpa ← S ; ARP_pkt.tha ← ff:ff:ff:ff:ff:ff;
17: broadcast ARP_pkt; // eth_dest = ff:ff:ff:ff:ff:ff
```

Any time the configuration file is modified, the updated version is transferred into both the SARPI and the ARP caches.

4.3 DARPI: Dynamic ARP Inspection

In the case of non-persistent addresses, ARP cache poisoning attacks are prevented by the DARPI module. DARPI

Algorithm 3. DARPI - Main Code

```
1: Main()
2: Clean();
3: start timer(1 s.);
4: when timeout ∨ ARP packet do
5:   if timeout then
6:     DARPI_cache ← DARPI_cache - DARPI_expired;
7:     start timer(1 s.);
8:   else if ARP_pkt.opcode=Request ∧ ARP_pkt.sha= myMAC then
9:     if ARP_pkt.spa= 0.0.0.0 ∨ ARP_pkt.spa=ARP_pkt.tpa then
10:      skip; // skip outbound probe/gratuitous ARP requests
11:    else if ARP_pkt.spa=myIP ∧ ARP_pkt.tpa∈ NETWORK(MyIP) then
12:      DARPI_cache ← DARPI_cache + ⟨ARP_pkt.tpa, time⟩;
13:    end if
14:  else if ARP_pkt.opcode=Request ∧ ARP_pkt.spa= S ∧ ARP_pkt.spa=ARP_pkt.tpa ∧ S ≠ 0.0.0.0 ∧ S ≠ myIP ∧ S ∈
       NETWORK(myIP) then
15:    Verify(S);
16:    Deny(S);
17:  else if ARP_pkt.opcode=Request ∧ ARP_pkt.spa= S ∧ ARP_pkt.tpa= myIP ∧ S ≠ 0.0.0.0 ∧ S ≠ myIP ∧ S ∈ NETWORK
       (myIP) then
18:    if ¬S ∈ DARPI_cache then
19:      Verify(S);
20:    end if
21:    new ARP_pkt': ARP_pkt'.opcode ← Reply;
22:    ARP_pkt'.tpa ← S, ARP_pkt'.tha ← ff:ff:ff:ff:ff:ff;
23:    ARP_pkt'.spa ← myIP; ARP_pkt'.sha ← myMAC;
24:    broadcast ARP_pkt; // eth_dest=ff:ff:ff:ff:ff:ff
25:    Deny(S);
26:  else if ARP_pkt.opcode=Request ∧ ARP_pkt.spa= 0.0.0.0 ∧ ARP_pkt.tpa=myIP then
27:    new ARP_pkt': ARP_pkt'.opcode ← Reply;
28:    ARP_pkt'.spa ← myIP; ARP_pkt'.sha ← myMAC;
29:    ARP_pkt'.tpa ← 0.0.0.0 ; ARP_pkt'.tha ← ff:ff:ff:ff:ff:ff;
30:    broadcast ARP_pkt; // eth_dest=ff:ff:ff:ff:ff:ff
31:  else if ARP_pkt.opcode=Reply ∧ ARP_pkt.sha= myMAC ∧ ARP_pkt.spa=myIP then
32:    skip; // skip outbound basic/probe/gratuitous ARP replies
33:  else if ARP_pkt.opcode=Reply ∧ ARP_pkt.spa=ARP_pkt.tpa ∧ ARP_pkt.spa= S ∧ S ≠ 0.0.0.0 ∧ S ≠ myIP ∧ S ∈ NET-
       WORK(MyIP) then
34:    Verify(S);
35:    Deny(S);
36:  else if ARP_pkt.opcode=Reply ∧ ARP_pkt.spa= S ∧ ARP_pkt.tpa= myIP∧ S ≠ 0.0.0.0 ∧ S ≠ myIP ∧ S ∈ NETWORK
       (MyIP) then
37:    if S ∈ DARPI_cache then
38:      DARPI_cache ← DARPI_cache − oldest{DARPI_cache(S)};
39:      Allow(S);
40:    else
41:      Verify(S);
42:      Deny(S);
43:    end if
44:  else
45:    skip; // skip inbound probe ARP reply
46:  end if
47: end do
```

adds a notion of state to the standard ARP protocol, which enables DARPI to detect the sources of ARP cache poisoning attacks, i.e., ARP requests, unsolicited replies, or gratuitous messages. Such notion of state is implemented by keeping track of the outbound messages generated by the host, in an internal cache of the DARPI module. Every inbound message received by the host that does not match any stored message is classified as unsolicited.

We describe below the general behavior of the DARPI protocol, referring to the details in Algorithms 2 and 3 that respectively describe the DARPI policies and the DARPI main code.

As a preliminary, we point out that in DARPI all messages are broadcast in the LAN. At start up, DARPI executes the *Clean* policy (lines 2 Algorithm 3, 1-5 Algorithm 2), which consists of removing all the entries, both static and dynamic, from the ARP cache, and all the entries in the DARPI cache.

When a host x receives an ARP request from ⟨ sha , spa ⟩ , it performs the following actions:

- **if** DARPI cache contains an entry with the IPv4 address equal to spa (lines 17-25 Algorithm 3)
- **then** x
  - sends an ARP reply to spa
  - removes from the ARP cache the mapping ⟨sha, spa⟩ inserted by the ARP protocol basing on the ARP request (*Deny* policy, lines 10-11 Algorithm 2). The DARPI cache entry is maintained;
- **then** x
  - sends an ARP request to spa (*Verify* policy, lines 13-17 Algorithm 2) followed by an ARP reply to spa,
  - removes from the ARP cache the mapping ⟨sha, spa⟩ inserted by ARP (Deny policy),
  - writes in the DARPI cache an entry ⟨spa, T_D⟩, where T_D is a timestamp allowing a 1 s. lifetime to the entry (lines 11-13 Algorithm 3). If within 1 s. the corresponding ARP reply message is not received, the entry is deleted from the DARPI cache (lines 5-7 Algorithm 3).

When a host x receives an ARP reply from ⟨ sha , spa ⟩ , it performs the following actions (lines 36-43 Algorithm 3):

- **if** DARPI cache contains an entry with the IPv4 address equal to spa
- **then** x removes the DARPI cache entry and creates the dynamic ARP cache entry ⟨sha, spa⟩ (*Allow* policy, lines 7-8 Algorithm 2);
- **else** x
  - sends an ARP request to spa (Verify policy),
  - removes the mapping ⟨sha, spa⟩ from the ARP cache (Deny policy),
  - creates the entry ⟨spa, T_D⟩ in the DARPI cache.

The reception of a gratuitous announce – be it either a request (lines 14-16 Algorithm 3) or a reply (lines 33-35 Algorithm 3) – is treated as the reception of a reply from a source for which no DARPI cache entry exists. The reception of an ARP probe is managed as in SARPI (lines 26-30 Algorithm 3)

These procedures aim at preventing ARP cache poisoning of hosts. Consider a host k which receives an ARP message (request, gratuitous, or unsolicited reply) from IP_h: in order to verify the information contained in the message, k sends an ARP request to h. At the same time, DARPI removes the entry related to IP_h from k's ARP cache and adds the ⟨IP_h, T_D⟩ entry in k's DARPI cache. In response to such messages, h sends an ARP reply to k. When k receives the ARP reply from h, k removes the ⟨IP_h, T_D⟩ entry from its DARPI cache, and inserts in its ARP cache the ⟨MAC_h, IP_h⟩ addresses contained in h's ARP reply. However, as also outlined by the model-checker we adopted, the above protocol has a transient flaw which occur when a malicious host m anticipates h's ARP reply, with a packet containing the ⟨MAC_m, IP_h⟩ pair. In such a case these data will be inserted in k's ARP cache thus poisoning it. Fortunately, the poisoning will last only until k receives the correct ARP reply by host h, which at this point is an unsolicited reply. When this occurs, DARPI removes the ⟨MAC_m, IP_h⟩ entry from the ARP cache and activates the Verify policy. We underline that, in a LAN environment, this procedure takes fractions of milliseconds, a period in which it is not possible to perpetrate any attack (see Section 5).

4.4 HARPI: Hybrid ARP Inspection

Most of the current LANs connect both hosts with persistent addresses and hosts with non-persistent addresses. In regard to these types of LANs, we merge the SARPI and DARPI modules into the HARPI module. Specifically, when an IPv4 address in the spa field of an ARP message is retrieved in the SARPI cache, HARPI behaves according to the SARPI policies; otherwise, HARPI adopts the DARPI policies.

5 Performance Evaluation

In this section, we present some preliminary performance measurements obtained by implementing ArpON in the OMNET++ network simulator version 5.1.1, in combination with the INET package version 3.6.4.

In all simulations, hosts use the standard UDP (User Datagram Protocol) and IP protocols. Every host may be both source and destination of UDP messages. As source, a host generates messages according to an exponential distribution with a parameter equal to 15 s., for a random destination chosen according to a uniform distribution. Among the hosts, one behaves as an attacker. It may either perpetrate a full MITM attack, or send poisoned Gratuitous Announces. Both victims (i.e., hosts that the attacker wants to impersonate) and targets (i.e., hosts whose ARP caches the attacker tries to poison) are chosen randomly. Every attack lasts around 6-7 minutes. When an attack ends, the attacker schedules the time to start a new attack according to an exponential distribution with variable parameter τ. All simulations reproduce 3600 s. of simulated time.

a) ArpON effectiveness: We evaluated the effectiveness of DARPI in comparison with classical ARP [42]. In a first set of measures, the network consists of n, $10 \leq n \leq 150$, hosts connected to a unique switch in a star topology via FastEthernet (100 Mbps) links. Every attempted attack succeeds in ARP, while *no attack is observed when hosts are configured with ArpON*. Fig. 3a shows the cumulative distribution of cache poisoning length for ARP, with 75 hosts in the LAN and $\tau = 300$ s, for both types of attacks. Approximately 20-30% of the attacks last more than 2 minutes, and around 35-40% of them last more than one minute. In the case of Gratuitous Announces, we observed that more than 50% of the hosts in the LAN suffered ARP cache poisoning within 10 minutes.

b) ArpON efficiency: The ArpON verification procedure increases the communication overhead. We compared the number of messages generated by both ARP and ArpON with variable number of hosts in the LAN. Fig. 3b shows both the number of generated messages (bars) and the ratio between the ArpON traffic and the ARP traffic (line), with

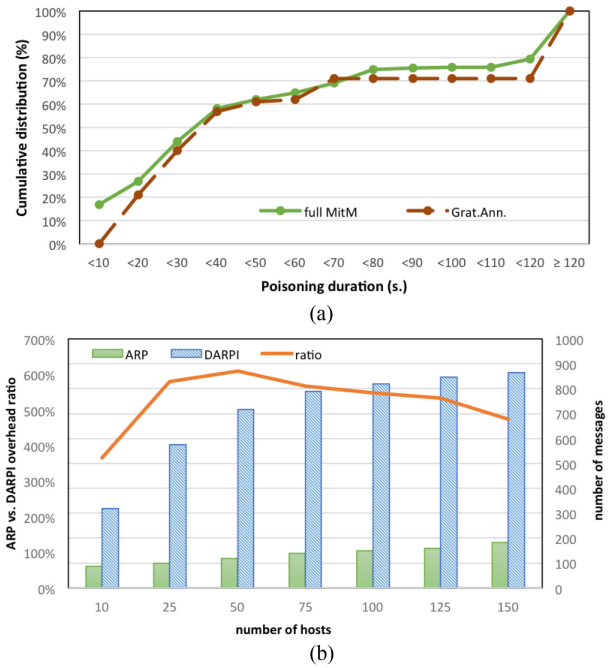

Fig. 3. (a) ARP performance in terms of poisoning duration. (b) Communication overhead for both ARP and DARPI, with variable LAN size.

$\tau = 300$ s. Although the increase in communication overhead is evident, it should be noted that it is roughly constant independently of the LAN size, varying between 3.5 and 6 times the ARP traffic with an average of 5.26. Despite both the higher amount of traffic generated by ArpON with respect to ARP and the need of verifying address correspondences, the latencies are lightly affected. Fig. 4 shows the end-to-end message latency measured on hosts, for variable number of hosts in the LAN, $\tau = 300$s and attacks perpetrated via Gratuitous Announces. The latency of ArpON is about 20 μs greater than that of ARP with the largest LAN considered. Furthermore, these results confirm the ArpON scalability in terms of latency: when increasing the number of hosts in the network from 10 to 150 (1500%), the latency increases only 80%.

All these measures show that – although ArpON imposes a degree of overhead in the network – it (i) effectively protects hosts from poisoning most of the time, (ii) scales well for both increasing LAN size and increasing frequency of attacks.

6 Address Translation Problem

6.1 Preliminary Definitions

In this section, we formally define the Address Translation Problem (ATP), the main problem underlying the ARP protocol, and we prove the impossibility of solving it in the more general context represented by networks with dynamic addressing, no cryptography, and at least one malicious host in the network. A trivial corollary of such a result is that there are no safe protocols for the above mentioned problem, where by safe we mean that no "bad things" occur during any execution of the protocol [8]. This also means that only approximate solutions to ATP can be found. One of such solutions is ArpON, which approximates a solution to the ATP problem by tolerating a

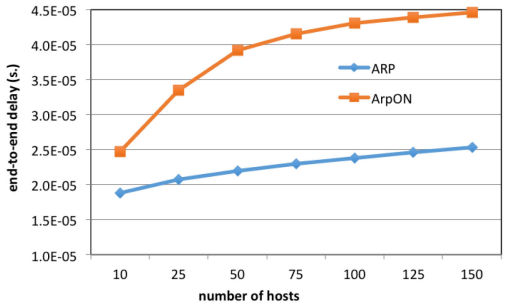

Fig. 4. End-to-end message latency on hosts, for variable number of hosts in the LAN, $\tau = 300$s and attacks perpetrated via Gratuitous Announces.

transient violation of the safety property, a violation which however lasts for periods of time too short to bring a successful MITM attack, as will be shown in Section 7.

In the following sections, we use equivalent terms *host* and *process*, denoting the process running the ARP protocol on a host. Hosts are connected by a broadcast medium and communicate either via broadcast or point-to-point primitives. The communication channel is reliable and synchronous, that is, an a-priori time bound can be established upon message delivery.[7]

Intuitively, each host h has a private value $\langle IP_h, MAC_h \rangle$ such that $(\forall h, k,\ h \neq k)$ $\text{MAC}_h \neq \text{MAC}_k \wedge \text{IP}_h \neq \text{IP}_k$. In the case of dynamic addressing (i) for every two hosts h, k, there is no way for k to know beforehand the IP associated to h, and conversely (ii) it is not possible for k to know a priori – given an IP address – which is the host (namely, MAC address) that owns it.

Each process is assumed to maintain a local vector $cv_i = (v_{i1}, \ldots, v_{in})$ where v_{ij} is process i's estimate of process j's private value, which will also be denoted by cv_{ij}; the MAC component of cv_{ij} may be the undefined value $\perp$.

In our system, processes evolve in steps. At every step a process i sends messages to other processes in the system and it may receive messages from them. Upon receiving messages, a host i can update its local vector cv_i using a deterministic function of its old local vector and the received messages. However, processes are not required to work in lockstep.

We have adopted the formalism in [41] in order to formalize our system processes.

A *virtual background* is a triple

$$\mathcal{S} = (P, \mathbf{M}, \mathbf{I}),$$

where $P, \mathbf{M}, \mathbf{I}$ are three finite sets; the set P is the set of (virtual) processes, the set $\mathbf{M}$ is called the set of MAC addresses and $\mathbf{I}$ is called the set of IP addresses. Among the elements of $P, \mathbf{M}, \mathbf{I}$, only some of them will be part of a scenario, i.e., of a concrete message exchange session in a specific LAN. We assume that the cardinality of $\mathbf{I}$ is less than or equal to the cardinality of P and $\mathbf{M}$ (in actual implementations, the cardinality of $\mathbf{I}$ is 2^{32}, the cardinality of $\mathbf{M}$ is 2^{48} and the cardinality of potential users P is unspecified).

7. Both assumptions are realistic thanks to the Binary Exponential Backoff algorithm used by Ethernet to overcome collisions.

For a virtual background $\mathcal{S}$, an *$\mathcal{S}$-scenario* (or simply a *scenario*) Σ is a 4-tuple (P_0, IP, MAC, σ) such that:

- $P_0 \subseteq P$ is the finite set of (actual) processes in a LAN;
- $IP : P_0 \longrightarrow \mathbf{I}$ is an injective function ($IP(p)$ - written IP_p - is said to be the IP-address of p);
- $MAC : P_0 \longrightarrow \mathbf{M}$ is an injective function ($MAC(p)$ - written MAC_p - is said to be the MAC-address of p).
- σ is a partial function, whose domain $dom(\sigma) \subseteq P_0^+$ is *finite*;[8] σ associates with $w \in dom(\sigma)$ a pair of values $\sigma(w) \in \mathbf{I} \times (\mathbf{M} \cup \{\perp\})$ (here $\perp$ is some fixed element not belonging to $\mathbf{M}$); for length-one words in $dom(\sigma)$, we stipulate that $\sigma(p) = \langle IP_p, MAC_p \rangle$ for all $p \in P$.

$\sigma(pwq)$ is the content p got from a chain of messages starting from q and reaching p.

We say that p is *honest* iff $p \in P_0$ and the following two conditions are satisfied for arbitrary $r \in P_0$ and $rpw \in P_0^+$:

i) if $rpw \in dom(\sigma)$ then $pw \in dom(\sigma)$ (a honest p does not forward messages it did not get);
ii) if $rpw \in dom(\sigma)$ then $\sigma(rpw) = \sigma(pw)$ (a honest p relays exactly the information it got without altering it in any manner).

The function σ is used by processes in order to update their cv vectors.

We denote by $\mathbf{I_0}$ the range of the function IP and by $\mathbf{M_0}$ the range of the function MAC (we have of course $\mathbf{IP_0} \subseteq \mathbf{I}$ and $\mathbf{M_0} \subseteq \mathbf{M}$). We say that Σ is *full* iff $\mathbf{I_0} = \mathbf{I}$: in a full scenario, all possible IP addresses are in use. A full scenario has to be considered *non* realistic. Indeed, $\mathbf{I}$ is the set of all possible IP addresses, while a LAN includes a very limited number of hosts (large LANs do not include more than a few hundreds of hosts).

6.2 Problem Impossibility

In this section, we prove that – under the system assumptions of Section 6.1 and of dynamic addressing – the Address Translation Problem cannot be solved because every algorithm either (i) might allow cache poisoning, or (ii) always terminates with the decision for the undefined identifier (which is completely useless) in order to avoid poisoning.[9]

We now formally define the notion of *address resolution algorithm*.

Definition 6.1. *An* address resolution algorithm *for a virtual background $\mathcal{S}$ is a function K associating to any $\mathcal{S}$-scenario $\Sigma = (P_0, IP, MAC, \sigma)$ and any pair of processes $p, q \in P_0$, a MAC address $K(p, q, \Sigma) \in \mathbf{M} \cup \{\perp\}$ (written $K_q^p(\Sigma)$) satisfying the following* invariance property. *$K(p, q, \Sigma)$ is the MAC address that p associates to IP_q.*

Definition 6.2 (Invariance property). *Let $\bar{\ } : P \longrightarrow P$ indicate a bijective function; for a finite word $w \in P^+$ and a process q, $\bar{w}$ denotes the word obtained from w by replacing everywhere $q \in P$ by $\bar{q}$. Suppose that $\bar{\ } : P \longrightarrow P$ is a bijection fixing p (i.e., with $\bar{p} = p$) and suppose that we are given two scenarios $\Sigma = (P_0, IP, MAC, \sigma)$ and $\Sigma' = (P_0', IP', MAC', \sigma')$ such that for every $w \in P^+$ we have (i) $pw \in dom(\sigma)$ iff $p\bar{w} \in dom(\sigma')$ and (ii) if $pw \in dom(\sigma)$, then $\sigma(pw) = \sigma'(p\bar{w})$. Then we must have $K_q^p(\Sigma) = K_q^p(\Sigma')$ for all $q \in P_0 \cap P_0'$.*

The intuitive meaning of the invariance property is that the result of a *deterministic* address resolution algorithm executed on a host p is always identical whenever "the same information" is available to p, that is, every time p receives the same sequence of messages. The $\bar{\ }$ function defines a permutation over the processes in P. It is used in the subsequent proof in order to substitute a honest process $q \in P_0 \subseteq P$ with a malicious process $\overline{q} \in P \setminus P_0$ pretending to be q, while all the other processes in P_0 remain fixed.[10] In such a context – where we have assumed the absence of cryptographic techniques that might be used for authentication – we must consider that p is not able to distinguish between two messages with the same content sent by two different hosts.

The ATP problem addressed in this paper, as well as by the ARP protocol, consists of guaranteeing that every time a LAN host p tries to resolve a certain IP address into the corresponding MAC address, it ends up by storing the correct binding in its local vector. We acknowledge that a host might record a default binding instead of the correct one, where the undefined identifier $\langle IP_x, \perp \rangle$ represents the default value used whenever the MAC address corresponding to IP_x is not known.

More formally:

Definition 6.3. *Given a virtual background $\mathcal{S}$ and an algorithm K, we say that K is* correct *if and only if for any $\mathcal{S}$-scenario Σ and for any $p, q \in P_0$ such that both p, q are honest and $p \neq q$, $K_q^p(\Sigma) \in \{MAC_q, \perp\}$. The ATP problem lies in classifying such correct algorithms K and in supplying one.*

The Correctness property posits that ARP cache poisoning cannot occur. Nonetheless, it admits undefined entries because of changed correspondence due to dynamic IP address assignment, or ARP cache entry expiration, or missing correspondence due to the lack of communication. It is worth noting that a correct algorithm should also be *valid*, i.e., it should supply a non-undefined correspondence as soon as a sufficient number of messages has been exchanged. Yet, we do not formally define such a validity property as the theorem below shows that any reasonable definition leads to impossibility.

A correct algorithm capable of producing non-undefined correspondence exists, but it works only in full scenarios, and thus is of no use in practical applications. Such a trivial algorithm can be described as follows. Let N be the cardinality of $\mathbf{I}$ (i.e., of all virtual IP addresses); we note that currently $N = 2^{32}$. If p receives less than N messages of the

8. We use P_0^+ to denote the set of finite non-empty words on the alphabet P_0. The fact that the domain is a finite subset of P_0^+ implies that not all messages are requested to arrive to their destinations before a decision about an address binding is taken. This is needed because in real systems processes do not know the cardinality of P nor that of P_0, but they have to decide within a finite time.

9. In the case of persistent addresses where an address database may be configured in the hosts (as in the case of SARPI), the impossibility does not hold.

10. We notice that assuming the invariance property for any permutation is formally equivalent to assuming it for any exchanges, since any permutation can be expressed as a composition of exchanges. This explains why in the proof we then use only the invariance for a single exchange.

kind $\sigma(pq)$ (i.e., if the set of the two-letter words pq belonging to $dom(\sigma)$ has cardinality less than N), then it sets $K_q^p(\Sigma) = \perp$ for each q; if it receives N messages of the kind $\sigma(pq)$, then it checks whether there is just one pair (IP_q, a) for each address IP_q. If this is not the case, again it sets $K_q^p(\Sigma) = \perp$ for all q; otherwise it sets $K_q^p(\Sigma)$ equal to the unique a such that (IP_q, a) belongs to the set $\{\sigma(pq) \,|\, pq \in dom(\sigma)\}$. It is justified by doing so, because, since it received N messages, then it knows that the scenario is full, so that if q is honest, then the unique pair (IP_q, a) it received must be such that $a = MAC_q$. If, on the other hand, there are two (or more) different pairs of the kind (IP_q, a) for the same address IP_q in the set $\{\sigma(pq) \,|\, pq \in dom(\sigma)\}$, then it is evident that some poisoning attack has occurred.

The problem with the above trivial algorithm is that it never produces a defined correspondence in a non-full scenario, so that it is bound to produce concrete effects only in unrealistic borderline cases. In fact, there are no better solutions:

Theorem 6.1. *Let K be a correct address resolution algorithm for $\mathcal{S} = (P, \mathbf{M}, \mathbf{I})$. Then for any non-full $\mathcal{S}$-scenario $\Sigma = (P_0, IP, MAC, \sigma)$ and for any distinct $p, q \in P_0$, we have that if p and q are honest and distinct, then $K_q^p(\Sigma) = \perp$.*

Proof. Suppose that, on the contrary, we have $K_q^p(\Sigma) = MAC_q$, for honest $p, q \in P_0$ and $q \neq p$ in a non-full $\mathcal{S}$-scenario $\Sigma = (P_0, IP, MAC, \sigma)$. Since Σ is not full, there is an $a \in \mathbf{I}$ not in the range of IP; since IP and MAC are injective functions and since the cardinality of $\mathbf{I}$ is less than or equal to the cardinalities of both $\mathbf{M}$ and P, there are $\bar{q} \in P \setminus P_0$ and $b \in \mathbf{M}$ not in the range of MAC.

Consider now a new scenario $\Sigma' = (P_0', IP', MAC', \sigma')$ obtained by Σ as follows (the modification is meant to simulate a man-in-the-middle attack - we add to the scenario an intruder $\bar{q}$, intercepting all messages originating from q and replacing into them q's MAC address with its own):

- $P_0' = P_0 \cup \{\bar{q}\}$;
- IP' extends IP by letting $IP'(\bar{q})$ be any $a \in \mathbf{I}$ not in the range of IP;
- MAC' is obtained from MAC as follows: (i) $MAC'_r = MAC_r$ for $r \neq q$ and $r \neq \bar{q}$, (ii) $MAC'_{\bar{q}} = MAC_q$, and (iii) MAC'_q is any b not in the range of MAC.

In other words, in the new scenario the intruder $\bar{q}$ happens to have the same MAC address that q had in the old scenario. We show that, as a consequence of the invariance property, $\bar{q}$ will be able to convince p that q's MAC address is the same as in the old scenario, whereas this is not anymore the case (the old q's MAC address is now $\bar{q}$'s MAC address).

Let $\bar{(\ )} : P \longrightarrow P$ be the bijection exchanging q with $\bar{q}$ and leaving all the other $r \in P$ fixed (notice that this induces an involution for words, i.e., we have $\bar{\bar{w}} = w$ for all $w \in P^+$).[11] We let $dom(\sigma')$ be equal to $\{\bar{w} \,|\, w \in dom(\sigma)\} \cup \{q\}$. We define $\sigma'(\bar{w}) := \sigma(w)$, for all $w \in dom(\sigma)$ of length bigger than 1 (if w has length 1, then $w = r$ for some r, and $\sigma'(r) := \langle IP_r, MAC_r \rangle$, according to the general definition of a scenario).

Since we have that $\bar{p} = p$ and $p \neq q$, the hypothesis of the invariance property applies, thus producing that $K_q^p(\Sigma') = K_q^p(\Sigma) = MAC_q = MAC'_{\bar{q}} \neq MAC'_q$. This demonstrates that K is not correct (contrary to the hypothesis of the theorem), provided we can show that p, q are still honest in the new scenario Σ'.

It turns out that q is certainly honest in Σ' because the only word mentioning q in $dom(\sigma')$ is the single-letter word q. It is clear also that p remains honest in the new scenario Σ', because (using the fact that $\bar{p} = p$) we have

$$\sigma'(\bar{r}p\bar{w}) = \sigma'(\overline{rpw}) = \sigma(rpw) = \sigma(pw) = \sigma'(\overline{pw}) = \sigma'(p\bar{w}),$$

for all $\bar{r}p\bar{w} \in dom(\sigma')$. Moreover, the first condition for p to be honest is also verified because, since p is honest in the old scenario, for all words of the form $\bar{r}p\bar{w}$ we have

$$\bar{r}p\bar{w} \in dom(\sigma') \Leftrightarrow rpw \in dom(\sigma) \Rightarrow \\ \Rightarrow pw \in dom(\sigma) \Leftrightarrow p\bar{w} \in dom(\sigma') \quad .$$

This concludes the proof of the Theorem. □

*Remark.*The entire proof relies on the absence of an authenticator, i.e. of an evidence that the identifier comes from the honest owner of the IP address under consideration. If such an evidence can be supplied (e.g., through cryptographic techniques) then the problem becomes solvable, as done by a few cryptography-based ARP proposals in the literature.

Remark. The definition of an "address resolution" algorithm K we gave in this section (Definition 5.1) is quite general and abstract: such a K can operate in *any* scenario (possibly giving undetermined values as results). We have proved that any such K (under realistic mild assumptions) is either incorrect or can only produce undetermined values. Only few such abstract K can give rise to real life address resolution algorithms; in fact, real life address resolution algorithms (which cannot be entirely correct according to the above theorem) operate only within scenarios obeying the rules of a precise protocol.[12] Such rules may include intermediate control steps and phases, which are also important for verification, but that are abstracted away in the general formalization of this section. In addition, the notion of a scenario we have introduced cannot be fully captured within first order logical contexts; consequently, only special scenarios can be considered in formal specifications for fully automated tools relying on decision procedures in first order logic (these are the decision procedures implemented in SMT (Satisfiability Modulo Theories) solvers). An effective way to introduce the special scenarios constructed using specific protocol rules is to introduce further *array*variables (in addition to the IP and MAC array variables mentioned in any scenario): for instance, in order to model the information $\sigma(qp)$ forwarded by a certain (malicious or correct) process p to all other processes q, one may use an appropriate array a_p. Array variables are at the core of array-based systems [28], [29], the

11. The fact that $\bar{(\ )} : P \longrightarrow P$ is an involution guarantees that any $w \in P^+$ can be written as $\bar{v}$ for a unique v: this fact guarantees the correctness of the definitions below.

12. Formally, if you like, one may say that they give undetermined results in scenarios not built up according to such rules.

formal framework underlying the tool MCMT. This is the tool we shall employ in our formal verification analysis.

7 Formal Verification of DARPI

7.1 Preliminaries on Formal Verification

The family of HARPI protocols under consideration belongs to the *infinite-state reactive parameterized systems*: although the behavior of a single host can be described by a finite state automaton, the number of components which constitute a system (i.e., a LAN), and whose behavior is determined by messages received by other system's components, is potentially unlimited.

Various techniques have been introduced in the literature to handle safety verification for such parameterized systems (see [2], [3], [4], [9], [11], [12], to name but a few entries). We chose the declarative approach of the *array-based systems* [23], [28], [29], because it offers a great flexibility and relies (at the deductive engine level) on the mature technology offered by state-of-the-art SMT-solvers, which is gaining relevance in the field of automatic theorem provers. In array-based systems (see [24], [30] for tool implementations), the state is represented by both global variables and array variables such that each array corresponds to a component of the state of the hosts, that is, the kth element of array a contains the value of component a for the host k. This representation, which is very natural, eases the modeling process. A system is specified via a pair of formulæ $\iota(\underline{p})$ and $\tau(\underline{p}, \underline{p}')$, where $\underline{p}$ is the set of parameters and array-ids, $\iota(\underline{p})$ is the formalization of possible initial states of the system, $\tau(\underline{p}, \underline{p}') := \bigvee_{i=1}^{n} \tau_i(\underline{p}, \underline{p}')$ symbolizes the possible state transitions of the system – according to the considered algorithm – modifying $\underline{p}$ into $\underline{p}'$. A safety problem is given by a further formula $\upsilon(\underline{p})$, which describes the set *Bad* of states verifying the unsafe condition. Each transition $\tau_i \in \tau$ is composed by a *guard* and a set of updates: if the current values of parameters and arrays satisfy the guard, then the transition may fire and the updates are applied. More guards may be verified simultaneously. In this case, one of the corresponding transitions fires nondeterministically. A *safety model checking problem* is the problem of checking whether the formula

$$(^{\star})_n \qquad \iota(\underline{p}_0) \wedge \tau(\underline{p}_0, \underline{p}_1) \wedge \cdots \wedge \tau(\underline{p}_n, \underline{p}_{n+1}) \wedge \upsilon(\underline{p}_{n+1}),$$

is satisfiable for some n, that is, whether a state in *Bad* can be reached from an initial state by applying the possible transitions. In order to verify whether a protocol is safe with respect to *Bad*, the tool we use in this work adopts a *backward reachability* policy. The search starts from *Bad* and, using the state transitions, for every element of *Bad* computes the pre-image, i.e., the set of states which can lead to *Bad*. For every set of obtained pre-images the same procedure is repeatedly applied, until one of the following two events occurs: either (i) a fixed point is reached (not intersecting initial states), meaning that the pre-image computation cannot reach other states different from the current ones, or (ii) an initial state is reached. In the former case, no formulæ of type $(^{\star})_n$ describing the reachability of *Bad* can be satisfied and the system is safe. In the latter case, some formula of type $(^{\star})_n$ is satisfiable and the system is unsafe.

We used the Model Checker Modulo Theories (MCMT) tool [30], whose state variables include arrays. Sets of states and transitions of a system are described by quantified first-order formulæ *of special kinds*. The tool leverages decision procedures (as implemented in state-of-the-art SMT solvers) to treat satisfiability problems involving various datatypes such as arrays, integers, Booleans, etc. Checks for safety and fix-points are performed by solving SMT problems (due to the special shape of the formulæ used to describe sets of states and transitions, such checks can be effectively discharged); MCMT uses Yices [26] as the underlying SMT solver. In addition to standard SMT techniques, efficient heuristics for quantifier instantiation, specifically tailored to model checking, are at the heart of the system. Termination of the backward search is guaranteed only under specific assumptions, but it commonly arises in practice (for a full account of the underlying theoretical framework, the reader is referred to [29]). MCMT guarantees the safety of a protocol for any number N of system components. MCMT maintains formulæ describing the set of states that can reach a *Bad* state in one, two, three, ... etc. steps. Such formulæ describe infinite sets of states. For this description to be appropriate, quantified variables are needed. The tool increases the number of quantified variables it uses as soon as it realizes that it needs more variables. It stops whenever it gets a fixed point (safe outcome) or a set of states intersecting the set of initial states (unsafe outcome)

The process of converting an algorithm into an MCMT model is performed manually: we extracted the pseudo-codes included in this paper from the UML diagrams included in the documentation available in the official site (and validated with the implemented source code), by abstracting away the implementation-dependent details. From the pseudo-codes, we derived the MCMT models, composed of both the transactions describing each event that might occur when running the protocols (policies execution, timer expiration, and if-statements in the pseudo-codes), and additional transactions modeling all the possible behaviors of a malicious attacker. This procedure requires deep comprehension of the algorithm.

We used MCMT to verify the correctness of both SARPI and DARPI. For the sake of brevity, in the next section we describe in detail the DARPI modeling; all developed models are available at https://homes.di.unimi.it/∼pagae/ARPON/index.html.

7.2 DARPI Modeling

We developed several models for DARPI, differing in the absence or presence of malicious processes, as well as in the considered *Bad* formula. A first model, which reproduced the behavior of the protocol for arbitrary number of honest hosts, was determined to be safe. It contains all possibile events, although serialized according to the generation and management of one event at a time. By adding one malicious host, though, we verified that the protocol is *unsafe* with respect to MITM attacks, according to the impossibility result of Section 6.2. In such a case, the sequence of events yielded by the model is the following: host v generates a honest basic request for a host h. h executes Verify(), Deny (), and subsequently it sends a reply (lines 17-25 of

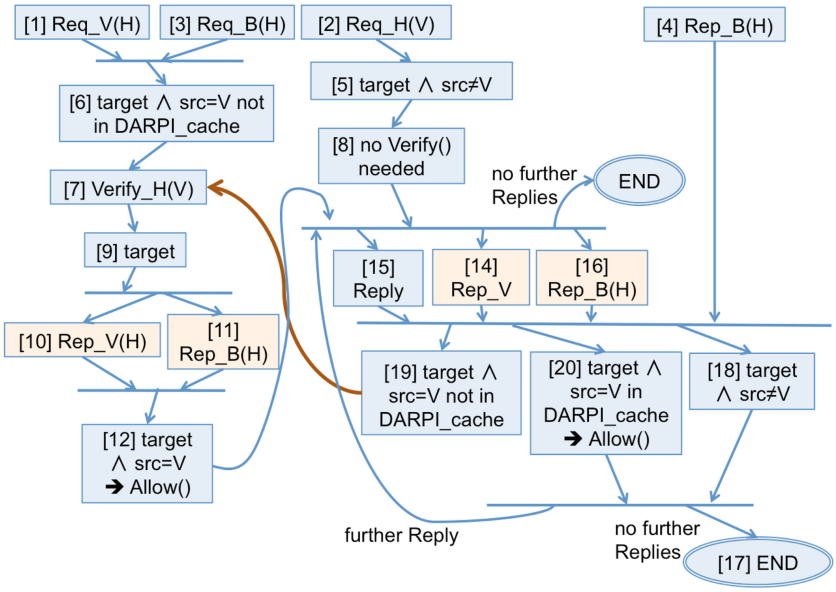

Fig. 5. Diagram summarizing the DARPI formal model.

Algorithm 3); furthermore, h adds v in its own DARPI_cache (lines 11-12). The transaction will be terminated as soon as v sends a confirmation reply to h. In the meantime, a malicious host sends a poisoned unsolicited reply to h impersonating v. Such a message wins the race and is received by h before the awaited reply from v. As a consequence, h interprets the reply as the awaited response and updates its ARP cache (lines 36-39) thus poisoning it.

Considering this negative result, it was necessary to understand if an intermediate result could be obtained; we proceeded as follows. Our aim was to prove that, although DARPI cannot escape from the impossibility result, when *ARP cache poisoning occurs, it is always removed*. The key point is that, in DARPI, all messages are broadcast, and no information is inserted into an ARP cache without verification. Cache poisoning in a host h may occur only upon the request of the Allow() policy, which is executed only when a reply is received from an IP address listed in h's DARPI cache. Furthermore, an IP address is inserted into the DARPI cache as a consequence of the local generation of an ARP request. By construction, these requests are sent broadcast; hence, the legal owner of the target IP (let us say k) always receives it and generates a reply that eventually arrives at h. If the reply matches with an entry in the DARPI cache, Allow() is executed and no cache poisoning occurs. If by contrast another malicious reply has been received earlier, which poisons h's ARP cache, then k's reply has yet to be received and – when it arrives – the Deny() policy applied by ArpON removes the poisoned entry from the ARP cache, and the verification procedure re-starts. Hence, what we aim at formally verifying is that – when cache poisoning occurs – *there is still the legal reply pending which must be received* regardless of which events are occurring in the network.

To this end, we have developed the DARPI model indicated as *comprehensive* model, described in the following lines. In our models, we assume that all hosts are correctly configured and that the network is reliable (see Section 6.1). We indicate with N the number of hosts in the network. We focus on systems composed of $N \geq 3$ processes. Among them we identify a malicious host b, aiming at impersonating a victim v, and a generic honest host denoted by h. The lower bound on system cardinality is enforced by adding it to the guards of all transitions. b wants to convince one of the honest hosts that IP_v corresponds to its MAC_b. In order for this to occur, b may send any message at any time in order to fool the other hosts; Gratuitous Announces sent by b may be unicast rather than broadcast. Yet, b cannot alter nor destroy messages sent by other hosts. By contrast, the h's and v honestly generate and process messages according to DARPI. To close the verification process, we serialize the events. The modeling of time is neglected and DARPI_cache entries never expire, which just eases the work of malicious processes by indefinitely allowing the unsafe event sequence described above. We take both the IP address and the MAC address of a host x to be equal to x.

The following global variables are used: φ indicates the logical clock of the computation; sh, sp and tp correspond to the `sha`, `spa` and `tpa` message fields respectively; a flag ww is used as a switch as explained below. The state of each process x is represented by the following local variables: $sm[x]$ indicates whether x must send a message and of what kind; $CM[x]$ and $CP[x]$ respectively contain the MAC and IP addresses of the victim in x's cache (cv_{xv}). The flag $fv[x]$ indicates whether x must execute the Verify() procedure; the target of the verification request is always v. An entry in DARPI_cache is represented by the field $tD[x]$ for the target of the verification request. Finally, $tg[x]$ remembers the destination of the message to be sent when $sm[x]$ is set.

Note that the primed notation ($'$) of a variable indicates its updated value in case a transition fires (if its guard is satisfied). In the following transition formulæ, existentially and universally quantified variables x, y, z etc. are processes whose local state satisfies the transition guard and is updated as indicated.

To provide further clarification of the model, we refer to Fig. 5 where a diagram describes the model and its transitions: numbers in square brackets indicate the corresponding transition τ. The points where the replies of v and b to a

verification request may arrive in a different order are highlighted in orange.

The initial state satisfies

$$\iota_D := \varphi = 0 \wedge sh = 0 \wedge sp = 0 \wedge tp = 0 \wedge ww = 0 \wedge (\forall x.sm[x] = 0 \wedge CM[x] = 0 \wedge CP[x] = 0 \wedge fv[x] = 0 \wedge tD[x] = 0 \wedge tg[x] = 0),$$

that is, no process has executed the current initial step, no message is present, no caches contain any information regarding v, all DARPI_caches are empty, and no process has a message to send. The unsafe state is

$$\upsilon_D := \varphi = 6 \wedge (\exists z1, z2, z3.CP[z1] \neq CM[z1] \wedge z2 = v \wedge sm[z2] = 0 \wedge z3 = CM[z1] \wedge sm[z3] \geq 0),$$

that is, a process $z1$ exists whose ARP cache is poisoned, since it remembers the MAC address of a process $z3$ instead of that of the victim $z2 = v$; the malicious host $z3$ may have a message to send, while the victim has *no* messages to send. If this occurs, $z1$ will be unable to call the Deny() and Verify() procedures and the poisoning will persist. This *Bad* formula precisely is the negation of the above reasoning, and aims at showing that – whenever cache poisoning occurs – there is still a honest reply somewhere that has yet to be received.

The model includes some invariants, not discussed here for the sake of brevity, that ease the closing of the verification. Apart from the initial event generation, the model mimics Algorithm 3; we indicate the Algorithm lines to which each transition corresponds. Transitions τ_1 to τ_4 are guarded by $\varphi = 0$ (initial step) and any one of them may fire to generate the initial event.

In the first transition, a honest request is generated by v to another host

$$\tau_1 := \varphi = 0 \wedge (\exists x, y.x \neq y \wedge x = v \wedge y \neq v \wedge N \geq x \wedge N \geq y \wedge \varphi' = 1 \wedge tp' = y \wedge sh' = v \wedge sp' = v).$$

In this and the following transitions, a condition such as $N \geq x$ forces process indexes to be within the system cardinality.

The second transition similarly describes the generation of a honest request from a h to v (which is not reported). By contrast, the generation of a malicious request is modeled as follows:

$$\tau_3 := \varphi = 0 \wedge (\exists x, y.x \neq y \wedge x = b \wedge y \neq v \wedge N \geq x \wedge N \geq y \wedge \varphi' = 1 \wedge sm'[x] = 3 \wedge tp' = y \wedge sh' = x \wedge sp' = v).$$

The value of $sm[x] = 3$ is never changed again. It indicates that the malicious process already fired, and prevents further generation of malicious messages to avoid repeating already visited patterns and from entering into infinite loops. A malicious unsolicited reply is similarly generated

$$\tau_4 := \varphi = 0 \wedge (\exists x, y.x \neq y \wedge x = b \wedge y \neq v \wedge N \geq x \wedge N \geq y \wedge \varphi' = 8 \wedge sm'[x] = 3 \wedge tp' = y \wedge sh' = x \wedge sp' = v),$$

and the model jumps to the transitions relative to the management of replies, that is, $\varphi' = 8$.

The next transitions describe the reception of requests either by the target process (line 17) which is also the victim and must just generate a reply (lines 21-24)

$$\tau_5 := \varphi = 1 \wedge (\exists x.x = v \wedge x = tp \wedge N \geq x \wedge sp \neq v \wedge \varphi' = 2 \wedge sm'[x] = 1 \wedge tg'[x] = sp),$$

or when the apparent source is the victim

$$\tau_6 := \varphi = 1 \wedge (\exists x.x = tp \wedge sp = v \wedge N \geq x \wedge tD[x] = 0 \wedge \varphi' = 2 \wedge sm'[x] = 1 \wedge CM'[x] = 0 \wedge CP'[x] = 0 \wedge fv'[x] = 1 \wedge tg'[x] = sp).$$

In both cases, it has a reply to send to the request source. In the latter case, the target also summons Deny() and it has to verify the identity of the source (lines 18-25).

According to the algorithm, a host first sends possible verification requests, and then sends its own reply. The next two transitions make it possibile to enter the verification phase. If a verification must be performed, a message is sent and the appropriate entry is inserted into the DARPI cache (lines 11-13)

$$\tau_7 := \varphi = 2 \wedge (\exists x.fv[x] = 1 \wedge x \neq v \wedge N \geq x \wedge \varphi' = 3 \wedge tp' = v \wedge sh' = x \wedge sp' = x \wedge fv'[x] = 0 \wedge tD'[x] = v).$$

If no process must perform Verify() – as in the case of a request received by the victim (τ_5) – then the execution moves to the transitions regarding reply generation

$$\tau_8 := \varphi = 2 \wedge (\forall x.fv[x] = 0) \wedge \varphi' = 7.$$

If a verification request is sent, it is processed by its target (lines 17-25)

$$\tau_9 := \varphi = 3 \wedge (\exists x.x = v \wedge x = tp \wedge sp \neq x \wedge N \geq x \wedge \varphi' = 4 \wedge sm'[x] = 2 \wedge tg'[x] = sp),$$

where the value of $sm[x]$ indicates that a reply to a verification must be sent, as described in the following transition:

$$\tau_{10} := \varphi = 4 \wedge (\exists x.x = v \wedge sm[x] = 2 \wedge N \geq x \wedge N \geq tg[x] \wedge \varphi' = 5 \wedge sm'[x] = 0 \wedge tp' = tg[x] \wedge sh' = x \wedge sp' = x \wedge tg'[x] = 0).$$

It is also possible that, at this step, a malicious host generates a poisoned unsolicited reply

$$\tau_{11} := \varphi = 4 \wedge (\exists x, y.x \neq y \wedge x = b \wedge sm[x] \neq 3 \wedge tD[y] = v \wedge y \neq v \wedge N \geq x \wedge N \geq y \wedge \varphi' = 5 \wedge sm'[x] = 3 \wedge tp' = y \wedge sh' = x \wedge sp' = v).$$

It is worth noting that this unsolicited reply is addressed to the process y waiting for a reply to its verification, thus reproducing the worst case that leads to ARP cache poisoning. The reply to a verification request is processed as follows (lines 36-39):

$$\tau_{12} := \varphi = 5 \wedge (\exists x.x = tp \wedge sp \neq x \wedge sp = v \wedge tD[x] = v \wedge \varphi' = 6 \wedge CM'[x] = sh \wedge CP'[x] = sp \wedge tD'[x] = 0).$$

This is a point in the algorithm (line 39 in Algorithm 3) at which the Allow() procedure is called upon and cache poisoning may occur. Hence, a fictitious transition – one that does nothing – is inserted for the sole purpose of taking a snapshot of the system just after the Allow() execution, so as to verify the unsafe condition before proceeding

$$\tau_{13} := \varphi = 6 \wedge ww = 0 \wedge (\exists x.N \geq x \wedge \varphi' = 7).$$

Subsequently, two cases may occur; either (i) the victim has yet to send its own reply to the verification because previously the transition τ_{11} fired instead of τ_{10}, or (ii) τ_{10} fired and the model moves to the transitions modeling reply generation and management. In the former case

$$\tau_{14} := \varphi = 7 \wedge (\exists x.x = v \wedge sm[x] = 2 \wedge N \geq x \wedge N \geq tg[x] \wedge \varphi' = 8 \wedge sm'[x] = 0 \wedge tp' = tg[x] \wedge sh' = x \wedge sp' = x \wedge tg'[x] = 0).$$

The transition modeling the generation of a basic reply is similar (τ_{15}), but fires in case of $sm[x] = 1$ for some x. Also, at this point, a malicious reply might be generated by b

$$\tau_{16} := \varphi = 7 \wedge (\exists x, y.x \neq y \wedge x = b \wedge sm[x] \neq 3 \wedge y \neq v \wedge sm[y] = 1 \wedge tg[y] = v \wedge tD[y] = 0 \wedge N \geq x \wedge N \geq y \wedge \varphi' = 8 \wedge sm'[x] = 3 \wedge tp' = y \wedge sh' = x \wedge sp' = v).$$

This reply is directed to the process that should send a reply to the victim, not having the victim in its DARPI cache. This event mimics the situation in which both v and b reply to a verification request, but v's reply (generated in τ_{10}) arrives before b's reply (generated here). If no reply must be sent, the system terminates

$$\tau_{17} := \varphi = 7 \wedge (\forall x.sm[x] \neq 1 \wedge sm[x] \neq 2 \wedge \varphi' = 11).$$

Since no transition is guarded by $\varphi = 11$, no further system evolution is possible.

Subsequently, three transitions model the processing of a reply (lines 36-43), for the cases respectively of reply addressed to v (which would be managed by SARPI), reply generated by v not in the DARPI cache of the process

$$\tau_{19} := \varphi = 8 \wedge (\exists x.x = tp \wedge sp \neq x \wedge sp = v \wedge tD[x] \neq v \wedge N \geq x \wedge \varphi' = 9 \wedge CM'[x] = 0 \wedge CP'[x] = 0 \wedge fv'[x] = 1),$$

or reply generated by v in the DARPI cache of the process

$$\tau_{20} := \varphi = 8 \wedge (\exists x.x = tp \wedge sp \neq x \wedge sp = v \wedge tD[x] = v \wedge N \geq x \wedge \varphi' = 6 \wedge ww' = 1 \wedge CM'[x] = sh \wedge CP'[x] = sp \wedge tD'[x] = 0).$$

The former raises the execution of a new verification procedure. The latter corresponds to a call to Allow() and – as before – a transition immediately following this call is inserted in order to verify υ_D:

$$\tau_{21} := \varphi = 6 \wedge ww = 1 \wedge (\exists x.N \geq x \wedge \varphi' = 9 \wedge ww' = 0).$$

Hence, the value of ww makes it possibile to differentiate between the instant after τ_{12} and the instant after τ_{20}, so as to appropriately update the value of φ after checking υ_D and continue the execution from the appropriate point.

The last transitions determine the subsequent actions. The state of all the processes in the system must be considered:[13] if some process exists that has yet to verify, the execution goes back to transition τ_7 for that process

$$\tau_{22} := \varphi = 9 \wedge (\exists x.x \neq v \wedge fv[x] = 1 \wedge N \geq x \wedge \varphi' = 2).$$

If some process exists that has yet to reply, the execution goes back to transitions τ_{14}-τ_{15} for that process

$$\tau_{23} := \varphi = 9 \wedge (\forall x.fv[x] = 0) \wedge (\exists x.sm[x] \geq 1 \wedge sm[x] \leq 2 \wedge N \geq x \wedge \varphi' = 7).$$

If no process in the system has any message to generate (universally quantified variable), then the system ends

$$\tau_{24} := \varphi = 9 \wedge (\forall x.fv[x] = 0 \wedge sm[x] \neq 1 \wedge sm[x] \neq 2 \wedge N \geq x \wedge \varphi' = 11).$$

7.3 Verification Results

The DARPI model described in Section 7.2 is the most complex of the models we developed for this work, and it highlights the problems we had to face in the verification process. As it can be noticed, most of the formulas describing either the algorithm or the attacker's actions involve quantifiers (see e.g., the modeling of Bad States υ_D, or transition τ_{23} that involves both existential and universal quantifiers). Such quantifiers are the logical counterpart of the fact that the system to be verified is a *parameterized system*, in the sense that it covers scenarios where a *finite but unbounded and not a priori known* number of actors occur.

There are well known results showing that even limited fragments of logics and theories involving quantifiers are undecidable and – also when the problem is decidable – checking for satisfiability might be extremely expensive in terms of both memory and computation power. The technique used by MCMT [30] is a *backward reachability analysis* that, when successful, automatically synthesizes invariants able to certify systems safety. Backward reachability analysis requires satisfiability tests both for fixpoints and safety checks; such satisfiability tests are shown to lie within quantified but decidable fragments via quantifier elimination or quantifier instantiations techniques [17], [28], [29] (when universal guards occur, however, overapproximations are adopted [7]). An alternative implementation of backward reachability is in the tool Cubicle [24]. In the last years, different techniques (mostly based on variants of the extension of the IC3 algorithm [13] to infinite state systems [34]) were introduced in order to analyze parameterized systems and to

13. As an example, recall that a gratuitous announce triggers the generation of a verification by *each* recipient.

TABLE 1
Results of HARPI Formal Verification

	outcome	time (s.)	depth	nodes	SMT calls	# literals
SARPI – no malicious: ATP	SAFE	0.128	1	1	1166	2
SARPI – broadcast malicious: ATP	SAFE	0.144	1	1	1346	2
SARPI – unicast malicious: ATP	SAFE	0.167	1	1	1624	2
DARPI – no malicious host: ATP	SAFE	0.381	2	1	1745	11
DARPI – broadcast malicious: ATP	UNSAFE	67011	12	4352	2597327	12
DARPI – **comprehensive model**: *un-poisoning*	SAFE	304.3	20	1258	308682	18
DARPI – generation of Request from victim - malicious host: *un-poisoning*	SAFE	25.35	19	291	35736	11
DARPI – generation of Request to victim - malicious host: *un-poisoning*	SAFE	17.33	11	127	14157	11
DARPI – generation of unsolicited Reply from malicious host: *un-poisoning*	SAFE	32.69	19	299	40276	11

synthesize universally quantified invariants, see e.g., [35]. A forthcoming paper [22] makes a thorough comparison between an IC3 implementation and the MCMT/Cubicle backward reachability technique, showing that the two methodologies are somewhat orthogonal to each other, at least regarding the number of solved problem instances.

In Table 1, the verification results are shown of the formal models which have been developed, in terms of their safety. For each experiment the following were reported: the outcome of the verification, the time spent by MCMT to perform the verification, the maximum depth of the tree of system states explored by MCMT, the number of states (nodes) explored, the number of calls to the underlying SMT-solver, the maximum number of literals in the formulæ passed to the SMT-solver. The outcome SAFE refers to the fact that the algorithm correctly solves the problem considered. The first five safety verifications test the unsafe condition

$$\upsilon_{ATP} := \exists z.CP[z] \neq CM[z],$$

that is, no process z exists such that for v it records a MAC address not corresponding to the IP address.

In the case of SARPI, we experimented with any number of hosts, and with malicious processes able to send either only broadcast messages (according to the specifications of SARPI), or unicast messages too. In all cases the algorithm proved to be safe, and verification was quickly computed by the tool. By contrast, DARPI has proved to be safe only when there is no malicious host in the system. When modeling the actions of just one malicious host able to send broadcast, but not unicast, messages, then, according to the impossibility proof, DARPI results are unsafe and the destructive sequence of events is the one supplied in Section 7.2. It is worth noting that unicast messages are more dangerous as they may trick one specific host while the others are not aware of what is happening as they do not receive any message.

The results found on line six of the table have been achieved by running the comprehensive model described in section 7.2. The verification conducted shows that DARPI is always able to remove poisoned information from the ARP caches. Note that the unsafe condition υ_D in this model describes states where ARP cache poisoning already occurred, but the victim has no message to send in order to remove it (Section 7.2). The SAFE outcome guarantees that, when a cache is poisoned, the victim always has yet to send a reply that will remove the entry from the cache.

To complete our representation, we also report the results achieved with non-comprehensive DARPI models analyzing specific cases.

We measured - through simulations in an ad hoc setting - the duration of the ArpON shaded area in DARPI, which turned out to be 0.11 ms, and this interval is not sufficient for carrying out a MITM attack. More precisely, in MITM attacks packets are intercepted by the attacker, analyzed, sometimes modified and subsequently re-injected in the network; an attacker must grab at least one packet of the attacked traffic flow. This implies that packets arriving to the NIC (Network Interface Card) have to be delivered to the application level where a specific application will analyze their content and decide the next move. This can be done in two ways: either, a packet has to traverse the entire operating system TCP/IP stack on the attacker host. The time to perform such an operation has been recently estimated in [31], where it turns out that a packet takes more than 5 ms to traverse the entire stack. Or, the grabbed packet may be directly delivered from the NIC to the attacker application (*flow bifurcation* and *zero-copy*), bypassing the TCP/IP stack. On the most optimized platforms, this takes not less than 0.53 ms [31]. Consequently, in both cases, DARPI removes a cache poisoning in a time shorter than the time spent by an attacker in capturing, modifying and injecting a packet back into the network.

8 Conclusion

In this study, we have explored a long-standing attack brought to Internet hosts, namely, MITM attack through ARP cache poisoning, studying how to solve it using protocols compatible with the existing ARP versions, thus not requiring significant changes to network devices. To this end, we formally define the Address Translation Problem (ATP) addressed by the Address Resolution protocols adopted in the Internet to find the correspondence between network and MAC addresses. We prove that – in the case of non-persistent addresses and in the absence of cryptography – no correct and effective solution exists for ATP because either (i) no correspondence is ever supplied, or (ii) a wrong correspondence might be supplied. In both cases, the problem properties are violated. As a consequence of this demonstration, we propose the ArpON algorithm that is perfectly back-compatible and interoperable with existing ARP implementations, and permits a balance between the described extreme behaviors. By formally validating the properties of ArpON, we verify that ArpON – according to the above impossibility proof – may provide an incorrect correspondence. Yet, when this occurs, *ArpON always removes cache poisoning* in a very short time.

The impossibility proof clearly refers to the worst case where attackers always interfere with normal ARP functioning trying constantly to trick other hosts regarding correct address correspondence. This does not always occur in practice. The source code of ArpON is publicly available, and has been downloaded by more than 100,000 users thus far. Although received feedbacks have been positive, we are currently accurately assessing its performance through simulations, with the aim of empirically proving the upper bound on cache poisoning duration for different network sizes, percentages of malicious hosts, and aggressiveness of attackers, in comparison with classical ARP.

REFERENCES

[1] C. L. Abad and R. I. Bonilla, "An analysis on the schemes for detecting and preventing ARP cache poisoning attacks," in *Proc. 27th Int. Conf. Distrib. Comput. Syst. Workshops*, 2007, pp. 60–70.

[2] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine ,"Regular model checking without transducers (on efficient verification of parameterized systems)," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2007, pp. 721–736.

[3] P. A. Abdulla, F. Haziza, and L. Holík, "All for the price of few," in *Proc. Int. Workshop Verification Model Checking Abstr. Interpretation*, 2013, pp. 476–495.

[4] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena ,"A survey of regular model checking," in *Proc. Int. Conf. Concurrency Theory*, 2004, pp. 35–48.

[5] M. Agarwal, S. Biswas, and S. Nandi ,"Advanced stealth man-in-the-middle attack in WPA2 encrypted wi-fi networks," *IEEE Commun. Lett.*, vol. 19, no. 4, pp. 581–584, Apr. 2015.

[6] M. M. Alani. *Guide to OSI and TCP/IP Models*. Cham, Switzerland: Springer, 2014.

[7] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi ,"Universal guards, relativization of quantifiers, and failure models in model checking modulo theories," *J. Satisfiability Boolean Model. Comput.*, vol. 8, pp. 29–61, 2012.

[8] M. W. Alford *et al.*, "Formal foundation for specification and verification,"*Lecture Notes Comput. Sci.*, vol. 190, pp. 203–285, 1985.

[9] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen ,"Flat acceleration in symbolic model checking," in *Proc. Int. Symp. Automated Technol. Verification Anal.*, 2005, pp. 474–488.

[10] M. Barnaba, "Anticap," 2003. Accessed: Jul. 21, 2021. [Online]. Available: http://cvs.antifork.org/cvsweb.cgi/anticap

[11] R. Bloem *et al.*, *Decidability of Parameterized Verification*. San Rafael, CA, USA: Morgan & Claypool Publishers, 2015.

[12] A. Bouajjani, P. Habermehl, and T. Vojnar,"Abstract regular model checking," in *Proc. Int. Conf. Comput. Aided Verification*, 2004, pp. 372–386.

[13] A. Bradley, "SAT-based model checking without unrolling," in *Proc. Int. Workshop Verification Model Checking Abstract Interpretation*, 2011, pp. 70–87.

[14] D. Bruschi, A. Di Pasquale , S. Ghilardi, A. Lanzi, and E. Pagani , "Formal verification of ARP (Address Resolution Protocol) through SMT-based model checking - A case study," in *Proc. 13th Int. Conf. Integrated Formal Methods*, 2017, pp. 391–406.

[15] D. Bruschi, A. Ornaghi, and E. Rosti, "S-ARP: A secure address resolution protocol, "in *Proc. 19th IEEE Annu. Comput. Secur. Appl. Conf.*, 2003, pp. 66–74.

[16] B. Bhushan, G. Sahoo, and A. K. Rai, "*Man-in-the-middle attack in wireless and computer networking" A review*, in *Proc. 3rd Int. Conf. Adv. Comput. Commun. Automat.*, 2017, pp. 1–6.

[17] A. Carioni, S. Ghilardi, and S. Ranise ,"MCMT in the land of parameterized timed automata," in *Proc. 6th Int. Verification Workshop*, 2010, pp. 47–64.

[18] Y. Chahid, M. Benabdellah, and A. Azizi, "Internet of Things protocols comparison, architecture, vulnerabilities and security: State of the art," in *Proc. 2nd Int. Conf. Comput. Wireless Commun. Syst.*, 2017, pp. 1–6.

[19] K. Cheng, M. Gao, and R. Guo ,"Analysis and research on HTTPS hijacking attacks," in *Proc. 2nd Int. Conf. Netw. Secur., Wirel. Commun Trusted Comput.*, 2010, pp. 223–226.

[20] S. Cheshire, *IPv4 Address Conflict Detection*, RFC 5227, Jul. 2008

[21] S. Cheshire, B. Aboba, and E. Guttman, *Dynamic Configuration of IPv4 Link-Local Addresses*, RFC 3927, May 2005.

[22] A. Cimatti, A. Griggio, and G. Redondi, "Universal invariant checking of parametric systems with quantifier-free SMT reasoning," in *Proc. Int. Conf. Automated Deduction*, 2021, pp. 131–147.

[23] S. Conchon, A. Goel, S. Krstic, A. Mebsout, F. Zaïdi, "Invariants for finite instances and beyond," in *Proc. Formal Methods Comput.-Aided Des.*, 2013, pp. 61–68.

[24] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi, "Cubicle: A parallel SMT-based model checker for parameterized systems," M. Parthasarathy, S. A. Seshia Eds., in *Proc. 24th Int. Conf. Comput. Aided Verification*, 2012, pp. 718–724.

[25] Droms R.: *Dynamic Host Configuration Protocol*. Request for Comments 2131, Mar. 1977.

[26] B. Dutertre and L. de Moura, "The Yices SMT Solver," Tech. Rep., SRI International, Menlo Park, CA, USA, 2006. [Online]. Available: http://yices.csl.sri.com

[27] O. Eisen, "Catching the fraudulent man-in-the-middle and man-in-the-browser," *Netw. Secur.*, vol. 2010, no. 4, pp. 11–12, 2010.

[28] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Towards SMT model checking of array-based systems," in *Proc. Int. Joint Conf. Automat. Reasoning*, 2008, pp. 67–82.

[29] S. Ghilardi and S. Ranise, "Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis," *J. Logical Methods Comput. Sci.*, vol. 6, no. 4, pp. 1–48, 2010.

[30] S. Ghilardi and S. Ranise, "MCMT: A model checker modulo theories," in *Proc. Int. Joint Conf. Automat. Reasoning*, 2010, pp. 22–29.

[31] M. Glübert, "Latency optimization and analysis through the use of a high-speed packet IO framework for high-bandwidth data processing," in *Proc. Seminar Master Applied Res.*, 2020, pp. 1–4.

[32] D. G.-Smith, "*Whatever happened to IPv6?*, ITPRO Newsletter, Accessed: Mar. 2, 2018. [Online] Available: http://www.itpro.co.uk/internet-protocol-version-6-ipv6/30657/whatever-happened-to-ipv6

[33] N. E. Hastings and P. A. McLean, "TCP/IP spoofing fundamentals," in *Proc. IEEE 15th Annu. Int. Phoenix Conf. Comput. Commun.*, 1996, pp. 218–224.

[34] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Proc. Int. Conf. Theory Appl. Satisfiability Testing*, 2012, pp. 157–171.

[35] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, "Property- directed inference of universal invariants or proving their absence," D. Kroening, C. S. Păsăreanu Eds., in *Proc. Int. Conf. Comput. Aided Verification*, 2015, pp. 583–602.

[36] Lawrence Berkeley National Laboratory: *ARPWATCH: ARP Spoofing Detector*. Accessed: Jul. 21, 2021. [Online] Available: ftp://ftp.ee.lbl.gov/ARPwatch.tar.gz

[37] W. Lootah, W. Enck, and P. McDaniel, "TARP: Ticket-based address resolution protocol," *Comput. Netw.*, vol. 51, no. 15, pp. 4322–4337, 2007.

[38] S. Nam, D. Kim, and J. Kim,"Enhanced ARP: Preventing ARP poisoning-based man-in-the-middle attack," *IEEE Commun. Lett.*, vol. 14, no. 2, pp. 187–189, Feb. 2010.

[39] T. Narten, E. Nordmark, W. Simpson, and H. Soliman *Neighbor Discovery for IP Version 6 (IPv6)*, RFC 4861, Sep. 2007.

[40] A. P. Ortega, X. E. Marcos, L. D. Chiang, and C. L. Abad, "Preventing ARP cache poisoning attacks: A proof of concept using OpenWrt," in *Proc. Latin Amer. Netw. Operations Manage. Symp.*, 2009, pp. 1–9.

[41] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234., Apr. 1980.

[42] D. C. Plummer, *An Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*, RFC 826, Nov. 1982.

[43] R. Shirey, *Internet Security Glossary, Version 2*, RFC 4949, Aug. 2007. Accessed: Jul. 21, 2021

[44] I. Teterin, Antidote. Accessed: Jul. 21, 2021. [Online] Available: http://online.securityfocus.com/archive/1/299929

[45] J. Tian, K. R. B Butler, P. D. McDaniel, and P. Krishnaswamy, "Securing ARP/NDP from the ground up," IEEE *Trans. Inf. Forensics Secur.*, vol. 12, no. 9, pp. 2131–2143, Sep. 2017.

[46] Middle approach to asynchronous and backward-compatible detection and prevention of ARP cache poisoning, by M. V. Tripunitara and P. Dutta, U.S. Patent 6,771,649, [Online]. Available: https://patentimages.storage.googleapis.com/59/e6/84/1362440b2128d9/US6771649.pdf

Danilo Bruschi received the PhD degree in computer science from the University of Milan. He is currently a professor of computer science and heads the System Security Laboratory, University of Milan. His research interests include system security, IoT security, social implications of ICT insecurity, formal methods, and computer security.

Andrea Di Pasquale is currently a technical lead engineer with Infovista, S.A. His research interests include software engineering, high performance systems, high-performance networks, computer networks, and network security.

Silvio Ghilardi is currently a full professor of mathematical logic with the Department of Mathematics, University of Milan. His research interests include automated reasoning and SMT-based model-checking. He is an associate editor for the *Journal of Automated Reasoning*.

Andrea Lanzi is currently an associate professor with Computer Science Department, University of Milan. His research interests include several aspects of cyber security, host intrusion detection systems, memory errors exploitation, reverse engineering, malware, and forensic analysis. He has studied the application of emulation or virtualization and compiler techniques for malware analysis and detection in the Android context.

Elena Pagani is currently an associate professor with Computer Science Department, University of Milan. Her research interests include network protocols and architectures, MANETs, opportunistic and wireless sensor networks, performance evaluation, and formal verification of network protocols. She is the area editor for *Elsevier Computer Communications Journal*.