Formal Verification of ARP (Address Resolution Protocol) Through SMT-Based Model Checking - A Case Study -

Danilo Bruschi[1], Andrea Di Pasquale[1], Silvio Ghilardi[2], Andrea Lanzi[1], and Elena Pagani[1](✉)

[1] Università degli Studi di Milano, via Comelico 39, 20135 Milano, Italy
{danilo.bruschi,andrea.lanzi,elena.pagani}@unimi.it, spikey.it@gmail.com
[2] Università degli Studi di Milano, via Saldini 50, 20133 Milano, Italy
silvio.ghilardi@unimi.it

Abstract. Internet protocols are intrinsically complex to understand and validate, due both to the potentially unbounded number of entities involved, and to the complexity of interactions amongst them. Yet, their safety is indispensable to guarantee the proper behavior of a number of critical applications.

In this work, we apply formal methods to verify the safety of the Address Resolution Protocol (ARP), a standard protocol of the TCP/IP stack i.e. the communication protocols used by any Internet Host, and we are able to formally prove that the ARP protocol, as defined by the standard Request for Comments, exhibits various vulnerabilities which have been exploited since many years and still are the main ingredient of many attack vectors. As a complementary result we also show the feasibility of formal verification methods when applied to real network protocols.

Keywords: ARP · Man-in-the-Middle attack · Denial-of-Service attack · Formal verification · Model evaluation · Satisfiability Modulo Theories

1 Introduction

Core of this work is the Address Resolution Protocol (ARP), a standard protocol of the TCP/IP stack i.e. the set of communication protocols used by any Internet Host. More precisely, we apply a formal method to verify the safety property of ARP, where by safety we mean that no "bad things" happen during any protocol execution [18]. As far as we know, this is the first time that a formal method is successfully applied to the analysis of ARP. The work has been conducted by using the Model Checker Modulo Theories (MCMT) tool [15], which is a fully declarative and deductive symbolic model checker for safety properties of infinite state systems.

The ARP protocol plays a very critical role in the transmission phase of Internet messages as it converts the network (IP) address of a host into its

© Springer International Publishing AG 2017
N. Polikarpova and S. Schneider (Eds.): IFM 2017, LNCS 10510, pp. 391–406, 2017.
DOI: 10.1007/978-3-319-66845-1_26

corresponding hardware (MAC or Ethernet) address, which is the address we need to specify for communicating directly with a host. We briefly recall that IP addresses identify hosts in Internet and they are used "only" to route messages across the Internet. By contrast, MAC addresses identify hosts inside a Local Area Network (LAN) where they are physically connected. When a message has to be delivered to a host h, both its network and hardware addresses have to be known. Contrarily to network addresses which are usually publicly available (in particular in their symbolic form www.yyy.zzz), hardware addresses are not. Thus, ARP has been introduced for translating network addresses into hardware addresses. The protocol has been initially defined by Request for Comment (RFC) 826 [19], and subsequently redefined by RFC 3927 [11] and RFC 5227 [10], which have tried to settle some problems arising in the original formulation.

As many protocols of the TCP/IP stack, during the last twenty years ARP has been subverted in order to perform various forms of computer attacks [4,6,20]. The most prominent attack performed via ARP is the *Man-in-the-Middle* attack (MitM), in which an attacker can impersonate a victim's host and intercept/modify all the traffic directed to the victim's host. ARP hosts can also be victim of a *Denial-of-Service* attack (DoS). In this case, a malicious host m can continuously induce a victim host v to dismiss its current network address and to select a new one. While v does not own a stable address, it is not able to communicate.

In this paper, by using Satisfiability Modulo Theories (SMT), we will formally prove that the ARP protocol – as specified by the RFC documents – lacks safety properties, more precisely there exist protocol executions in which a MitM attack can be successfully perpetrated against some host. The same turns out to be true for a DoS attack.

2 Preliminaries on Formal Verification

The considered family of protocols 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 infinite.

Various techniques have been introduced in the literature to handle safety verification for such parameterized systems (see [1–3,5,8,9], to name but a few entries). We chose the declarative approach of the *array-based systems* [12,14,16], because it offers a great flexibility and relies (at deductive engine level) on the mature technology offered by state-of-the-art SMT-solvers, which is gaining relevance. In array-based systems (see [13,15] for tool implementations), the state is represented by both global variables, and by array variables such that each array corresponds to a component of the state of the hosts, and the k-th element of an array a contains the value of component a for the host k. This representation is very natural, and eases the modeling process. A system is specified via a pair of formulæ $\iota(\underline{p})$ and $\tau(\underline{p}, \underline{p}')$, and a safety problem via a further formula $\upsilon(\underline{p})$, where

$\underline{p}$ is the set of parameters and array-ids, $\iota(\underline{p})$ is the state 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 $\underline{p}'$, and $\upsilon(\underline{p})$ is 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 at the same instant; 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 any element of *Bad* computes the pre-image, i.e. the set of states which can lead to *Bad*. For any 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 with respect to the property described by *Bad*. 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 [15]. MCMT is a fully declarative and deductive symbolic model checker for safety properties of infinite state systems whose state variables include arrays. Sets of states and transitions of a system are described by quantified first-order formulae *of special kinds*. The tool exploits decision procedures (as implemented in state of the art SMT solvers) to cope with satisfiability problems involving various datatypes like 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). Besides standard SMT techniques, efficient heuristics for quantifier instantiation, specifically tailored to model checking, are 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 [16]). MCMT guarantees the safety of a protocol for any number N of system components.

The process of converting an algorithm into a MCMT model is performed manually: it requires deep comprehension of the algorithm, which must be broken down into its fundamental mechanisms and all possible cases, that are then translated into model transitions.

3 Address Resolution Protocol (ARP)

The main task of ARP is to enable a host h of a local network to discover, given the (32-bits) IP address of a host k (usually a well known data), the corresponding (48-bits) MAC address associated to k.[1] For efficiency reasons any host h maintains in a private data structure, known as ARP cache, all mappings $\langle$MAC, IP$\rangle$ it has so far discovered. Whenever h has to get in touch with host k, it will first look for k's MAC address in its own ARP cache. In case of failure it will initiate the ARP protocol, and it will proceed in the following way: h sends to all hosts in the LAN an *ARP request message*, asking for the MAC address of the owner of the address IP_k. Once k receives such a message it sends an *ARP reply* message, unicast to h, providing its own MAC address MAC_k. Once h receives the ARP reply it updates its own cache with the entry $\langle \mathrm{MAC}_k, \mathrm{IP}_k \rangle$. Similarly, k updates its own cache with the mapping $\langle \mathrm{MAC}_h, \mathrm{IP}_h \rangle$ provided by the ARP request from h. The same action is performed by other hosts already knowing h, so as to maintain their information updated. These operations are more precisely described in the following Algorithm 1. RFC 826 requires that ARP messages have a predefined format. The Ethernet header includes, among others, both the source and destination MAC address, `eth_src` and `eth_dest` respectively. The ARP message payload includes among others: the `opcode` identifying whether the message is a Request or a Reply, the source hardware (`sha`) and network (`spa`) addresses, and the target hardware (`tha`) and network (`tpa`) addresses, where *target* is the host destination of the ARP message.

3.1 ARP Formal Verification

In our verification, we assume that either (*i*) all hosts are honest, or (*ii*) one malicious host p_m exists, trying to perform a MitM attack against a victim p_v. Case (*ii*) is able to capture all the behaviors possible in real LANs. Indeed, real attackers focus on a specific victim, usually chosen after a preliminary analysis of the target LAN aiming at individuating the most vulnerable device in it. On the other hand, in case safety against MitM should be proved, any number of both attackers and victims should be checked. By contrast, we want to verify *un*safety; hence, finding counterexamples with just one attacker is sufficient.

Honest hosts send Requests when they need to know the identity of a message destination; they manage ARP messages according to Algorithm 1. p_m may send either Requests or Replies at any time, containing fake information; it may also send unicast Requests to a specific host, not processed by other hosts. According to RFC 826 [19], we do not model cache entry expiration: at any time a host may generate a request even if the target information is already in its cache, as if its cache has expired in the past. We model the processing of one ARP message at a time. We take both the MAC address and the IP address of a host p_x to be

[1] We briefly recall that the MAC address of any device is hardwired into the device by its manufacturer, and is not publicly available.

Algorithm 1. Classical ARP (RFC 826 [19])

```
1:  RequestGeneration()
2:  when MAC address for some target IP needed do
3:     new ARP_pkt: ARP_pkt.opcode ← Request; ARP_pkt.spa ← myIP;
4:     ARP_pkt.sha ← myMAC; ARP_pkt.tpa ← targetIP; ARP_pkt.tha ← ⊥;
5:     broadcast ARP_pkt;
6:  end do
7:
8:  PacketReception()
9:  when ARP_pkt received do
10:    Merge_flag ← false;
11:    if ARP_pkt.spa ≠ 0.0.0.0 ∧ ARP_pkt.spa ϵ ARP_cache then
12:       corresponding ARP_cache.sha ← ARP_pkt.sha;
13:       Merge_flag ← true;
14:    end if
15:    if ARP_pkt.tpa = myIP then
16:       if ARP_pkt.spa ≠ 0.0.0.0 ∧ not Merge_flag then
17:          ARP_cache ← ARP_cache ∪ ⟨ ARP_pkt.spa, ARP_pkt.sha ⟩;
18:       end if
19:       if ARP_pkt.opcode = Request then
20:          new ARP_pkt': ARP_pkt'.opcode ← Reply; ARP_pkt'.spa ← myIP; ARP_pkt'.sha ←
            myMAC;
21:          ARP_pkt'.tpa ← ARP_pkt.spa; ARP_pkt'.tha ← ARP_pkt.sha;
22:          send ARP_pkt to ARP_pkt.tha;
23:       end if
24:    end if
25: end do
```

equal to x. For the sake of space, in this section we just discuss the modeling of the unsafe case; the safe model is equal to the unsafe one without the transitions describing the p_m's behavior.[2] In the following, let N be the number of hosts.

In our models, the following global variables are used: φ indicates the current step of the computation, I counts the number of processes having processed the message in the current step; sh, sp and tp correspond to the `sha`, `spa` and `tpa` message fields respectively. The state of each process p_x is represented by the following array variables: $sm[x]$ indicates whether p_x must send a message; $cu[x]$ indicates whether p_x has processed the received message and possibly has updated its own cache. Both $sm[x]$ and $cu[x]$ are boolean variables. A MitM attack succeeds when in the ARP cache of some host $h \neq p_v$ the entry corresponding to p_v does not contain p_v's MAC address; such a situation is modeled by introducing the variables $CM[x]$ and $CP[x]$ which contain respectively the MAC address and IP address of p_v as contained in p_x ARP cache. For the sake of conciseness, in the transitions below we do not display the variables whose value stays unchanged.

The initial state satisfies:

$$\iota_1 := \varphi = 0 \land I = 0 \land sh = 0 \land sp = 0 \land tp = 0 \land \\ (\forall x.\ sm[x] = 0 \land cu[x] = 0 \land CM[x] = 0 \land CP[x] = 0) \quad (1)$$

that is, no message is around, no process has executed the current step, all caches do not contain any information about p_v, and no process has a message to send.

[2] Both source codes and results of all the models described in this work are available at http://homes.di.unimi.it/~pagae/ARPmodel/index.html.

The unsafe state capturing MitM attacks is described by the following formula:

$$\upsilon_M := \exists z.\ CM[z] = m \wedge CP[z] = v \tag{2}$$

that is, a process z exists whose cache entry for p_v was poisoned with the value of p_m.

The first three transitions model the `RequestGeneration()` procedure in Algorithm 1: we non-deterministically choose both the sender and the target of the new message. This is written as:

$$\tau_1 := \varphi = 0 \wedge \exists x, y.\ x \neq y \wedge \varphi' = 1 \wedge I' = 1 \wedge sm'[x] = 1 \wedge cu'[x] = 1 \wedge tp' = y$$

The sender parameters in the message are set in the next two transitions; in the former the host behaves honestly, in the latter the sender is p_m and generates a poisoned Request:

$$\tau_2 := \varphi = 1 \wedge \exists x.\ sm[x] = 1 \wedge \varphi' = 2 \wedge sp' = x \wedge sh' = x$$
$$\tau_3 := \varphi = 1 \wedge \exists x.\ sm[x] = 1 \wedge x = m \wedge \varphi' = 2 \wedge sp' = v \wedge sh' = m$$

If the source IP is different from that of p_v, a transition allows all processes to fire – one at a time – without changes to the cache entry concerning the victim:

$$\tau_4 := \varphi = 2 \wedge sp \neq v \wedge I < N \wedge \exists x.\ cu[x] = 0 \wedge \varphi' = 2 \wedge I' = I + 1 \wedge cu'[x] = 1$$

The same actions are performed (τ_6) when $sp = v$ but the host is not the target ($x \neq tp$) and it has nothing in its cache about p_v ($CP[x] = 0$). Otherwise, two cases must be considered. First, the receiving process is not the target but it has information about p_v in its cache, so it updates the cache entry:

$$\begin{aligned}\tau_5 := \varphi = 2 \wedge sp = v \wedge I < N \wedge \exists x.\ cu[x] = 0 \wedge CP[x] > 0 \wedge x \neq tp \wedge \\ \varphi' = 2 \wedge I' = I + 1 \wedge cu'[x] = 1 \wedge CP'[x] = sp \wedge CM'[x] = sh\end{aligned}$$

Transitions τ_4-τ_6 model lines 10–14 of Algorithm 1. By contrast, if the host is the target (lines 15–18 of Algorithm 1), it must also generate a Reply, which is recorded by appropriately setting its $sm[x]$:

$$\begin{aligned}\tau_7 := \varphi = 2 \wedge sp = v \wedge I < N \wedge \exists x.\ cu[x] = 0 \wedge x = tp \wedge \varphi' = 2 \wedge \\ I' = I + 1 \wedge sm'[x] = 1 \wedge cu'[x] = 1 \wedge CP'[x] = sp \wedge CM'[x] = sh\end{aligned}$$

When all hosts processed the Request, the Reply is sent (lines 19–23 of Algorithm 1). Two transitions describe this event: either the target generates a honest Reply (τ_8) or, if the target is p_m, it may generate a poisoned Reply. We report here just the latter; the former can be easily derived:

$$\begin{aligned}\tau_9 := \varphi = 2 \wedge I \geq N \wedge \exists x.\ cu[x] = 1 \wedge sm[x] = 1 \wedge x = m \wedge \\ \varphi' = 3 \wedge I' = 0 \wedge tp' = sp \wedge sp' = v \wedge sh' = m\end{aligned}$$

Table 1. Results for the formal verification of ARP (RFC 826)

	MitM					
	Outcome	Time (s)	Max. depth	# nodes	SMT calls	# literals
No malicious	Safe	0.222	2	3	249	7
Broadcast p_m	Unsafe	0.211	5	12	395	10
Unicast p_m	Unsafe	0.150	5	12	457	10

According to [19], Replies are sent unicast (line 22 of Algorithm 1); hence, the message is processed just by the target (lines 15–18 of Algorithm 1), and afterwards a re-initialization – leaving caches unchanged – is performed before repeating all over again:

$$\tau_{10} := \varphi = 3 \wedge \exists x.\ x = tp \wedge \varphi' = 4 \wedge CP'[x] = sp \wedge CM'[x] = sh$$
$$\tau_{11} := \varphi = 4 \wedge \varphi' = 0 \wedge I' = 0 \wedge tp' = 0 \wedge sp' = 0 \wedge sh' = 0 \wedge (\forall x.\ sm'[x] = 0 \wedge cu'[x] = 0)$$

Verification results. Table 1 shows the results obtained by running the described models on an Intel Core i7 running Linux Ubuntu 14.04 64 bits. We report the running time, the depth of the status tree, the number of tree nodes explored, the number of calls to the SMT solver, and the maximum number of literals in the constraint describing a node.

4 Link-Local Addresses

RFC 3927 [11] adds new functionalities to ARP for enabling the protocol to work in local networks where hosts may automatically configure their own network address interface, without human intervention. Address configuration is performed by randomly choosing an IP address in the range 169.254.1.0–169.254.254.255 and then verifying that the chosen address is not already in use by some other host.

Algorithm 2 describes RFC 3927. *All messages* – both Requests *and* Replies – *are broadcast*. A host h wishing to adopt a certain IP address ip *probes* it by broadcasting a Request with `spa` = 0.0.0.0 – which is an invalid address so as to avoid polluting caches if ip is already in use by another host – and `tpa` = ip. If h receives an ARP message with either `spa` = ip, or null `spa` and `tpa` = ip, it deduces that another host is using or probing ip and selects a different address. Otherwise, h *announces* that it will use ip by broadcasting a Request with both `spa` and `tpa` equal to ip, so as to overwrite previous ARP cache entries related to ip. From now on, for any received packet, h compares ip against the `spa` contained in the packet; if the two are equals, the address conflict detection (ACD) procedure is executed.[3] According to ACD, a host may try to defend its

[3] It is worth to notice that the lack of this check allowed the MitM attack in RFC 826 against the victim itself.

Algorithm 2. Dynamic configuration of Link-Local addresses (RFC 3927 [11])

```
1: Select()
2: when network interface becomes active do
3:    myIP ← rand(seed(MAC, previous IP), 169.254.1.0, 169.254.254.255); Probing();
4: end do
5:
6: Probing()
7: new ARP_pkt: ARP_pkt.opcode ← Request; ARP_pkt.spa ← 0.0.0.0;
8: ARP_pkt.sha ← myMAC; ARP_pkt.tpa ← myIP; ARP_pkt.tha ← 0;
9: timer ← rand(0, PROBE_WAIT); count ← 0;
10: repeat
11:    when timeout do
12:       broadcast ARP_pkt; count++;
13:       if count < PROBE_NUM then
14:          timer ← rand(PROBE_MIN, PROBE_MAX);
15:       end if
16:    end do
17: until count < PROBE_NUM;
18: timer ← ANNOUNCE_WAIT;
19: when (ARP_pkt received s.t. (ARP_pkt.spa = myIP) ∨ (ARP_pkt.opcode = Request ∧
    ARP_pkt.spa = 0.0.0.0 ∧ ARP_pkt.tpa = myIP ∧ ARP_pkt.sha ≠ myMAC) do
20:    give myIP up; LimitConflicts(); //FAILURE!
21: end do
22: when timeout do
23:    conflict_num ← 0; Announce(ANNOUNCE_NUM); //SUCCESS!
24: end do
25:
26: Announce(limit)
27: count ← 0;
28: new ARP_pkt: ARP_pkt.opcode ← Request; ARP_pkt.spa ← myIP;
29: ARP_pkt.sha ← myMAC; ARP_pkt.tpa ← myIP; ARP_pkt.tha ← 0;
30: repeat
31:    broadcast ARP_pkt; count++; wait(ANNOUNCE_INTERVAL);
32: until count < limit;
33: ConflictDetection();
34:
35: ConflictDetection()
36: while true do
37:    when ARP_pkt received do
38:       if ARP_pkt.spa = myIP ∧ ARP_pkt.sha ≠ myMAC then
39:          ACD(); //CONFLICT!
40:       else
41:          ARP.PacketReception(ARP_pkt); //processing according to RFC 826
42:       end if
43:    end do
44: end while
45:
46: LimitConflicts()
47: conflict_num ++;
48: if conflict_num ≥ MAX_CONFLICTS then
49:    timer ← RATE_LIMIT_INTERVAL;
50: else
51:    timer ← 0;
52: end if
53: when timeout do
54:    Select();
55: end do
56:
57: ACD()
58: if want to defend ∧ current_time - start_defend > DEFEND_INTERVAL then
59:    start_defend ← current_time; Announce(1);
60: else
61:    give myIP up; start_defend ← 0; LimitConflicts();
62: end if
```

address at most once by sending a new Announce. If another conflict is detected, the host dismisses its own network address and selects a new one. In case of no conflict, the original ARP (Algorithm 1) is executed.

4.1 Verification of ARP as in RFC 3927

In order to analyze this protocol, three models have been developed:

M1: Probe and Announcement messages have been added to the ARP model, but *not* the address conflict detection mechanism
M2: the ACD mechanism has been modeled, with address give up in case of a detected conflict
M3: the ACD mechanism has been modeled, by introducing the defense procedure above mentioned in case a conflict is detected. When a second conflict is detected, the host – who already defended – dismisses the used address.

For all the three models the safety with respect to MitM attacks has been analyzed; for M2 and M3 we also investigated the safety property with respect to DoS attacks. No cache expiration is considered.

For the sake of space, we describe here just the more complex model, i.e. M3, and we focus on the new features introduced with respect to the ARP model as described in Sect. 3.1. This new model includes an additional global variable GA whose value indicates the type of message considered: Probe (1), Announce (2), Request (3), or unsolicited Reply (4) – not corresponding to any Request – from p_m. Additional local variables are: $st[x]$ which indicates the state of a host, that is, if it has to send the Probe (0), or the Announce (1), or its IP address is configured and it may send Requests (2). p_m may send any message independently of its own state. The variable $cd[x]$ indicates whether this is the first time that the host has detected a conflict and must thus defend. The variable $gu[x]$ indicates how many times a host gives up its current address. The new initial state is defined as:

$$\iota_2 := \iota_1 \wedge GA = 0 \wedge (\forall x.\ gu[x] = 0 \wedge cd[x] = 0 \wedge st[x] \geq 0 \wedge st[x] \leq 2)$$

where ι_1 is defined in Eq. (1). This formula provides the maximum generality as it does not force any initial state to the network hosts. The unsafe state for MitM, υ_M, is defined as in Eq. (2).

DoS attacks can be modeled by an host that dismisses its address an indefinite number of times. Yet, this is actually a *liveness* property that cannot be verified with the adopted technique. Hence, we shall re-write it as a weaker *safety* property, whose negation is:

$$\upsilon_D := \exists z.\ gu[z] \geq threshold \tag{3}$$

for some finite value of *threshold*. This is weaker than a DoS attack, as it says that a host dismisses its address a finite number of times. We discuss this aspect in more detail at the end of this section, when analyzing the verification results.

A description of the model now follows. In the first six transitions, we describe the event to be reproduced, amongst either generation of Probe, Announce or Request issued by a host,[4] or generation of an Announce, Request or unsolicited Reply from p_m. For the sake of space, we report here just the more complex case, that is, the generation of a Request:

$$\begin{aligned}\tau_3 := \varphi = 0 \wedge \exists x.\ st[x] = 2 \wedge \varphi' = 1 \wedge I' = 1 \wedge GA' = 3 \wedge sm'[x] = 1 \wedge \\ cu'[x] = 1 \wedge sh' = x \wedge sp' = x \\ \tau_7 := \varphi = 1 \wedge \exists x, y.\ x \neq y \wedge sm[x] = 1 \wedge \varphi' = 2 \wedge I' = 1 \wedge sm'[x] = 0 \wedge \\ cu'[x] = 1 \wedge tp' = y \wedge sp' = x \wedge sh' = x\end{aligned}$$

The former transition selects the source while the latter selects the target. All other cases are modeled in one step, as just the source identifier must be indicated in the message, and lead to transitions guarded by $\varphi = 2$. Similarly for p_m's messages, where always $sp' = v \wedge sh' = m$.

Subsequently, there are eight transitions modeling the processing of the message generated by one of the first six transitions. The following cases are modeled as in the case of ARP (Sect. 3.1): (τ_8) Request processing when $sp \neq v$; (τ_9) $sp = v$ and the host is not the target but can update the cache; (τ_{10}) $sp = v$ and the host is not the target and cannot update the cache; (τ_{11}) $sp = v$ and the host is the target (but not the victim) that generates a Reply. Other four cases involve the victim in case the message is poisoned: p_v is the target of the message and detects the conflict; if this is the first conflict then it defends its address (τ_{13}), otherwise it discards the address (τ_{12}). Or, p_v detects the conflict but it is not the target. We analyze in more detail these latter cases, as they are more complex since two messages have to be modeled: both the target Reply and the victim defense.

The two messages cause different cache updates: if the target is different from p_v, its reply does not change the cache entries concerning the victim. Hence, they can be processed in whatever order, and we decided to model the processing of the Reply first. In case p_v renounces, the following transition applies:

$$\begin{aligned}\tau_{15} := \varphi = 2 \wedge I < N \wedge sp = v \wedge \exists x.\ cu[x] = 0 \wedge x \neq tp \wedge x = v \wedge cd[x] > 0 \\ \wedge I' = I + 1 \wedge sm'[x] = 3 \wedge cu'[x] = 1 \wedge gu'[x] = gu[x] + 1 \wedge cd'[x] = 0\end{aligned}$$

The value of $sm[x]$ is not changed afterwards and allows to remember – once all hosts have processed the Reply – that the victim has changed its address; $cd'[x]$ is reset because the victim is dismissing its current address, and it has not observed any conflict on the new address it is going to adopt. By contrast, in case p_v defends ($cd[x] = 0$), the transition τ_{14} is applied; such a transition is conceptually equal to τ_{15} apart for the assignments $sm'[x] = 5$ and $cd'[x] = 1$.

As for ARP, once all hosts processed the Request, the Reply is sent in broadcast, and consequently processed by all hosts. Four transitions (τ_{23}-τ_{26}) replicate for the Reply the same cases as for the Request modeled by transitions τ_8-τ_{11}

[4] Also p_m, who may nondeterministically behave honestly.

above described. Transitions τ_{27}-τ_{28} describe the cases in which p_v observes a poisoned Reply – generated by the malicious – and it either renounces or defends.

In the case p_v is the target of the Request and defends, no Reply is generated and the system goes to the defense modeling (fired by $sm'[x] = 4$). The defense is modeled by the following transitions:

$$\begin{aligned}\tau_{18} := {} & \varphi = 2 \wedge I \geq N \wedge \exists x.\ cu[x] = 1 \wedge sm[x] = 4 \wedge \varphi' = 4 \wedge I' = 1 \wedge sm'[x] = 0 \\ & \wedge cu'[x] = 1 \wedge tp' = v \wedge sp' = v \wedge sh' = v \wedge (\forall y. y \neq x \wedge cu'[y] = 0) \\ \tau_{19} := {} & \varphi = 4 \wedge I < N \wedge \exists x.\ cu[x] = 0 \wedge CP[x] > 0 \wedge I' = I + 1 \wedge \\ & cu'[x] = 1 \wedge CM'[x] = sh \wedge CP'[x] = sp\end{aligned}$$

The former describes the generation of the Announce after all hosts processed the Request. The latter describes the processing of the Announce on behalf of the receiving hosts having information for p_v. A transition τ_{20} describes the case of a host receiving an Announce but not having a cache entry for p_v, and thus skipping any processing. When all hosts processed the Announce, the system can restart:

$$\begin{aligned}\tau_{31} := {} & \varphi = 4 \wedge I \geq N \wedge \exists x.\ cu[x] = 1 \wedge (\forall y. sm[y] = 0) \wedge \varphi' = 0 \wedge I' = 0 \wedge \\ & GA' = 0 \wedge sm'[x] = 0 \wedge cu'[x] = 0 \wedge tp' = 0 \wedge sp' = 0 \wedge sh' = 0\end{aligned}$$

The untouched variables are the cache, the state, the record of giveups and defenses occurred so far. A similar re-initialization is performed every time nothing harmful occurred. Transitions similar to τ_{18}-τ_{20} above apply when the target is different from p_v and all hosts already processed the Reply, and are fired by guards containing $sm[x] = 5$.

By contrast, if p_v renounces, this is modeled by a transition like this (triggered after all hosts processed a possible Reply):

$$\begin{aligned}\tau_{29} := {} & \varphi = 3 \wedge I \geq N \wedge \exists x.\ cu[x] = 1 \wedge sm[x] = 3 \wedge \varphi' = 0 \wedge I' = 0 \wedge \\ & GA' = 0 \wedge sm'[x] = 0 \wedge cu'[x] = 0 \wedge sp' = 0 \wedge sh' = 0 \wedge tp' = 0 \wedge \\ & (\forall y.\ CM'[y] = 0 \wedge CP'[y] = 0)\end{aligned}$$

which records that no host has information concerning the new address p_v is going to adopt.

Verification results. The first three lines in Table 2 show the outcome (Safe or Unsafe) obtained by running the described models for RFC 3927, and the running time; for DoS attack, a *threshold* = 5 was used. If no malicious host exists, the protocol is safe with respect to the MitM attack. By contrast, one malicious host sending either broadcast or unicast messages is able to pollute other processes caches. The three models reveal the impact of the different mechanisms adopted for ACD.

In M1, no ACD mechanism is implemented, that is, hosts do not check the `spa` field in incoming ARP messages. Hosts never dismiss their address and the

Table 2. Results of the verification of RFC 3927 (M1, M2, M3) and RFC 5227 (M4)

	MitM			DoS		
	no p_m	bcast p_m	ucast p_m	no p_m	bcast p_m	ucast p_m
M1	[S] 0.392 s	[U] 0.260 s	[U] 0.379 s	–	–	–
M2	[S] 0.268 s	[U] 0.439 s	[U] 0.311 s	[S] 0.330 s	[U] 44.85 s	[U] 104.93 s
M3	[S] 0.380 s	[U] 0.417 s	[U] 0.409 s	[S] 0.419 s	[U] 843.2 s	[U] 1716.8 s
M4	[S] 0.306 s	[U] 0.401 s	[U] 0.415 s	–	–	–

DoS attack cannot occur. By contrast, MitM may happen, and a counterexample provided by the prover – with p_m sending broadcast messages – consists in the following sequence of events: p_m generates an Announce with $sp = v$, $sh = m$ and $tp = v$. Any host h receiving it (*the victim included*) updates its cache with $CM[h] \leftarrow m$ and $CP[h] \leftarrow v$. When p_m sends unicast messages, the MitM attack is achieved with p_m sending a poisoned Request to a target that records the fake information in its own cache.

In M2, the protocol is proved unsafe with respect to MitM, and the following counterexample is supplied by the prover for p_m sending broadcast messages: p_m generates a poisoned Request to a random target $h \neq p_v$, containing $sp = v$ and $sh = m$; the target records in its cache $CM[h] \leftarrow m$ and $CP[h] \leftarrow v$. It is worth notice that the models do *not* capture the temporal duration of the attack, that is, the unsafe outcome of the model for MitM in M2 lasts for the time needed by p_v to configure a new IP address. Afterwards, entries still existing in some caches and coupling the MAC of p_m with the dismissed IP of p_v are refreshed as soon as p_v sends its own first Announce with its new address, and starts using it. By contrast, with ARP [19] the attack may last indefinitely. Similarly, if p_m sends unicast messages, the unsafe sequence of events is the same as for M1 above, the victim is unaware of the problem and never raises a conflict detection event.

In M2, the verification of the possibility of a DoS attack has been conducted with different values of *threshold* (see Eq. (3)). With threshold 20 and no constraints on the number of hosts, a sequence of 78 events is produced as counterexample, which clearly shows loops (Fig. 1): p_m sends an unsolicited (broadcast) poisoned Reply to a random target; p_v receives it and gives up. Afterwards, p_v takes another address but p_m sends a poisoned Announce with $tp = v$, $sp = v$, $sh = m$; p_v (which owns the target IP address) detects the conflicts and gives up again. The existence of loops shows that a sequence of events exists such that the host may indefinitely dismiss its address, thus implying that the DoS attack holds. With $threshold = 20$, running times were of 434.9 s. and 1171.6 s. for the broadcast and unicast case respectively.

In M3, MitM may arise and the counterexample supplied by the prover, when p_m generates broadcast messages, is the following: p_m generates a poisoned base Request to a random target h, with $sh = m$ and $sp = v$; the target receives such a Request and sets in its cache $CM[h] \leftarrow m$ and $CP[h] \leftarrow v$. The cache poisoning lasts until the victim defends by announcing its new IP address.

Fig. 1. M2 - Event loop in the verification of DoS attack for RFC 3927.

The unsafe sequence of events for p_m sending unicast messages is the same as for M1 and M2 above, and the same considerations apply. The verification of the DoS possibility is harder than before, due to the fact that a renounce may occur just *after* a previous conflict detection which the host coped with by defending. Hence, the sequences of events are longer and the prover has to explore a larger tree of possible sequences of events with more nodes of higher depth. Yet, with no constraints on the number of processes, we were able to achieve an unsafe outcome for v_D with $threshold = 5$; the prover supplied a sequence of events of length 49 involving a loop (Fig. 2). As for M2, the loop implicitly shows that the DoS attack may verify.

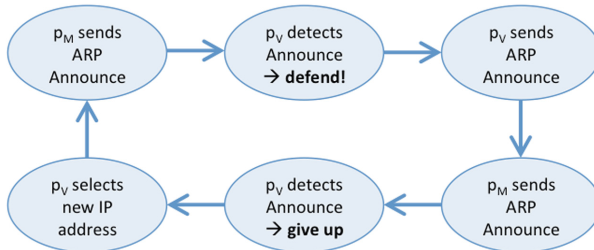

Fig. 2. M3 - Event loop in the verification of DoS attack for RFC 3927.

The highest computation complexity is reached by the M3 model verifying DoS, with p_m generating unicast messages: the maximum depth of the status tree is 49, the number of explored tree nodes is 8180, and 1355491 calls to the SMT solver are performed; the longest formula involves 26 literals. For the sake of space, all the results are reported in our website for interested readers.

5 Extended ARP: Address Conflict Detection

In order to deal with misconfigurations, RFC 826 [19] was further updated by RFC 5227 [10] which, with respect to RFC 3927, introduced a more aggressive Address Conflict Detection (ACD) mechanism. For the sake of brevity, in this section we just describe the differences with the latter, taking as reference Algorithm 2. According to RFC 5227, in the `Select()` procedure an address is assigned to a host in one out of three ways, namely: a static address is configured by a network administrator, or a dynamic address is supplied by either a

DHCP server or the Link-Local mechanism. Then, the `Probing()` procedure is run every time an interface is configured or booted. In RFC 5227, in case of conflict detection a host may (*i*) cease to use its IP address, or (*ii*) defend its address *once*, or (*iii*) defend its address *indefinitely*. In cases (i)-(ii), the behavior is exactly as in RFC 3927. Case (iii) is adopted e.g. when the host is a server needing to maintain its well-known stable address, and is not included in RFC 3927. In this case, the `Announce` procedure is called with $limit = 1$, and then the host continues using its IP address ignoring further conflicts. The `ACD()` procedure is modified accordingly, while the other procedures are equal in the two standards.

As a consequence of the above, cases (i)−(ii) are modeled by the M2 and M3 described in Sect. 4.1, while case (iii) is modeled by an additional model **M4** obtained from M3 by modifying the transitions where the victim dismisses its IP address so that the victim just fires without performing further actions.

Verification results. The last line of Table 2 reports the results obtained by running M4. The DoS attack cannot occur as hosts never dismiss their IP addresses. The sequence of events describing a MitM attack is the same whether p_m sends broadcast or unicast messages: p_m sends a poisoned Request to a target that pollutes its cache by recording the fake information. In the former case, the victim detects the conflict the first time and sends an Announce. Afterwards, it ignores the conflicts, and the malicious host may continue sending poisoned messages while the victim does not take any action.

6 Conclusions

In this paper, the modeling and formal verification of the three standard protocols for address resolution in Internet is described. The relevance of our work lies in two main achievements: first, under a practical point of view, our experiments formally show the weaknesses of currently adopted technologies with respect to security aspects, thus providing formal foundation to well known phenomena discovered and exploited by the underground community since many years. Second, the work highlights the maturity of existing formal approaches and tools in verifying the safety and correctness of real distributed systems. These approaches have been so far validated with several problems, included other network protocols (e.g. [7,17]). Yet, to the best of our knowledge, this is the first time that these techniques are applied to the analysis of ARP, with excellent results: the verification was possible for all deployed models, for any number N of system components, and within acceptable computation time.

In the future, we plan to apply these techniques to the verification of algorithms proposed in the literature but not yet standardized aiming at securing ARP – thus contributing to the development of safer networks – as well as possibly to other Internet protocols.

References

1. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71209-1_56
2. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35873-9_28
3. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004). doi:10.1007/978-3-540-28644-8_3
4. Alqahtani, A.H., Iftikhar, M.: TCP/IP attacks, defenses and security tools. Int. J. Sci. Mod. Eng. (IJISME) **1**(10) (2013)
5. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005). doi:10.1007/11562948_35
6. Bellovin, S.M.: Security problems in the TCP/IP protocol suite. ACM SIGCOMM Comput. Commun. Rev. **19**(2), 32–48 (1989)
7. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM **49**(4), 538–576 (2002)
8. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, San Rafael (2015)
9. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27813-9_29
10. Cheshire, S.: IPv4 Address Conflict Detection. RFC 5227, July 2008
11. Cheshire, S., Aboba, B., Guttman, E.: Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927, May 2005
12. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Proceedings of FMCAD (2013)
13. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31424-7_55
14. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 67–82. Springer, Heidelberg (2008). doi:10.1007/978-3-540-71070-7_6
15. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14203-1_3
16. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. J. Log. Methods Comput. Sci. **6**(4) (2010)
17. Islam, S.M.S., Sqalli, M.S., Khan, S.: Modeling and formal verification of DHCP using SPIN. Int. J. Comput. Sci. Appl. **3**(6), 145–159 (2006)

18. Alford, M.W., Ansart, J.P., Hommel, G., Lamport, L., Liskov, B., Mullery, G.P., Schneider, F.B.: Formal foundation for specification and verification. In: Paul, M., et al. (eds.) Distributed Systems. LNCS, vol. 190, pp. 203–285. Springer, Heidelberg (1985). doi:10.1007/3-540-15216-4_15
19. Plummer, D.C.: An Ethernet Address Resolution Protocol - or - Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826, November 1982
20. Wagner, R.: Address Resolution Protocol Spoofing and Man-in-the-Middle Attacks. The SANS Institute, Reston (2001)