# Implementation of ALPHA: An Adaptive and Lightweight Protocol for Hop-by-hop Authentication [*]

Johannes Gilger
RWTH Aachen University
Aachen, Germany
johannes.gilger@rwth-aachen.de

Florian Weingarten
RWTH Aachen University
Aachen, Germany
florian.weingarten@rwth-aachen.de

Tobias Heer[†]
Distributed Systems Group,
RWTH Aachen University
Aachen, Germany
heer@cs.rwth-aachen.de

## ABSTRACT

This paper describes our work on the ALPHA protocol [3] for the duration of our UROP project. Based on the original ALPHA paper, which purely specifies the basic algorithmic details of the protocol, we developed a fully functional ALPHA implementation for IP networks. In this process, we significantly extended the functionality and behaviour of the ALPHA protocol. We begin with a short reminder of the ALPHA protocol before highlighting the key features that we developed beyond the original ideas. At the end we evaluate the maturity of the implementation by measuring its performance.

## Keywords

Mobile network, Authentication, Hop-by-Hop

## 1. INTRODUCTION

ALPHA is a protocol to provide hop-wise authentication of network packets on top of an already existing network. ALPHA can be applied in a range of multi-hop networks. One example are IP based networks. ALPHA uses lightweight cryptographically secure functions in the form of an interactive hash-chain based signature scheme [5] to provide authentication of subsequent packets. To be flexible for different traffic patterns it also defines three different transmission modes: ALPHA-N sends each packet immediately while ALPHA-C / ALPHA-M collect and send packets in bursts. While the former is suited for low-latency and low-throughput situations, the latter modes perform much better in high-throughput scenarios by making a trade-off in latency. The difference between them are different requirements in storage and computation.

As part of our UROP project, we implemented and consistently documented a working prototype of the ALPHA software, which consists of software used by the clients as well as a filter-program for the intermediate hop nodes called ALPHA-filter. We developed the ALPHA software from scratch.

## 2. FEATURES

In this section we describe the key features we implemented in the ALPHA software.

### 2.1 ALPHA modes

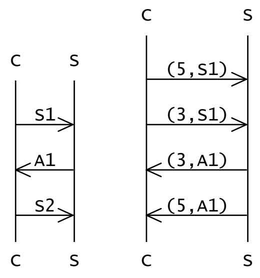
The three ALPHA-modes provide flexibility in the scope of use-scenarios for the ALPHA software. The ALPHA software is supposed to work for a wide range of devices and networks. Using the different modes separately or even in combination provides the user with the possibility to adapt the protocol to his needs.

The different modes for sending packets (ALPHA-M, ALPHA-N, ALPHA-C) were explained in detail in [3], so there was no ambiguity when implementing them. We wanted to work with the different modes easily, that is why we abstracted the methods called directly as much as possible. This way, the caller does not have to deal with the (implementation-specific) differences between the modes.

### 2.2 Associations

While the ALPHA-modes already provide some level of flexibility (low-latency with ALPHA-N vs. high-bandwidth with ALPHA-M, ALPHA-C) by themselves, it became apparent that only being able to send data on a single predetermined channel with a fixed ALPHA-mode per peer was a serious limitation. When using the burst-modes for high-volume traffic, small packets used for signaling were being sent with the rest of the data stream instead of being prioritized. We suspected this behaviour to result in decreased performance. Furthermore, the symmetry of this single connection meant that each side had to generate and store the same amount of hash-chain elements, even if the expected traffic flow was mostly uni-directional, thus depleting the signature chain on the sender more quickly and wasting precomputed hash-chain elements. And lastly, because of the inherent latency of ALPHA, the daemon always had to wait a full round-trip before actually sending the packet.

To address these issues we introduced *associations*, which form different communication channels between two peers, roughly speaking. These associations share common principles with the ones found in the IPsec protocol[4]: they have an ID and a type, and a lookup method is used to map an incoming packet to an actual association in the local memory. We decided to use uni-directional channels to adapt to typical asymmetrical bandwidth requirements. Since the actual network interface can only send one packet at a time we had to devise a scheduler which first distributed outgoing packets to the available outgoing associations, and then made sure that the resulting packet queues stored at these associations were processed in a fair manner. A small example of two ALPHA clients which send and receive packets on two different associations is given in Fig. 1. Here, the first element of the tuple identifies the association while the second entry signals the packet type.



(a) $\alpha$ (b) $\alpha$ Ass.

**Figure 1: Regular ALPHA and ALPHA with two associations**

[*]This research was supported by the Undergraduate Research Opportunities Programme (UROP Offer 182) of RWTH Aachen University and was completed in 02/2010.

[†]Coordinator for the UROP-project

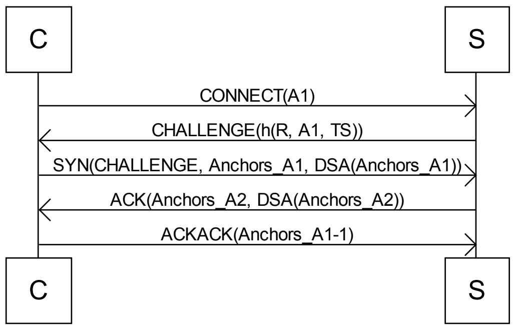

C
S
CONNECT(A1)
CHALLENGE(h(R, A1, TS))
SYN(CHALLENGE, Anchors_A1, DSA(Anchors_A1))
ACK(Anchors_A2, DSA(Anchors_A2))
ACKACK(Anchors_A1-1)
C
S

**Figure 2: Complete ALPHA handshake**

Since neither the mode nor the number of required associations is known to either side before connecting, we also needed a flexible mechanism not only for creating new associations, but also for signaling the end of existing association. We decided that the best way to achieve this was to designate a special Alpha association, which we called the *control association* (similar to the two phases in the IPsec protocol). The control association has the unique property of being bi-directional. It is used for signaling requests between the Alpha hosts when a hash-chain is depleted and a new association has to be established.

## 2.3 Authenticated anchors

The original paper mentioned the need for guaranteeing the authenticity of the initial hash chain anchors sent by both parties. To that end it suggested, amongst other methods, the use of public-key cryptographic systems. We chose to use the Digital Signature Algorithm (DSA) in order to sign and verify the initial anchors during connection. This allows us to provide a secure exchange of hash anchors, while fulfilling the requirement for Alpha-filters to be able to read and store the anchors during the exchange.

The next step was to design a simple handshake procedure for establishing a connection. We used the SYN, ACK, ACKACK terminology to denominate the first three packets sent. Denial of service (DoS) attacks based on exhausting the resources of a node were a realistic concern during the design. Storing data received with the first packet from an unknown peer meant that a sufficiently large number of forged connection attempts could result in the node running out of free memory or in a serious slowdown due to a large number of hash-chain computations.

To counter this attack, we preceded the handshake by a return-routability check that does not involve storing information on the server before the return-address of a client has been verified. This is the same concept as the cookies used during the IPsec Internet Key Exchange (IKE)[2]. When a client connects, the server responds with a challenge, which he generates using a secret random number, a timestamp and the client's claimed address. The client has to return this challenge with the SYN packet, which is also used to send the client's hash anchors to the server. The server can now verify that the client received the challenge and did not supply a false IP address. Only if the information is correct, a state for the client is created and the actual handshake continues. The server adds the client to its list of peers, stores the hash-chain anchors it received, generates its own hash-chains and replies using an ACK packet. The client replies with the final ACKACK-packet to signal the completion of the handshake. The packets carrying the anchors are signed (using DSA) by the peers, enabling the other side to verify that the data has not been tampered with during transmission and that the packet has not be received by an impersonator. A sequence-chart showing the packets sent during connection is given in Fig. 2.

## 2.4 Storage-strategies for hash-chains

One of the main objectives of Alpha was to stay flexible with regard to different classes of devices. While computing hash chains can be seen as the lowest common denominator those devices need to match, being able to store the complete hash-chains is a not a mandatory requirement. The authors did not mention any strategies to be able to trade storage space for computing time, but we came up with an intermediate storage format which would only store every $n$-th element of the hash-chain and computing intermediate entries as needed. We were able to verify the validity of our approach when we discovered [6].

## 2.5 ALPHA configuration

While developing Alpha we gradually enhanced the frontend for supplying options to the (running) Alpha daemon. We implemented support for a small and simple configuration file which not only holds system-wide parameters but also the settings for each individual peer known in advance. Most of the options can also be specified using commandline switches to the Alpha daemon, where they will override previous directives read from a configuration file. We differentiated between options that an end-user will be able to set and switches solely meant for the development of Alpha.

For controlling a local Alpha daemon and getting status information, a small program, called *alphacontrol*, was created. It communicates with the Alpha daemon using BSD sockets.

## 2.6 Platform compatibility

During development, we focused on the Linux operating system for both client and filter-software. Because of our own experience and the simple demand for performance, we chose the C programming language to implement both parts of the software. Another factor contributing to that choice was the necessity to easily manipulate data on a low abstraction level and to be able to directly use the libc in order to maintain platform compatibility.

Ensuring platform compatibility turned out to be one of the biggest problems we faced. Traditional VPN-solutions use a designated tunnel destination to send their packets to, so they can simply add an extra route for just that destination to be contacted directly, while the rest of the traffic has to pass through the VPN-software first. The software encapsulates the packets in a new packet to the tunnel destination, i.e. it will not pass through the software again. Alpha presents a different situation since the tunnel destination is the same host to be contacted using Alpha. We therefore had to devise a method for traffic to only pass through the Alpha daemon once. This is pictured in Fig. 3: An outbound packet P arrives at the routing table which sends it to the Alpha daemon. There it is encapsulated in an Alpha packet and sent to the routing table again, which will now send it directly to the network, i.e. the remote peer.

Linux enabled us to direct the traffic based on firewall-marks. Those marks are attached to packets by the netfilter[1] architecture and we simply marked all non-Alpha packets. The second step was to tell the routing algorithm that those marked packets should use a different routing ta-

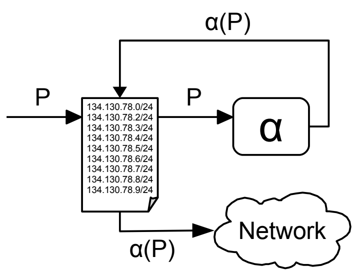


**Figure 3: ALPHA routing situation**

ble, one that lead through the tunnel interface controlled by the ALPHA daemon first. This method required minor or no modifications to existing systems.

Mac OS X, lacking the netfilter architecture, employs its own version of the UNIX ipfw firewall. After several failed attempts to set up a similar path, we put the OS X support on hold to focus on one architecture. We made sure that every version of ALPHA still compiled and ran on Mac OS X, including the daemon and the virtual network interface.

## 3. ORGANIZATIONAL CHALLENGES

ALPHA, as proposed in [3], is only described as a set of core ideas for the protocol, which are the correctness and evaluation of the different ALPHA modes. Not mentioned are more specific details when it comes to the data-structures and authentication-schemes to be employed and the behaviour to show in the face of failures and unexpected requests. The lack of a strict specification meant that we had to discuss new features and details with our UROP-coordinator frequently. It was also the opportunity for us to show him new improvements we had devised or point out problems not anticipated by the original paper.

## 4. EVALUATION

For evaluating the efficiency of the ALPHA-prototype we performed a range of measurements. We used a setup of dedicated machines to avoid any side effects by concurrent processes or network traffic. The setup consisted of two (resp. three) identical machines: AMD Athlon X2 4800+ processor, 3GB RAM, RTL8111 NIC, connected to an isolated GBit switch. The machines ran a 32bit Linux kernel (version 2.6.28). Throughput was measured with the *iperf* tool, using a simple uni-directional stream of bulk data from the client/sender to the server/receiver. Each single test was run seven times for a duration of 30s each and the results were averaged. Latency was introduced using traffic control (tc) of the Linux kernel. The ALPHA-daemon on the client was restarted after each test, the daemon on the server side ran continuously with 30 ALPHA-N associations to send data.

The same measurements were also performed with an intermediate node running the ALPHA-filter software and the results were similar, yet less smooth, which is why we chose the plots resulting from a direct connection here.

### 4.1 Plain throughput

The first thing we were interested in was the impact of ALPHA on the *maximum transmission rate* (bandwidth) with an increasing number of associations. Results of this test-run can be seen in Fig. 4. Discarding the initial step from one to two associations, it is obvious that without a delay, ALPHA does not benefit from using more than a few associations. Worse yet, the overhead for storing and accessing the different associations increases with a rising number of associations, reducing overall performance.

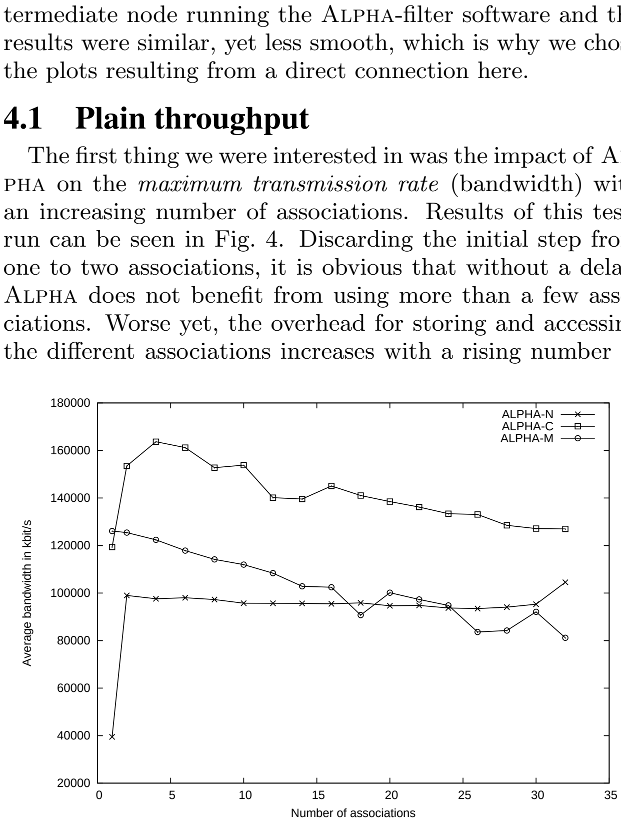


**Figure 4: Bandwidth without delay**

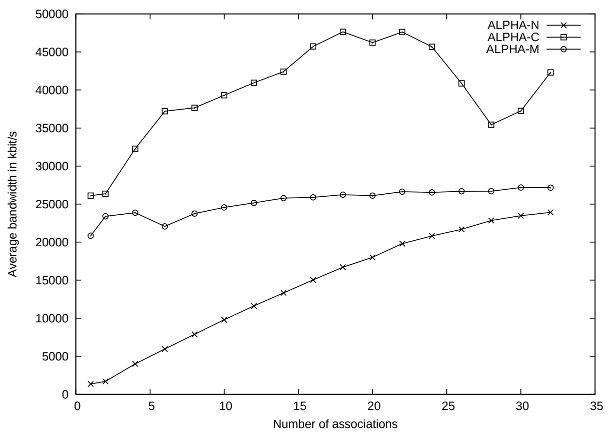


**Figure 5: Bandwidth with 10ms delay**

### 4.2 Impact of latency

We also wanted to show how the transmission rate behaved for different ALPHA-modes when we introduced an *artificial delay* (latency) between the nodes. The expected results were that the two bulk modes, ALPHA-C and ALPHA-M, would handle the delay better than ALPHA-N when compared to the respective test runs without delay. We also assumed that for the delayed test run, increasing the number of associations up to a certain number would have a positive effect on bandwidth, regardless of mode. The results of this run are given in Fig. 5. The impact of increasing numbers of associations can clearly be seen and behaves linearly for ALPHA-N. ALPHA-M bandwidth stays nearly constant, pointing to possible inefficiencies in the handling of ALPHA-M associations in the ALPHA-client.

## 5. CONCLUSION

While the measurements we performed point to some performance bottlenecks yet to address in the ALPHA prototype, they were a good indicator of the benefit of associations. Using multiple associations meant an improvement of bandwidth in the face of latency. We learned that the optimal number of associations depends on the network latency as well as the resources of the nodes running ALPHA. An intelligent scheduler together with different combinations of associations should improve the performance noticeably and would be the next logical step in the development of the ALPHA-prototype.

In the course of this project we learned how to succesfully implement, document, test and improve a stable prototype for the ALPHA-protocol, while at the same time working on the protocol itself.

## 6. REFERENCES

[1] http://www.netfilter.org/.

[2] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[3] T. Heer, S. Götz, O. Garcia Morchon, and K. Wehrle. Alpha: An adaptive and lightweight protocol for hop-by-hop authentication. In *Fourth Conference on emerging Networking EXperiments and Technologies, CoNEXT 2008*, Madrid, Spain, 2008. ACM.

[4] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), Dec. 2005.

[5] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):772, 1981.

[6] Y. Sella. On the computation-storage trade-offs of Hash chain traversal. *Lecture notes in computer science*, 50(2):270–285.