Time-Based Link Layer Authentication Simulation with Elixir / Erlang

Ian Johnson

1 The Time-Based Authentication Protocol

The simulator described in this report is used to collect data pertaining to the efficacy of the Time-Based Authentication (TBA) protocol. The TBA protocol is a link-layer authentication protocol which operates under the assumption that a user can send a message whenever it becomes available on their system (this exists in a CDMA network, for example). Both parties in a TBA-authenticated system run parallel stream ciphers, which they initialize using a pre-existing shared secret, or through a Diffie-Hellman key exchange. The two parties use the parallel ciphers to encode a shared secret delay value. Whenever a party has a message to send to the other, it induces an artificial delay based on the most recent output of the stream cipher. Upon receiving the message, the other party measures the total response time since their prior message, and ensures that the correct amount of time was induced. The protocol has specifications for setup, tear-down, error recovery, and threat response, all of which are explained in the appendix of this report.

The remainder of this report will consist of a description of the simulation tool used to test this proposed protocol, as well as some initial results from running the simulator.

2 Simulating the TBA Protocol

The simulator for the TBA protocol is a parallelized Elixir program which can run on any arbitrary number of separate processes, processor cores, or nodes on a server cluster. However, note that running the simulator on a network-connected cluster which does not provide constant message delay violates the simulator's assumption that there is negligible propagation time for messages between nodes.

2.1 Elixir/Erlang Background

Erlang is a VM-based functional programming language with strong, deep-seeded support for parallelization and distributed programming. Elixir is, in essence, a modern syntax and library set for the Erlang language. A few nomenclature distinctions must be made with respect to the two languages before the simulator itself can be discussed.

First, the (simplified) architecture of the Elixir/Erlang environment must be known. Erlang runs a VM called Beam. The Beam VM can contain multiple Erlang processes. Erlang processes have very low overhead, so millions of them can run on a single Beam instance at once. However, the Beam VM runs on a single operating system-level process. Therefore, a simple parallelized program running in a single Beam VM is not truly parallel, as all computations are occurring on the same physical CPU core. However, Erlang processes running on Beam are all isolated from one another, and can send eachother messages based on their Beam process IDs (PIDs).

Therefore, they can, with reasonable accuracy, be used to simulate truly parallel processes with neglibile message propagation delay.

The data presented in this report was generated using parallel Erlang processes running in a single Beam VM on a single CPU core. However, the simulator itself is configurable to use any number of cores on any number of machines.

2.2 Simulator Architecture

Every node on the link-layer network is modelled using an Erlang process with its own unique PID which is used for addressing. All nodes are run using OTP servers, which provide error recovery and a standardized message communication model. There are additional OTP servers running in their own Erlang processes which the network nodes can poll for information. These servers manage a synchronous clock among all of the nodes, as well as an address list of all active nodes on the network. Nodes can join or leave the network as they please, and the OTP server in charge of topology synchronization communicates their presence/absence to the remaining nodes on the network. All nodes on the network maintain connections with all other nodes on the network, with the exception of a few special ”attacker” nodes, which are Erlang processes that send messages to active nodes in an attempt to derail their communications, or intrude upon them.

All network nodes are supervised by an OTP supervisor using a one-for-one supervision strategy. The same is true for the network services servers (time, topology), which are supervised by a separate OTP supervisor.

Because all of the Erlang processes are running on a single Beam instance, none of them can send a message to another at exactly the same time. However, this does introduce a bit of artificial delay into the network. This artificial delay represents the random distribution of time during which the system takes over the CPU core and temporarily halts the Beam instance. This delay is a reasonable approximation of actual processing delay. The data shown in this report is gathered from a Linux machine running on a quad-core Intel I7 processor, with no user tasks running. (The only other tasks running on the processor are system tasks related to the Ubuntu 14.04 distribution).

2.3 Using the Simulator

Currently, the simulator can only be run from the command line, although a proper deployment is under way. In order to run the simulator, one can clone the source code from *www.github.com/ianjjohnson/AuthenticationSimulator.git*. Note that the Elixir/Erlang runtime and Mix toolkit must be installed to run it. Once the deployment is complete, the Erlang runtime will no longer be a dependancy, as it will be built into the deployment. Once the code has been cloned, navigate into the root directory of the Git repository, and type:

iex -S mix

This will open the Interactive Elixir shell. Once in the shell, type:

Simulator.run

This will begin running the simulator. While the simulator is running, it will produce output about time delays caused by system CPU use. In order to, at any point, write the current authentication statistics to a file, without interrupting the simulation, type:

Simulator.Loger.write_authenticators

This will produce a file called *data/auth.dat*, which is an ASCII-encoded text file containing data about the authentication rates of the protocol using various window sizes. Note that the window sizes are encoded in milliseconds, and by default the maximum induced delay is 1024 milliseconds. This is configurable within the code.

3 Results

3.1 Delay

While the authenticator runs, it logs the amount of delay incurred above the artificial delay for every message that is received on the simulated network. Figure 3.1 shows that delay as a timeseries.

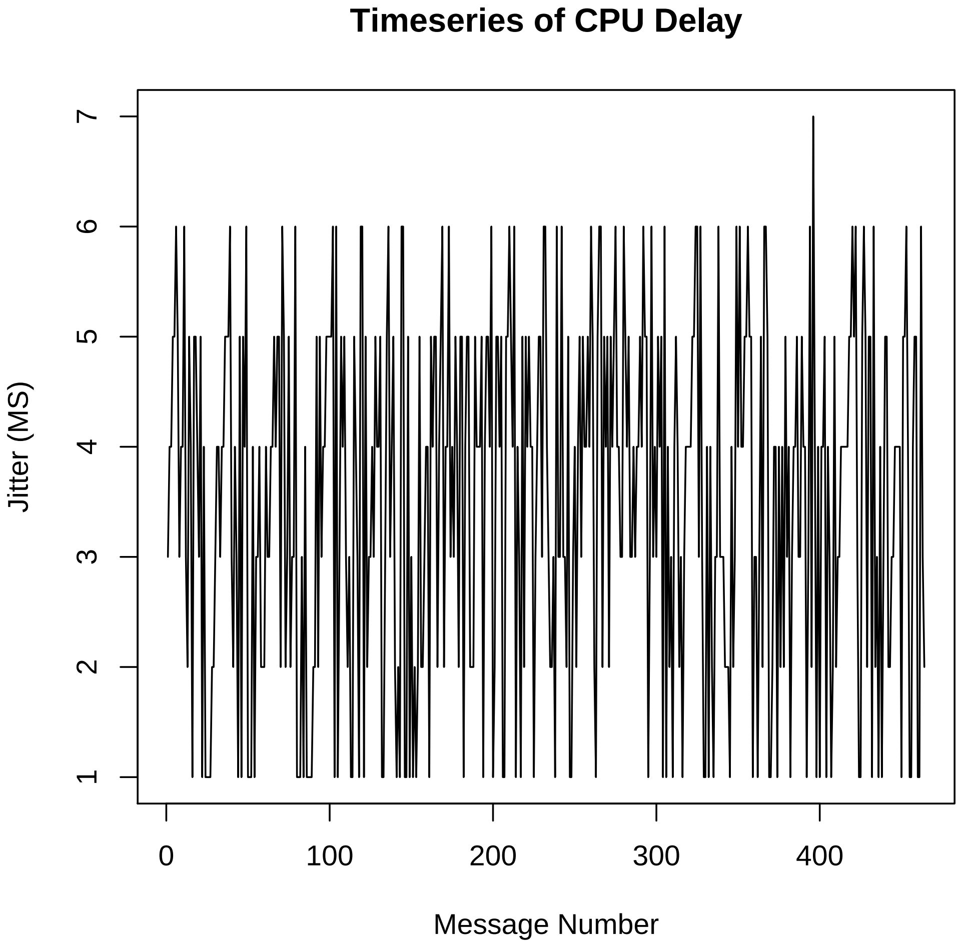

Figure 3.1: Timeseries of Extra (Non-Induced) Delay

The time series shows that there is no strong signal in the delays. It appears to simply be noise caused by the OS taking over the CPU occasionally. Nonetheless, the distribution of delay is still of interest, and is shown in Figure 3.2.

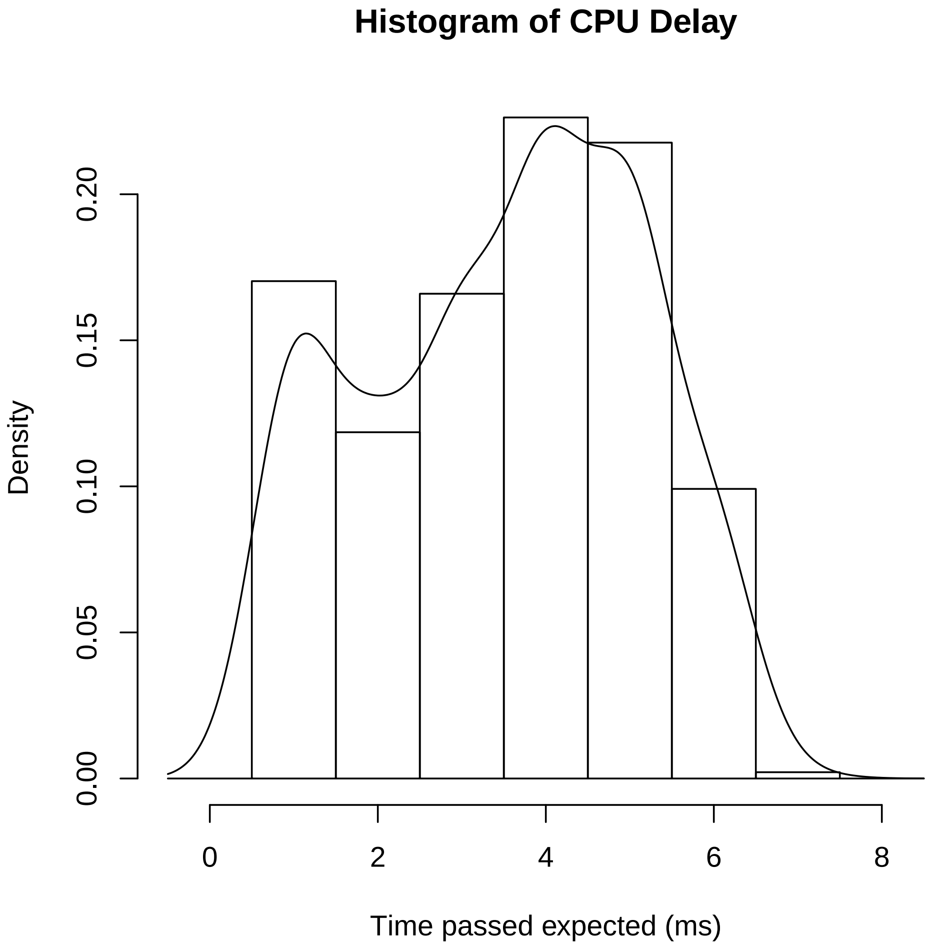

Figure 3.2: Histogram of Extra (Non-Induced) Delay

A vertical line must be drawn in the Figure 3.2 such that all messages received in an amount of time to the left of the line are authenticated, and all those to the right of the line are treated as invalid. The remainder of the presented data pertains to the process of deciding where that line should be drawn.

3.2 Authentication Window

While the simulator runs, it constantly logs the amount of time passed the expected time for both valid messages, and messages from attackers, and keeps track of how frequently a specific authentication window size correctly or incorrectly labels both valid and attacker messages. The simulator writes a confusion matrix for each authenticator size. The confusion matrix identifies the number of true positives and true negatives as well as the number of false positives and false negatives.

Figure 3.3 shows the percentage of safe messages that were labeled as safe, and the percentage of attacker messages which were successfuly marked as unsafe.

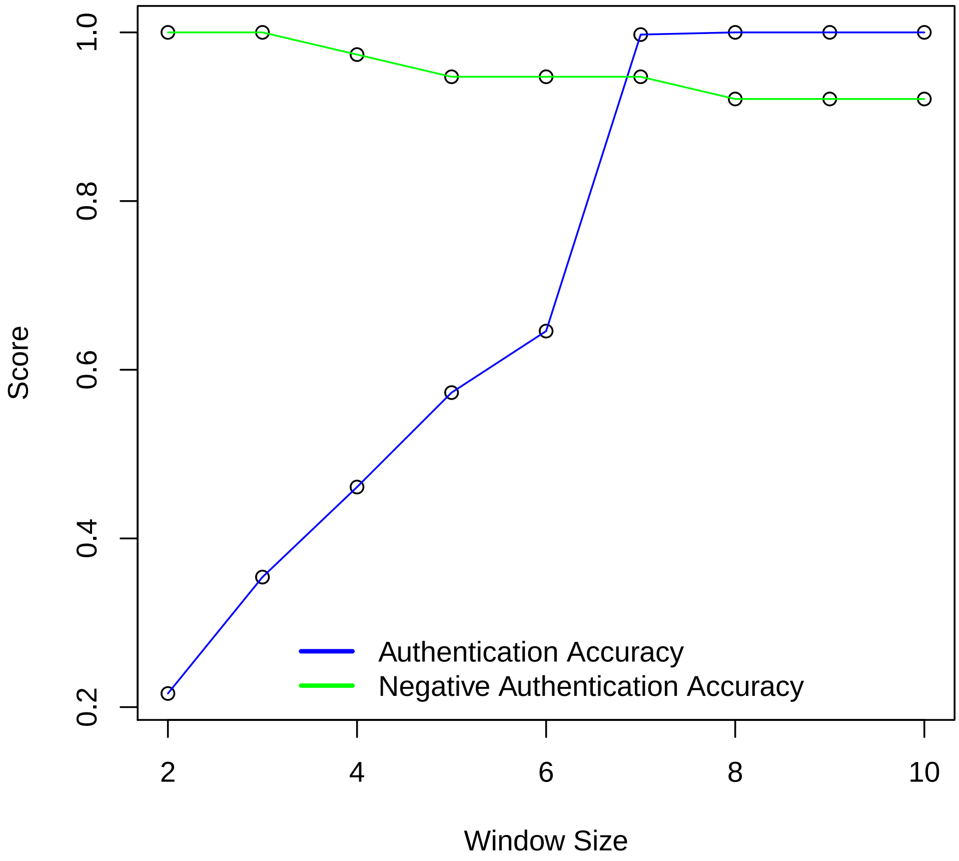

Figure 3.3: Recall and precision scores for various window sizes

Figure 3.3 shows that, in general, as window size increases, authentication accuracy increases, and negative authentication accuracy decreases. This is expected, as a greater window size in general causes more messages to be labeled as ”safe.” Figure 3.4 shows the cost function of the various window sizes, where the cost function is defined as the item-wise cost matrix defined by Table 3.1.

Actual	Predicted	Cost
Safe	Safe	0
Safe	Unsafe	0.5
Unsafe	Safe	1
Unsafe	Unsafe	0

References

[1] here is the citation