

Web Timing Attacks Made Practical*

Timothy D. Morgan[†] Jason W. Morgan[‡]

August 3, 2015

Abstract

This paper addresses the problem of exploiting timing side channels in web applications. To date, differences in execution time have been difficult to detect and to exploit. Very small differences in execution time induced by different security logics, coupled with the fact that these small differences are often lost to significant network noise, make their detection difficult. Additionally, testing for and taking advantage of timing vulnerabilities is often hampered by the tools available. To that end, we perform a thorough Monte Carlo comparison of several statistical techniques meant to identify the existence of differences in computation time in remote web applications. We then implement a tool that allows penetration testers to more thoroughly identify potential exploits.

1 Introduction

Security controls in software often perform complex logic in order to verify authenticity, integrity, or appropriate authorization of user requests. While the final result of any security logic is typically exposed to users, it is sometimes critical that the system avoid telling the user *how* it arrived at a particular result. In some cases, information about how a decision was made is revealed via error messages or other distinguishing behavior that is easy to identify. In other cases, systems unintentionally leak information about security decisions through “side channels.” These include a broad array of vectors such as radio frequency emissions, heat dissipation, and execution time. Such side channels have been extensively studied in the context of cryptographic algorithms, and practical attacks have

*Manuscript prepared for Blackhat USA, Las Vegas, Nevada, August 1–6, 2015.

[†]Founder, Blindspot Security, Portland, Oregon.

[‡]Post-Doctoral Researcher in Political Methodology, The Ohio State University, Columbus, Ohio.

been demonstrated on a variety of specific systems (e.g., AlFardan et al. 2013; Bernstein 2005; Percival 2005).

Execution time is one side channel that has not been exploited to the degree that it could be. It is common that one particular path through a particular security logic takes longer to perform than another. For example, if a website authenticates usernames and passwords in separate stages, execution time will be longer when a user provides a valid username and an invalid password than if they had provides both an invalid username and password. In this case the security logic would have leaked information about the username being valid the via the execution time side channel. To date, such differences in execution time have been difficult to detect and to exploit. This is mostly due to the very small differences in execution time relative to network latency, often causing any small differences in execution time to be lost in network noise. This combined with the dearth of tools available to penetration testers for identifying the presence of such vulnerabilities, there are likely numerous yet undetected vulnerabilities in the wild.

This research focuses on the problem of exploiting timing side channels in web applications. Building on prior work (Lawson and Nelson 2010; Mayer and Sandin 2014b), we employ improved sample collection and statistical modeling techniques—including a multidimensional Kalman Filter (Kalman 1960) and various L -estimators, which are validated through a Monte Carlo analysis—in order to more accurately identify differentials in execution times across a network. These techniques are embedded in a set of tools and work-flow that allow a penetration tester to not only detect timing flaws, but also help exploit them within the time frame of a typical security assessment.

2 The Problem

2.1 Previous Work

Previous researchers have made great progress in developing methods that can identify and exploit remote timing vulnerabilities. Crosby et al. (2009), for instance, focus on measuring individual packet round-trip times and compare several tests for distinguishing timing differences. Their best performing classifier is the “box test,” in which lower quantiles of

each distribution of round-trip times are used in order to identify differences between these distributions.

Following up on this work, Lawson and Nelson (2010) presented their research at Blackhat 2010 where they too compared several statistical methods for distinguishing independent distributions of timing measurements. They settled on a percentile range approach very similar to that described by Crosby et al. (2009).

More recently Mayer and Sandin (2014b) provided an updated analysis of what kinds of timing attacks might be practical in real-world systems. They also released a tool, Time Trial,¹ which can help the typical penetration tester or vulnerability researcher determine if a timing difference exists on a remote system. In their analysis, the box test was the primary method of distinguishing timing distributions.

2.2 Goals

While significant progress has been made, past research has largely focused either on exploiting specific vulnerabilities, or on identifying the minimum timing differences that are theoretically measurable in controlled environments conditional on different sample sizes. While crucial for conducting further research, this information doesn't directly address the key question that penetration testers need to answer before moving forward with an exploit: *How many samples do I need to collect to exploit the timing side-channel I just discovered?* Answering this question is the primary motivation behind our research and the tool set we develop.

On the statistical side, our research also seeks to improve the techniques employed by previous researchers through the application of rigorous analysis and testing. In particular, we utilize statistical techniques that degrade more gracefully when applied to data sets with the extensive "jitter" (or "noise") that would typically be found in real-world, long-distance communications.

Finally, previous researchers Lawson and Nelson (2010) left open the question of whether or not TCP timestamps could be used to improve round-trip time measurements. Exploring this possibility was a secondary goal of our research.

¹https://github.com/dmayer/time_trial

2.3 Timing Side-Channel Scenarios

Execution time side-channel leaks can appear in a wide variety of situations. In cryptographic systems, where they have been most closely studied, information leaks through execution time can introduce catastrophic weaknesses in specific contexts. We were more interested, however, in looking at situations where typical custom applications might be exposed to these kinds of attacks. Consider for a moment the following screen shot of a registration web page shown in Figure 1.

Figure 1: Typical Health Insurance Registration Site

The screenshot shows a web registration page titled "Personal Information:". At the top, a progress bar indicates four steps: 1 Personal Information (active), 2 Create a Secure Log In, 3 Preferences, and 4 Terms. Below the title, a red asterisk indicates a required field. The registration options are "Member ID Number" (unselected) and "Social Security Number" (selected). The form fields are: "Social Security Number:" (with a help icon), "Re-enter SSN:", "Full First Name:", "Full Last Name:", "Date of Birth:" (with dropdowns for Month and Day, and a YYYY input), and "Zip Code:" (with a help icon). A yellow callout box points to the SSN field with the text "Please enter your 9 digit Social Security Number." At the bottom, there are "BACK" and "CONTINUE" buttons.

This page allows existing health insurance customers in the US to register for an online account on the insurance provider’s website. This kind of identity verification could be considered a form of knowledge-based authentication (KBA), which are often susceptible to “divide and conquer” attacks. *Note that we know of no vulnerability in this page and do not want to imply it is not secure.* However, it is commonly the case that forms such as this

leak information through error messages that allow for brute force attacks.

For instance, let us assume for a moment that when a user submits this form with an invalid social security number, the site returns an error message “Invalid SSN. Try again.” However, in this hypothetical scenario, entering a *valid* SSN along with an *invalid* name causes the site to return “Invalid Name.” In this case, an attacker could iterate over a large number of possible SSN numbers until the site returned the “Invalid Name” error. Next, the valid SSN’s name fields could be brute-forced using common names. From there, guessing the user’s date of birth and zip code might be very easy, since SSNs are typically closely associated with the month a person was born (assuming they were born in the US) and what state they lived in when they first received the SSN (which could be correlated with their current region of residence). On the other hand, if the site always responded with the same error message for every invalid combination of values, then it would be much more difficult for the attacker to guess valid field values, since the number of possible combinations of these field values becomes significantly higher.

Returning to the context of timing attacks, now assume the target site does respond securely with the same error message for every invalid form submission. However, perhaps the site requires a slightly different amount of time to validate each field of the form. That is, perhaps the site submits one database query to a back-end server to look up a user’s SSN, and then submits a separate database query to validate each subsequent field. If this were the case, then attackers would once again be in the position to conduct a divide and conquer attack, assuming they could accurately measure the execution time of the form’s processing. However, given the limited tools available today, penetration testers and security analysts would have a difficult time testing for this kind of issue and, consequently, evaluating the associated level of risk.

Another scenario where accurate timing measurements would be valuable to penetration testers would be in the case of a truly blind SQL injection attack. In testing for these flaws, it is fairly common knowledge in the security testing community that an attacker can force a database query to pause for a period of time (either through explicit `WAITFOR DELAY . . .` calls or computationally intensive operations). This is often sufficient to prove that a SQL injection flaw exists, but to extract information from the database quickly would require

accurate timing measurements of specific requests. A penetration tester could try to speed up an attack by reducing the delay they create in the query processing, but how small of a delay can be accurately measured with a minimal number of requests? Without tools to conduct a statistical profile of the application's response time variability, one is likely to spend too much time obtaining only limited results or obtaining inaccurate information due to misinterpreted timing delays.

Many other scenarios exist where timing flaws could exist, but are likely going unnoticed by the security testing community. Mayer and Sandin (2014b) do a great job of enumerating a number of other common cases in their 2014 Blackhat research.

3 Our Approach

Past researchers have taken the general approach of collecting samples for each test case and then comparing the individual distributions of response times. These distributions are clearly not Gaussian in nature and it is not always clear what distribution best fits the data, as it is often multi-modal and almost never symmetric. This greatly reduces the set of standard statistical techniques that can be applied directly to the individual RTT distributions. In addition, this approach often has difficulty taking into account any disruption that may occur in network or application conditions over time. For instance, if in the middle of a long-running sample collection job the network experiences a momentary outage and subsequent rerouting of traffic, the underlying distribution is likely to change for a period of time. In other cases, the application being tested could suddenly come under heavy load from outside users. In both of these situations, characteristics of the RTT distributions can change significantly, complicating the subsequent identification of any timing differences.

We approach the statistical analysis of RTT distributions from several angles. In one approach, we develop a set of tests, which attempt to accurately estimate the central tendency of the distribution of pair-wise round-trip time differences. If the central tendency of differences is clearly non-zero, in a statistically significant way, then we can say with some confidence that a difference in timing exists. In another approach, we employ a Kalman

filter, which is a robust statistical method commonly used on data sets such as this. We discuss each approach in the following sections.

3.1 Central Tendency of Pairwise RTT Differences

Instead of treating the timing measurements for each test case as separate distributions, we collect our samples for each test case as pairs² (i.e., they are collected at nearly the same time, randomizing which test case—“long” or “short”—is sent first). The objective of doing the collection in pairs was to reduce the heterogeneity in the distributions of the two test cases, thus making them more comparable than if we had collected them in sequence. For instance, suppose we had first decided to collect the set of long-case observations, then went on to collect the short-case observations. Following this collection approach, a systematic change in the network conditions during any of this period would make the two distributions of the RTT data incomparable. By collecting the data in pairs, however, a systematic change in the network would affect subsequent *pairs* of samples; i.e., round trip times within pairs would remain comparable.

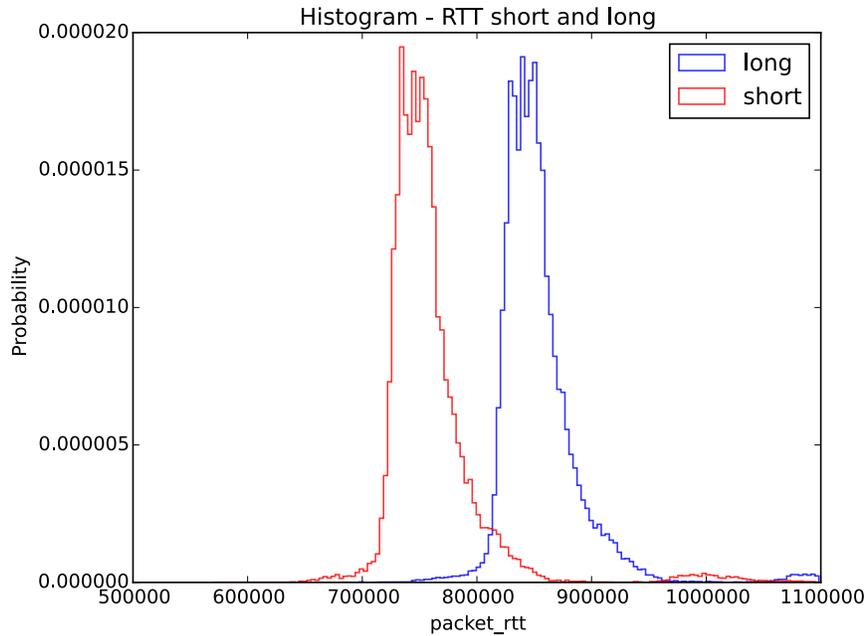
With these sample pairs in hand, we then compute the difference in round-trip time for each pair of requests. The distribution of these differences turns out to be fairly symmetric about the mean and allows us to use more traditional statistical techniques. Consider the histogram of test case data in Figure 2.

Here we have two distributions, one for the “short” critical computation, and one for the “long” critical computation. Each distribution is very similar, but obviously different in their central tendency. Neither is symmetric. However, if we measure the difference in round-trip time for each pair of RTT measurements, we obtain the distribution in Figure 3. This is clearly much more symmetric than the individual RTT distributions.

The fundamental question to be answered in an analysis of the distribution of differences is whether the central tendency of this distribution different than zero. This is, of course, a common task in statistics and the symmetric nature of the distribution allows us to appeal to standard measures of central tendency; e.g., mean or median. Additionally, working with the distribution of pair-wise differences also makes the analysis more resilient to changes in

²Note that Nanown is designed to handle more than two test cases per sample collected, though for this analysis we focused on simple pairs of test cases

Figure 2: Histogram of Test Case RTT Distributions for Short and Long Computations

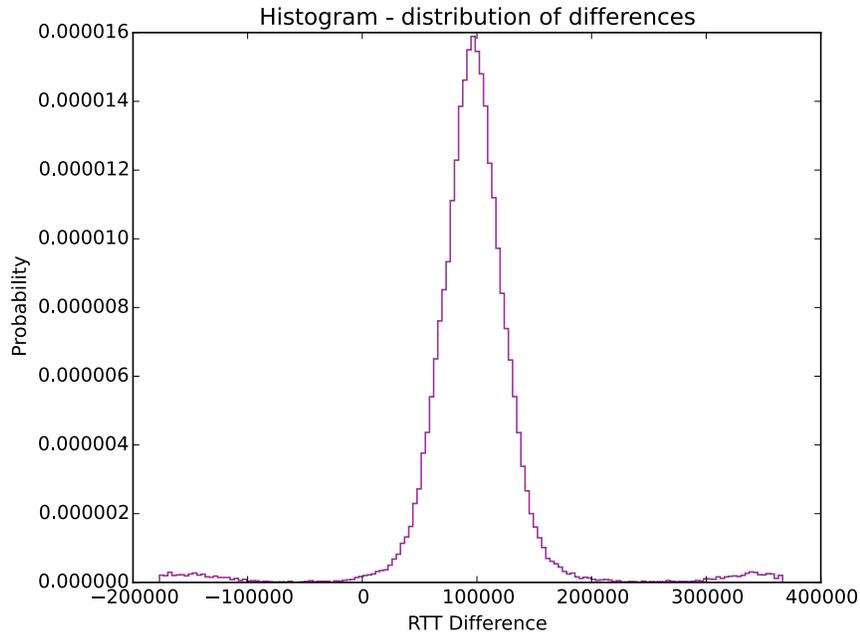


network and host conditions. As an example of this, during one early test over the Internet, we encountered a data set with a distribution of that shown in Figure 4.

This was initially a confusing result, as it wasn't clear what would generate such an extremely multi-modal data set. However, after reviewing the round trip times as a function of time-of-day, it became obvious what the issue was (see Figure 5).

Here, it appears we fell victim to a feature of Comcast's Internet service: PowerBoost™. This feature is designed to allow subscribers to download the first small portion of a file at unlimited speeds (as fast as the infrastructure permits), but once some threshold is reached, Comcast resumes throttling the user to the rate they are subscribed at. It appears this is what had occurred, since the first small portion of the test ran with much less delay. This could clearly throw off any simple statistical methods that operate on the distributions individually in an uncorrelated way, and in this case it would be tempting to reject the data at the beginning of this test and focus on the later data. However, if we perform our statistics on the distribution of pair-wise differences, the data is automatically normalized around a

Figure 3: Histogram of Pair-Wise Differences in RTT for Short and Long Computations

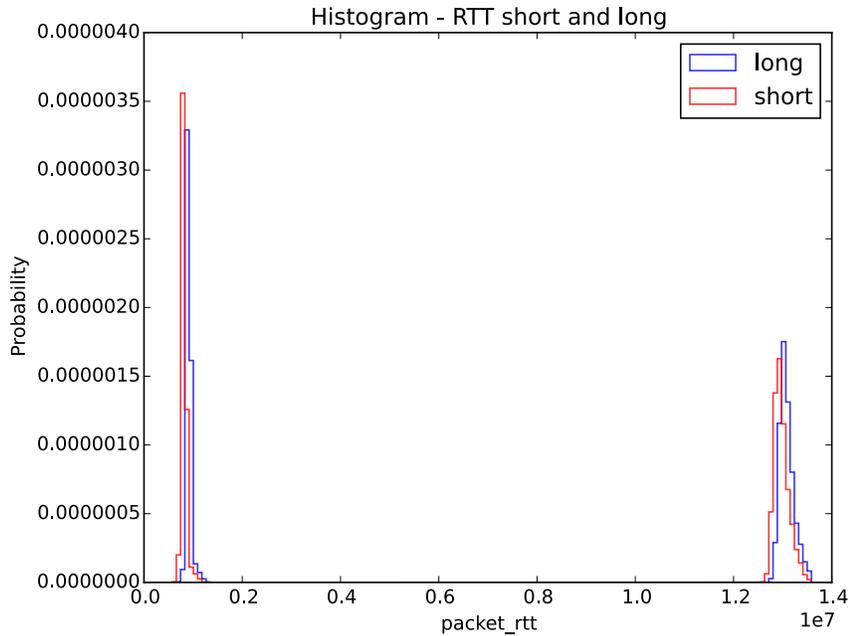


central tendency that describes the difference in application server processing time. This is clearly shown by the purple “difference” series in the above plot. While the variance of data in each case may change, at least all data is usable to approximate the critical computation delta.

3.2 TCP Timestamp Measurement Approach

One approach for leveraging TCP timestamps to measure round-trip times, as mentioned by Lawson and Nelson (2010) in their 2010 Blackhat presentation, involves trying to synchronize the local system clock with the clock of the target system. Samples would then be collected in some small amount of time (call this t_b) before the target host’s TCP timestamp clock is expected to roll over to the next increment. If the resulting response did contain an increment in the timestamp clock, then one can conclude the response time was greater t_b , otherwise it was smaller. Under perfect conditions, one could use a binary search to quickly determine the server processing time. However, to successfully perform this measurement,

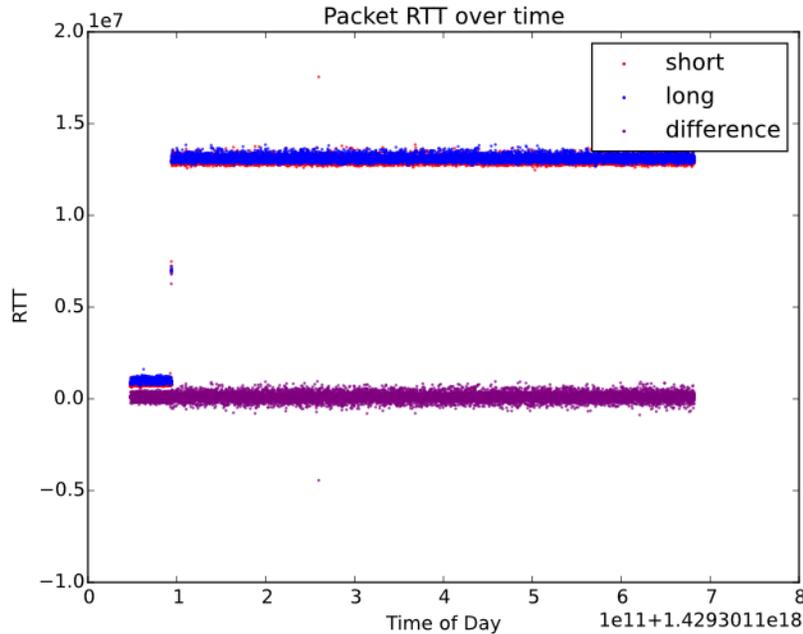
Figure 4: Histogram of RTT Values for each Test Case



one would need to have extremely accurate measurements of both the target clock rate, the target clock offset (when, in terms of local time, the clock changes), and be able to submit requests at very accurate times. Lawson and Nelson (2010) rejected this approach due to the perceived implementation difficulties and we currently agree that this approach is likely impractical.

However, an important observation about low-precision clocks is that they can still be used to measure very small time differences statistically without any synchronization, so long as one can obtain many samples. For instance, if a clock measures time in increments of 1ms and we wish to measure the time of some event that takes 0.1ms, then we could collect those samples at random points in time and count the number of samples where the low-precision clock incremented by one. If the samples were collected in an evenly-distributed manner over a long period of time, then one would expect approximately 1 in 10 samples to show an increment in the clock. Simply taking the mean of sample times can therefore provide a measure of this very short event time. For example, if 10,000 samples were taken and 9000

Figure 5: RTT Measurements Over Time



of them showed no increment in the 1ms clock, while 1000 showed a 1ms increment, then the measured time is: $(9000 \times 0\text{ms} + 1000 \times 1\text{ms})/10000 = 0.1\text{ms}$.

Note that this form of measurement is very sensitive to sampling error. If the number of samples taken that indicate a 1ms time difference is incorrect by even a small amount, this has a significant impact on the overall measurement. In addition, if the times when samples are taken are not evenly distributed over the clock precision time frame (that is, the sample time-of-day modulo the clock precision is not evenly distributed), then this can quickly skew the results. Even with these apparent difficulties, TCP timestamps do offer the promise of potentially negating network jitter, so we decided to capture this information and determine just how accurate or inaccurate this data would be.

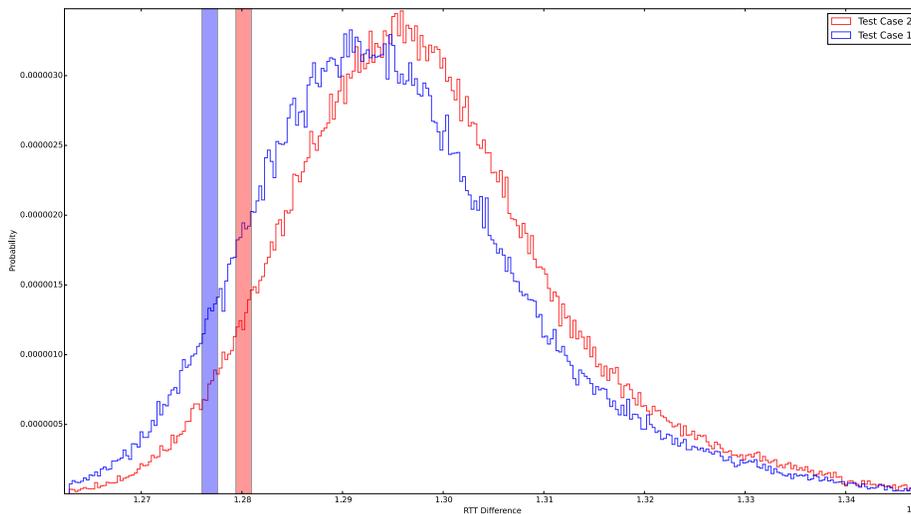
3.3 Classifiers

We tested several classifiers on the collected RTTs. These are described below.

3.3.1 Box Test

The “box test” is a simple classifier based on order statistics, apparently introduced in Crosby et al. (2009). The box test has two parameters, the “low” and “high” values, which are percentiles that designate the bounds of the “box.” When comparing two independent distributions to determine if they are equal, one simply computes the low and high percentiles for each distribution. If the resulting boxes do not overlap, then one of them is declared to be larger than the other and when the percentiles do overlap at all, then the two distributions are considered equal. For example, suppose we are using a box test with parameters: `low=6`, `high=8` and we applied this test to the distributions shown in Figure 6.

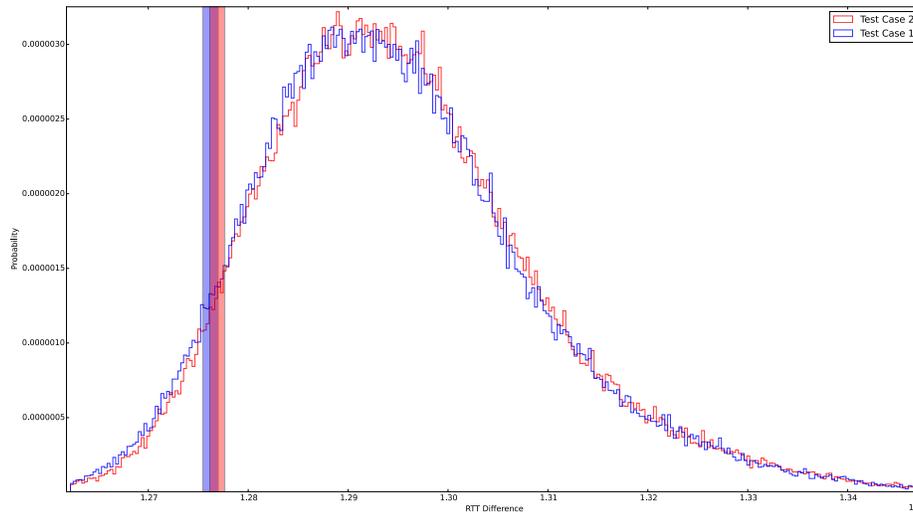
Figure 6: Box Test — Clearly Distinct Distributions



Here we can see the entire box for the first distribution is clearly less than the lower bound of the the second distribution. Therefore, the boxes do not overlap and we would consider the first distribution to be smaller. Conversely, Figure 7 shows the same box test parameters applied to more closely aligned — data sets. It is clear in this case that the boxes overlap, indicating the box test would classify these two distributions as indistinguishable.

The box test is nice in that it relies on order statistics which are robust in the face of outliers and network jitter. However, by considering the two distributions as entirely separate, any correlation between the collected RTT measurements for each test case is

Figure 7: Box Test — Indistinguishable Distributions



ignored. As discussed earlier, measurements of different test cases captured at nearly the same time could both be affected by the same network disturbances, but the box test cannot take this into account. Finally, the box test is only a classifier, it cannot estimate the observed time difference between box test cases, which is useful in some cases.

The box test has no canonical algorithm to choose the best low and high parameter values. Previous researchers have found that there are no “safe” parameters that work on all distributions, so one must employ a training algorithm to identify the parameters most appropriate to a given purpose.

We developed a box test training algorithm that is both relatively efficient and effective. We first observed that the distance between the low and high values (the “box width”) is directly related to the ratio of false positives to false negatives produced by the classifier. A larger box width tends to produce a larger number of false negatives (since the boxes will overlap more often). A narrow width produces a greater number of false positives and fewer false negatives. Relying on this observation and experience from past researchers, our algorithm works approximately as follows:

1. Select a box width of 1.0. Test all possible low values from 0 to 50, measuring the overall error rate for each. From these results, identify the 5 candidate values that

have the lowest error.

2. For each of 5 low values from step 1, estimate the false positive and false negative rates for all widths in the range $[0.5, 6.0]$ in increments of 0.5. Take the mean of all false positives and all false negatives separately for each width over the 5 low values. Let w^* be the candidate width that produces the most balanced false positive and false negative rate (smallest absolute difference between the two).
3. Using w^* from step 2, repeat step 1 by iterating over all low values from $[0, 50]$. Isolate l^* , value with the lowest overall error rate.
4. Using the l^* from step 3, test all widths in the range: $w^* \pm 0.7$ in increments of 0.05. Let the l^* value be the width which returns the most balanced false positive and false negative rates.

The reasoning behind this algorithm is that we should be able to quickly isolate a width that works best through a process similar to gradient descent or binary search, if only we knew the correct location to place our box. So we start by finding a set of candidate locations for our box and use those to find a candidate width value that should be close to optimal. We then finalize the location of our box using that candidate width and later fine-tune the width using the finalized location.

3.3.2 *L*-Estimators

L-estimators are a broad category of statistical estimators based on linear combinations of order statistics. In our case, we are interested in measuring the central tendency of distributions. One of the simplest, relatively robust *L*-estimators for measuring this is the median. That is, simply taking the 50th percentile of a distribution produces a reasonable estimate of its location. An alternative, yet still simple *L*-estimator would be to measure the 25th and 75th percentiles and then take the mean of those two values. This measure is called the midhinge and often slightly better than the median since it measures two separate values, while remaining less susceptible to outliers.

One can generalize the midhinge further by parameterizing it. With a parameter, w , in the range $[0, 50]$, we could define an *L*-estimator of (where P is the percentile function):

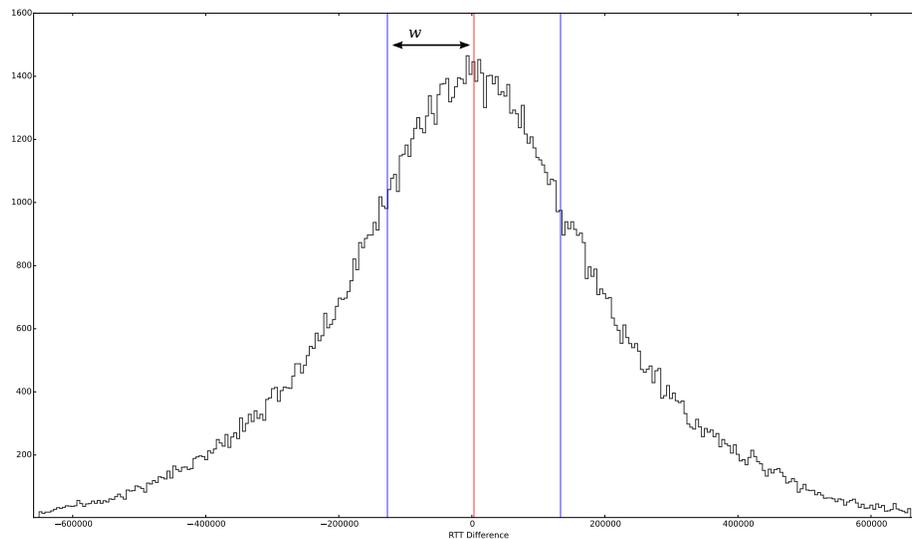
```

LET L1 = 50 - w
LET R1 = 50 + w
RETURN mean(P(L1), P(R1))

```

For instance, with $w = 10$, we would take mean of the percentiles at 40 and 60. This parameterized L -estimator is well known and is referred to as the midsummary. As an example, Figure 8 shows the midsummary percentiles as blue vertical lines and the resulting central tendency estimate in red.

Figure 8: Midsummary, $w = 25$



Of course one can imagine any number of linear combinations of order statistics in used to estimate the central tendency of a distribution. Adding more order statistics to the mix, one might be able to make the measure more reliable, but possibly sacrifice some flexibility in focusing on the most important portions of a distribution. After experimenting with a number of possibilities, we settled on trying three different L -estimators: the midsummary, a 4-statistic measure we named the quadsummary, and a 7-statistic measure we named septasummary.

The quadsummary is just like the midsummary, except it has two additional order statistics calculated halfway between $L1$ and the median and $R1$ and the median. In other words, the quadsummary procedure is:

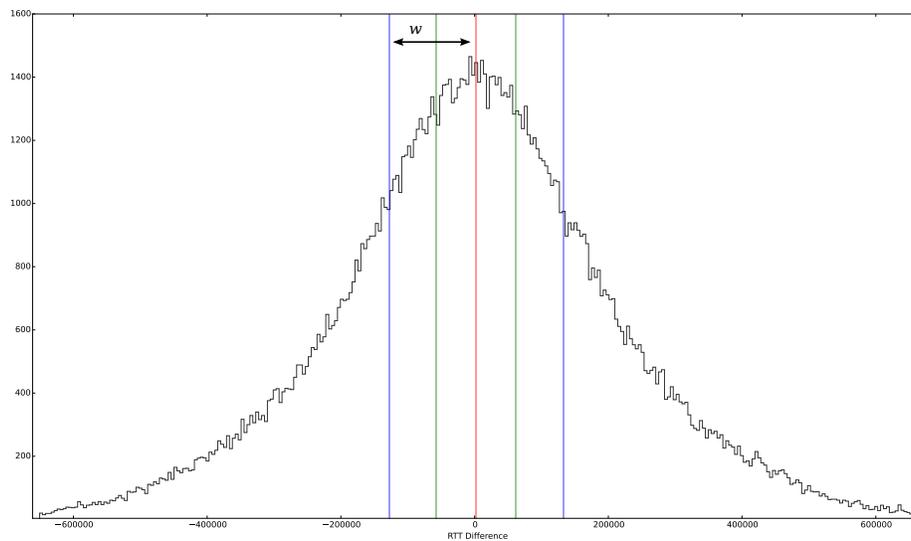
```

LET L1 = 50 - w
LET L2 = (L1 + 50) / 2
LET R1 = 50 + w
LET R2 = (R1 + 50) / 2
RETURN mean(P(L1), P(L2), P(R1), P(R2))

```

Figure 9 shows an example of the quadsummary with the L2 and R2 locations in green.

Figure 9: Quadsummary, $w = 25$



Finally, the septasummary is the same as quadsummary, except that it adds three more order statistics: the median, one halfway between L1 and the left tail of the distribution, and a third one halfway between R1 and the right tail. The pseudocode is:

```

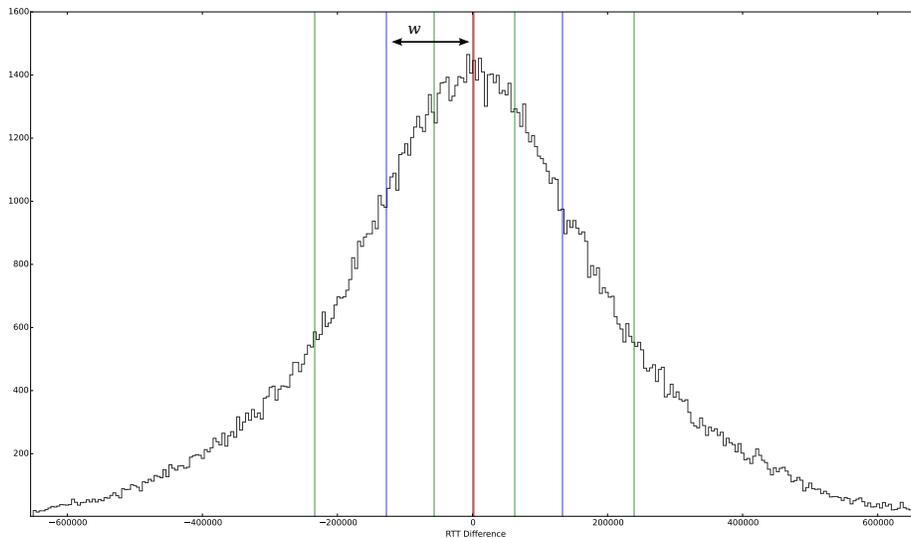
LET L1 = 50 - w
LET L2 = (L1 + 50) / 2
LET L3 = L1 / 2
LET M = 50
LET R1 = 50 + w
LET R2 = (R1 + 50) / 2
LET R3 = (R1 + 100) / 2

```

RETURN $\text{mean}(P(L1), P(L2), P(L3), P(M), P(R1), P(R2), P(R3))$

See Figure 10 for an example of the septasummary with the additional L3 and R3 locations in green and the median in dark grey (at nearly the same location as the estimate).

Figure 10: Septasummary, $w = 25$



The reason all of the order statistics locations are parameterized from w is that it simplifies the implementation and training algorithm. Having a separate parameter for each pair of statistic locations could improve performance and may be investigated in the future.

Of course all of these measures simply tell us the approximate location of a distribution, but they do not directly answer the question: *is test case A's distribution greater than test case B's?* To determine the answer, we first obtain the distribution of differences between test cases, as discussed earlier. We then introduce a new parameter, the threshold, and apply our L -estimator to the distribution of differences. If the L -estimator's result is greater than the threshold, then we classify the test cases as different. Otherwise they are classified as being similar.

Each of the L -estimators is trained using the same algorithm to determine the best w and threshold values, which is approximately:

1. The L -estimator is used across the entire training data to obtain a central tendency

value. An initial threshold is then set to $1/2$ of this value (halfway between the expected L -estimator output and 0).

2. Next, every w value in the range $[1, 49]$ is tested to identify the one with the lowest overall error.
3. Using the candidate w value from step 2, a series of potentially better threshold values are tested. The values tested are in the range of $\pm 20\%$ of the threshold from step 1. A new candidate threshold value which best equalizes the false positive and false negative error rates is selected.
4. Using the candidate threshold from step 3, a series of potentially better w values is tested in the range $[w - 4, w + 4]$ in increments of 1 percentile. The value which minimizes overall error is selected as the final w parameter.
5. In this final step, step 3 is repeated over the candidate threshold values $\pm 10\%$ (in increments of 1%). Once again, the threshold chosen best equalizes the false positive and false negative rates.

3.3.3 Mean of TCP Timestamps

The TCP timestamp data has a precision level so low that it is inappropriate to use order statistics. Instead we are forced to the mean or some variant thereof. During testing we attempted to use the mean of timestamp differences between pair-wise test cases. We also attempted several weighted means, choosing weights based on whether the associated packet time information indicated certain samples might be outliers. Ultimately, none of these attempts were successful. In even the best results yielded error rates near 40%, meaning this approach was little better than a coin flip. Therefore we omitted the direct timestamp analysis from the results below. However, we are hopeful that there may be some way to leverage the timestamp information to help augment packet time data in a more sophisticated algorithm.

3.3.4 Kalman Filters

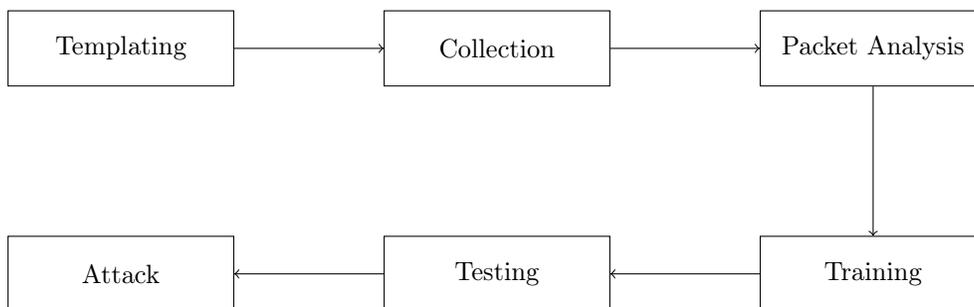
The Kalman filter is an iterative, “on-line” process of estimating what is essentially a weighted running average of a time series. The method is often put to use in signal processing due to its robustness to noisy data. This makes it ideal for our purposes since, as described above, the RTT data are riddled with the unexpected spikes. In our application, we use the Kalman filter to estimate RTT for each test case series simultaneously (a 2-dimensional model). This allows us to account for correlations in the series over time as well as the overall variance in the series. Unfortunately, due to the computational complexity of the algorithm, the Kalman filter approach has not yet been incorporated into tools and full statistical results are not available at this time. That said, the approach does appear promising and warrants further study.

4 Implementation

4.1 Overview of Nanown Workflow

As part of this research we have developed the new tool suite “Nanown” (Morgan 2015) Nanown is designed around a multi-stage workflow outlined in Figure 11.

Figure 11: Nanown Workflow



In the templating stage the user needs to identify two or more separate test cases which are suspected of responding with differing round-trip times. For instance, if one were attempting to brute force valid user names in a login form, then two test cases could be provided: one with a valid user name and one with an invalid user name. The user must

also provide a small amount of code to inform Nanown how to send the precise HTTP(S) requests the application expects.

In the next stage, the user initiates sample collection. A significant number of samples are required (perhaps 100,000) in order to profile the site behavior reliably and train Nanown on the most optimal parameters to use during an attack.

After all samples are collected, packet analysis is performed which associates observed packets with specific TCP connections. Several analysis steps are then performed to eliminate duplicate packets, identify out of order packets, and to isolate the pair of packets most closely correlated with the critical computation.

Next, each classifier is trained using a dedicated subset of the samples collected using bootstrapping techniques. Each classifier is trained on a variety of different sample sizes and the total classification error (mean of false positives and false negatives) is minimized for each of these sample sizes. As a result of this stage, a series of candidate parameters for each classifier are saved.

During the testing stage, each candidate classifier and associated parameters are tested against a separate subset of the samples dedicated to testing. Each classifier is tested to see if it can reach a 5% or lower error rate given the quantity of data available. If this is possible, then the sizes are progressively reduced to isolate the smallest number of observations that still results in a 5% error rate. As an output of this stage, the user is informed of which classifier is most successful and how many samples will be required to reliably arrive at each classification.

At this point the user can easily determine how realistic an attack would be, as they know approximately how many requests would be required for each classification and the rate of sample collection is known from the second stage. If the user chooses to formulate an attack, they would write a script that leverages the classification method and parameters learned during the training and testing stages. For each timing difference classification needed, the attacker simply needs to collect the recommended number of samples and then apply the classifier to the data. There is always some probability that a classification attempt may return an incorrect classification, so any key result the attacker needs to be sure of can be re-checked with additional rounds (e.g. best 2 out of 3).

4.2 Round Trip Time Measurement

Since TCP software stacks do not expose TCP timestamp information to callers, it was clear that the only way to obtain this information would be to either use a custom TCP implementation, or to employ a packet sniffer to monitor transactions between standard HTTP clients and services. The latter option was chosen as it seemed simpler and less error prone.

Our current implementation spawns a long-running network sniffer to monitor all TCP connections between the local system and a designated target service. This sniffer records basic metadata about each packet observed, including port information, observed time stamp, raw TCP timestamp (if available), TCP sequence and ACK numbers, and whether the packet was sent or received. This information is written to a log file for later processing.

Once the network sniffer is enabled, the Python-based sample collector sends groups of requests, one for each test case (designated with simple names, such as “long” and “short”), and repeats this process as quickly as possible. Each group of test case RTT measurements is considered one “sample” in our terminology. The order in which the test case groups are sent for each sample is randomized to minimize the possibility of sampling error due to local or remote scheduler biases. A variety of diagnostic information is collected by the sampling process, but most importantly, the local TCP port numbers are logged for each request. Later, once all data has been collected, the TCP port numbers are used (along with time information) to link up every HTTP request with the raw packets observed for that request.

Next, each request’s packet metadata must be carefully analyzed to obtain measured round-trip times, both for locally measured time differences, and for TCP timestamp differences. This can be a very tricky task, since TCP itself is a complex protocol and one must account for situations such as re-sent packets, out of order packets, and determining which packets best capture the time delay associated with the web application’s critical computation. The following sections discuss how we dealt with these issues.

4.2.1 Re-sent Packets

Since TCP attempts to guarantee that no packets are lost during transmission, it is common to observe duplicate or retransmitted packets. Either end of a TCP connection may

retransmit a packet if it doesn't receive confirmation that the packet was received within a particular time frame. The decision to retransmit varies based on current network conditions and the particular TCP implementation.

The obvious problem with duplicate packets in our project was to decide which packet to trust. For instance, if two packets with the same HTTP data were observed, which time stamp should we use? We decided to adopt a conservative policy of "first packet wins". In the case of sent packets, there is no practical way to know if a duplicate packet appeared because the first copy was lost/corrupted or if it was just resent because the remote end didn't acknowledge it quickly enough. In the latter case, the HTTP server may have already been processing the request, unbeknownst to us, so we conservatively use the first copy we observed.

In the case of duplicated received packets, we know that the earliest first packet arrived successfully, so it is likely that the client was just slow to acknowledge the first copy. The first copy indicates that the server had finished any associated critical computation at some point prior to that, so using the observed time from the first copy makes the most sense. (There is a tiny chance that the first copy of the received packet was actually corrupt, failing a CRC check, in which case packet metadata may have caused our analysis to be mixed up. We estimate this to be a very low probability event, so we have not dealt with it specifically.)

Ultimately, any TCP connection where retransmissions occur could be considered less trustworthy from a data collection standpoint, regardless of the de-duplication policy used. For this reason, for any probes where a duplicate packet was observed, our processing routines record a flag indicating that duplicate packets were observed, which could assist in later classification efforts.

4.2.2 Out of Order Packets

When dealing with out of order packets, we rely on the assumption that a single HTTP request will be made over the TCP connection. This means all request packets with payload will be sent first, then the response will appear afterward with a series of response payloads. We have experimented with several ordering strategies to deal with packets that arrive out of order, but we found the simplest and most reliable approach is to first eliminate packet

duplicates, as discussed above, and then simply keep packets in order based on the time they were observed. Indeed it should be rare to see any request packets be observed out of order (since we choose the earliest copy of any duplicates and ignore the others). When it comes to reordered received packets, any data payload that was delayed would at most add some small additional delay to our RTT measurements. Failing to include the critical computation time in our RTT measurement should not be possible, given our request/response procedure. As with duplicate packets, any probe where an out-of-order packet is observed is flagged as such for future processing.

4.2.3 Advantages of Packet Sniffing

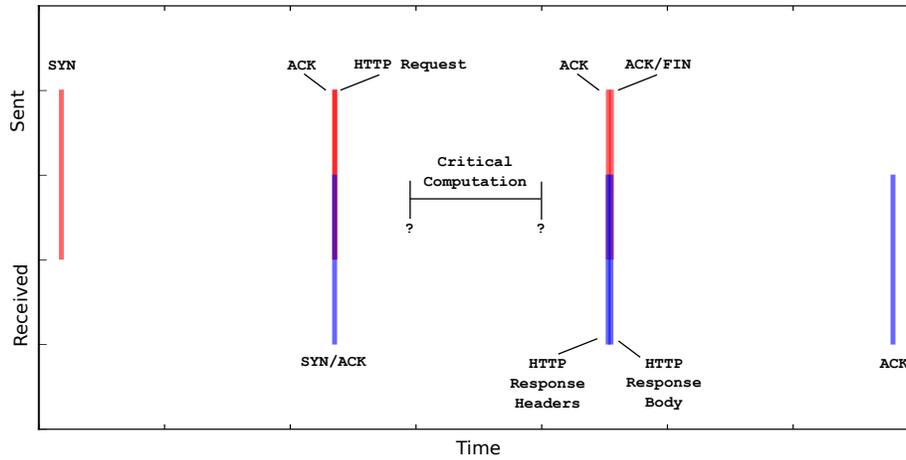
In addition to providing us the ability to collect TCP timestamps, we found that measuring round-trip times based on packet sniffing to be far more precise than any procedure that relies entirely on userspace time measurements. The timestamps obtained from raw sockets are assigned to each packet by network adapter and/or kernel timers, which log packet arrival before this data is processed and delivered to userspace. This time delay between packet arrival and delivery to userspace applications adds variance to all measurements, not to mention any additional time delay that may be introduced by multiple userspace processes competing for CPU time. While real-time priority settings can diminish this negative impact in userspace, it does not eliminate the overhead of transitioning back and forth between userspace and kernel space multiple times for each timing measurement.

Besides eliminating this local host variance, having access to raw packet data allows one to filter out additional network latency introduced by factors such as the TCP handshake and other non-application packet exchanges. For instance, in the Figure 12 we see that including the latency of the TCP handshake would more than double the overall RTT measurement:

With raw packet data, it is easy to conservatively measure round-trip times as the difference between the observation times of the first sent packet that contains TCP payload and the last received packet containing payload.

We can also see the negative effects of measuring round-trip time from within userspace experimentally. In Figure 13 we have histograms of two RTT data sets compared side by side. In both cases the same HTTP service is tested under the same conditions and parameters

Figure 12: HTTP Packet Times



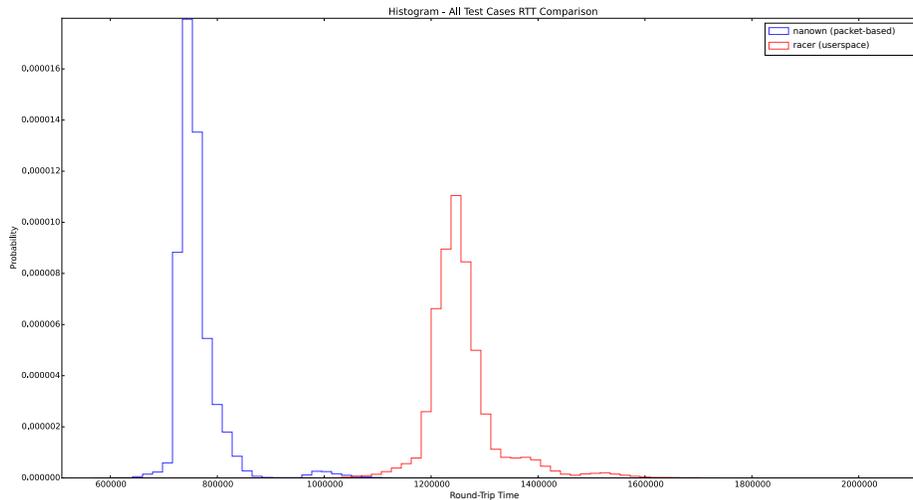
(a virtual machine on the LAN). In one case we use our packet sniffing approach, and in the other, the Time Trial racer tool (Mayer and Sandin 2014a) is used:

While the racer tool is a very good userspace implementation (easily beating the collection rate and quality and of data we have collected with Python in userspace), it is clear that the blue histogram both has shorter time measurements and is also more compact (with a higher, more distinct peak). Indeed, the median absolute deviation of the packet sniffing data are 40% smaller. (Note that racer was run with the real-time priority and CPU affinity settings suggested by the authors.)

4.2.4 Isolating the Critical Computation Packets

While choosing the first and last data packets appearing in a TCP connection is a safe conservative way to measure round-trip time, it is often the case that the critical computation occurs somewhere in the middle of the multi-packet exchange. For instance, the targeted web page may perform the critical computation and subsequently return a large body of content that cannot fit within a single packet. In this case, it doesn't make sense to measure the arrival time of the last packet in that buffer. Instead, the first data packet to arrive might be more appropriate and yield less variance in the measurements. In other instances, some web applications are written in integrated scripting languages (think: .php, .asp, and .jsp

Figure 13: Racer vs. Packet Distributions



scripts) where the HTTP response headers and perhaps HTML header content are written out to the client immediately, followed by the critical computation, and then finally the remainder of the page is written after this.

One might be tempted to try and identify content signatures for a particular application in order to zero in on the first packet after the critical computation based certain content elements. However, this starts to become much more difficult if HTTPS is in use since only SSL/TLS metadata could be profiled. HTTPS applications happen to be a scenario where trimming extraneous packet time from RTT measurements is most useful due to the additional latency the protocol introduces. Finally attackers are likely to be more interested in targeting SSL/TLS based applications to begin with. For these reasons we took the approach of trying to isolate the best pair of sent and received packets for RTT measurement based on TCP metadata and statistical metrics alone.

Our current algorithm for isolating the packets most closely associated with the critical computation starts by measuring the delta and variability of the pair-wise RTT differences. The delta is measured using our L -estimator while the variability is measured using the median absolute deviation. From there, we progressively “trim” the leading data packets from the sample. For instance, in the first iteration we re-measure all round-trip times starting from the second sent data packet and ending at the last received data packet, effectively

ignoring the sent time of the first sent packet. We calculate the delta and variability for this new candidate distribution. If the second sent packet occurs prior to the critical computation, then we would expect the measured delta to change very little, while the variability should drop slightly. However, if the second sent packet occurs after the critical computation, then we would expect the delta to drop significantly (to near 0). Based on this expectation, we currently use the heuristic: if the delta increases or drops by no more than 15% *and* the measured variability decreases, then we accept the new candidate distribution as a valid one.

We repeat this process over the series of sent data packets, stopping at the last one as our latest starting point for measuring RTT. Then this process is repeated again on received packets, starting from the latest received packet and working forwards. In each case, as soon as a candidate distribution appears to be unacceptable, then the process stops and further combinations are not considered.

We regard this current algorithm as being very conservative and there is room for improvement. The current variability and delta estimators may not be entirely reliable. In the future we hope to revisit this approach and perhaps use a trained delta estimator to more accurately trim leading and trailing packets from the RTT time frame. We would also like to investigate measuring RTTs strictly between two received packets (we always measure the beginning of the RTT from at least one sent packet in the current implementation).

4.3 TCP Timestamps

4.3.1 Measurement Difficulties

The TCP timestamp option is defined in RFC 1323 and is designed to support algorithms that improve performance. When a host supports this option, every packet is labeled with the timestamp option which contains (amongst other items) a “TSval” field. TSval contains the current time of the sending host. For this reason, it is attractive to try and use this timestamp to estimate the run-time of a server-side operation, since this value would not be affected by most network jitter. However, in practice there are a number of problems with using TCP timestamps.

First and foremost is the fact that the majority of operating systems use a TCP times-

tamp clock resolution in the range of 1ms to 5ms, while many of the operations an attacker would want to measure have time differences that are hundreds or thousands of times smaller than that. In addition, the clocks used for TCP timestamps by operating system kernels typically rely on real-time clocks that are not corrected for any hardware clock skew, which means different systems with the exact same software would not have precisely the same TCP timestamp clock rate.

Further complications stem from the fact that TCP timestamps values are measured not from the current time of day, but some arbitrary starting point. In many operating systems, all TCP timestamps start from 0 when the system is first booted and continually increment thereafter. However, in at least some BSD systems, the starting point for timestamps on each TCP connection is randomized. That is, timestamps within a given TCP connection increment normally as expected, but subsequent TCP connections will start with an unpredictable initial timestamp. Note, however, that even on these BSD systems the clock frequency of the TSval field remains the same on all TCP connections.

4.3.2 Estimating Clock Precision

In order to use the TSval field, we first must find out what the clock precision is for the target host. That is, we want to answer the question: how many nanoseconds pass between each TSval clock increment? To obtain this information empirically on a per-host basis, we need to obtain a large number of packets sent to us from the target and measure the arrival times. While the arrival times may vary due to network jitter, with enough arrival times and associated TSval information, we should be able to estimate how often the TSval field increments.

Our implementation starts by sending a crafted HTTP request to the target. The HTTP headers in this request are intentionally “trickled” slowly to the target. For a large portion of the data sent, small delays (on the order of one tenth of a second) are inserted between each byte or each line. Since these delays are much larger than the TCP RTT, the target is forced to send a separate ACK message to confirm each small bit of packet data. Meanwhile, our packet sniffer is recording metadata on all received packets. We then repeat this process, sending multiple trickled HTTP requests to the target and log all associated data.

Once the requests are complete, we analyze the received packets. Each TCP connection is analyzed independently (so as to function even against BSD-based systems). The TSval fields and arrival time are organized into a sequence of two dimensional data points and an ordinary least square regression is performed. This gives us a simple linear approximation of the rate at which the TSval field increments for each connection.

After applying this regression to each TCP connection, the resulting slopes of all connections are analyzed. In some cases we have found that some public websites use load balancers in a way that causes the TCP timestamp results to be erratic from one connection to the next. This could be because different operating systems are load balanced behind a single IP address. For this reason we apply a simple heuristic and warn the user if the standard deviation of the regression slope values is greater than 5% of the mean. (In most cases where load balancing isn't in use, we find the standard deviation to be far less than 1% of the mean). Assuming the results appear reliable, the mean value is taken as the TSval clock rate.

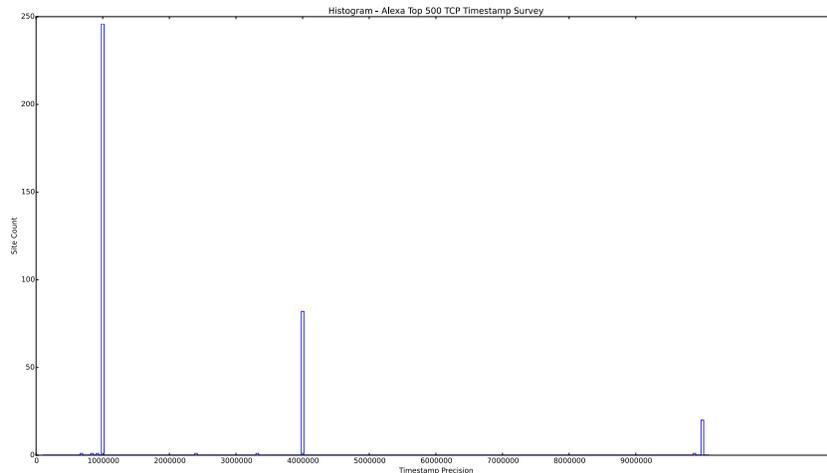
4.3.3 Survey of Support

After developing a the method for measuring TSval clock precision, we decided to find out how common TCP timestamp support is on Internet-facing hosts. It may be that many major sites disable it, or we may find TSval clock rates we didn't expect. We surveyed the Alexa top 500 web sites³ and found that 2.2% of them were down, 26.6% did not support TCP timestamps, and a further 0.2% returned inconsistent timestamp precision measurements. Of the remaining 71% of sites, the measurements fell into a few very distinct categories, as shown in Figure 14.

This makes it clear that just a few timestamp precision levels dominate most high-end web sites. Of those that support timestamps to begin with, nearly half use a 1ms precision (which implies that many of these these likely use BSD-based load balancers). The next most common case is 4ms precision, which is indicative of Linux hosts. Overall there are no surprises here, given that these are some of the most popular web sites.

³<http://www.alexa.com/topsites>

Figure 14: Alexa Top 500 Timestamp Survey



4.3.4 Measuring Round-Trip Times

When measuring the round-trip times with packet arrival times, we have been measuring the time difference between a specific sent packet and a specific received packet. However, in the case of the TSval field, we are only interested in the values sent by the target website, which requires us to measure from a different starting point. In the specific case of TCP timestamp RTTs, we first isolate the sent packet we are using for the packet RTT measurements, and then locate the received packet that ACK-ed this sent packet. This allows us to observe the time when our request was received at the server, based on the server’s clock.

5 Experimental Methods

5.1 Scenarios

Throughout the development of Nanown, a number of timing attack scenarios have been tested. However, for the purpose of comparing the statistical classifiers, we settled on 4 separate scenarios, all of which use a custom “delayed echo” web server. This web server is a simple Python-based script which responds to HTTP requests. It accepts a URL parameter

specifying the number of nanoseconds the script should pause. The script uses a busy-wait loop which repeatedly checks a monotonic clock to determine if enough time has passed.⁴ After waiting (approximately) the specified amount of time, the service responds with a trivial response body containing the number of nanoseconds it believed it waited. This information is used during testing to ensure the collected data sets do appear to provide the desired delta.

The echo service was installed on four separate hosts at different network distances away from a single simulated attacking system.

Table 1: Network Scenarios

Name	Type	OS	Network Hops	Approx. Latency (ms)	TSval Precision (ms)
lnx	physical	Linux 3.16	1	0.25	4.00
vm	Qemu VM	Linux 3.16	2	12.00	4.00
vps	Linode VM	Linux 4.0	12	31.00	3.33
bsd	physical	FreeBSD 10.1	13	84.00	1.00

Here we have two hosts located a number of hops away over the Internet, and two on the local network. One host on the LAN is a VM and one remote host is a VM, which makes for a balanced comparison. None of the hosts tested were dedicated to this research. All systems and local network equipment have other day-to-day tasks to perform, which we hope makes the data more realistic than using a sterile lab environment.

5.2 Choosing Deltas and Sample Quantity

Early in our research we conducted some basic tests to try and understand the typical time execution time of specific operations that we expect could produce timing differences in vulnerable applications. The following operations were executed under Python 3.4 in a controlled environment. A distribution of timing measurements were obtained from within the script using a real-time clock and the central tendency estimated using order statistics. See Table 2.

⁴Prior researchers have indicated this is the most accurate way to pause for designated amount of time in userspace.

Table 2: Approximate Execution Time

Task	Approx. Execution Time (ns)
MD5 of an 8 byte string	6150
Parse trivial JSON string	13700
Parse moderately complex JSON string	29300
Parse a 2 parameter HTTP query string	42900
SQLite memory SELECT	27800
SQLite on-disk SELECT	45500
Open and read (but not close) a 1 byte file	83000

Note that we did not include any string comparison operations in our research. This was intentional, since previous researchers have made it clear that measuring these remotely is almost certainly impractical in the vast majority of cases.

Considering the above time frames, it is certainly the case the Python is not the most efficient language for interacting with external libraries and I/O. However, these are all very trivial “toy” operations and any real-world application that needed to conditionally interact with a database or the filesystem is likely to perform more complex logic. These “ballpark” time measurements led us to settle on 5000ns as a reasonable time frame as a starting point for attacks. That is, if we can measure time differences of this size in a remote application, then most conditional timing side-channel leaks should be exploitable. From there, we expanded our set of test deltas up and down by factors of five, arriving at the set: 40ns, 200ns, 1000ns, 5000ns, and 25000ns.

Next, it was important to understand how many samples an attacker would consider feasible to obtain in an exploit. Based on collection rates and typical attack scenarios, how many samples should an attacker find reasonable to collect for each time-distinguishing operation? Let us perform some simple thought experiments in a couple of attack scenarios: a CBC mode padding oracle attack against a URL token, and brute forcing user information in a knowledge-based authentication page.

Suppose we identify a timing difference during decryption of a CBC-encrypted URL token. This could indicate that the programmer used MAC-then-encrypt construction (unsafe), rather than encrypt-then-MAC construction. By measuring timing differences in the

decryption and verification operations, we may be able to decrypt the token. In a padding oracle attack, one needs to perform an average of 128 tests per byte decrypted. Suppose we have a 128 byte ciphertext (which is a reasonable length given a MAC may be embedded in it). Then decrypting this token would require $128 \times 128 \times N_o$ requests, where N_o is the number of observations required accurately measure the timing difference. Let us assume an attacker can send 100 HTTP requests per second (perhaps optimistic over the Internet without alarming the target administrators). If $N_o = 100$, and we need 2 requests per sample (two test cases), then this works out to $128 \times 128 \times 100 \times 2 = 3,276,800$ HTTP requests and just over 9 hours to complete. This is completely plausible. However, if $N_o = 20000$, then it would require just under 76 days to complete, which seems pretty unreasonable for the typical attacker.

Consider a separate scenario where an attacker wishes to guess the personal details of a user through a knowledge-based authentication form, such as the real-world example described earlier in this paper. If every form field exhibited the same subsequent timing difference (that is, upon guessing an SSN successfully, the first name field can be guessed using the same timing delta, then the the last name field, and so on), then we just need to estimate the number of guesses required on each field and add them up. Perhaps the target database has a large number of SSNs inside, allowing us to send only 10,000 guesses to identify a valid one. Next, each of the name fields might require 1000 guesses to have a high level of success at identifying the correct name (since many names are re-used). Next, a the birth date would need to be guessed, which might require only 100 attempts given information from the SSN value. Lastly, the zip code would need to be guessed. There are over 40,000 zip codes in the US, but given SSN information, let us assume it requires only 5000 guesses before identifying the correct one. Putting this together with 100 samples per timing difference, it might require about $(10000+1000+1000+100+5000) \times 100 \times 2 = 3.42e6$ requests and 9.5 hours to complete. If $N_o = 20,000$, this jumps to 79 days.

In summary, we chose to limit the number of observations for tests to a maximum of 20,000. This isn't really practical in most real-world attacks, but at least gives us some sense as to what could work if only small improvements were made to statistical or data collection efficiency.

5.3 Monte Carlo Procedure

Monte Carlo methods were used to systematically compare the performance of the different classifiers discussed above. To that end, for each of the network scenarios, three data sets were collected for $\delta \in \{40, 200, 1000, 5000, 25000\}$, the artificially induced execution time between the short and long messages. For each δ , training (50,000 pairs), control (100,000 pairs), and test data sets (100,000 pairs) were collected. Overall, up to 1.25 million pairs of messages were collected for each network scenario.

Given these data sets, the Monte Carlo procedure consisted of taking the following steps.

1. Train the different classifiers on subsets of the training and control data sets. That is, take a random subset (of different lengths) of the training and control data sets, then calculate the error rates for each of the methods across a range of parameter values. Repeat this many times, saving the error rates for each iteration and classifier-parameter combination.
2. From the training data, identify the best (or a set of best) performing classifier-parameter combinations.
3. Using the set of best performing classifiers, select a random subset of the test data and calculate error rates. Repeat this many times and save the results.
4. Compare the performance of the classifiers.

Using this procedure, the objective was to identify the best performing classifier across a range of network scenarios. Broadly speaking, analogous approaches are used in the statistical and machine learning literatures to compare different estimation methods. However, because we are interested in an efficient implementation that requires as few samples and as little time as possible in real-world cases, we had to extend the procedure to account for the data-time complexity of the problem. Further details are provided in the sections below.

5.4 Training and Testing

With the exception of the Kalman filter, all of the training algorithms discussed in previous sections use a Monte Carlo bootstrapping process to measure performance and adjust

parameters as needed. We found that the amount of data (the length of the sub-series) provided to each algorithm greatly influenced how well-trained the algorithm became. If a small number of observations are provided to a training algorithm in each iteration of the bootstrapping, the parameters learned tend to be poor-performing as there was not enough information available to find good parameters. However, if a large number of samples are provided, the classifier tended to be over-trained, honing in on local minimal that didn't generalize well from the training data set to the test data set.

For this reason we chose to train each classifier on groups of observations that varied widely in size. We start with just 10 observations and increase this size by a factor of 1.5 repeatedly until the training algorithm achieves less than 5% error, or until reaching a maximum of 10,000 observations in size. This results in an array of up to 19 different sets of learned parameters for each classifier. While the training algorithms for each classifier may not be perfect, each classifier has many different chances at finding the global optimum.

The subsequent test phase attempts to determine which classifier/parameters combination can successfully classify the test data at a 5% overall error rate with the *fewest* number of observations. (This is the information a penetration tester will want to know, and simply testing for which classifier has the lowest error at a given number of observations doesn't directly address this question.) In order to achieve, this we use the following procedure for each of the classifier/parameters candidates:

1. Test the candidate on the same number of samples that were used during training to obtain an overall error rate.
2. Use a heuristic to estimate how many more samples might be required to reach an error rate less than 5%.
3. Repeat steps 1 and 2 until the the error rate is less than 5%, or until the maximum observation set size (20,000) is reached. If the error rate is above 5% at 20,000 observations, exit.
4. Reduce the number of observations by 5% and re-test the candidate.
5. If the error rate is still below 5%, repeat step 4. Otherwise, return the number of

observations that last had an error rate below 5%.

After this process is complete on all candidates, the candidate that successfully obtained less than 5% error and had the fewest number of samples required is considered the “winner” on that data set.

6 Results

Table 3 presents the overall results of the Monte Carlo experiment comparing the different classifiers discussed in previous sections. Each cell reports the results for a particular network scenario and delta value. In the cell, when the classifier was able to achieve a $< 5\%$ total error rate (i.e., including false positives and false negatives) in fewer than 20,000 observations, the number of observations used to achieve this is reported. When an error rate below 5% was not achieved using a particular classifier, the lowest error rate is reported (as a %). The best performing classifier is highlighted in bold for each case.

Several things are clear from these results. First, none of these classifiers works well with deltas below 1000ns. It is likely that for deltas this small, the number of observations needed to reach a reasonable error rate is well above 20,000. The amount of noise in the network data is simply too large relative to the signal. Second, the box test doesn’t perform all that well. In only one case (5000ns over the VM) did it have the best performance. In fact, in most cases (13 of 17 cases), the box test performed worse than all other classifiers. Third, the best performing classifier was the midsummary, which either required the fewest observations to achieve an error rate below 5% or, when that level of error was not achievable, had the lowest overall in 10 of 17 the cases.

In comparing the performance of the three L -estimators, it seems that when the classification task is easier, due to a high signal to noise ratio, then the estimators with more order statistics (septasummary or quadsummary) are the most efficient. However, as the classification task becomes difficult, the midsummary tends to edge out the other estimators. This makes sense from a robust statistics stand point. Consider a degenerate case of an L -estimator where 1000 different order statistics, each at a different location, are measured across a data set and then the mean is taken. If the data set has 1000 data points, then

Table 3: Results: Reported observations indicate that method achieved $< 5\%$ error rate in that size of sample. A percentage error indicates the lowest error rate by that test with fewer than 20,000 samples.

Scenario	Classifier	Delta (ns)				
		25000	5000	1000	200	40
lnx						
	midsummary	29 obs	894 obs	17147 obs	16.60% err	38.60% err
	quadsummary	26 obs	894 obs	16289 obs	20.55% err	47.30% err
	septasummary	15 obs	894 obs	17147 obs	22.35% err	45.20% err
	boxtest	146 obs	20.80% err	36.30% err	47.55% err	49.85% err
vm						
	midsummary	242 obs	10898 obs	15789 obs	19.45% err	23.05% err
	quadsummary	344 obs	10583 obs	8.30% err	18.40% err	30.05% err
	septasummary	356 obs	9706 obs	8.30% err	22.40% err	31.10% err
	boxtest	615 obs	7909 obs	7.50% err	47.00% err	36.00% err
vps						
	midsummary	21.80% err	31.80% err	19.00% err	33.10% err	35.85% err
	quadsummary	32.75% err	31.55% err	34.95% err	32.25% err	37.35% err
	septasummary	22.40% err	43.50% err	30.05% err	46.55% err	36.70% err
	boxtest	48.15% err	39.70% err	41.00% err	46.70% err	44.75% err
bsd						
	midsummary	21.30% err	21.80% err			
	quadsummary	22.35% err	28.65% err			
	septasummary	27.65% err	18.00% err			
	boxtest	24.35% err	46.80% err			

this calculation would be nearly equivalent to taking the mean of the data point values and would therefore be just as susceptible to outliers in the data as the mean is. So as we add more order statistics to an estimator, we can more quickly estimate the location of a clean data set, but may perform worse on one containing more noise.

7 Conclusion

This paper has looked at the issues associated with identifying timing side-channel vulnerabilities in web applications. The chief impediment to determining whether a timing attack is viable is overcoming the low signal to noise ratio that exists over network connections and under variable host conditions. To address this problem, we have developed and tested novel round-trip time measurement methods which include packet sniffing and a paired request sampling method. We went on to provide a thorough test of several different classifiers, which target the difference in execution times between request types. These tests show that the current state of the art in detecting execution time differences—the box text—does not perform particularly well when compared to other methods, namely the midsummary estimator, which performs significantly better. Finally, we have developed a tool, Nanown, that incorporates our methodology, making it accessible to other researchers and penetration testers.

7.1 Avoiding Timing Flaws

While this research has focused on improving identification and assessment of remote timing weaknesses, it is worth discussing approaches for prevention and mitigation. Here we list a few recommendations for helping reduce the risk of these attacks: It is sometimes the case that applications expose the application runtime to clients, either through HTTP headers, HTML comments, or in other locations as a way to help developers analyze application performance. As a basic first mitigation step, these metrics should be removed from responses in production environments, since they can making timing attacks much easier.

In situations where a security critical interface is exposed to users, one possible mitigation is to use a CAPTCHA challenge to block scripted attacks. So long as the CAPTCHA is secure and well implemented, then any timing discrepancies would likely become unexploitable due to the difficulty of collecting many samples.

Finally, the most effective universal fix for timing attacks is to simply remove the timing leaks from all security-critical operations. While this sounds simple, it is often difficult to do in practice. In some cases applications must process one piece of information before

another due to design constraints. In other cases, an application may appear to have no timing difference in the code, but after compilation, the application may exhibit a timing difference due to compiler optimizations. (Indeed, while testing an application in Java, simple changes that didn't appear to affect the critical computation did indeed change the timing difference in unexpected ways. We believe this may have occurred due to instruction reordering.) For this reason, any attempts to eliminate or mask a timing difference must be tested empirically. Until recently, there were few tools available to help with this, but we hope Nanown will fill this gap in developers' and penetration testers' tool kits.

For additional information on ways to prevent application timing differences, see the detailed recommendations in (Mayer and Sandin 2014b).

7.2 Future Work

We leave the following thoughts on areas where additional techniques could be explored. Upon request, we are happy to share our data sets for further research and analysis.

7.2.1 Explore More Statistical Methods

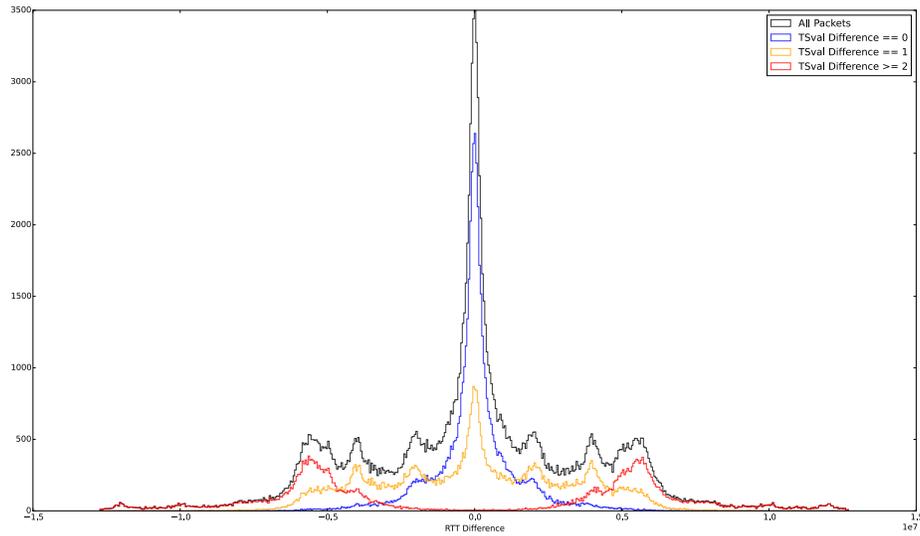
Early in our analysis, Kalman filters worked well in a preliminary data set. We subsequently spent countless hours trying to make this signal processing technique work on our problem. We learned that there are many different ways to apply and tune Kalman filters to a data set and that this process can be more art than science. We still believe this approach may yield a strong result and should be pursued. In addition, it is worth considering alternative L -estimators as well as robust M -estimators which can be more efficient.

7.2.2 Combine TCP Timestamps with Packet RTT Data

While directly using the TCP timestamp data does not seem to be effective, there may be some information contained in these timestamps that could assist a more sophisticated algorithm in dealing with host and network noise. Consider Figure 15 where we have partitioned the VPS data set into groups based on the timestamp data. There are only a limited number of discrete values recorded for each TSval RTT. If the two test cases exhibited the same

TSval RTT, then they have a 0 difference and are included in the blue histogram below. Likewise, a ± 1 difference in the TSval RTT groups them into the orange histogram.

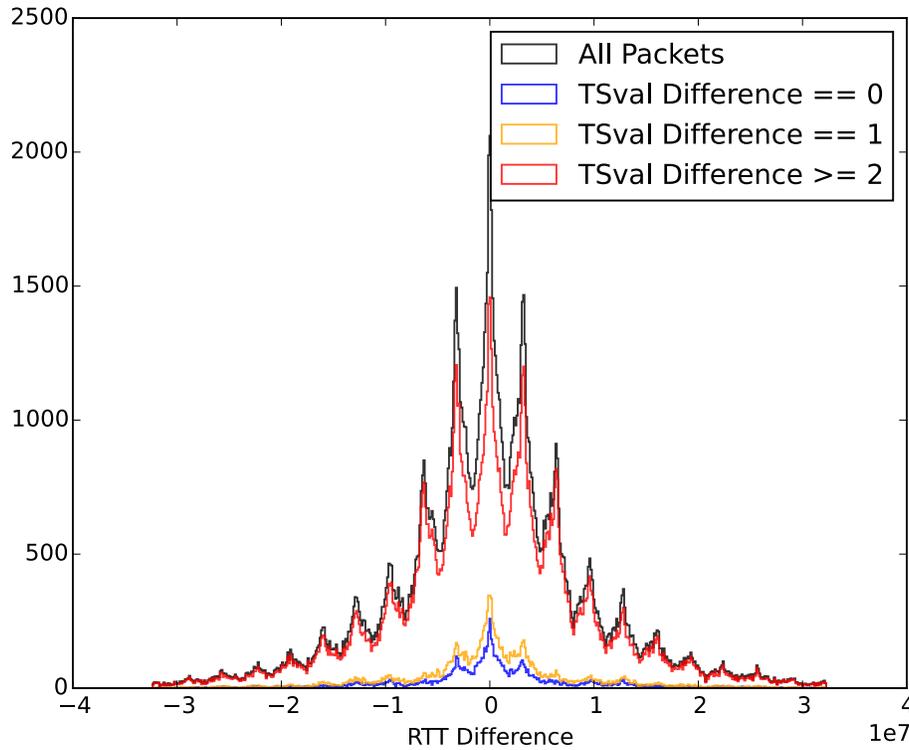
Figure 15: TSval Partitioning - “vps” scenario



Here we can see that the 0 difference data has much lower variance and a very clear signal in the center. As the TSval difference grows, the distribution becomes much wider and there is less and less clear signal in the center. This variance is almost certainly generated by inconsistencies in response time by the host itself, since TSvals are generated there. Based on this one result, it seems like TSvals could provide a very powerful tool for triaging the data set and focusing on the more trustworthy information. However, if we apply this technique to the BSD system scenario, we get a very different result, as shown in Figure 16.

Here it seems the most noisy data, that with a TSval difference of 2 or greater, looks almost identical to the whole data set. The data we would expect to be more trustworthy (0 and 1 differences) also exhibits the profile of the whole data set, though much smaller. It doesn't appear as if TSvals would help us weed out noisier data in this case. Clearly, further research is needed to determine if and how the TSval information could augment the collection of packet data overall.

Figure 16: Tsval Partitioning - “bsd” scenario



7.2.3 Explore Request Rate Trade-Offs

This research focused largely on finding ways to minimize the number of samples required to reliably detect a timing difference in a remote application. For an attacker, however, the goal is often to minimize the time required to exploit a flaw. The Nanown data collection phase is not particularly fast and it could easily be optimized to send HTTP requests at a much higher rate. However, in doing so it is possible that the quality of the collected samples will diminish due to possible network congestion and resource limitations on the target system. Some reduction in data quality may be acceptable. For instance, in a particular attack if lower quality RTT measurements cause us to require twice as many samples to classify a timing difference, but our request rate is ten times greater, then the attack overall would be five times faster.

As an area of future research, these attacks could be made much more practical if, for a given target system, the degradation in data quality could be accurately measured as the

request rate is increased. Ideally, a “sweet spot” would be identified that minimizes overall attack time as a function of this request rate.

8 Appendix

8.1 Glossary

Central Tendency the central or “typical” value for a probability distribution; commonly used central tendency measures include the mean, median, and mode, though many others exist

Critical Computation the portion of the target application’s computation that we wish to measure a timing difference in

Delta the time difference between critical computations, as produced by different test cases

RTT round-trip time

Sample a collection of individual round-trip time measurements probes, one for each test case; at least two test case probes exist in each sample

Test Case a type of request sent to the application to elicit a specific type of behavior in the critical computation

Unusual Case the test case which appears to be most different in response time than the others; deltas are measured between this test case and all others

References

- AlFardan, Nadhem J., Kenneth G. Paterson, and Royal Holloway (2013). *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. IEEE Symposium on Security and Privacy, <http://www.isg.rhul.ac.uk/tls/TLStiming.pdf>.
- Bernstein, Daniel J. (2005). *Cache-Timing Attacks on AES*. Tech. rep. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>. Department of Mathematics, Statistics, and Computer Science, The University of Illinois at Chicago.
- Crosby, Scott A., Dan S. Wallach, and Rudolf H. Riedi (2009). “Opportunities and Limits of Remote Timing Attacks”. In: *ACM Transactions on Information and System Security* 12.3. <http://www.cs.rice.edu/~dwallach/pub/crosby-timing2009.pdf>.
- Kalman, Rudolph Emil (1960). “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME—Journal of Basic Engineering* 82.Series D, pp. 35–45.
- Lawson, Nate and Taylor Nelson (2010). *Exploiting Remote Timing Attacks*. Research Presented at Blackhat USA 2010, <https://www.youtube.com/watch?v=hVXP8git7A4>.
- Mayer, Daniel A. and Joel Sandin (2014a). *Time Trial*. Open Source Tool Presented at Blackhat USA 2014, https://github.com/dmayer/time_trial.
- (2014b). *Time Trial: Racing Towards Practical Remote Timing Attacks*. Research Presented at Blackhat USA 2014, <https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf>.
- Morgan, Timothy D. (2015). *Nanown*. Open Source Tool Presented at Blackhat USA 2015, <https://github.com/ecbftw/nanown>.
- Percival, Colin (2005). *Cache Missing for Fun and Profit*. Research Presented at BSDCan 2005, <http://www.daemonology.net/papers/htt.pdf>.