Time Trial

Racing Towards Practical Remote Timing Attacks

Daniel Mayer (daniel@matasano.com)
Joel Sandin (jsandin@matasano.com)

August 7, 2014

Abstract:
Attacks on software become increasingly sophisticated over time and while the community has a good understanding of many classes of vulnerabilities that are commonly exploited, the practical relevance of side-channel attacks is much less understood.

One common side-channel vulnerability that is present in many applications today are timing side-channels which allow an attacker to extract information based on different response times. These side-channel vulnerabilities are easily introduced wherever sensitive values such as credentials or API keys are processed before responding to a client. Even though there is basic awareness of timing side-channel attacks in the community, they often go unnoticed or are flagged during code audits without a true understanding of their exploitability in practice.

In this paper, we provide both a tool 'time trial' and guidance on the detection and exploitability of timing side-channel vulnerabilities in common application scenarios. Specifically, the focus of this paper is on remote timing attacks, which are performed over a LAN, in a cloud environment, or on the Internet. To illustrate this, we first present experimental timing results that demonstrate how precisely timing can be measured and, more importantly, which timing differences can be distinguished remotely. Second, we compare our results with timing differences that are typically encountered in modern web frameworks and servers. The discussed attack scenarios include database queries, message authentication codes, web API keys, OAuth tokens, and login functions.

This paper has significance for a wide spectrum of the conference audience. Readers in defensive security roles will gain a better understanding of the threat timing side-channel vulnerabilities pose and, based on the demonstrated attacks, will be better able to evaluate the severity and impact of a successful side-channel attack. Readers in a penetration testing role will learn how to distinguish theoretical timing attacks from legitimately exploitable flaws by using our tool 'time trial'. Finally, readers focused on research implications will receive a comprehensive update on the state-of-the-art in exploiting timing attacks in practice.

0 Contents

1 introduction

Software can be affected by a wide array of security vulnerabilities. Many of these, such as memory corruption flaws [DMS06] and web application flaws [SP11, Zal12] have been studied in depth and are well understood by the information security community. However, even if systems do not present any direct flaws, they often remain susceptible to side-channel attacks. These attacks exploit secondary information from a system – things like execution time, power consumption, and Radio Frequency (RF) emissions – to increase the effectiveness of attacks.

One common side-channel vulnerability that is present in many web or other network applications today are timing side-channels which allow an attacker to extract information about a sensitive credential from a system based on differences in the response time. In this white-paper, we provide both guidance on the detection and exploitability of timing side-channel vulnerabilities in common web application scenarios and a tool 'time trial'. Specifically, the focus is on remote timing attacks, which are performed over a LAN, in a cloud environment, or over the Internet.

Outline: Below, we first introduce the topic of timing side-channels in more depth. In Section 2 we then present several timing side-channels with examples. Section 3 describes the process of performing a timing attack in more detail and Section 4 introduces our new tool *time trial*. Finally, Sections 5 and 6 present our results on using *time trial* in order to determine whether a system is vulnerable to timing attacks and how to defend against them respectively.

1.1 SIDE-CHANNEL ATTACKS

When exploiting a side-channel, the adversary does not attack an explicit flaw but is able to leverage information that is exposed due to specific implementation details or even physical characteristics of the overall system. Typically, the goal is to deduce some information about a secret kept by the target system. Since cryptographic systems generally operate on such secrets (the keys) it comes as no surprise that side-channel attacks have frequently been developed in order to break cryptographic systems.

Despite a significant volume of (mostly) academic research, compared to traditional software vulnerabilities, the practical relevance of side-channel attacks is much less understood by the community. Examples of side-channel attacks include power analysis (e.g. [KJJ99, KJJR11]), differential fault analysis (e.g., [BDL97, Gir05]), acoustic analysis (e.g. [GST13]), involuntary transmission of electromagnetic radiation (so-called *Van Eck phreaking* [vE85]), and timing side-channels—the focus of this paper.

1.2 TIMING SIDE-CHANNEL

As mentioned, timing side channels are prevalent in modern web applications. Using a timing side-channel, one can deduce some information about the inner workings of the application based on its response time. Let us look at a basic example to illustrate this. Listing 1 shows a simple login

LISTING 1: BASIC TIMING SIDE-CHANNEL IN RUBY SINATRA

```
post '/login' do
  if not valid_user?(params[:user])
      "Username or Password incorrect"
  else
      if verify_password(params[:user], params[:password])
        "Access granted"
      else
        "Username or Password incorrect"
      end
  end
end
```

function written in Ruby and Sinatra which takes a `user` and `password` as parameters. To check if valid credentials have been submitted, the code first checks if the username is valid and only then verifies the password in a second step. Note that the returned error message does not give an indication on whether the supplied username was valid. However, by measuring the response time on such a system, we can distinguish between requests that triggered the password validation function and requests that did not. As a result, we are in a position to enumerate valid users of this application. We will look at this scenario in more detail later in Sections 2 and 5.

1.3 PREVIOUS WORK

The work in this paper builds upon the large body of prior work on timing side-channels. Timing attacks have been studied by both academic and industry security researchers. Traditionally, the academic side focused on (local) timing attacks against cryptographic systems. Notable early results include Paul Kocher's timing attack research [Koc96], as well as the David Brumley and Dan Boneh 2003 attack on OpenSSL over a LAN resulting in successful private key recovery [BB03]. More recently, Crosby et al. [CWR09] did a comprehensive study on statistical methods that can aid in mounting successful remote attacks. Empirical results include the talk by Nate Lawson and Taylor Nelson presented at BlackHat 2010 [LN10] as well as Sebastian Schinzel's presentations at CCC [Sch11, Sch12]. Schinzel also developed a tool for attacking systems with timing side-channels, but it is not publicly available. Our work builds on these empirical studies.

2 timing-based vulnerabilities

Every programming language provides operations that demand caution when used. Some examples of common tasks that have developed a well-deserved reputation for being dangerous when used incorrectly are copying memory, construction of database queries, and parsing of XML data. In contrast, timing side-channels may even be introduced when a developer uses what can rightfully be seen as "innocent" operators such as branching and string comparisons. The subtlety of their manifestation may, in part, explain their prevalence in software products today.

The following discussion presents settings where timing side-channels are commonly introduced by developers. This section is a purely theoretical source-code analysis and we do not imply that each and every instance is exploitable by a remote attacker over the internet. We defer this analysis to Section 5.

Below, the different timing flaws are grouped by the vulnerable underlying operation and each section is accompanied by one or more modern code samples pulled from publicly available sources that illustrate actual vulnerabilities. We canonicalized the examples as to not publicly expose particular projects. The ease in which these examples can be found in both classic and new languages reflects the lack of general awareness among developers. In some cases, these examples are included in publicly available tutorials or up-voted forum posts that may have served as templates for other implementations. A discussion on how to remediate or limit the impact of timing flaws is given in Section 6.

2.1 BRANCHING

We already have seen timing vulnerabilities based on branching in our introductory example in Listing 1. The basic idea is that the application follows a different major code path based on a secret value. Depending on the application, this flaw may occur in various different ways. Below we discuss two of the more general kind: authentication and timing-based padding oracles.

2.1.1 Authentication

In the login example discussed in the Section 1, the verification of the password only occurs if the supplied username is known to the system. Since the verification will involve additional computation time, this introduces a measurable timing difference.

Figure 1 illustrates this behavior. Here the attacker sends a request at the time marked with "Start". The target then processes the request and either returns immediately after checking the username (time t_0) or after the password was verified (time t_1). Note that even if the application returns the same error message, we can distinguish the different execution paths due to the difference in response time.

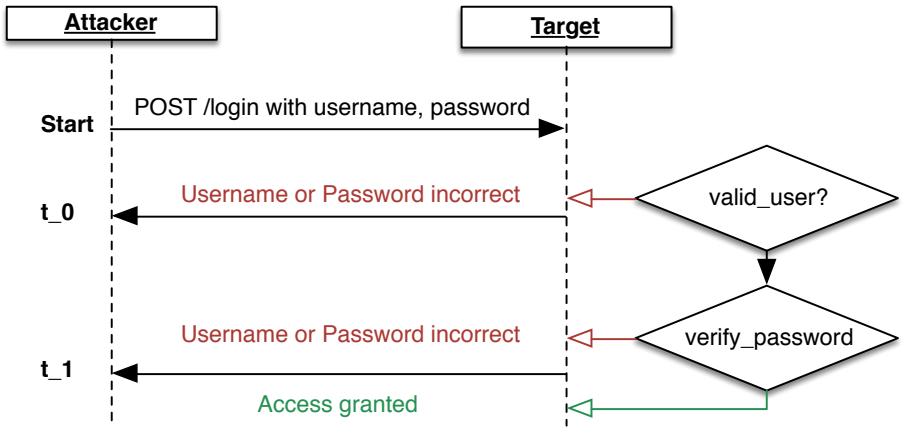


FIGURE 1: Illustration of timing differences for user authentication.

2.1.2 Padding Oracles

A padding oracle [Vau02, PY04] is a well-studied cryptographic vulnerability that, even without considering timing, is considered a side-channel flaw. When encrypting data using a block cipher, for most cipher modes of operation, e.g. CBC mode (see [Wik14b]), the plaintext has to be a multiple of the block size. Since this is not always naturally the case, the plaintext in the last block has to be padded such that it fulfills this requirement. There are different schemes that can be applied but typically *PKCS#5* or *PKCS#7* is used [Hou09, Wik14c]. When decrypting, most libraries verify that the last block had correct padding according the agreed-upon padding scheme. If this is not the case, a padding error is raised.

If an attacker is able to learn whether a submitted ciphertext had correct padding, i.e., if the application communicates padding errors to the attacker, they are in the position to decrypt arbitrary ciphertexts without knowing the secret key. A full treatment of padding oracles is outside the scope of this paper but [RD10a] and [Hol10] give a detailed review of the subject from an applied perspective. In the past, many (prominent) software and hardware implementations were affected by it, e.g., [RD10b, DR11, BFK+12].

Even when the application does not explicitly disclose whether the padding was correct (see Listing 2), timing side-channels may still allow an attacker to learn this information [San10] as illustrated in Figure 2. When a padding error occurs during decryption, the application typically does not process the decrypted data. Instead it is likely to return a (generic) error message (t_0). If the padding is correct, regular processing of the request will occur (t_1), which itself may encounter an error if the decrypted data mangled by the attacker is malformed. Therefore, an attacker may be able to distinguish this timing difference and exploit the padding oracle.

LISTING 2: PADDING ORACLE IN RUBY

```
require 'openssl'
require 'sinatra'

get '/decrypt' do

  begin
    cipher = openssl::cipher::aes256.new(:cbc)
    cipher.decrypt
    cipher.key = $key #key is generated somehere
    plain = cipher.update(params[:data]) + cipher.final
    begin
        process_data plain
    rescue
        return "an error occurred."
    end
    return "data processed successfully."
  rescue
    return "an error occurred."
  end
end
```

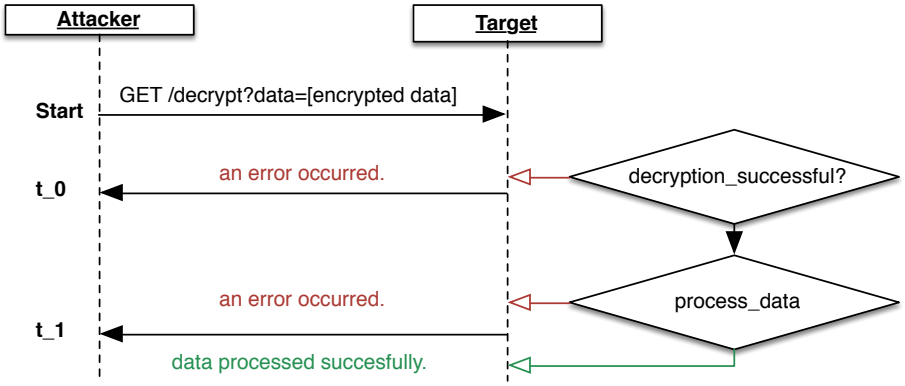


FIGURE 2: Illustration of the timing difference for a padding oracle.

2.2 STRING COMPARISON

String comparison is one of the classic examples for timing vulnerabilities. While they may seem trivial, they can have far-reaching implications depending on the values involved in the comparison. Let us first examine the basic principle behind timing flaws in string comparison. As you can see in Figure 3, string comparison functions tend to be implemented as a loop over all characters of

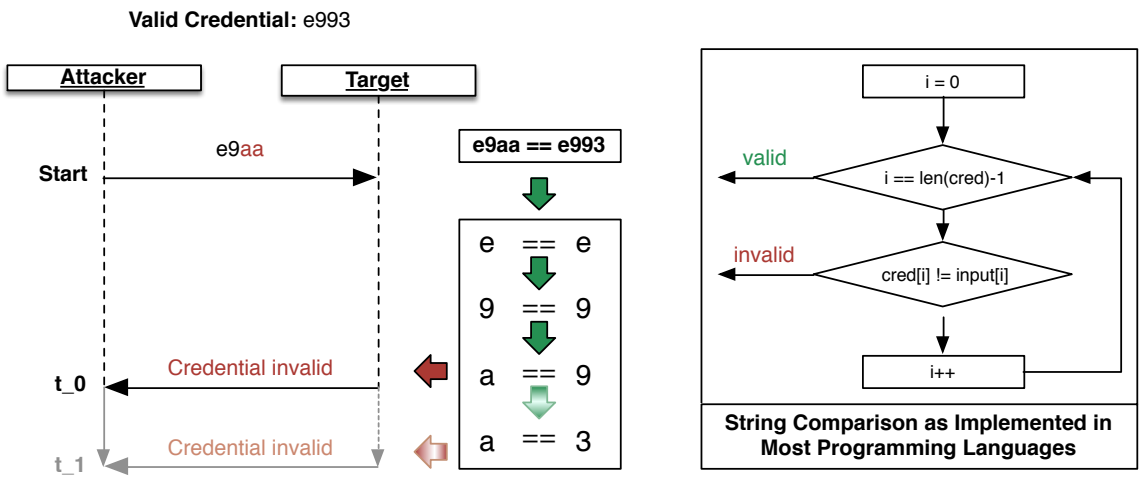


FIGURE 3: Illustration of the principle behind string-based timing attacks.

the input strings. Once the two strings differ in a character, the comparison function returns with `false`. Only if all characters are identical, is `true` returned.

By simply looking at the return value for the string comparison function, one cannot tell which character causes the mismatched, i.e., the result generally does not say "The strings are not equal. The first character they differ in is at position 3.". However, by leveraging timing measurements, one is able to learn exactly this information. In Figure 3, the target knows a valid credential of e993 (shortened for simplicity) and compares a user-supplied value to that secret value in order to authenticate the user. In the depicted example, the attacker supplies the input e9aa and the internal string comparison will proceed as shown in the center of the figure. As long as the characters in both strings are equal, the next character is compared. As soon as one differs, the function returns. Since each additional comparison takes extra time, an attacker can leverage this to brute-force the character for each position one at a time as follows:

1. Start with a string aaaaa.
2. Take the first position and iterate over all valid characters (baaaa, caaaa, etc.). Once one of the request for a certain character takes longer to process (t_1) than the others (t_0), this indicates that the character was correct and that the server performed an addition comparison for the following character (e.g., eaaaa).
3. Remember the string determined thus far, move to the next character, and repeat the process (ebaaa, ecaaa, etc.) until all characters have been determined.

What is important to note is that the attacker can brute-force a single character at a time. Compared to a regular brute-force attack this requires effort that is linear in the length of the secret instead of exponential.

Below we discuss some common scenarios which involve the comparison of sensitive string values. This includes Message Authentication Codes (MACs), OAuth tokens, API keys, and HTTP Basic Auth usernames and passwords.

2.2.1 MAC Authentication

MACs are commonly used in order to cryptographically authenticate some piece of data in a similar, yet more efficient, manner than cryptographic signatures do. The goal for applying MACs is to ensure that the corresponding piece of data has not been tampered with since only the originator is able to create the valid MAC. For this, both the server and the legitimate client share a common secret value. This secret value is required to compute a valid MAC for arbitrary data. In order to verify a MAC, the verifier computes the valid, expected MAC and compares it to the value submitted by the client. If this comparison is performed using regular string comparison, the MAC is prone to a timing attack in which the attacker is able to forge a MAC for arbitrary data. By following the approach described above, the attacker is able to construct a valid MAC byte by byte. One of the most common ways of implementing a MAC us by using Hash-Based MAC (HMAC) algorithms such as HMAC-SHA1-256.

MACs are used in a variety of contexts such as authenticating ciphertexts, proving the origin of data passed in an API request, etc. One frequent use case is to authenticate requests sent to web APIs. When MACs are used with web APIs, a common pattern is to require the client to "authenticate" their API request with an attached MAC value such that the server can verify that the request comes from a legitimate client. If the server cannot verify the MAC in the manner describe above, the server returns a 403 or otherwise indicates that the request is unauthorized. The use of a regular (early terminating) string comparison function by the server in this context introduces a timing side channel. A malicious client can adaptively infer the expected MAC value for a given request and execute this request without authorization.

A perusal of publicly available libraries quickly identifies numerous examples of this antipattern. Below, as an illustration, Listing 3 shows one example in PHP, Listing 4 for Java, and Listing 5 for Ruby.

LISTING 3: HMAC TIMING VULNERABILITY IN PHP

```
function compareHash(){
  $hash_client = hash_hmac('sha256', $string, $secret);
  if($hash_client == $this->hash_server) {
    $this->valid_hash = true;
    return true;
  } else {
    $this->valid_hash = false;
    return false;
  }
}
```

LISTING 4: HMAC TIMING VULNERABILITY IN JAVA

```
public static Boolean checkRequest(final Request request, final String secretKey) {
    final String requestSignature = request.getHeader(X_HMAC_AUTH_SIGNATURE);
    final String generatedSignature = createRequestSignature(request, secretKey);
    return generatedSignature.equals(requestSignature);
}
```

LISTING 5: HMAC TIMING VULNERABILITY IN RUBY

```
def authenticated?(request)
  rx = Regexp.new("#{@service_id} ([^:]+):(.+)$")
  if md = rx.match(authorization_header(request))
    key_id = md[1]
    hmac = md[2]
    secret = @credential_store[key_id]
    !secret.nil? && hmac == signature(request, secret)
  else
    false
  end
end
```

Systems that authenticate session cookies using MACs may fall victim to the same vulnerabilities given above (e.g., Listing 6).

LISTING 6: SESSION MANAGEMENT TIMING VULNERABILITY IN NODE.JS

```
    if (originalHash == crc32.signed(val)) return debug('unmodified session');
```

2.2.2 OAuth Tokens

OAuth [HL10, Har12, JH12] is an authorization standard that can be used to delegate access to a third-party resource without disclosing the main access key or password of the resource owner. This delegation is performed by means of so-called OAuth tokens which are opaque identifiers that are used by a third party in place of a user's credentials to gain authorized access to a resource. Implementors can choose to construct OAuth tokens in various ways. They often turn to HMACs, leading to vulnerabilities similar to those shown in the HMAC section. A well known vulnerability identified by Sebastien Martini [Mar] follows the anti-pattern in Listing 7 but similar flaws exist in implementations in other languages such as, for example, PHP (see Listing 8).

LISTING 7: OAUTH TIMING VULNERABILITY IN PYTHON

```
def check(self, request, consumer, token, signature):
    built = self.sign(request, consumer, token)
    return built == signature
```

LISTING 8: OAUTH TIMING VULNERABILITY IN PHP

```
public function check_signature($request, $consumer, $token, $signature) {
  $built = $this->build_signature($request, $consumer, $token);
  return $built == $signature;
}
```

2.2.3 Web API Keys

In contrast to other identifiers discussed before, API keys typically do not have a internal structure but are long, randomly generated opaque blobs unique to a given API. While they serve a similar function as passwords, due to their random structure, they are less likely to be stored and processed securely. For instance, at least one online best practices guide [Sto13] actually recommends *against* hashing API keys in the interest of performance without mentioning the security drawbacks of this approach. Similarly, side channels and secure storage of API keys aren't mentioned in the OWASP Cheat Sheet [OWA14].

As should be apparent from the discussion above, if a submitted API key is compared against a valid one using regular string comparison, they may be prone to timing attacks. Listing 9 shows a validation function for a request handler in C# ASP.net. In this example, the handler responds with a "403 Forbidden" response if the key included in the client query doesn't match the expected key.

LISTING 9: API KEY TIMING VULNERABILITY IN C# ASP.NET

```
private bool ValidateKey(HttpRequestMessage message) {
  var query = message.RequestUri.ParseQueryString();
  string key = query["key"];
  return (key == Key);
}
```

2.2.4 HTTP (Basic) Authentication Middleware

Newer "batteries not included" microframeworks (e.g., node.js, clojure, flask, sinatra) encourage users to graft in various HTTP basic auth middleware implementations, or roll their own. The example handlers below (Listings 10 to 12) are pulled from middleware documentation, and are meant to be called in order to verify basic auth credentials when handling requests. If these patterns are used directly, a timing side channel may be introduced. Note that similar patterns are easy to introduce into regular authentication functionality for systems where the userbase is small, account lockout is not implemented, and developers are unaware of the need to store passwords securely (see Section 6 for a more detailed discussion).

LISTING 10: BASIC AUTH TIMING VULNERABILITY IN RUBY SINATRA

```
authorize do |username, password|
  username == "john" && password == "doe"
end
```

LISTING 11: BASIC AUTH TIMING VULNERABILITY IN NODE.JS

```
connect.createServer(
 basicAuth(function (user, password) {
   return user === "admin" && password == "secret";
 }),

 function (req, res) {
   res.writeHead(200);
   res.end("welcome " + req.headers.remote_user);
 }
);
```

LISTING 12: BASIC AUTH TIMING VULNERABILITY IN CLOJURE

```
(defn authenticated? [name pass]
  (and (= name "foo")
       (= pass "bar")))
```

Being aware of the various instances of timing vulnerabilities in difference languages and frameworks, we discuss the details of how to perform an actual timing attack in the next section.

3 timing attacks in practice

Now that we understand the different ways in which timing attacks can be introduced into applications, we'll discuss how one would go about exploiting such timing vulnerabilities. As mentioned above, at their core timing attacks boil down to an attacker being able to distinguish server response times as a function of the input.

These attacks can be local or remote. In a local attack, the attacker either has access to a piece of hardware and can interact with it directly, or is able to execute the attack on the same physical host on which the target is running. In contrast, remote attacks are executed over a network which may be a LAN or the Internet. In this paper, we focus on remote attacks but we point out settings in which a remote attacker may take advantage of local computation.

Below, we first describe in detail how timing attacks are performed in practice. We then talk about the potential and limitations of parallelized attacks and how to conduct black-box identification of timing side-channels.

3.1 ANATOMY OF A TIMING ATTACK

In a successful timing attack, an attacker interacts with a remote server, measures the round-trip time of messages, processes these round-trip times using statistical techniques to infer the remote execution time at the target, and progressively learns the contents of a secret value one byte (or many bytes) at a time.

In this process, the attacker attempts to elicit different response times by providing carefully crafted input values to the server. The exact inputs to send depends on the attack scenario. In general, the attacker will have a set of candidate values which cover potential inputs (e.g., usernames, session tokens, signatures). The timing attack then proceeds in multiple rounds. In each round the attacker learns one additional piece of data about the secret value (e.g., validity of the username, byte of the session token or signature). To do so, the attacker iterates through all of the candidate values and selects the value that results in the longest (or shortest) execution time. Note that this process may terminate early when the desired candidate is found. In an ideal world, this question can be answered by taking a single measurement for each candidate. In real systems, however, there are multiple sources of variation– called *jitter*–involved in each measurement of time (see Section 3.1.1). For this reason, real-world attacks require multiple messages to be sent for each candidate value, and statistical analysis of the resulting response time distributions. The concrete steps involved in each round are:

1. For each candidate, multiple measurements of the round trip time are collected using a sensor able to precisely measure the elapsed time.
2. These measurements may then be *filtered*: filtering is an optional process which selects a single value (or set of values) based on the measurements, that is (are) reliably correlated with the remote execution time.

3. The (filtered) values are then evaluated using statistical hypothesis testing in order to identify whether they come from different statistical distributions, or more plainly, whether one candidate resulted in a signficantly longer or shorter execution time than the others.

There are many approaches to filtering and hypothesis testing, and thus our discussion only covers the techniques that previous empirical studies have found to be most effective. For an excellent in-depth treatment of this topic, refer to Crosby et al.'s paper [CWR09] on which large parts of this subsection are based.

3.1.1 Response Time and Jitter

When interacting with a remote target, the only variable we can directly measure is the *round trip time*. This quantity includes propagation times for both the request and the response, along with the variable we are actually interested in: the execution time used in processing the request at the target. If the propagation time remained constant for all our requests, we could simply compare two measurements and decide immediately if the execution time was different or the same. However, interaction with a remote system introduces multiple sources of variation that can cause the propagation time to differ significantly between requests. Network latency, caused by varying delays at intermediate hops on the path to the target, and even varying system loads on the target will introduce jitter into the overall response time. These distortions impede our ability to estimate the execution time at the target.

The following model of propagation developed by Crosby et al. illustrates the terms that contribute to the overall round-trip time [CWR09]:

$$response\_time = processing\_time + propagation\_time + jitter \tag{1}$$

Where the different terms are defined as follows:

response_time: The measured round trip time.

processing_time: Time the target used to process the request. Assumed constant for all requests for the same message.

propagation_time: The average latency over the network link. Assumed constant over all requests.

jitter Overall jitter.

Here the propagation time is the sum of the average request and response propagation times. The jitter term can be positive or negative and, for the purpose of our analysis, this term encapsulates the cumulative sources of variation in our measurements. The term we are actually interested in is the *processing_time* at the remote target. If this processing time is significantly longer (or shorter) for a given candidate value, we likely have identified another piece of the secret.

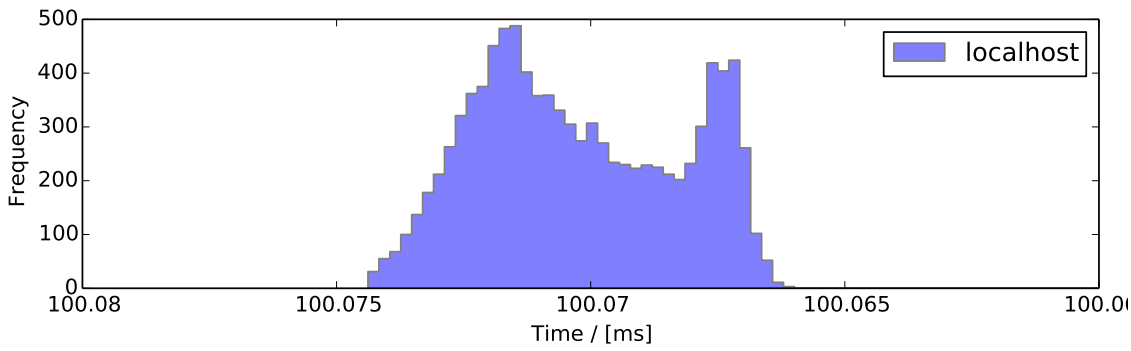


FIGURE 4: Example of a typical response time distribution (100 ms processing time) on the same host.

3.1.2 Measuring with Precision

As a prerequisite to performing a successful timing attack, we need to be able to precisely *measure* time (the response_time) on the system executing the attack. While we don't have any influence on the amount of jitter introduced by propagation and at the target, we are able to reduce the jitter added due to the measurement setup. This includes choosing a proper clock and reducing outside interference with the timing measurement as much as possible. Crosby et al. have shown that the network hardware may also have a measurable impact on the overall quality of the timing results. All these parameters should be taken into consideration.

3.1.3 Filtering

As discussed above, by performing many measurements we build a statistical picture of the jitter based on the measured response times. We can then use this data to compute a value that is (hopefully) correlated with the execution time on the target. This step, known as *filtering*, processes the measurements and produces a value or set of values that act as a "fingerprint" for the execution time. Susbequently, this fingerprint can be used in comparisons when proceeding with the last stage of the analysis.

One filtering approach would be to simply take the mean of all the measurements. The mean is only a good correlation when one is dealing with a distribution that is symmetric around the mean value (e.g. Gaussians/Normal distributions). While this may work for local attacks where no network propagation is at play, as previous researchers have observed, response times in a wide-area network can't be modelled by such a distribution. As a result, the mean has a weak correlation with the remote execution time.

Figure 4, Figure 5, and Figure 6 show typical timing measurement performed on the same host, a LAN, and over the Internet (between broadband connection and an Digital Ocean droplet),

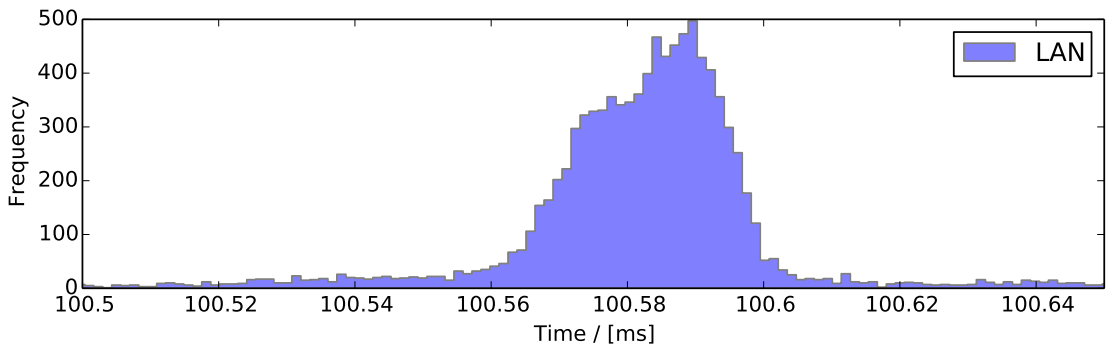


FIGURE 5: Example of a typical response time distribution (100 ms processing time) on a switched LAN (1 hop).

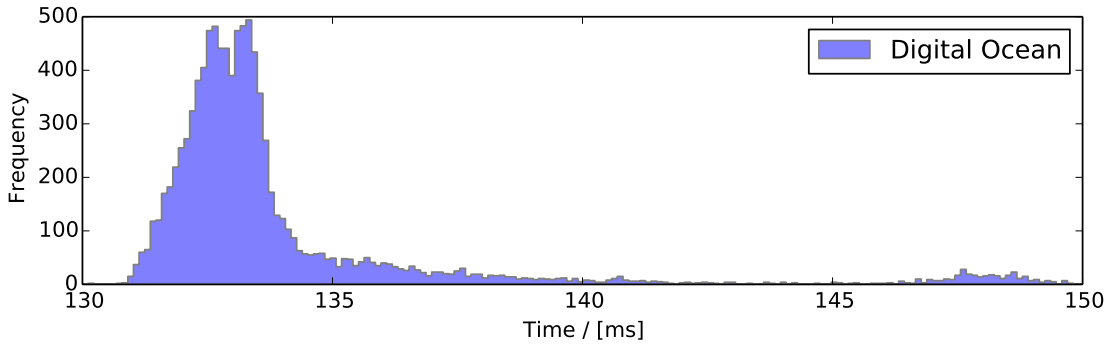
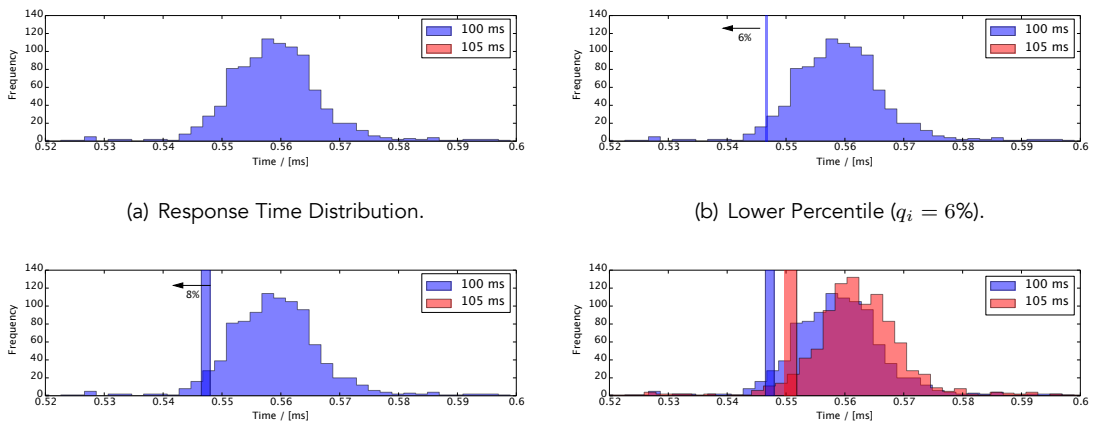


FIGURE 6: Example of a typical response time distribution (100 ms processing time) from a broad band connection to Digital Ocean.

respectively. As one can see, the distribution measured over the Internet shows a somewhat rapid onset and a long decaying tail on the upper end. In particular, the actual distribution of response times, as seen in these measurements, are clearly not normal-distributed but rather highly skewed, asymetric, with more than one mode, with a long-tail.

Previous empirical studies found that low percentiles–the value below which a given percentage of samples fall–are strongly correlated with the processing time on the target [CWR09]. Intuitively, one might expect selecting the minimum response time (0th percentile) would yield an ideal filter, as jitter can only add to this shortest response time. However, in practice this percentile filter turns out to be noisier than other low-percentile filters. Armed with the knowledge of which values are strongly correlated with the processing time, we can now construct a hypothesis test.



(a) Response Time Distribution.

(b) Lower Percentile ($q_i = 6\%$).

(c) Upper Percentile ($q_j = 8\%$).

(d) Both Distributions with both percentiles.

FIGURE 7: Illustration of the Box Test.

3.1.4 Hypothesis Testing

For our purposes, the best performing test identified in previous research is the so-called "Box test". It is based on the observation discussed above, that certain percentiles are well-correlated with the processing times. Developed by Crosby et al. [CWR09], this test has two parameters, i and j, corresponding to low empirical percentiles extracted from the sample measurements. The box test computes the percentiles q_i and q_j from both sets of samples, and declares the samples as coming from different distributions (rejecting the null hypothesis) if the intervals are non-overlapping and properly ordered.

The box test is illustrated in Figure 7: For a response time distribution (Figure 7(a)) the lower percentile of $q_i = 6\%$ (Figure 7(b)) represents the response time below which 6% of the samples were recorderd. Similarly, the upper percentile $q_j = 8\%$ (Figure 7(c)) indicates the response time below which 8% of the samples lie. Finally, in Figure 7(d) the same percentile range is plotted for both distributions. As one can see, the two "boxes" do not overlap and the one corresponding to the shorter proessing time (blue) is located at lower response times than the one for the longer processing time (red). Thus, in this case, the two distributions are assumed distinct with blue corresponding to a smaller processing time than red.

By using this technique, one implicitly applies the filtering step discussed above as part of the analysis. In fact, one can feed raw data sets to the box test and get reasonable results as we will see in Section 5. In practice, low-percentiles (with i and j both less than 10 percent) yielded the best results both in prior research [CWR09] and in our experiments. Crosby's work also showed that based on the assessed alternatives, this test has the lowest false negative rates while also maintaining a low (less than 5 percent) false positive rate.

3.2 PARALLELIZING TIMING ATTACKS

In a scenario where an attacker has multiple targets to choose from and multiple sensors at his disposal, a high degree of parallelism may be possible. There are many concrete situations in which multiple targets are available. For example, e-Commerce sites, social networks, and content delivery networks often have tens, hundreds, or even hundreds of thousands of machines that process requests using the same sensitive credentials (e.g., API keys, SSL certificates). Imagine a timing side channel in a popular library implementation of TLS. In this situation, an attacker can perform measurements against any SSL terminating endpoint holding the certificate in use, and can use multiple sensors to interrogate multiple endpoints in parallel.

3.2.1 Opportunities for Parallelizing Timing Attack

The problem of learning the next unknown byte (or multiple bytes) in a credential can be viewed as a search problem. As mentioned above, the sensor tests each candidate value and selects the value that results in the longest (or shortest) execution time at the target. This set of candidate values can be divided among multiple sensors. Each sensor takes a slice of the set, and selects pairs of candidates from within its individual slice. For each pair, the sensor performs the measurements, filtering (if required), statistical analysis (e.g., the box test), and *rejects both candidates* if their execution time is the same. At the end of the search, the "lucky winner" among these sensors has found two inputs which result in different response time distributions and can announce to the others that the correct value has been identified. The sensors can then tackle the next round of the attack.

Of course, this scenario assumes that there is only one valid value for each set of candidate values such that they can be tested independently from another. Also, it is important to note that it is generally not possible to combine the measurements from different sensors for the same candidate. This is due to the fact that different sensors will have different jitter distributions which makes the later analysis challenging or impossible.

3.2.2 On Learning Multiple Bytes Per Round

Timing attacks resulting from early terminating comparisons (see Section 2.2) have historically been studied in settings where a single attacker probes a single target and progressively learns an unknown credential one byte at a time. But an attacker isn't required to proceed a single byte at a time. Multiple bytes can be probed as part of a single trial. Expanding our search space to chunks of multiple bytes at a time causes the difference in execution time between a correct and incorrect candidate at the target to increase, making it easier to detect when we have found our correct candidate. But this increase comes at a cost: the set of candidate values increases exponentially with the size of the chunk probed at once.

While a parallel timing attack can't help us reduce the number of messages sent when testing multiple bytes per round, it does allow us to reduce the total execution time of each round.

3.3 BLACK-BOX DETECTION AND TEMPLATING

In side-channel attacks, *templating* involves performing the initial measurements against a second, attacker-controlled target (the "template") before launching the attack against the actual target [CRR03]. An example of this is in an attack against a hardware device such as a Hardware Security Module (HSM). The attacker can obtain an HSM with specifications that are identical to the target, perform repeated measurements to learn the exact attack parameters needed to extract the secret from the target device, and then perform the attack against the live target. This allows the attacker to reduce the required time and the message footprint of the actual attack.

In the context of web applications, a similar kind of templating is often possible. With a valid credential, an attacker can perform detailed measurements and understand the exact properties of the target system before an attack is launched. Knowledge of an existing credential allows us to detect timing side channels remotely in a black-box fashion, and can significantly increase the effectiveness of our attack in the following ways.

3.3.1 Black-box Detection

Timing side-channels are tranditionally identified through source code review. In situations where source code is unavailable, an existing credential allows us to *remotely* identify endpoints that may be susceptible to timing attacks. To detect a timing side-channel, we craft two inputs that purposefully trigger the longest and shortest execution time. For this, we take our valid credential and generate two candidates based on it. The first has the first byte mutated (shortest execution), and the second has the last byte mutated (longest execution). Figure 8 illustrates this setting. We can then compare the remote execution times of the endpoint in question for each of these two candidates. If a side-channel exists, the execution times will be different and we can detect the side channel remotely if this difference is large enough to be distinguishable.

3.3.2 Examining Percentile Filters

The box-test is effective in settings where low empirical percentiles are strongly correlated with remote processing time on the target. An existing credential allows us to verify that percentiles exhibit this correlation *before* we actually attempt to perform a timing attack. This optional verification step can give us greater confidence that the attack will succeed.

To test percentile filters using an existing credential, we mutate this credential in each position to cause the comparison to fail at that position, and collect many timing measurements for each of these mutated credentials. We then confirm that the low empirical percentiles increase linearly as more characters match.

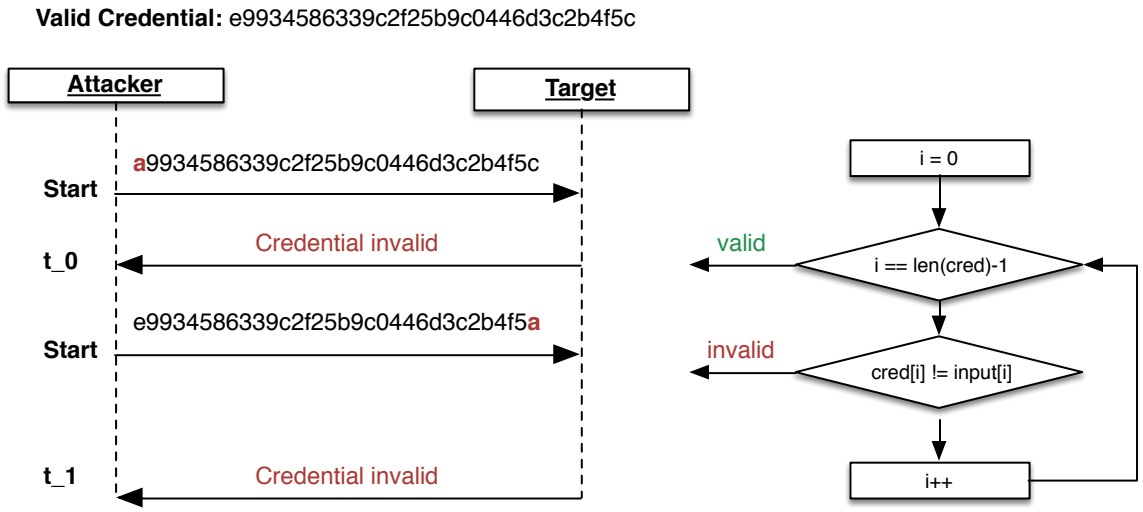


FIGURE 8: Illustration of the short and long request for black-box detection.

3.3.3 Calibrating Our Hypothesis Test

The box-test, our hypothesis test used for comparing sets of measurements, uses a configurable pair of percentiles extracted from each set of samples for testing. We can identify the best pairs of low percentiles to reliably detect differences to a low false-negative rate and acceptable false positive rate.

To calibrate our hypothesis test for a specific target, we again mutate our credential in each position to cause the comparison to fail at that position, and collect many timing measurements for each of these mutated credentials. We then try every i, j pair and select the pair that performs best over our sample set. This effectively mimics the empirical study of Crosby et. al. when selecting an i, j that formed an effective hypothesis test for all samples collected during their study, but here we select the best hypothesis test for *the specific system we are targeting*.

3.3.4 Smallest Detectable Timing Difference

Our sensor may not be sensitive enough to detect differences in execution time at the resolution of a single extra byte comparison. But it may be possible to detect larger timing differences. By progressively mutating an existing credential starting from the last byte and performing our hypothesis test on the resulting timing measurements, we can identify the smallest number of characters that cause a timing difference that is remotely detectable. This quantity informs any analysis on the expected message complexity of the attack.

3.3.5 Selecting the Ideal Sensor

In a situation where we have multiple sensors, an existing credential can be used to select the sensor with the highest sensitivity of detection. We perform our analysis with all sensors, and select the sensor that can detect the smallest difference in execution time.

3.3.6 Avoiding Detection

Using an existing credential associated with our account reduces the number of measurements we need to perform against the unknown credential. This may assist an attacker in avoiding detection, by generating fewer logging artifacts or security events for the unknown credential being attacked.

4 our tool: time trial

During software assessments, the authors of this paper have encountered numerous potential timing attacks through source code analysis (see Section 2). Without proper tools, it is challenging to assess the exploitability of such flaws and discovery via blackbox testing is hard. Similarly, it is difficult to communicate the exploitability of these flaws to open-source projects, clients, or vendors. In particular, except for the most pronounced timing differences, providing proofs-of-concept to illustrate the vulnerability is not feasible. In order to help the community to reduce the gap between theoretical timing attack scenarios and provably exploitable ones, we developed our tool *time trial* [MS14].

The core functionality of *time trial* is the ability to schedule, execute, and capture precise timing data—each called a *trial*. When developing *time trial*, the focus was not on creating weaponized attacks but to perform a feasibility analysis of the attack. Since a successful attack relies on the ability to distinguish two different response times, *time trial*'s feasibility analysis takes two recorded trials and performs a statistical analysis as previously discussed in Section 3.1.4.

We describe *time trial*'s implementation details in the following subsections.

4.1 DESIGN GOALS

As discussed in the Section 3.1 one crucial step in performing timing attacks is the capture of precise timings. An attacker can only influence its own contribution to the overall jitter of the measurement which means timing measurements should avoid introducing additional jitter as much as possible. Moreover, in order to accommodate different attack scenarios, one should be able to place the sensor capturing the timing data–called *racer* in the following–at an arbitrary position relative to the target. For instance, one should be able to perform a trial from the local LAN,

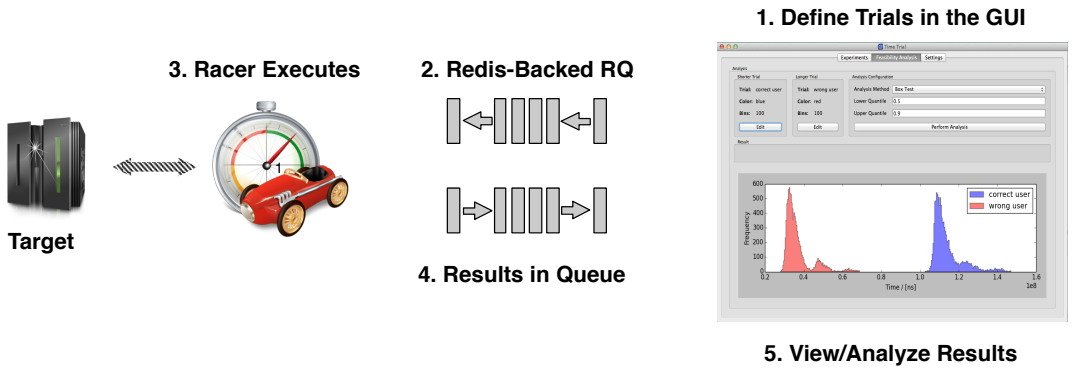


FIGURE 9: Basic workflow and design of *time trial*.

against a target in a cloud environment, or over the Internet. At the same time, scheduling of trials and the analysis of the timing data should be convenient and efficient.

4.2 IMPLEMENTATION

In order to achieve the goal of being able to place the racer on an arbitrary host (e.g., close to the target) and still having a convenient way to analyze the data, *time trial* is implemented as a two component system. The first is the racer which performs the actual timing measurements and the second is a GUI application which is used for scheduling and analysis. For most trials it is necessary to perform a large number of requests in order to obtain a reliable, statistical distribution of the response times. The number of measurements to perform can be set in the GUI along with all other parameters for the different trial types (see Section 4.3).

Both of *time trial*'s components communicate via a Python RQ [Dri] that is backed by the Redis key-value store [Red].

4.2.1 The Time Trial GUI

The GUI component is written in Python using Qt. The interface allows the specification of different trials and their parameters (see Section 4.3 for details on which kind of scenarios *time trial* supports at time of writing). All data about trials and the timing results are stored in a SQLite database. We chose SQLite to make the setup as easy as possible and since we are using SQLAlchemy as an ORM, we can easily switch to a more powerful database backend if needed.

Once a trial is defined in the GUI, it can be assigned to a racer and queued. Queuing pushes the trial into an `rq` from which the corresponding racer retrieves the trial parameters and executes them. After completion, the results are pushed back into rq, the GUI receives them, and persists the results in the SQLite database.

Once the results are available in the *time trial* GUI, they can be plotted in the form of a histogram (frequency as a function of response time) and analyzed using the statistical tests (see Section 3.1.4). As we will see in Section 5, the parameters going into the analysis, e.g. the box test, are not fixed and should be tuned depending on the target.

4.2.2 The Racer

The main factor for an effective implementation of the racer is that it can capture precise timing data. Specifically, we need to be able to capture the response times for each individual request with high precision. It is not sufficient to determine the cumulative or average response time for a large number of requests.

We have brainstormed various ideas on how to do this in the most effective manner. From using a network interface which tags incoming packets with an arrival time (often used for IDS systems),

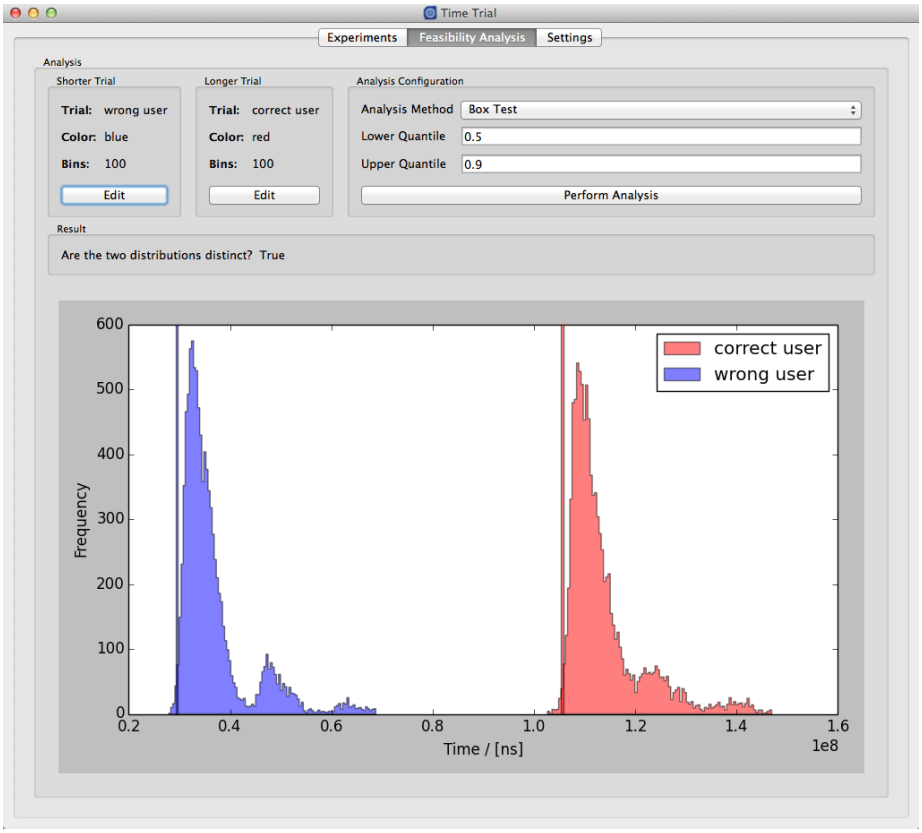


FIGURE 10: Screenshot of the *time trial* GUI.

over embedded devices with real-time operating system, to FPGAs. In the end we decided to go with an optimized Linux setup for several reasons:

1. Most of the jitter is out of the control of the racer and we cannot do anything to reduce it.
2. There are many ways in which we can optimize a Linux host to significantly reduce the locally introduced jitter.
3. When targeting cloud systems, in order to get close to the target, it is rather impractical to deploy custom hardware in the target's data center. A dedicated Linux host or at least a VM are available most of the time.

Leaving operating system optimizations aside for now, the main property required of the Linux host is a reliable source of time. Since our timing process will spend most of its time waiting for a response from the target, we cannot rely on the user time, which captures the duration the process is actually processing on the CPU, but need to use the real time, which corresponds to the actual time passed. On a regular Linux system, a high precision timer is available via the `clock_gettime`

LISTING 13: PORTABLE HIGH-RESOLUTION TIMER ON LINUX AND OS X[jbe11]

```
#include <time.h>
#ifdef __MACH__
#include <mach/clock.h>
#include <mach/mach.h>
#endif

void current_utc_time(timespec * ts) {
#ifdef __MACH__ // OS X does not have clock_gettime, use clock_get_time
  clock_serv_t cclock;
  mach_timespec_t mts;
  host_get_clock_service(mach_host_self(), CALENDAR_CLOCK, &cclock);
  clock_get_time(cclock, &mts);
  mach_port_deallocate(mach_task_self(), cclock);
  ts->tv_sec = mts.tv_sec;
  ts->tv_nsec = mts.tv_nsec;
#else
  clock_gettime(CLOCK_MONOTONIC, ts);
#endif
}
```

call using the `CLOCK_MONOTONIC` clock. Compared to the `CLOCK_REALTIME` clock the monotonic clock is not adjusted due to Network Time Protocol (NTP) updates or other OS adjustments.

Despite `clock_gettime` being a POSIX-specified function, it is not available on OS X. However, OS X has a similar system service which can be queried using `clock_get_time` and we integrate it into the *time trial* racer using the code shown in Listing 13.

Given a suitable time source, there are still several ways to improve the precision of our measurements. By default, the racer process will be preempted by the operating system in order to schedule other processes to run. Even on a host that is dedicated for time measurements, there are kernel and other processes that may be scheduled while running timing measurements. In order to eliminate the influence of the scheduler as much as possible, we run the racer on a Linux host that reserves one CPU core entirely for the racer. By using the `grub` boot options shown in Listing 14, the kernel and all scheduled processes are only ever using a maximum of 2 "CPUs". On a host with hyper-threading and two cores (4 virtual "CPUs"), this leaves an entire core unused. But this CPU core can still be assigned to processes by requesting it explicitly. Listing 15 shows how to achieve this by means of the `CPU_SET` function. As a result we have a dedicated CPU core that is not used by any other process such that the timing measurement should not be preempted during its execution. To further ensure that the process remains scheduled, we run it with real-time priority.

LISTING 14: LIMITING THE CORES AVAILABLE TO LINUX

```
GRUB_CMDLINE_LINUX_DEFAULT="maxcpus=2 isolcpus=1"
```

LISTING 15: C++ SCHEDULING OPTIMIZATIONS

```
#include <sys/resource.h>
#include <sched.h>
void set_cpu_affinity(int cpu) {
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(cpu, &mask);
}

void enable_real_time(int priority) {
        int which = PRIO_PROCESS;
        id_t pid;
        int ret;
        pid = getpid();
        ret = setpriority(which, pid, PRIO_MIN);
}
```

4.3 SUPPORTED TRIAL TYPES

Currently, *time trial* supports timing attacks in three different scenarios: a proof-of-concept echo setup, and two HTTP scenarios.

4.3.1 Echo Trial

The echo trial serves basic feasibility testing purposes and can be used to create settings with known timing delay. It consists of a custom TCP client and server. When executed, the client sends a single integer (`int`) to the server who interprets the integer as a wait time in nanoseconds, sleeps for said amount of time, and returns the same integer to the client. The client uses the timing functionality shown above to record the time for each exchange.

The sleep was originally implemented using `nanosleep(2)`. However, for delays below 1 µs this sleep timer proved unreliable and could not be used to generate a constant delay for each request. To remedy this, we implemented busy waiting based on Intel's Time Stamp Counter (TSC) which increments with every clock tick. The counter is kept in a 64-bit CPU register which was accessed using inline assembly.

4.3.2 HTTP Request Trial

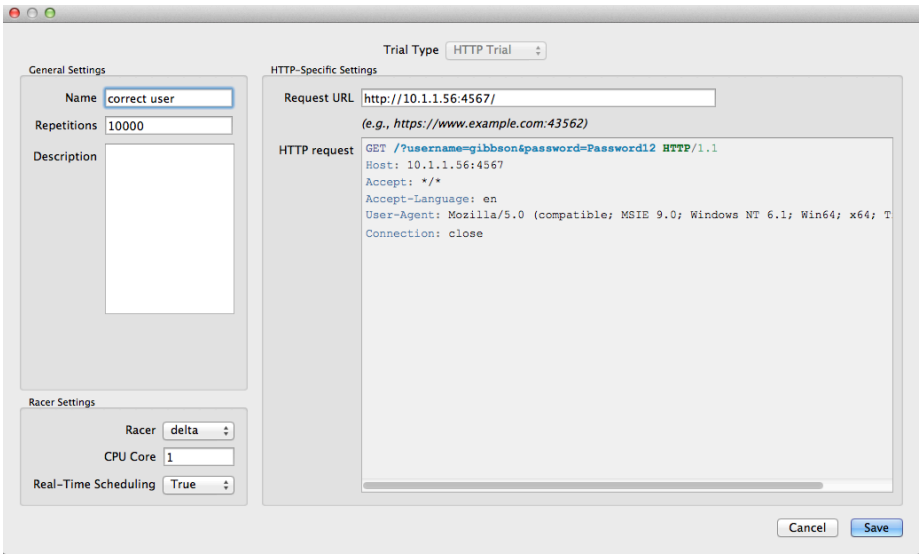


FIGURE 11: Configuring an HTTP Trial

Our work so far has focused on remote timing attacks in web services. With the HTTP request trial type we are able to perform timing measurements against arbitrary HTTP hosts. To setup such a trial, one needs to specify a target URL and a raw HTTP request to be sent. The racer parses the HTTP request and then uses cpp-netlib [otcnp] to execute the request against the target. We took special care to not include any HTTP parsing but only the execution of the request itself in the timing measurements. If desired, we have the ability to remove certain HTTP error codes from the set of response times.

4.3.3 Timing Extraction Racer

Many frameworks include the execution time at the server directly in the response. For instance, some frameworks set the X-Runtime response header, while others include HTML comments that indicate the time taken to process the request on the server. These debugging options can be leveraged to obtain precise timing data which is not prone to distortion by network jitter. To support analysis of these systems, *time trial* includes a racer that uses regular expressions to extract the response time from a server response. This client can be configured to extract response time from headers or the response body, and automatically formats the timing data in the form expected by the GUI for analysis. This racer is easily extended in cases where additional processing of responses is required.

4.4 PLANNED EXTENSIONS

There are various directions in which we plan to extend *time trial*'s functionality. As a major new function we are working on implementing an actual attack feature that given a known-vulnerable endpoint, a request, an injection point, and parameters for the statistical analysis, is able to automatically exploit the target.

To make *time trial* applicable to a wider range of targets, additional trial types should be added such that, e.g., custom TCP protocols or even local targets (e.g., cryptographic systems) can be analyzed. Similarly, the import of externally captured timing data may be useful for settings where *time trial* cannot be deployed directly and timing data is captured by other means.

While we currently have settled on a Linux host for performing timing measurements, it may be worthwhile to explore other techniques such as FPGAs or networking interfaces with packet tagging. In particular, if one wishes to get an upper bound on what the most sophisticated attackers may be capable of, this is a valuable path of research.

5 survey of timing attack targets

In this section, we discuss several practical results we obtained using our tool *time trial*. First, in Section 5.1 we present a generic feasibility analysis on what response time difference can be distinguished in different network settings. Following that, in Section 5.2, we discuss what these results mean for practical attack scenarios such as string comparison and branching-based timing side-channels.

5.1 GENERIC FEASIBILITY ANALYSIS

To determine which timing differences are exploitable in principle for different network settings, we performed timing measurements using the *Echo Trial* described in Section 4.3.1. The methodology used for this basic feasibility analysis is similar to the one used by Crosby et al. in [CWR09]. However, instead of using UDP as Crosby did, we decided to use TCP connections as they are more likely to be used by attacked targets. Due to the additional packets involved in TCP, our approach may incur more jitter and thus reduce the measurement resolution as compared to UDP.

Armed with this setup, we investigated several scenarios. First, we configured a *switched LAN* in our lab by connecting two computers with a NetGear Gigabit switch. Second, we simulated *Internet* conditions by running experiments between a broadband cable connection and a Digital Ocean Virtual Private Server Droplet. Finally we looked at *Cloud Computing Environments*. Modern cloud solutions such as Amazon's EC2 make it much easier for an attacker to get closer to the target. To analyze this, we used two Amazon EC2 instances in the same availability zone and measured response times between those instances.

In the following sections, we will discuss the limits for distinguishing different processing times as determined by our experiments in each environment.

5.1.1 LAN Results

We start our analysis with a rather long processing time difference of 1 ms (100 ms vs. 101 ms). As shown in Figure 12, this long difference can easily be distinguished with the naked eye.

When the difference is reduced to 100 µs (100 µs vs 200µs), the two response time distributions move closer together, but remain clearly distinct (see Figure 13).

As seen in Figure 14, for a difference of 10 µs (100µs vs. 110µs), the distributions start to overlap but remain easily distinguishable.

The situation becomes less clear when looking at processing time differences of only 1 µs. Figure 15 shows the same processing time difference measured 1,000 times, 10,000 times, and 100,000 times. While the first two plots don't show any clear separation of the two distribution,

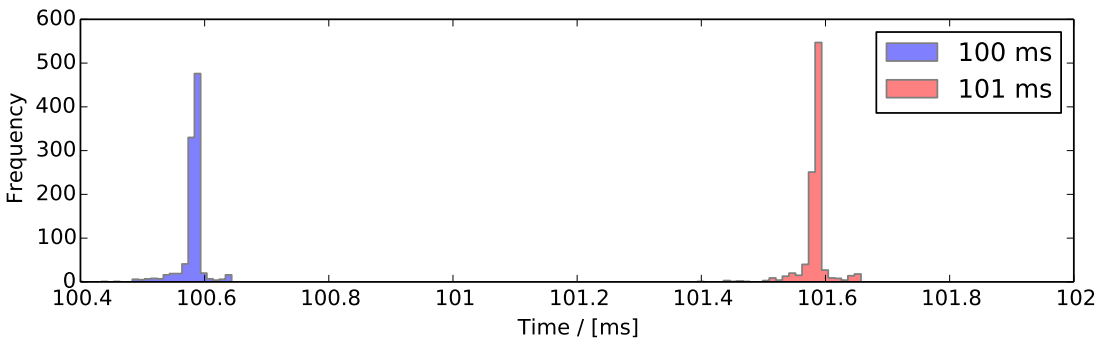


FIGURE 12: LAN: 100 ms vs. 101 ms using 1,000 repetitions.

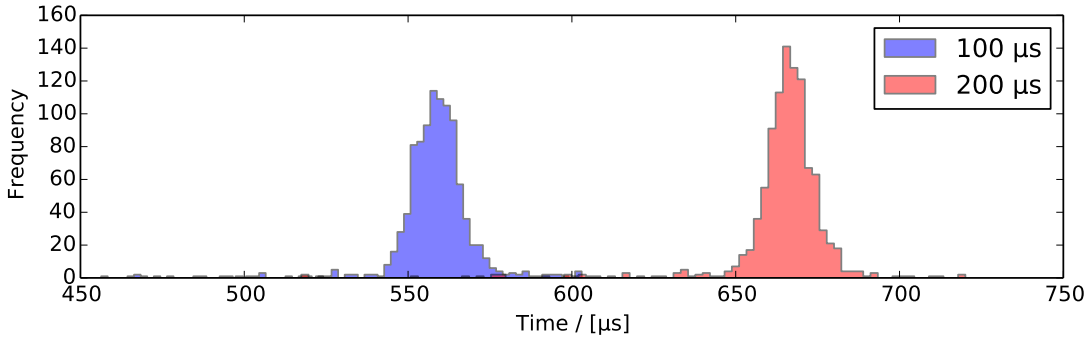


FIGURE 13: LAN: 100 µs vs. 200 µs using 1,000 repetitions.

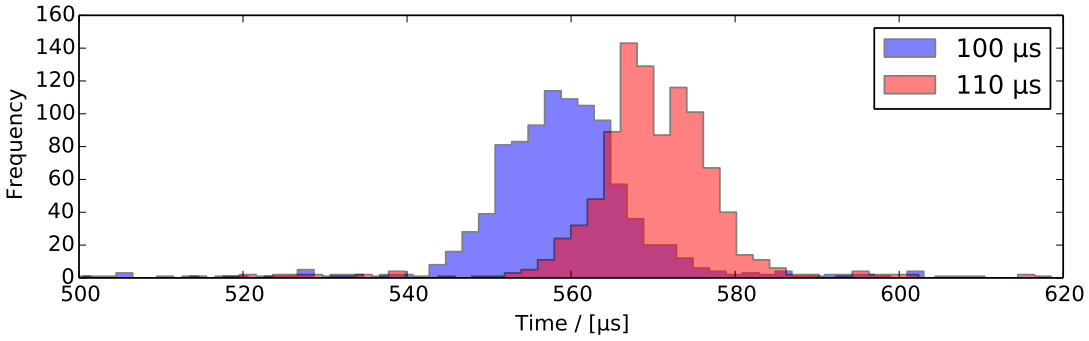


FIGURE 14: LAN: 100µs vs. 110µs using 1,000 repetitions.

for 100,000 measurements, one can see a clear shift between them. Furthermore, as indicated by the two lines in the figure, the box test identified the two distributions as distinct.

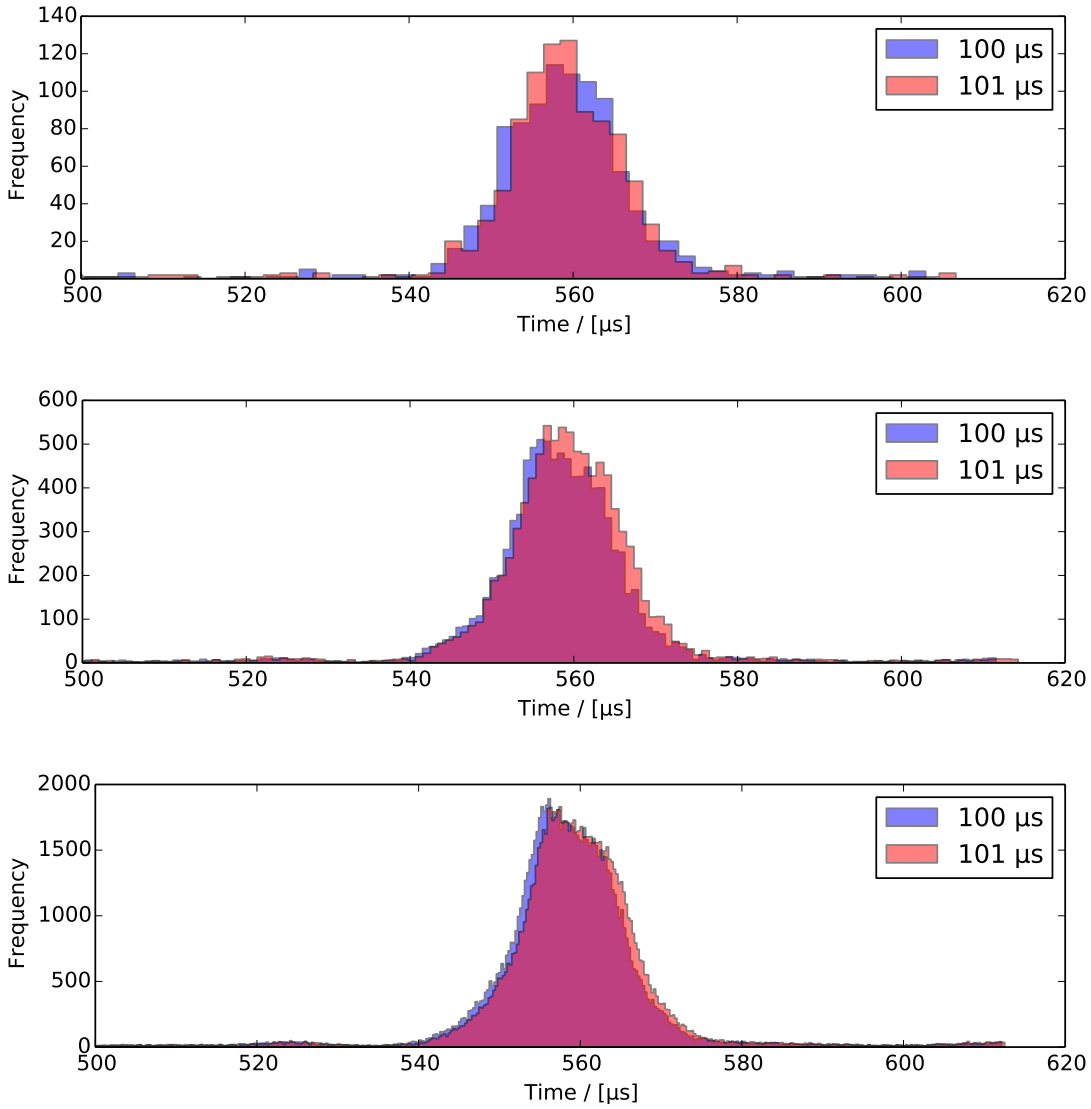


FIGURE 15: LAN: 100 µs vs. 101µs using 1,000, 10,000, and 100,000 repetitions respectively.

For any shorter processing times we switched from `nanosleep` to busy waiting based on the TSC. With this, we hit the limit of what can be resolved on a LAN somewhere soon below 100 ns (100 ns vs. 200 ns) in Figure 16.

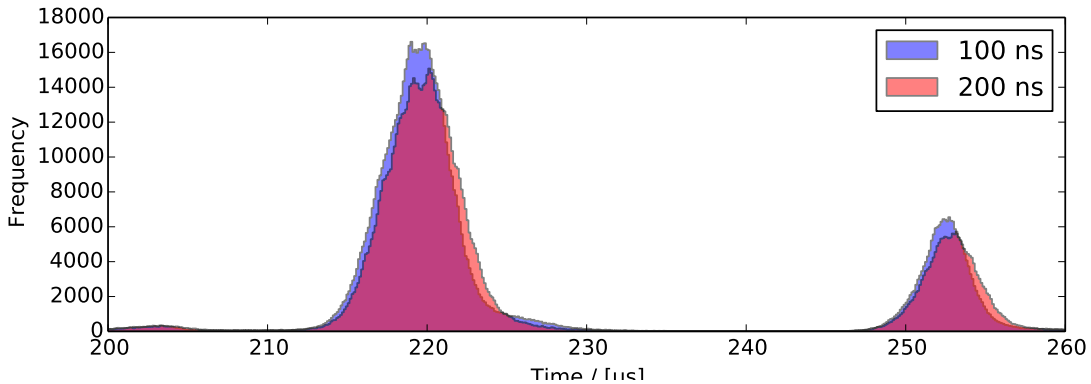


FIGURE 16: LAN: 100ns vs. 200ns using 1,000,000 repetitions. Less repetitions would be sufficient in order to distinguish the difference.

It should be mentioned that it may be possible for an attacker to resolve even smaller timing differences but that our implementation of busy waiting on the target server is not stable enough to produce the consistent processing delays to test this scenario.

5.1.2 Loopback

Local attacks are a less likely attack scenario that move an attacker even closer to the target and eliminate the influence of the network. We measured how an attacker executing on the same physical host would fare in a timing attack. This attack was still performed via the network interface and not by measuring the execution time directly. As seen in Figure 17, distinguishing a 30 ns difference is still possible. For anything below 30 ns, our measurements were not consistent. It was unclear in testing if this was a limitation of the tool, technique, or noise in the simulated target server itself due to variance in TSC values for such short delays.

5.1.3 WAN Results

Due to the increased jitter on a WAN connection, the resolution of our measurements via the Internet at large was expected to be smaller. This was confirmed through active testing.

For the WAN environment, we performed a measurement with a 1 ms processing time difference. In contrast to the LAN, the resulting distributions overlap significantly but can still be distinguished easily (see Figure 18).

When the difference in response time is reduced, one quickly needs to add additional requests in order to keep the noise low enough to be able to distinguish the results. Figure 19 shows that for a 100 µs difference (100 µs vs. 200 µs) one can still distinguish when 10,000 repetitions are being used.

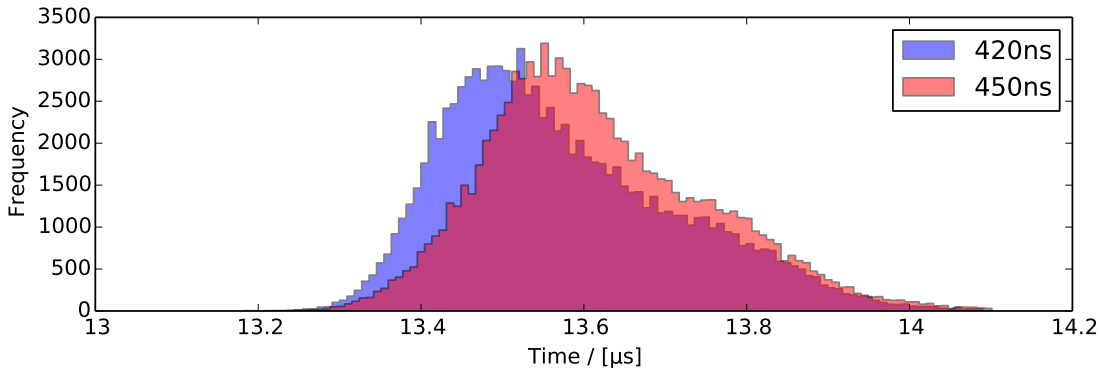


FIGURE 17: Loopback: 100ns vs. 200ns using 1,000,000 repetitions. Less repetitions would be sufficient in order to distinguish the difference.

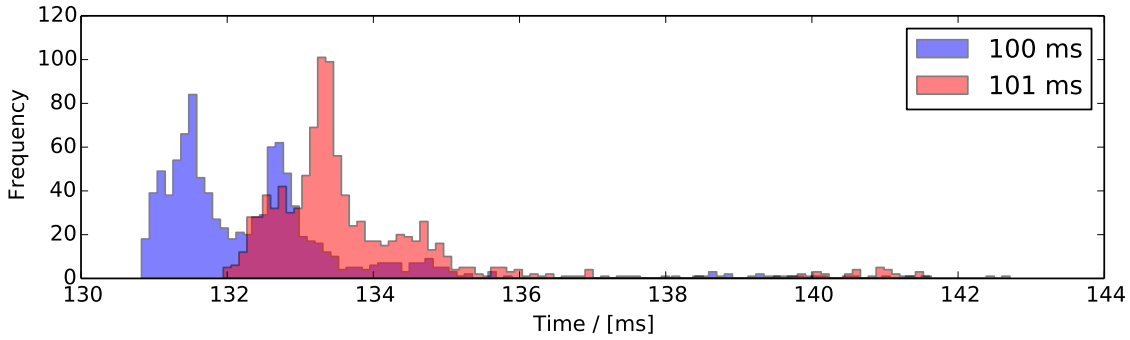


FIGURE 18: WAN: 100 ms vs. 101 ms using 1,000 repetitions.

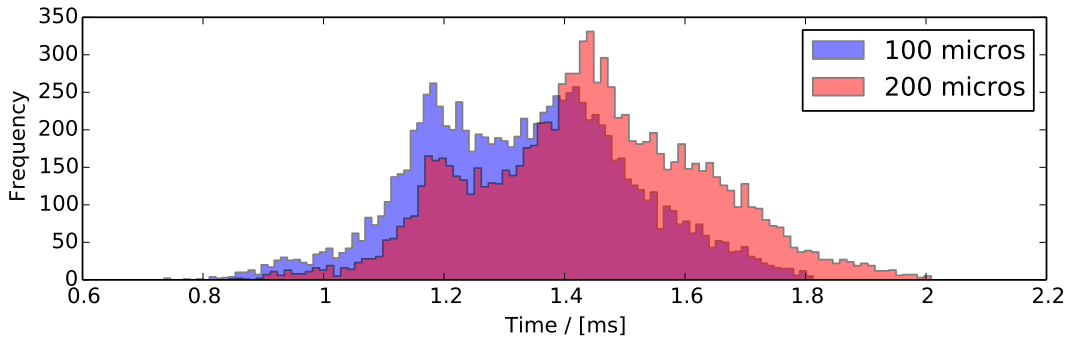


FIGURE 19: WAN: 100 µs vs. 200 µs using 10,000 repetitions.

For the WAN setting we determined that timing differences of 1 µs could still be distinguished (see Figure 20) while 100 ns was generally infeasible. However, there is potential for achieving greater resolution in-between. One limiting factor was that the network quality was rather inconsistent which we speculate may lead to changes in jitter while the measurements took place. These variations made it difficult to draw final conclusions. While it made our experiments more difficult, this result illustrates that the selection of a high-quality vantage point is important in order to gain maximal timing resolution. In Section 5.1.5 we discuss in some more detail how one could compensate for the change in environment in future experiments.

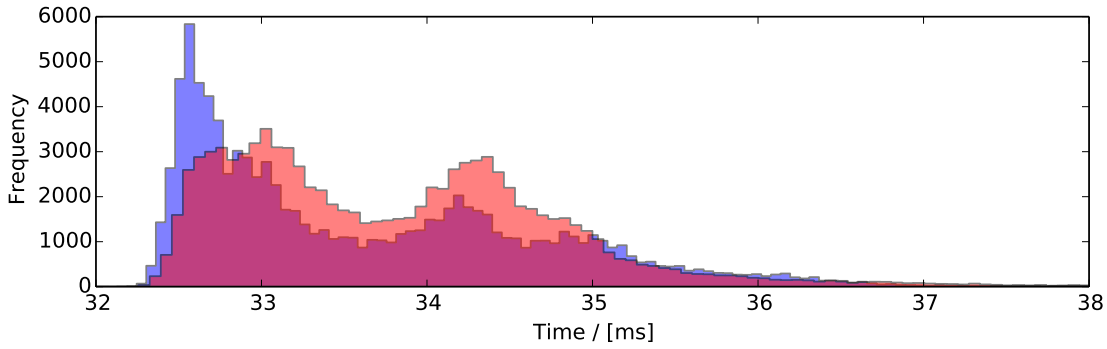


FIGURE 20: WAN: 100 µs vs. 101 µs using 100,000 repetitions.

5.1.4 EC2 Results

As we showed in the previous section, timing measurements quickly become unreliable over long-distance Internet connections. If the target is located in a cloud environment, by renting a virtual server in the same environment, the attacker can cut out large amounts of Internet jitter and put

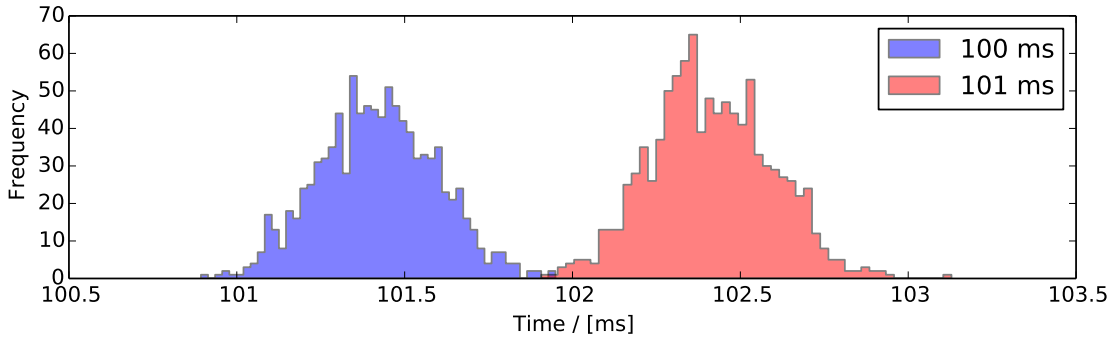


FIGURE 21: EC2: 100 ms vs. 101 ms using 1,000 repetitions.

themselves into a LAN-like environment. As it turns out, the network within an EC2 availability zone is excellent. This is likely to high-quality networking hardware used by Amazon's service.

The two main take-aways for our EC2 analysis are: 1) Attacks within the EC2 environment are effectively as feasible as on a local network and 2) if we used better hardware for our LAN measurements, we are likely to obtain better results in that environment as well. Below are the detailed plots for different timing differences as obtained for EC2.

Similar to the LAN environment, the distributions for 1 ms timing difference (100 ms vs. 101 ms) are clearly separated in Figure 21.

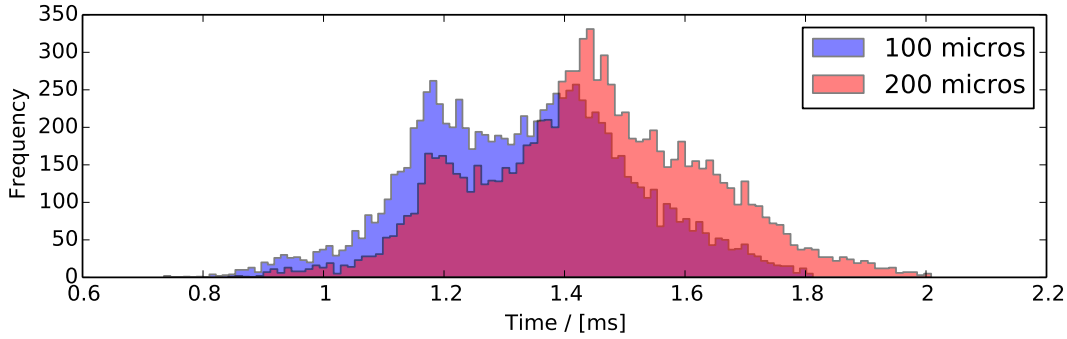


FIGURE 22: EC2: 100 µs vs. 200 µs using 10,000 repetitions.

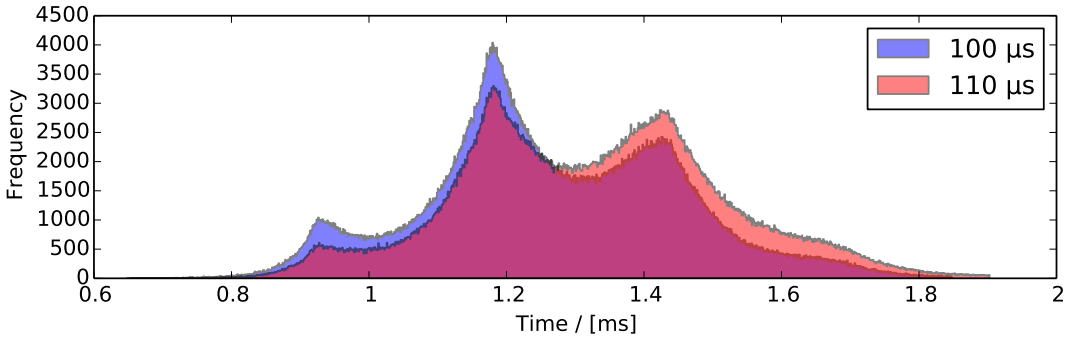


FIGURE 23: EC2: 100 µs vs. 110 µs using 100,000 repetitions.

For 100 µs (Figure 22) and 10 µs (Figure 23) processing time differences the distributions quickly start to overlap but remain distinguishable with the naked eye.

Using the `nanosleep` timer, the resolution limit was reached at around 1 µs (101 µs vs. 100 µs) where the box test is still able to distinguish the distributions (Figure 24).

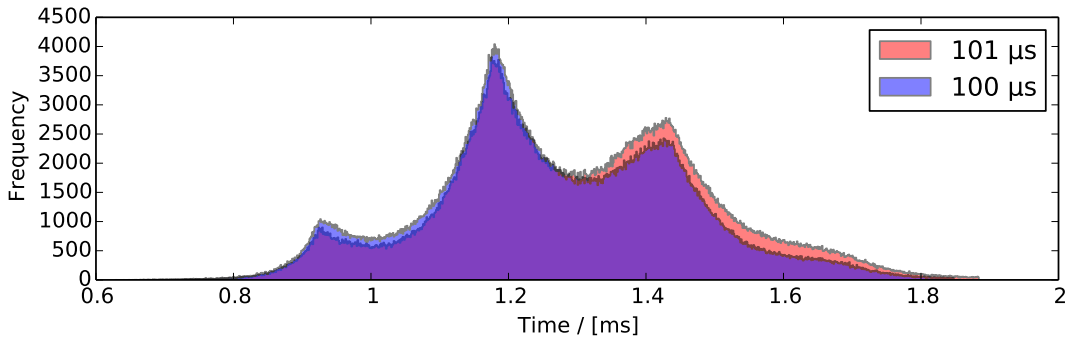


FIGURE 24: EC2: 100 µs vs. 101 µs using 100,000 repetitions.

After switching to the more precise timer using busy-waiting based on the TSC, we were able to distinguish even about 100 ns of timing difference in the EC2 environment (Figure 25).

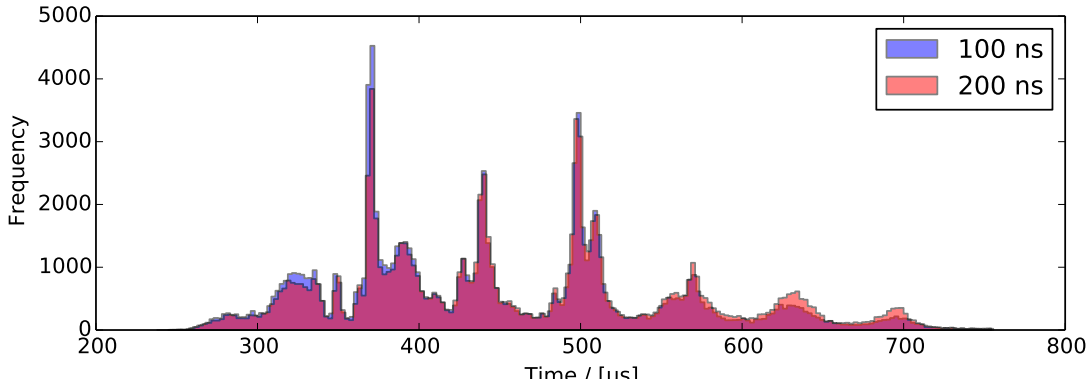


FIGURE 25: EC2: 100 ns vs. 200 ns using 100,000 repetitions.

5.1.5 Summary

Table 1 summarizes our results discussed above and shows that the resolution for attacks over the Internet is significantly lower than on a LAN. At the same time, the ability to move into the same cloud environment as target may give a LAN-like advantage to the attacker. These results are consistent with the ones obtained by Crosby et al. [CWR09]. For some environments we obtained a slightly smaller resolution in our experiments which may be due to the use of TCP instead of UDP.

Potential Improvements: As mentioned above, since we artificially introduced specific processing times, the delay function itself may have added additional jitter to the measurement. While

	1 ms	100 µs	10 µs	1µs	100 ns	< 100 ns
Loopback	✓	✓	✓	✓	✓	✓
LAN	✓	✓	✓	✓	✓	⊘
Amazon EC2	✓	✓	✓	✓	✓	⊘
WAN	✓	✓	✓	✓	⊘	⊘

TABLE 1: Overview of the response time differences that could be distinguished in each network environment

our CPU scheduling optimizations should minimize these influences, a better resolution may be achievable in practice.

Moreover, especially for the WAN environment we saw rapid changes in the network quality and propagation delays. For all of our experiments we first obtained the data for the longer (or shorter) processing time before performing measurements for the other time. If the network quality changes during this time, it is difficult to line up the resulting response time distributions. In the future, the process could be improved by making alternating requests that switch between short and long response times.

5.2 REAL-WORLD TARGETS

Now that we have gained a better understanding on which differences can be distinguished in the different network settings, we will relate these to timing differences that are encountered for different timing side-channels in practice. We first discuss string comparisons and then cover branching-based timing flaws.

5.2.1 String Comparison

Objective and Methodology: Microbenchmarks were developed for popular languages and used to better understand the timing measurement resolution required to exploit timing attacks against early-exit comparison functions. These benchmarks compute a per-byte (or per-word, where appropriate) time measurement representing the execution time difference that may be introduced when an additional byte (or bytes) of an input match with the target value. For these measurements, "exploitation" is taken to mean adaptively learning the contents of a hidden credential on a target in a reasonable time. These numbers are not meant to imply that information isn't leaked by the execution time, as it clearly is, they only illustrate the challenge faced by an attacker who wants to perform the attack in a reasonable time frame.

In practice, it turns out that target platforms are not guaranteed to "exit early" on the first non-matching byte, and may compare entire CPU words (or larger values) in a single instruction [Law10]. To compute the per-word microbenchmark for our modern platform, measurements were taken when comparing two strings that differed in the 128th and 256th position. The difference between these two measurements was divided by the number of words and iterations to obtain an average execution time.

These microbenchmarks are accompanied by an analysis of execution time as a function of matching bytes in the first 32 bytes of buffers being compared. These positions were chosen in part because many of the benchmarked platforms exhibited execution time behavior that *didn't vary* with the number of matching characters for smaller strings. This behavior makes timing attacks against these platforms more difficult in practice, as discussed below in more detail. This behavior is platform specific and other platforms may perform true 'early exit' comparisons that make attacks easier.

Microbenchmarking is a difficult art, and the challenges faced in capturing these benchmarks illustrates how small these timescales are on modern systems. Benchmarking challenges and limitations are discussed in more detail in the results. These measurements will be improved in a future version of this paper to provide a more accurate picture into the behavior of comparison functions on current CPUs.

Environments: Benchmarks were implemented and run on a mid-range desktop as well as an affordable embedded single-board computer. Benchmarks were also performed on an EC2 "small" instance (CPU Single Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz) but performance was similar to the desktop and is omitted from the results reported below. Benchmarking platform specifications are as follows:

Platform 1: Mid-2013 MacBook Pro

CPU: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz (4 cores), 4 gigs RAM (DDR3)

OS: Ubuntu 14.04 Server

Linux kernel: Ubuntu 3.13.0-32-generic #57-Ubuntu SMP

libc version: 2.19-0ubuntu6

gcc version: 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

Python: Python 2.7.6

Java: Java SE Development Kit 8 Update 11 for Linux 64

Clojure: Leiningen 2.4.2, but Clojure benchmarks were compiled as a standalone JAR

Ruby: ruby 1.9.3p484 (2013-11-22 revision 43786) [x86_64-linux]

Platform 2: BeagleBone Black Rev C

CPU: ARM Cortex-A8 1GHz (single core), 512 megs RAM

OS: Debian 7.6 (wheezy)

Linux kernel: 3.8.13-bone47

libc version: 2.13-38+deb7

gcc version: 4.6.3 (Debian 4.6.3-14), compiled -lrt

Python: Python 2.7.3

Java: Java SE Development Kit 8 Update 6 for ARM (1.8.0_06)

Clojure: Leiningen 2.4.2, but Clojure benchmarks were compiled as a standalone JAR

Ruby: ruby 1.9.3p194 (2012-04-20 revision 35410) [arm-linux-eabihf]

Early-Exit and Multi-Byte Comparison: Timing attacks on string comparison rely on the (often unstated) assumption that execution time increases smoothly as each successive byte matches. If multiple matching bytes do not cause an increase in execution time, an attackers job becomes harder, as they must guess many correct bytes before inducing a measurable time difference in the target. Modern CPU and compiler combinations may compare entire words at a time, and thus leak no information about sub-word matches to the attacker.

Java was chosen specifically to study this due to its susceptibility to timing attacks in past studies. On Linux, the execution time remains relatively constant within a word (of 64 bits), which makes a remote attack unworkable for an attacker (see Figure 26.

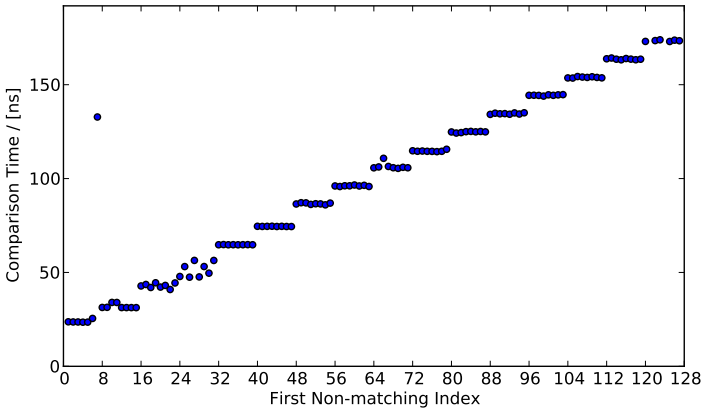


FIGURE 26: String Comparison using Java on an Intel i5 CPU. Processing time within a CPU work does not increase but once a word boundary is crossed, there is a corresponding jump.

Java on the ARM platform did *not* exhibit this behavior (see Figure 27, and execution time increases smoothly with each successive byte. This helps an attacker, and along with the numbers stated above, put these platforms within reach of a timing attack.

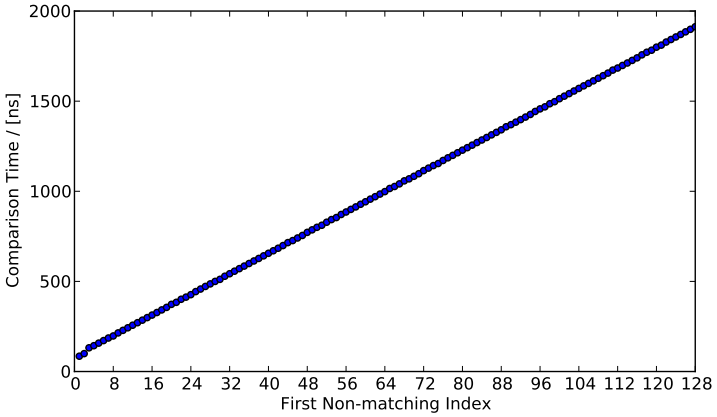


FIGURE 27: String Comparison using Java on a BeagleBone Black. The procesing time increases for each additional byte that is compared.

Per-Byte Microbenchmarks: All measurements are in nanoseconds and the i5 benchmarks are per-word whereas the other platforms are per-byte.

		Lawson 2010 Athlon X2 2.7GHz*	Cortex-A8	i5-3210M 2.50GHz
C	*memcmp*	.719	1.3770	.243
	strcmp	-	4.0441	.41
ruby	==	.840	1.7586	.36
python	==	1.4	1.4867	.224
java	*String.equals*	40.594	18.91	7.65

* Benchmarks from the Lawson and Taylor 2010 Blackhat Presentation

The contrast between the Lawson data and our own microbenchmarks is striking. Part of the difference may be attributed to hardware: The Intel i5 can compare two 8-byte words atomically in a single instruction, and has multiple execution units that allow multiple comparisons to be done in parallel. Given the CPU frequency of 2. 7GHz, our results correspond to about 2 comparisons per clock-cycle. While this may indeed be the case, it could also suggests possible limitations in the microbenchmarking approach. Measuring the per-byte (or per-word) microbenchmark requires us to run our comparison in a tight loop to reduce measurement error, but this may defeat its own purpose in some cases (notably python) as it allows the processor to optimize and compare multiple words in a single clock cycle. C exhibited the most striking behavior, with performance counter-intuitively increasing as the first non-matching word increased, thus requiring the comparison between differences of many words.

Regardless of these limitations, these benchmarks show that once a runtime has done everything it needs in order to perform the comparison, the actual byte-to-byte timing differences are very

small. The numbers were daunting already in Lawson's study in 2010 and timing attacks against these languages on a modern platform are even less feasible today.

Java on the i5 remains somewhat of an exception, but because timing information is only leaked at word boundaries, attacks remain intractable on modern systems. In contrast, the performance of Java on the embedded system may put it in reach of a remote timing attack in certain environments.

Conclusion: Modern implementations still leak timing information, but increased hardware performance and optimized comparison functions greatly reduce the remote exploitability of these side channels. Against these implementations, an attacker must guess multiple bytes correctly, instead of just a single byte, and the time difference the attacker must discern remotely. Embedded systems remain vulnerable due to a lower performance, smaller word size, and the use of unoptimized comparison functions in platforms like Java.

5.2.2 Microcontrollers and the Internet of Things

The rise of the *Internet of Things* brings a plethora of Internet-connected devices onto modern networks. To understand the susceptibility of these platforms to timing attacks we performed experiments against embedded microcrontrollers. The slower, low-power processors in use in these devices allow us —in some way—to "travel back in time" to when processor speeds were lower. Since these embedded processors perform less operations per second, they require more time for each operation which leads to larger timing differences for different operations. We already gave a glimpse at that with our BeagleBone Black discussed above. To take this a step further we have also performed a basic analysis of the Arduino Mega.

The Arduino Mega is a popular device whose processor operates at 16 MHz. We measured the processing time for string comparison on the device and determined that it takes approximately 0.46 µs per byte—significantly longer than on any other device we investigated. With such a large difference per byte, it was worthwhile exploring whether this timing difference can be exploited remotely over a LAN. As shown in Figure 28, there is a difference in the response times for strings that differ in the first character vs. strings that differ only at the 10th character. This indicates that timing attacks on regular string comparison have to be assumed feasible for any embedded system.

5.2.3 Branching

In contrast to the string comparison discussed above, there is no generic analysis for branching-based flaws. This is due to the fact that the difference in processing time for each branch depends on the underlying application. We have studied a few common functions that are frequently used in sensitive contexts and may disclose information via conditional branching.

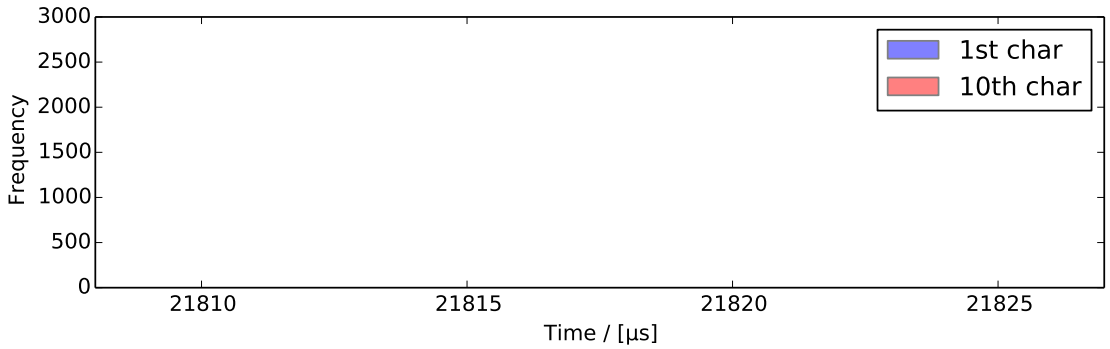


FIGURE 28: Arduino: Comparing strings that differ at the first character and strings that differ at the 10th character respectively.

Hash Functions Hash functions are frequently used in sensitive application areas that range from login function, over HMACs, to data integrity checks. In order to determine whether one could distinguish the execution of a hash function remotely, we implemented a basic web service in ruby (sinatra using thin) which verifies a user's password depending on whether the username was correct (see Figure 1). The web service was configured to use OpenSSL's implementation of SHA1 and SHA256 or ruby's bcrypt implementation. For SHA, we were able to detect whether the function was executed in both the LAN and the WAN environments (Figure 29).

Bcrypt is hash function specifically designed for password storage. It's runtime is intentionally be slow in order to prevent rapid dictionary or brute-force attacks on compromised password hashes. As a side-effect of it being slow, it is trivially possible to detect whether an execution of bcrypt occurred. This is illustrated by the obvious separation of the two response time distributions for the LAN and WAN environments in Figure 30.

Note: It is important to realize that this is not a flaw in the hash functions themselves, but rather a timing side-channel introduced via the manner they are used in this case.

Database Queries The performance of database queries is dependent on both the query and the data in the database. Due to the complex and optimized structure of data storage in modern database systems, it is not trivially possible to draw conclusions about the stored data based on the response time. This is a subject that certainly warrants future research as it has the potential to disclose sensitive data.

Since a full study of the timing of database queries was outside the scope of this research, we only assessed whether the execution of a simple query on an in-memory SQLite database could be detected. In order to keep the query time as short as possible, the database included only a single row. This attack is feasible in a LAN environment (see Figure 31) and, while not experimentally

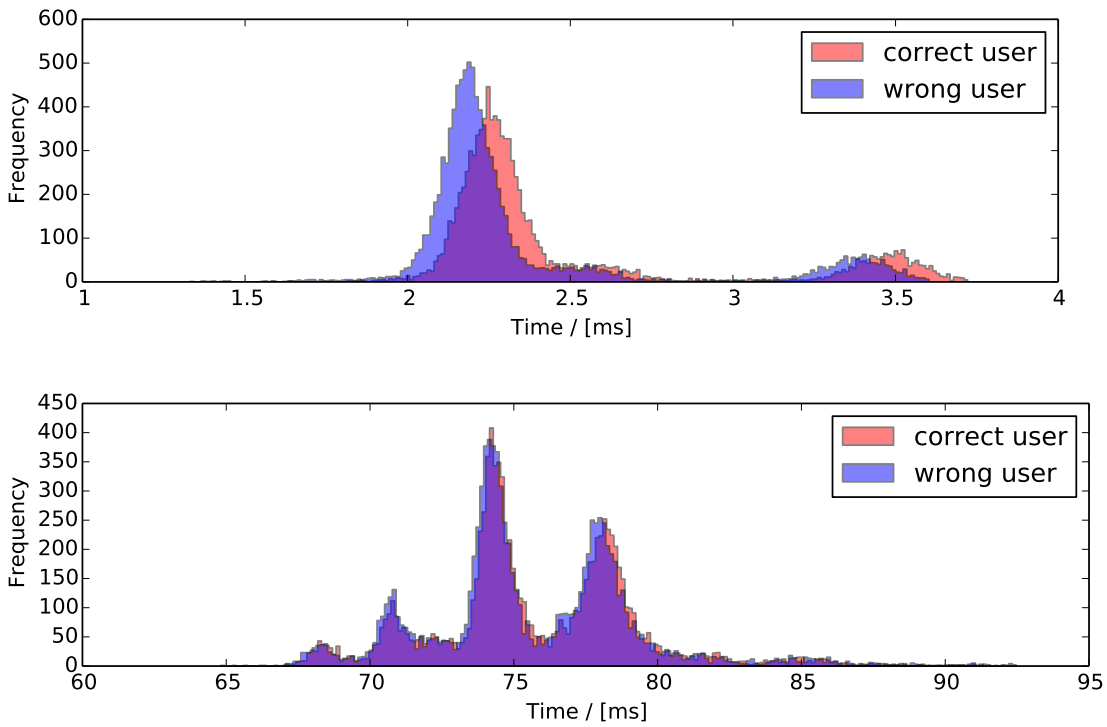


FIGURE 29: SHA256: Detecting whether SHA256 was executed in a LAN (top) and WAN (bottom) environment. Measured using 10,000 requests for each case (less are likely sufficient on the same LAN).

verified, the timing difference appears large enough that it should be detectable over the Internet as well.

CBC Padding Oracle In Section 2.1.2 we introduced the padding oracle attack on the CBC cipher mode. As mentioned, the attack relies on the ability to distinguish a decryption with incorrect padding from one with correct padding. When correct padding is encountered, the application will typically attempt to process the decrypted data in some manner. Which processing takes place after decryption depends on the application but a wide-range of actions (e.g., those involving I/O) will consume significant amounts of processing time and the difference can thus be detected remotely. It is important to note that this can be exploited even if the application returns the same message for both cases!

For example, assume the decrypted data includes an access token. Upon successful decryption, the application looks up this token in a database in order to verify that it is valid. Other scenarios include logging of the request, backend requests, XML parsing, or any other kind of processing.

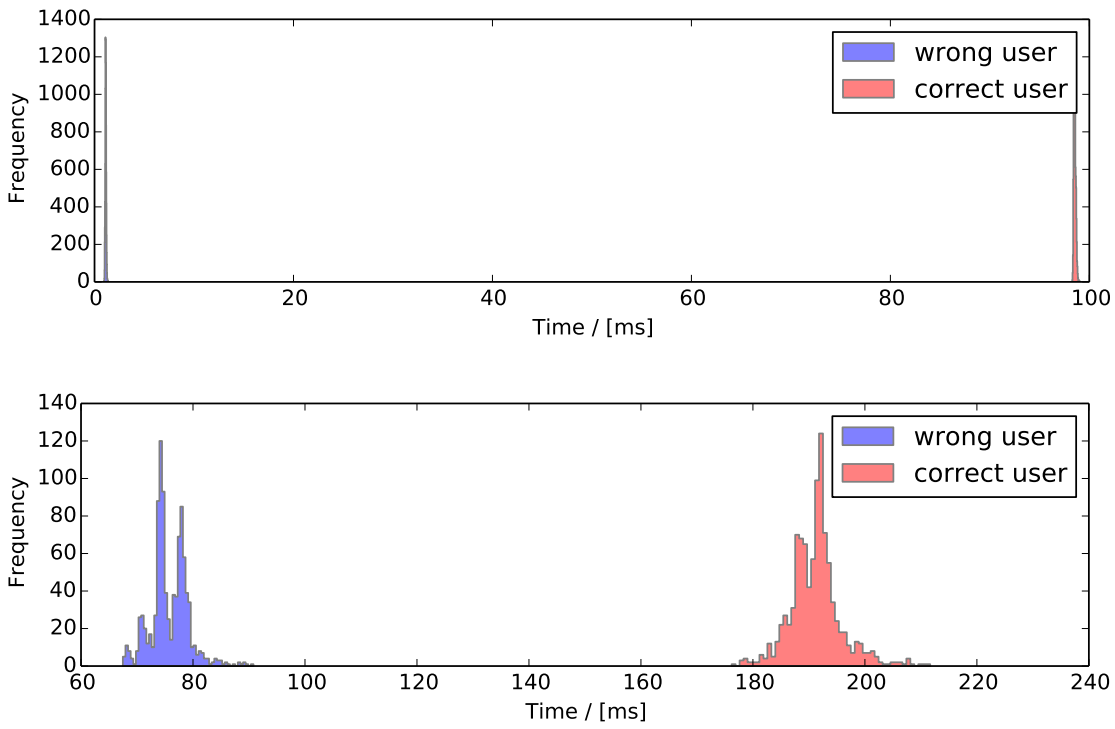


FIGURE 30: Bcrypt: Detecting whether bcrypt (work factor of 10) was executed in a LAN (top) and WAN (bottom) environment. Measured using 1,000 requests for each case (less are likely sufficient in both cases).

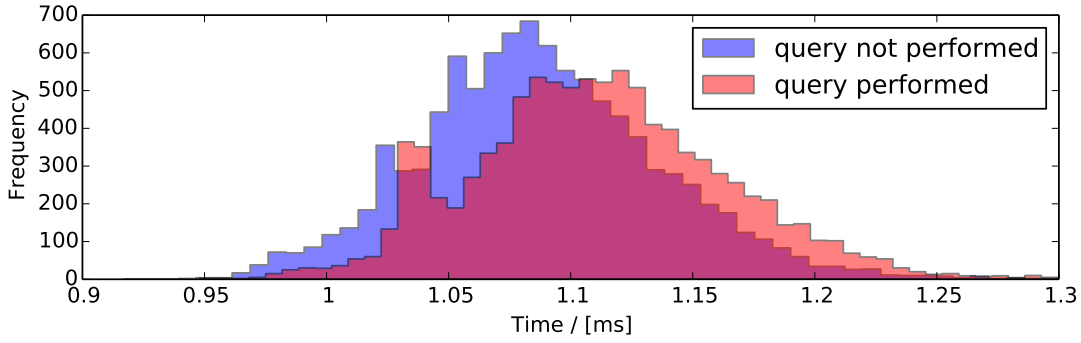


FIGURE 31: Database Query: Timing difference for a simple database query against an in-memory SQLite database containing a single entry.

We implemented the padding oracle attack using the timing difference as the "oracle". Figure 32 shows a screen shot of our attack running against a mock-up server.

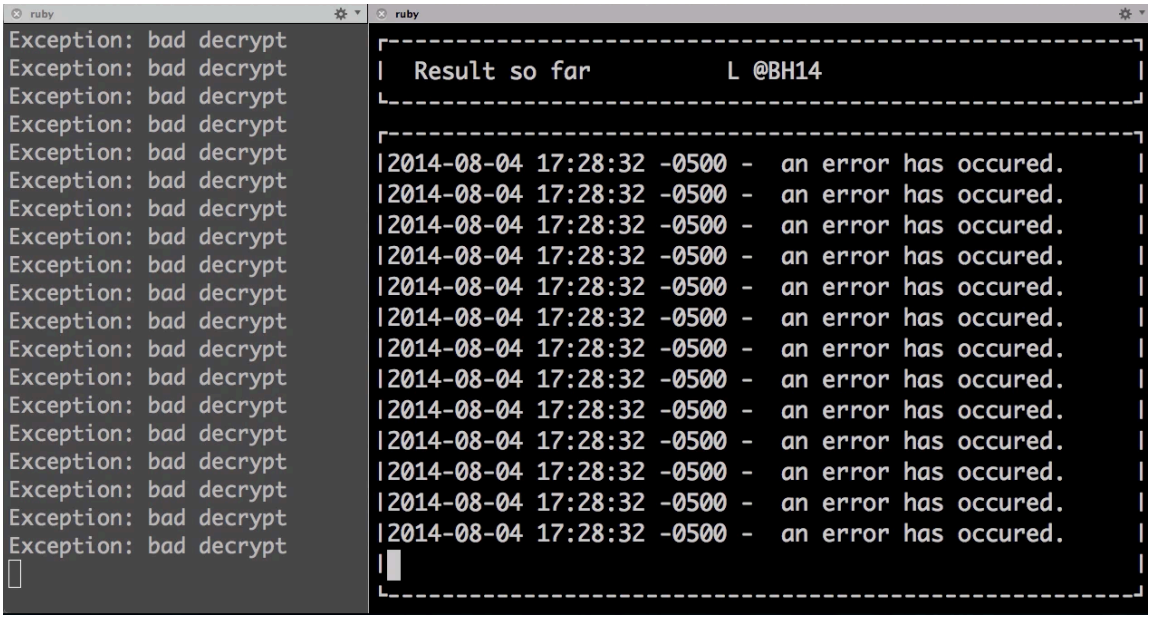


FIGURE 32: CBC Padding Oracle: Screenshot of a CBC padding oracle attack solely based on response time analysis.

5.3 CONCLUSIONS

The use of optimized string comparison functions and support for atomic comparisons of 64-bit words makes exploitation of comparison-based timing side channels highly unlikely for applications that leverage those features on modern hardware. A very privileged network position or attacks performed on the same host increase the resolution of attacks significantly and may give an attacker an additional advantage. However, based on our analysis, this advantage appears insufficient for practical remote attacks. Since regular string comparison of sensitive data still does leak some information (even if not exploitable remotely) as part of a best practice approach one should implement defenses as discussed in the next section. In particular, embedded systems remain at greater risk due to their slower processing speeds, and Arduino and other embedded systems appear to be feasible targets.

The impact of branching-based side-channels depends on application, but common complex operations appear to introduce delays that are significant enough to distinguish by a remote attacker. For instance, any form of I/O, database queries, even computation of hash functions adds enough processing time for a remote attacker to detect and potentially exploit.

6 preventing timing attacks

In the previous section we have seen which kind of timing flaws can potentially be exploited in practice. Below, we discuss the counter measures one can take in order to minimize or eliminate timing flaws in applications.

The general recommendation is that all operations involving sensitive data should execute in constant time, i.e., the execution time should not depend on the sensitive value involved. As it turns out, while this can be achieved for some high-level operations, it is hard to realize for all algorithms. However, there are some well-established solutions for common operations that we discuss below.

6.1 BRANCHING

For branching-based vulnerabilities there is no cut-and-paste solution against timing attacks. Custom work is required based on the underlying scenario.

6.1.1 Authentication

For user authentication, one needs to compute the password hash regardless of whether the user is valid and perform the comparison of both the user and the password hash using constant time comparison functions (discused below). If database queries are involved, the joint query for both username and password hash may hide timing differences well enough such that they are not exploitable.

6.1.2 Padding Oracles

Padding oracles are only exploitable if the attacker is able to tamper with the ciphertext sent to the server. In order to prevent this kind of attack, ciphertexts should be authenticated [Wik14a] which prevents (or rather detects) any modification performed by a third party. When using AES, the preferred way of doing this is to use a cipher mode that provides both confidentiality and authentication at the same time. *Galois/Counter Mode* (GCM) is a widely available example of such a mode. Should the language used not support this, CBC mode combined with an HMAC is the next best choice. When using an HMAC combined with CBC mode, verifying the HMAC before attempting decryption will detect and thus prevent any unwanted modification. There are a few caveats when implementing this. First, the HMAC should use a different secret key than the AES encryption. Second, the HMAC has to be verified using a constant time comparison function (see next section). Only if the HMAC is valid should the ciphertext be decrypted.

6.2 STRING COMPARISON

In order to prevent timing vulnerabilities in string comparisons, the function must not return early when a different byte is found in the input strings. The generic way of achieving (see Listing 16) this is to first determine if the compared strings have the same length and return false otherwise. The algorithm then continues by performing a byte-wise XOR operation while "summing-up" the results using a bit-wise OR operation. If two bytes are equal, the XOR will be zero and one otherwise. The result of the oerall OR operation will only be equal to zero if all bytes XORed to zero. Thus checking if the result yielded zero allows us to determine whether the strings were equal. The use of XOR on each pair of bytes being compared ensures that the computation time per byte is constant, regardless of the values of the strings.

LISTING 16: PSEUDOCODE FOR CONSTANT-TIME COMPARISON

```
function constant_equal(a, b):
    if length(a) != length(b):
        return false

    result = 0
    for (i = 0; i < length(a); i++)
        result = result OR (x XOR y)
    return (result == 0)
```

Note that by first checking whether the strings are of equal length introduces a more subtle timing channel: an attacker can learn the *length* of the credential being checked. While disclosing the length typically is not a concern, these solutions should not be used for applications where this is a security issue.

There has been research [Hil11] indicating that some compilers optimize constant-time comparison functions and may thus re-introduce timing leaks. For HMAC verification, the work recommends to compute an additional HMAC of the values before comparing them. In fact, just applying a cryptographically secure hash function will have a similar effect.

6.2.1 Ruby

Rails provides `secure_compare` in `ActiveSupport::MessageVerifier`. Similar functions are available in other frameworks such as Rack.

LISTING 17: CONSTANT-TIME COMPARISON IN RUBY [Gro14]

```ruby
def secure_compare(a, b)
 return false if a.empty? || b.empty? || a.bytesize != b.bytesize
 l = a.unpack "C#{a.bytesize}"

 res = 0
 b.each_byte { |byte| res |= byte ^ l.shift }
 res == 0
end
```

6.2.2 Python

Python's `hmac.compare_digest` performs constant-time comparisons.

6.2.3 PHP

PHP 5.6 is introducing the `hash_equals()` function for constant-time comparisons.

6.2.4 Java

LISTING 18: GENERIC CONSTANT-TIME COMPARISON IN JAVA

```java
public static final boolean isEqual(final String a, final String b) {
  if (a.length() != b.length()) {
    return false;
  }
  int result = 0;
  for (int i = 0; i < a.length(); i++) {
    result |= a.charAt(i) ^ b.charAt(i);
  }
  return result == 0;
}
```

6.2.5 C# / ASP.net

LISTING 19: GENERIC CONSTANT-TIME COMPARISON IN C#

```
private static bool SlowEquals(byte[] a, byte[] b)
   {
       uint diff = (uint)a.Length ^ (uint)b.Length;
       for (int i = 0; i < a.Length && i < b.Length; i++)
           diff |= (uint)(a[i] ^ b[i]);
       return diff == 0;
   }
```

6.2.6 Node.js

The `buffer-equal-constant-time` node package performs constant-time buffer comparisons.

6.2.7 Clojure

LISTING 20: CONSTANT-TIME COMPARISON IN CLOJURE [Ree14]

```
(ns crypto.equality
  "Securely test sequences of data for equality.")

(defn eq?
  "Test whether two sequences of characters or bytes are equal in a way that
  protects against timing attacks. Note that this does not prevent an attacker
  from discovering the *length* of the data being compared."
  [a b]
  (let [a (map int a), b (map int b)]
    (if (and a b (= (count a) (count b)))
      (zero? (reduce bit-or (map bit-xor a b)))
      false)))
```

6.3 PASSWORD COMPARISON AND SALTED HASHING

Use a constant time comparison function when handling and comparing password hashes, API keys, session identifiers, or other authentication tokens directly in memory.

It is important to note that using a non-salted hash with an early-terminating comparison function (e.g., for password verification) still leaks information about the correct password. This is due to the fact that the attacker is able to compute the hash value and thus can infer which value

in the comparison of the hash values differed. Armed with this information, they are then able to eliminate candidate password whose hash value does not begin with the correct substring. However, it is not possible to fully recover a valid password in this manner, since this would require a pre-image attack on the underlying hash function.

7 References

[BB03] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *USENIX Security Symposium*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[BDL97] Dan Boneh, RichardA. DeMillo, and RichardJ. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology (EUROCRYPT)*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 1997.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer Berlin Heidelberg, 2012.

[CRR03] Suresh Chari, JosyulaR. Rao, and Pankaj Rohatgi. Template Attacks. In BurtonS. Kaliski, çetinK. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer Berlin Heidelberg, 2003.

[CWR09] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Transactions on Information and System Security*, 12(3):17:1–17:29, 2009.

[DMS06] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Pearson Education, 2006.

[DR11] Thai Duong and J. Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. In *Security and Privacy*, pages 481–489, May 2011.

[Dri] Vincent Driessen. Python RQ (Redis Queue). `http://python-rq.org/`.

[Gir05] Christophe Giraud. DFA on AES. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard - AES*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer Berlin Heidelberg, 2005.

[Gro14] Levi Gross. Constant Time Comparison Functions in… Python, Haskell, Clojure, Java etc.. `http://www.levigross.com/2014/02/07/constant-time-comparison-functions-in-python-haskell-clojure-java-etc/`, 02 2014.

[GST13] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. `http://eprint.iacr.org/`.

[Har12] D. Hardt. The OAuth 2.0 Authorization Framework. Technical report, Internet Engineering Task Force (IETF), 10 2012. RFC 6749.

[Hil11] Brad Hill. Double HMAC Verification. https://www.isecpartners.com/blog/2011/february/double-hmac-verification.aspx, 2011.

[HL10] E. Hammer-Lahav. The OAuth 1.0 Protocol. Technical report, Internet Engineering Task Force (IETF), 4 2010. RFC 5849.

[Hol10] Brian Holyfield. Automated Padding Oracle Attacks With PadBuster. http://blog.gdssecurity.com/labs/2010/9/14/automated-padding-oracle-attacks-with-padbuster.html, 2010.

[Hou09] R. Housley. Cryptographic Message Syntax. Technical report, 9 2009. RFC 5652 - Section 6.3.

[jbe11] jbenet. clock_gettime alternative in Mac OS X. https://stackoverflow.com/questions/5167269/clock-gettime-alternative-in-mac-os-x, 2011.

[JH12] M. Jones and D. Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. Technical report, Internet Engineering Task Force (IETF), 10 2012. RFC 6750.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397. Springer-Verlag, 1999.

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction To Differential Power Analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[Law10] Nate Lawson. Optimized Memcmp Leaks Useful Timing Differences. http://rdist.root.org/2010/08/05/optimized-memcmp-leaks-useful-timing-differences/, 08 2010.

[LN10] Nate Lawson and Taylor Nelson. Exploiting timing attacks in widespread systems. In *Black Hat Briefings*, 2010.

[Mar] Sebastien Martini. Security advisory: Timing attack in HMAC signature verification of Python OAuth. Technical report.

[MS14] Daniel Mayer and Joel Sandin. Time Trial Code on Github. https://github.com/dmayer/time_trial, 2014.

[otcnp] Contributors of the cpp-netlib project. The C++ Network Library Project. http://cpp-netlib.org.

[OWA14] OWASP. REST Security Cheat Sheet. https://www.owasp.org/index.php/REST_Security_Cheat_Sheet, 4 2014.

[PY04] KennethG. Paterson and Arnold Yau. Padding Oracle Attacks on the ISO CBC Mode Encryption Standard. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*,

volume 2964 of *Lecture Notes in Computer Science*, pages 305–323. Springer Berlin Heidelberg, 2004.

[RD10a] Juliano Rizzo and Thai Duong. Practical Padding Oracle Attacks. In *Black Hat Europe*, 2010.

[RD10b] Juliano Rizzo and Thai Duong. Practical Padding Oracle Attacks. In *WOOT*, 2010.

[Red] Redis Key-Value Store. `http://redis.io/`, Citrusbyte.

[Ree14] James Reeves. Clojure Constant Time Comparison. `https://github.com/weavejester/crypto-equality/blob/master/src/crypto/equality.clj`, 2014.

[San10] Eloi Sanfèlix. On Padding Oracles, CBC-R and timing attacks.... `http://www.limited-entropy.com/po_cbc-r_and_timing/`, 2010.

[Sch11] Sebastian Schinzel. Time is on my Side - Exploiting Timing Side Channel Vulnerabilities on the Web. In *28th Chaos Communication Congress - Behind Enemy Lines*, 2011.

[Sch12] Sebastian Schinzel. Time is NOT on Your Side - Mitigating Timing Side Channels on the Web. In *29th Chaos Communication Congress - Not My Department*, 2012.

[SP11] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, 2011.

[Sto13] Stormpath. Secure Your REST API... The Right Way. `https://stormpath.com/blog/secure-your-rest-api-right-way/`, 4 2013.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... In LarsR. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–545. Springer Berlin Heidelberg, 2002.

[vE85] Wim van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4(4):269 – 286, 1985.

[Wik14a] Wikipedia. Authenticated encryption — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Authenticated_encryption&oldid=612536176`, 2014. [Online; accessed 30-June-2014].

[Wik14b] Wikipedia. Block cipher mode of operation — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=614311551`, 2014. [Online; accessed 29-June-2014].

[Wik14c] Wikipedia. Padding (cryptography) — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Padding_(cryptography)&oldid=598396110`, 2014. [Online; accessed 29-June-2014].

[Zal12] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.