

# Detecting Hidden Storage Side Channel Vulnerabilities in Networked Applications

Felix C. Freiling and Sebastian Schinzel\*

University of Mannheim, Laboratory for Dependable Distributed Systems

**Abstract.** Side channels are communication channels that were not intended for communication and that accidentally leak information. A *storage* side channel leaks information through the *content* of the channel and not its *timing* behavior. Storage side channels are a large problem in networked applications since the output at the level of the protocol encoding (e.g., HTTP and HTML) often depends on data and control flow. We call such channels *hidden* because the output differences blend with the noise of the channel. Within a formal system model, we give a necessary and sufficient condition for such storage side channels to exist. Based on this condition, we develop a method to detect this kind of side channels. The method is based on systematic comparisons of network responses of web applications. We show that this method is useful in practice by exhibiting hidden storage side channels in three well-known web applications: Typo3, Postfix Admin, and Zenith Image Gallery.

## 1 Introduction

**Covert Channels and Side Channels.** A *covert* communication channel is a communication channel that was not intended to transfer information at all [15]. Today it is accepted that covert channels are impossible to avoid and hard to control. In practice, the sender  $P$  often does *not intend* to communicate with  $C$  but still leaks information. To distinguish this from the scenario of covert channels, the term *side channel* was used. Side channels abound and have been the focus of much research in areas like cryptographic hardware [11], API design [5] or non-electronic media [3].

In this paper we focus on side channels in applications running in the World Wide Web (web applications). Web applications are a prime communication mechanism today and side channels in web applications are relevant. Side channels are known as a special type of covert channel and covert channels are categorized into timing and storage channels. We thus adopt this discrimination for side channels and extend the general area of covert channels by distinguishing *storage side channels* from *timing side channels*. Whereas timing side channels are well researched in the World Wide Web, to our knowledge there is no well-founded research on the detection of storage side channels in web applications.

To fill this gap, we give a general method by which storage side channels can be detected in web applications. As main example we show that in many existing web applications with user management it is possible to find out whether a certain high-privileged user account exists or not. This information is usually treated as confidential because knowledge of high-privileged user names eases password guessing and phishing attacks.

**Side Channels in Web Applications: Related Work.** Most related work has focused on constructing or detecting *timing* channels on the web (see for example Felten and Schneider [12], Bortz, Boneh and Nandy [7] or Nagami et al. [17]). A timing covert channel appears when one “process signals information to another by modulating its own use of

---

\* Sebastian Schinzel was supported by Deutsche Forschungsgemeinschaft (DFG) as part of SPP 1496 “Reliably Secure Software Systems”.

system resources (e.g., CPU time) in such a way that this manipulation affects the real response time observed by the second process” [18].

Only comparatively few authors have investigated possible *storage* channels on the web, i.e., channels that reside in the direct output of web applications. Bowyer [8] and Bauer [4] describe the possibility of hidden channels in HTTP. Bowyer suggests using superfluous URL parameters to form an upstream connection, and HTTP header fields or image steganography as downstream connection. Bauer chooses HTTP redirects, Cookies, referrer headers, HTML elements or active content to hide communication over the HTTP protocol. Kwecka [14] summarizes five different methods that were used to create covert channels within application layer headers: Reordering of headers, case modifications, use of optional fields and flags, adding a new field, and using linear white space characters.

We are aware of only two papers on the detection of hidden storage channels on the web. Chen et al. [9] detected a side channel in the packet size of encrypted traffic between browser and several web applications that process critical information such as healthcare information. Their target was to detect side channels that are visible for a man-in-the-middle who has access to the encrypted network traffic. Borders and Prakash [6] present a method to determine the maximum bandwidth of covert channels. They focus on covert channels where  $P$  is the web browser and  $C$  is the web server, which is the opposite direction of our approach. Their method neither detects, nor prevents covert channels, but focuses on determining the upper boundary of a covert channel’s bandwidth. They analyze storage covert channels that enabled  $P$  to *purposefully* pass information to  $C$ . However, storage *side* channels appear if  $P$  *accidentally* and *unknowingly* passes information to  $C$ .

**Detecting Storage Side Channels in Web Applications.** In this paper, we exhibit a method for detecting storage side channels by comparing the differences in multiple responses. We call a storage side channel *hidden* if users will not detect it by observing only the visible elements of responses. In contrast to timing attacks, we measure differences in the *content* of the responses of web applications. The idea is to correlate the differences of web application responses with some secret information stored in the web server. Storage side channels are noisy, i.e. if a user performs the same requests multiple times, many web applications will seemingly display the same content in the user’s web browser. However, looking closely at the data of the web server’s responses, one discovers that the responses differ slightly in many cases. We show that by a structured analysis it is possible to detect those differences that correlate with secret information, and are thus leaking information. Applying this method to three practical systems (Typo3, Postfix Admin, and Zenith Image Gallery), we were able to extract confidential information about existing user names and private images in public galleries.

**Contributions.** In this paper, we study deterministic hidden storage side channels in web applications and make the following three contributions:

- We formalize the context in which side channels in networked applications occur and identify necessary and sufficient conditions for hidden storage side channels to exist.
- We develop a general method to detect hidden storage side channels in networked applications and apply the method to web applications.
- We use our method to identify side channels in three common web applications: the popular content management system Typo3 [2], Postfix Admin [1], which is a web-based administration system for the popular email server Postfix, and Zenith Image Gallery [10], a web-based image gallery.

**Outlook.** The paper is structured as follows: In Sect. 2 we introduce a formal model of hidden storage side channels and give a necessary and sufficient condition for their existence. In Sect. 3 we derive a detection technique for such channels in networked

applications and apply the method to three well-known web applications in Sect. 4. We conclude in Sect. 5.

## 2 Hidden Storage Side Channels in Networked Applications

We now abstract from concrete network protocols and investigate general networked applications and their susceptibility to side channels.

**System Model.** Our system model, depicted in Fig. 1, consists of an information producer  $P$  and an information consumer  $C$ . Both are modelled as *deterministic* state automata. Both  $P$  and  $C$  are connected through a shared resource  $R$ . In our case,  $R$  can be thought of as a network connection such as one using TCP over the Internet. Abstractly,  $R$  just allows to reliably transport a stream of bytes from  $P$  to  $C$ . The information that

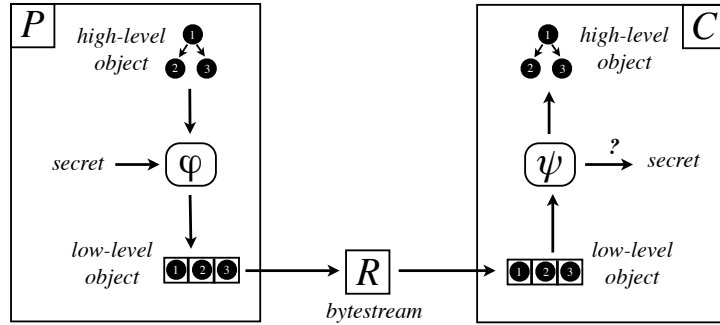


Fig. 1. System model.

$P$  sends to  $C$  can be any “high-level” digital object, e.g., a graph, a web page or a spreadsheet.  $P$  carefully constructed the high-level object in a way so that it only contains information that  $C$  is allowed to see. Producer  $P$  uses an encoding function  $\varphi$  that maps the high-level object into a bytestream suitable for transport over  $R$ . Similarly,  $C$  uses a decoding function  $\psi$  that transforms the received bytestream into the same high-level object again. Note that  $P$ ’s high-level object is the same as  $C$ ’s high-level object.

Formally, let  $\mathcal{H}$  denote the set of all abstract “high-level” objects and  $\mathcal{L}$  denote the set of all allowed byte-sequences that can be transported over  $R$ . We assume that  $\mathcal{L}$  is closed under subsequences, i.e., if  $x \in \mathcal{L}$  and  $x'$  is a subsequence of  $x$  then  $x' \in \mathcal{L}$  as well. We use  $x \cdot y$  to denote the concatenation of two byte-sequences  $x, y \in \mathcal{L}$ .

Using this notation,  $\varphi$  is a function that maps elements from  $\mathcal{H}$  to  $\mathcal{L}$ , and  $\psi$  is a function that maps elements from  $\mathcal{L}$  to  $\mathcal{H}$ .

**Definition of Hidden Storage Side Channels.** A covert channel is a unidirectional communication channel that allows  $P$  to covertly communicate with  $C$  over  $R$ .  $P$  and  $C$  agreed on  $\varphi$  and  $\psi$  beforehand. The purpose of those channels is to hide the fact that  $P$  communicates with  $C$  over  $R$ .

A side channel is a special covert channel in which  $P$  *unintentionally* communicates with  $C$  over  $R$ . Side channels do not appear on purpose but appear accidentally during system construction.  $C$ ’s challenge is to discover and decode the side channel in order to get the information from  $P$ .

Intuitively, a storage side channel allows  $C$  to infer a secret value from the content that  $P$  sent. A hidden storage side channel is a storage side channel that can only be detected

in low-level values from  $\mathcal{L}$ , as the high-level values are identical so that an observer of  $\mathcal{H}$  will not detect the differences.

In general, we assume that the decoding function  $\psi$  is the inverse of the encoding function  $\varphi$ , i.e.,  $\forall h \in \mathcal{H} : \psi(\varphi(h)) = h$ . However, many encoding formats in practice contain redundancy in the sense that the same high level objects can be encoded differently by  $\varphi$ . This can cause hidden storage side-channels, as we now explain.

Abstractly, a storage side channel regarding some secret information exists if  $P$  encodes some object  $h \in \mathcal{H}$  dependent on some secret (see Fig. 1). For example, if the secret consists of just one bit,  $P$  could encode  $h$  into  $l_0$  if the bit is 0, and into  $l_1 \neq l_0$  if the bit is 1. If both  $l_0$  and  $l_1$  will be decoded into the same high-level object  $h$  again, hidden storage side channels appear. Investigating the encoding of  $l_0$  and  $l_1$  reveals the secret.

**A Necessary and Sufficient Requirement.** Note that our system model subsumes the scenario of Kemmerer [13] since  $P$  and  $C$  are connected through a direct communication link. Therefore, it is clear that side channels may exist. However, we refine Kemmerer’s conditions to our case and derive a sufficient requirement for hidden storage side channels to exist in our system model.

It is sufficient for storage side channels to exist in our system model if there exist  $\psi$ -synonyms in  $\mathcal{L}$ , i.e. there exist distinct values  $l_1$  and  $l_2$  in  $\mathcal{L}$  that  $\psi$  maps into the same value in  $\mathcal{H}$ , formally:

$$\exists l_1, l_2 \in \mathcal{L} : l_1 \neq l_2 \wedge \psi(l_1) = \psi(l_2)$$

To prove that this is a sufficient condition we have to show how a hidden storage side channel can be constructed. This is rather straightforward, as explained above: Depending on the secret  $s$ , the producer (possibly unintentionally) selects either  $l_1$  (for  $s = 0$ ) or  $l_2$  (for  $s = 1$ ). By investigating the low-level encoding of the message, i.e., before applying  $\psi$ , the consumer can learn the value of  $s$ .

We now argue that the condition of  $\psi$ -synonyms is also a necessary condition. For this we have to prove that the existence of hidden storage side channels implies that  $\psi$ -synonyms hold on the encoding.

So assume that we have a system in which hidden storage side channels exist. From the definition of *storage* side channel and in contrast to timing side channel, we know that information is transported using the *content* of messages exchanged over  $R$ . Since the channel is hidden, the channel is not visible on the level of  $\mathcal{H}$ .

There are two cases to consider:

1. The content directly reflects secret information, i.e., a value  $l_1$  implies  $s = 0$  and  $l_2$  implies  $s = 1$ . In this case, we directly have the condition of  $\psi$ -synonyms.
2. The content reflects secret information via the order of byte sequences exchanged over  $R$ . In the simplest case, this means that there are two byte sequences  $l$  and  $l'$  such that  $l \cdot l'$  encodes  $s = 0$  and  $l' \cdot l$  encodes  $s = 1$ . Note that  $\psi(l \cdot l')$  must be equal to  $\psi(l' \cdot l)$ . In this case, choose  $l_1 = l \cdot l'$  and  $l_2 = l' \cdot l$  and the condition of  $\psi$ -synonyms holds again.

So overall, if hidden storage side channels exist, then the requirement of  $\psi$ -synonyms must hold. Therefore, the requirement is not only sufficient but also necessary.

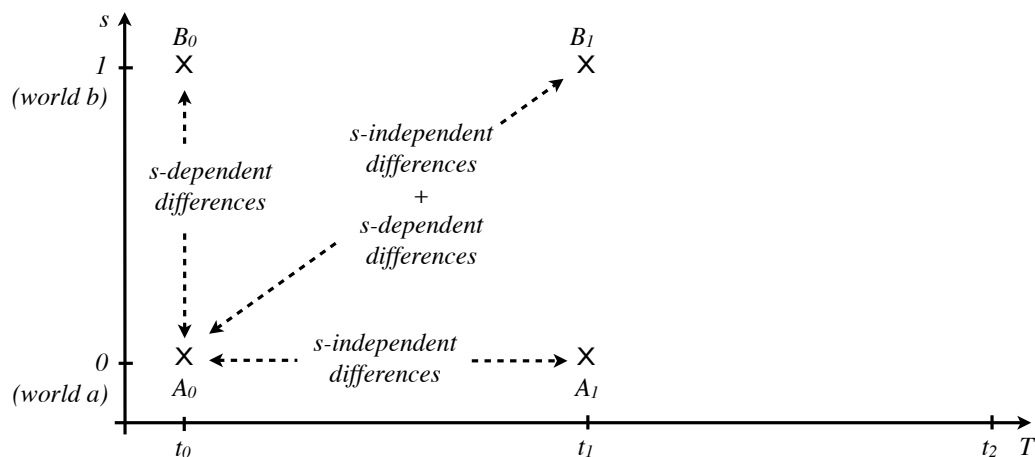
**Attacker Model.** We assume that the attacker is in full control over the the information consumer. The goal of the attacker is to deduce the secret that resides in the information producer  $P$ . The allowed interactions between the attacker and  $P$  must conform to the protocol used on the shared resource (e.g., HTTP), i.e., we do not consider protocol-specific attacks.

### 3 Detecting Storage Side Channels in Networked Applications

We now study the question, how to detect the existence of storage side channels for a given networked application. This is a non-trivial task since the dependency of the low-level encoding on the secret may not be known or may be time or context dependent.

**Secret-dependent and Secret-independent Differences.** Consider the case where  $P$  leaks secret information over a storage side channel to  $C$  and this secret consists of one bit  $s$ . Now assume two possible worlds  $a$  and  $b$  that are identical in all aspects except the value of the secret bit  $s$ : In world  $a$  we have  $s = 0$  and in world  $b$  we have  $s = 1$ .  $P$  now sends a message via  $R$  to  $C$ . Let  $A$  denote the message sent in world  $a$ , and let  $B$  denote the message sent in world  $b$ . If  $a$  and  $b$  are identical except for the values of  $s$ , then any difference between  $A$  and  $B$  is a direct influence of the difference of the secret value. (Recall that both  $P$  and  $C$  are deterministic automata.) Hence, measuring the differences between  $A$  and  $B$  yields *secret-dependent* differences in messages.

Ideally, we want to identify secret-dependent differences since they can be used to infer the value of  $s$ . However, in practice, it is difficult to construct two identical worlds  $a$  and  $b$  in which  $P$  has absolutely the same state (even using virtualization). To make our method more robust and practical, we allow observing messages at different points in time as shown in Fig. 2. If we send two successive messages  $A_0$  and  $A_1$  in world  $a$ , for example, then we can measure the differences between these two messages. Since  $s = 0$  for both messages, any differences between  $A_0$  and  $A_1$  must be *secret-independent*. Secret-independent differences can result from different times or relative orders of the sending of  $A_0$  and  $A_1$ . Now assume that a message in world  $a$  is sent at time  $t_0$  and another



**Fig. 2.** Realistic measurements yield a mix of secret-dependent and secret-independent differences. The challenge is to reliably extract secret-dependent differences from the mix.

message in world  $b$  is sent at time  $t_1$ . Let  $A_0$  denote the former and  $B_1$  denote the latter message. If we compare  $A_0$  and  $B_1$ , we find a composition of secret-dependent and secret-independent differences (see Fig. 2). The challenge is to identify the secret-dependent differences and associate them with particular values of  $s$ . In the following section, we describe our method that we used to successfully uncover storage side channels in web applications.

**Definitions and Notation.** Let  $A$  be a byte sequence  $A = a_0, a_1, a_2, \dots$ . We define an *edit*  $e$  of  $A$  to be a tuple  $(p, q)$  such that

1.  $p \in N : 0 \leq p < |A|$  and
2.  $q \in N : 0 \leq q \leq |A| - p$ .

Intuitively, an edit describes a “change” of the byte sequence, more specifically a removal of  $q$  bytes starting at position  $p$ . We formalize this by defining what it means to apply an edit to a byte sequence.

Let  $e = (p, q)$  be an edit of  $A$ . The *application of  $e$  to  $A$* , denoted  $A|_e$  is the byte sequence resulting from removing  $q$  bytes starting at index  $p$  in  $A$ , formally:

$$A|_e = a_0, a_1, \dots, a_{p-1}, a_{p+q}, a_{p+q+1}, \dots$$

Here are some examples: Assume  $A$  is the byte sequence “01234”. Then  $A|_{0,2} = 234$ ,  $A|_{1,3} = 04$ ,  $A|_{4,1} = 0123$ ,  $A|_{2,1} = 0134$  and  $A|_{1,0} = 01234$ . In contrast, both  $A|_{2,4}$  and  $A|_{5,0}$  are undefined because both  $(2, 4)$  and  $(5, 0)$  are not edits of  $A$ .

We now describe what it means to apply multiple edits to  $A$ . We say that an *edit  $(p, q)$  is compatible with edit  $(p', q')$*  iff (if and only if)  $p + q < p'$ . Intuitively, two edits are compatible if they affect different parts of  $A$ . They are not compatible (incompatible) if they overlap. For example, for  $A = 01234$ , edit  $(1, 3)$  and  $(4, 1)$  are compatible whereas  $(1, 4)$  and  $(4, 1)$  are incompatible.

Let  $\mathcal{E}$  be the set of all possible edits of  $A$  and  $E = e_1, e_2, e_3, \dots, e_n$  be a sequence of  $n$  edits of  $A$  such that for all  $i$ ,  $0 < i < n$ ,  $e_i$  is compatible with  $e_{i+1}$ . The *application of  $E$  to  $A$* , denoted  $A|_E$ , is defined as:

$$(\dots((A|_{e_n})|_{e_{n-1}})|_{e_{n-2}}\dots)|_{e_1}$$

Note that the definition applies edits in reverse order, i.e., starting with  $e_n$ . This is not a necessary requirement as edits could be applied in any order. However, it simplifies the definition since index numbers for subsequent edits  $e_{n-1}, e_{n-2}, \dots$  remain the same.

In the next step, we use the notion of edits to define the longest common subsequence and a special difference operator  $\Delta$ . Let  $A$  and  $B$  be two byte sequences. The byte sequence  $x$  is a *common subsequence (CS)* of  $A$  and  $B$  iff there exists two sequences of compatible edits  $E_A$  and  $E_B$  such that  $A|_{E_A} = x$  and  $B|_{E_B} = x$ . In other words,  $x$  can be constructed from  $A$  by applying  $E_A$  to  $A$  and  $x$  can be constructed from  $B$  by applying  $E_B$  to  $B$ .

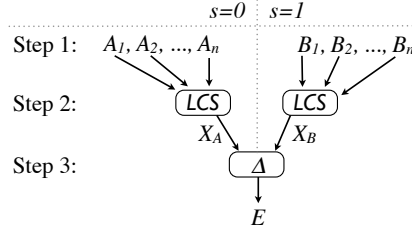
The byte sequence  $x$  is a *longest common subsequence (LCS)* of  $A$  and  $B$  if  $x$  is a common subsequence of  $A$  and  $B$  and  $|x|$  is maximal. We denote this by writing  $x = LCS(A, B)$ .

Note that there can be multiple *LCSs* for pairs of byte sequences. For example, for  $A = 212$  and  $B = 121$  there are two longest common subsequences, namely 21 and 12. In our implementation, we chose to return the “leftmost” *LCS* of the left parameter, i.e.  $LCS(212, 121) = 21$  and  $LCS(121, 212) = 12$ .

We are now ready to define the “difference” operator  $\Delta$  that will be important in our method to detect storage side channels. Let  $A$  and  $B$  be two byte sequences. The operation  $A \Delta B$  results in a compatible sequence of edits  $E$  of  $A$  such that  $A|_E = LCS(A, B)$ . In other words,  $E$  is the sequence of edits needed to produce out of  $A$  a longest common subsequence of  $A$  and  $B$ . For example, let  $A = 212$  and  $B = 121$ , then  $A \Delta B = (2, 1)$  and  $A|_{(2,1)} = 21$ .

**The Difference Algorithm.** In practice, secret-independent differences often result from dynamic values like session identifiers or time stamps. Note that for time dependent values, the time period during which the measurements were taken is important. For example, if the day field of a time stamp changes only after 24 hours (once every day), we will not be able to observe this dependency if all measurements are taken within the same day. For two such byte strings  $A_1$  and  $A_2$ , we must expect  $|LCS(A_1, A_2)| > 0$ , i.e., even two randomly generated byte strings are likely to have an *LCS* with non-zero length. So the idea is to incrementally compute the *LCS* of a *sequence* of byte strings  $A_1, A_2, A_3, \dots$ , i.e.,

first  $LCS(A_1, A_2)$ , then  $LCS(LCS(A_1, A_2), A_3)$ , then  $LCS(LCS(LCS(A_1, A_2), A_3), A_4)$ , etc. At some point, say while going from  $A_i$  to  $A_{i+1}$ , the length of the resulting byte sequence will not change any more. This means that all random elements have been eliminated from all  $A_i$ , i.e., we have computed the “static core” of the response sequence  $A_i$ .



**Fig. 3.** The difference algorithm.

In the following, we will now generalize our observations into an algorithm to identify secret-dependent differences. Fig. 3 shows a graphical representation of the steps.

1. Record  $n$  responses with  $s = 0$  (denoted  $A_1, A_2, \dots, A_n$ ) and  $n$  responses with  $s = 1$  (denoted  $B_1, B_2, \dots, B_n$ ). The value of  $n$  depends on how many responses are required to detect all dynamic parts in the response’s content. Section 4 introduces empirical values of  $n$  for different applications.
2. For all  $1 \leq i \leq n$  recursively calculate

$$X_A = \begin{cases} A_1 & \text{for } i = 1 \\ A_i |_{A_i \Delta X_{i-1}} & \text{for } 1 < i \leq n \end{cases}$$

Compute  $X_B$  similarly using responses  $B_i$ . Intuitively, the byte sequences  $X_A$  and  $X_B$  correspond to the “static” parts of  $A_i$  and  $B_i$ , respectively, in which any secret-independent differences are removed. Finding the secret-independent differences  $E_A$  of a response  $A_1$  is straight forward:  $E_A = X_A \Delta A_1$ .

3. Now compute  $E = X_A \Delta X_B$ . The set  $E$  contains an approximation of all secret-dependent differences. If  $E \neq \emptyset$  the probability of a storage side channel is high.

Looking closely at the edits in  $E$  it is often possible to decode the storage side channel, as we show in the following sections where we apply this method to real-world web applications.

## 4 Application of Method to HTTP/HTML

We now apply our method to web applications in which we wish to detect storage side channels. Such side channels can be, for example, the information whether a certain user account exists in a web application. Another example is the fact whether certain data items (such as images) are stored in a database. We start by verifying whether the condition of  $\psi$ -synonyms is satisfied in HTTP/HTML. Then, the attacker has to perform two steps. First, the attacker needs to find out whether the targeted application leaks the required information over a storage channel. Second, given that the targeted application is vulnerable to storage side channels, the attacker creates a search pattern. If the search pattern is applied to a single response, the attacker can find out the secret.

**Storage Side Channels in HTTP and HTML.** We now apply our system model to HTTP and HTML, and define that  $P$  is the web server,  $C$  is the web client, and  $R$  is the TCP channel between  $C$  and  $P$ . The set of high-level objects  $\mathcal{H}$  consists of all web pages in the form of bitmaps displayed in  $C$ 's web browser. The set of low-level objects  $\mathcal{L}$  is the set of all ASCII sequences sent over  $R$ . Therefore,  $\varphi$  is the function used by  $P$  that “turns” a web page’s intended bitmap into a valid HTML document and attaches it to a HTTP response before it is sent via  $R$  to  $C$ . The function  $\psi$  is the parser in  $C$ 's web browser that reads the content of the HTTP response, creates the page’s bitmap, and displays it on the screen of  $C$ .

HTTP responses contain a set of common headers, one of which is the *Server* header. The value of this data field is usually set to the particular type and version of the web server. However, this value is almost never used by the client and can therefore be set arbitrarily without affecting the web page displayed in  $C$ 's web browser. Other examples for synonyms come from the fact that HTTP header fields are not case sensitive, i.e. “date” and “dATe” denote the same header field name. Moreover, HTML allows including meta information (e.g., author) in the HTML header which can be set to arbitrary values without affecting the content displayed in the browser.

To summarize, HTML as well as HTTP allow synonymous values and are therefore potentially vulnerable to storage side channels.

**Detecting Storage Side Channels in Web Applications.** As a precondition for our first attack scenario, the attacker needs to know at least two existing user names and two non-existent user names in one instance of the targeted software. The attacker performs  $n$  login requests with known to be valid user names and records the responses  $A_i$ . All login requests in this scenario will fail because the attacker does not know any valid password in the target system. Thus, the system will always return an error message. He then calculates  $X_A$  from all responses and observes the decreasing length of the static part. In parallel, the attacker performs  $n$  login requests with *invalid* user names and records the responses ( $B_i$ ). Again, these login attempts will fail because the user name does not exist. Then, he calculates  $X_B$ .  $n$  was large enough if the *LCS* of the responses stays constant for larger  $n$ .

The attacker then calculates the set of edits ( $E$ ) representing secret-dependent differences. If  $E \neq \emptyset$ , the application leaks information about whether a user name exists through a storage side channel.

For our second attack scenario, the attacker needs to have access to at least one gallery containing no private pictures ( $A$ ) and another gallery with at least one private picture ( $B$ ). The attacker records  $n$  responses of  $A$  and  $n$  responses of  $B$  and calculates  $X_A$ ,  $X_B$ , and  $E$ . If  $E \neq \emptyset$ , the application leaks information about whether private images exist in a gallery through a storage side channel.

For the implementation of our algorithm, we chose Myers’ *LCS* algorithm [16]. It has near linear time and space requirements with regard to the size of the two responses that need be compared. A Java implementation of the algorithm performed around 70 *LCS* operations per second on responses with 12 Kilobytes.

**Exploiting Storage Side Channels in Web Applications.** The attacker now creates a search pattern from the secret-dependent edits in  $E$ . The application of this pattern to the response of a login request to any instance of the vulnerable web application determines whether the login request was performed using a valid or invalid user name.

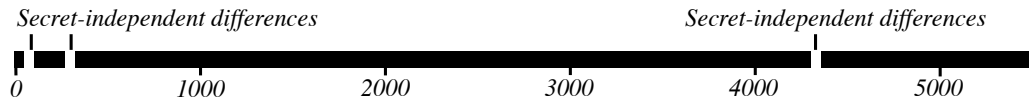
The calculation of  $E$  was a one time effort. From now on, the attacker can learn about the existence of a chosen user name with a single request and on any system on the web which runs the vulnerable software. The quality of the results is independent of the network conditions and of the amount of changed bytes in the secret-dependent difference.

In the following, we apply our method to three real-world examples.

**Example 1: Storage Side Channels in Typo3.** In our first example, we analyze whether the user authentication logic of the backend administration interface of Typo3 [2] version 3.6.0rc2-0.0.1, an open source and well-known content management system (CMS), is vulnerable of storage side channels. For this, we installed a copy of Typo3 in our lab. By default, Typo3 already creates one administrative user account (*admin*) during installation. It is possible to delete or rename this user name after installation, but we can assume that not all web administrators do this. Although we performed the measurements in our lab, this measurement could have been performed at any Typo3 instance on the web.

Let  $s$  be the boolean value denoting whether a user name exists. We now record a set of responses from login attempts with valid user names and another set with invalid user names. We then generate the “static core” of both sets, namely  $X_A$  and  $X_B$ . The set of edits  $E = X_A \Delta X_B$  yields the set of secret-dependent differences.

Interestingly, the different sizes of  $X_A$  and  $X_B$  correlates with  $s$  and therefore suggests that the analyzed functionality is vulnerable of storage side channels. Unfortunately, the size of the response will most certainly differ among several installations because of site-specific customizations of the web page and the web server type and configuration. An attacker would therefore need to analyze the response size for each target before performing the actual attack, which makes the attack easier to detect. Furthermore, if the web page contains secret-independent dynamic data with differing size, e.g. from a news ticker or changing ads, the response size is dynamic as well, which would require additional probabilistic analysis and even more measurements. Our method is immune to these disturbances as it produces the exact positions of secret-dependent differences, which allows the creation of search patterns for this particular change.



**Fig. 4.** The positions of secret-independent differences in a response of Typo3.



**Fig. 5.** The positions of secret-dependent differences in a response of Typo3.

Fig. 4 shows the positions of secret-independent differences in the byte stream of  $X_A$ . It is apparent that some values in the header, presumably time stamps, and one position in the HTML body change independently of  $s$ . Fig. 5 shows secret-dependent differences which form a storage side channel. This storage side channel leaks the existence of a given administrative user name. Note, that our method distinguishes secret-independent differences and secret-dependent differences even if they overlap, as it is the case here. It seems that the differences are located in the HTTP headers, because the positions of the secret-dependent differences are far to the left.

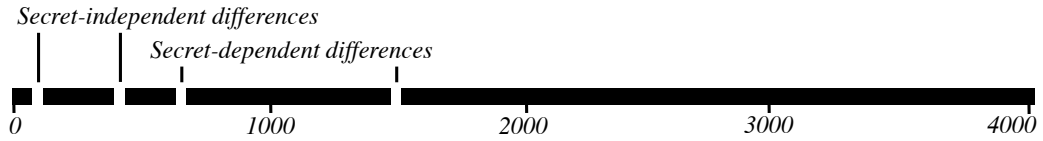
Fig. 6 shows a direct comparison of the HTTP headers generated with  $s = 0$  and  $s = 1$ , respectively. The secret-dependent differences are highlighted in both responses. A manual analysis of these exposed secret-dependent differences yields that not only the values of the HTTP headers in Typo3 change depending on  $s$  but also the order in which the headers are declared. Note that the content sizes of both responses are equal, i.e.

Non-existent user name ( $s=0$ )	Existing user name ( $s=1$ )
<pre> HTTP/1.1 200 OK Date: Mon, 25 Jan 2010 11:47:55 GMT Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny4 with Suhosin-Patch X-Powered-By: PHP/5.2.6-1+lenny4 Expires: <u>Thu, 19 Nov 1981 08:52:00 GMT</u> Last-Modified: <u>Mon, 25 Jan 2010 11:47:55 GMT</u> Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0 Pragma: no-cache Vary: Accept-Encoding Content-Type: text/html; charset=iso-8859-1 Content-Length: 5472 </pre>	<pre> HTTP/1.1 200 OK Date: Mon, 25 Jan 2010 11:47:45 GMT Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny4 with Suhosin-Patch X-Powered-By: PHP/5.2.6-1+lenny4 Expires: <u>0</u> Cache-Control: <u>no-cache, must-revalidate</u> Pragma: <u>no-cache</u> Last-Modified: <u>Mon, 25 Jan 2010 11:47:45 GMT</u> Vary: Accept-Encoding Content-Type: text/html; charset=iso-8859-1 Content-Length: 5472 </pre>

**Fig. 6.** Secret-dependent changes in Typo3’s HTTP header that depend on whether a user name exists.

the change of response size we observed earlier in this section does not affect the HTML content but solely happens in the HTTP header.

**Example 2: Storage Side Channels in Postfix Admin.** In our second example, we analyze whether the user authentication logic of Postfix Admin [1] version 2.3 rc7, an open-source Email administration frontend [1], is vulnerable of storage side channels. We calculated  $X_A$ ,  $X_B$ , and  $E$  as described previously. After roughly  $n = 15$  comparisons, the sizes of  $X_A$  and  $X_B$  do not change any more.



**Fig. 7.** The positions of secret-dependent and secret-independent differences in a response of Postfix Admin.

Non-existent user name ( $s=0$ )	Existing user name ( $s=1$ )
<pre> ... Content-Length: <b>3612</b> ... &lt;tr&gt;   &lt;td&gt;Login (email):&lt;/td&gt;   &lt;td&gt;&lt;input class="flat" type="text" name="fUsername"     value="" /&gt;&lt;/td&gt; &lt;/tr&gt; </pre>	<pre> ... Content-Length: <b>3626</b> ... &lt;tr&gt;   &lt;td&gt;Login (email):&lt;/td&gt;   &lt;td&gt;&lt;input class="flat" type="text" name="fUsername"     value="<u>admin@admin.de</u>" /&gt;&lt;/td&gt; &lt;/tr&gt; </pre>

**Fig. 8.** Secret-dependent changes in Postfix Admin’s HTTP header and HTML content that depend on whether a user name exists.

Fig. 7 shows the positions of the secret-dependent and the secret-independent differences in the responses. Fig. 8 shows a direct comparison of the secret-dependent differences in the HTTP headers and in the HTML content generated with  $s = 0$  and  $s = 1$ , respectively. The secret-dependent differences are highlighted in both responses. A manual analysis of the exposed secret-dependent differences yields that Postfix Admin automatically fills a user name in the form field of the response if and only if the user name exists in the database and is an administrative user. Strictly speaking, by the definition given in this paper, this vulnerability does not qualify as a *hidden* storage side channel,

as it *can* be observed in the browser. In practice, however, it is unlikely that this storage side channel is detected in a manual security test, because of the subtle nature of the difference. Actually, common web browsers may remember user names that a user filled in the same login form earlier, in order to automatically fill in the same user name for subsequent requests. We thus want to highlight, that our method will also detect “visible” storage side channels that would probably slip through most manual security analysis.

**Example 3: Storage Side Channels in Zenith Image Gallery.** In our last example, we analyze whether the Zenith Image Gallery [10] version v0.9.4 DEV, a well-known open source picture gallery, is vulnerable of storage side channels. Zenith allows administrators to upload pictures, which anonymous users can view. Administrators can also delete pictures from the gallery or mark them as private pictures. Private pictures are still shown to administrators but are hidden for anonymous users. Here, we analyze if an attacker can find out the existence of private pictures in the public gallery.

Let  $s$  be the boolean value denoting whether private pictures exist in a chosen gallery with seven public images and one additional private picture. We then record a set of responses  $B$  from this gallery with the additional private picture ( $s = 1$ ) and another set  $A$  from the same gallery in which the private picture is deleted ( $s = 0$ ). The measurements are performed as anonymous users and only 7 pictures are visible during both measurements. We then calculate  $X_A$ ,  $X_B$ , and  $E$ . After roughly  $n = 11$  comparisons, the sizes of  $X_A$  and  $X_B$  do not change any more.

```

7 public images, 0 private image (s=0)


---


<div style='float:left'>Pictures -
<a href='display.php?t=bycat&amp;q=4&amp;nr=7&amp;st=0&amp;upto=12&amp;p=1'>
  <span style='color:#fff'>Other</span>
</a>
</div>
7 public images, 1 private image (s=1)


---


<div style='float:left'>Pictures -
<a href='display.php?t=bycat&amp;q=4&amp;nr=8&amp;st=0&amp;upto=12&amp;p=1'>
  <span style='color:#fff'>Other</span>
</a>
</div>

```

**Fig. 9.** The secret-dependent difference in Zenith’s HTML content of a gallery that depends on the existence of private pictures in a gallery.

Fig. 9 shows a comparison of the HTML content of a gallery web page. Interestingly, only a single number depends on  $s$ .  $A$ -responses (no private images) have a constant size of 12460 bytes.  $B$ -responses (1 private image) have the same size with 12460 bytes.  $A$ -responses as well as  $B$ -responses have 35 secret-independent bytes. Only a single byte forms the hidden storage side channel. A manual analysis yields that the secret-dependent number represents the sum of public image and private images. In case of  $s = 1$ , the number is 8 as there are 7 public pictures plus 1 private picture in the gallery. Thus, the hidden storage side channel not only leaks the fact that there are private pictures in a gallery, but also the amount of private pictures in the gallery.

## 5 Conclusions

The success of our detection method critically depends on the secret that is to be observed and therefore it is hard to automate. This makes it extremely difficult to quantitatively assess the significance of side channel vulnerabilities in web applications in general. For

this paper, we analyzed 15 applications for *hidden* storage side channels and quickly identified the three examples documented here.

In principle, it is rather straightforward to avoid storage side channels by avoiding the condition of  $\psi$ -synonyms that we identified in this paper. This can be done, for example, by “fixing” the encoding function  $\varphi$  so that there are no synonymous values in  $\mathcal{L}$ . In reality, however, the encoding scheme is often given and, for the case of HTTP and HTML, it is not feasible to remove the general existence of synonymous values.

So in practice, more pragmatic approaches are necessary, such as trying to remove the relation between  $s$  and the choice of the synonymous values, to make the inference from  $\mathcal{L}$  objects to the value of  $s$  (and vice versa) impossible. This approach, however, largely depends on the vulnerable application and on how this application chooses synonymous values depending on  $s$ . In practice, therefore, side channels in web applications will have to be treated in a similar way as covert channels in other systems: Educate developers and give them effective techniques for detecting such channels in their systems. Furthermore, future protocol designers should carefully avoid conditions in which synonymous values are acceptable.

**Acknowledgments.** We thank Martin Johns for the helpful comments on a previous version of this paper.

## References

1. P. Admin. Web based administration interface, 2010. <http://postfixadmin.sourceforge.net/>.
2. T. T. Association. Typo3 content management system, 2010. <http://www.typo3.org/>.
3. M. Backes, M. Dürmuth, and D. Unruh. Compromising reflections-or-how to read LCD monitors around the corner. In *IEEE Symposium on Security and Privacy*, pages 158–169. IEEE Computer Society, 2008.
4. M. Bauer. New covert channels in HTTP. *CoRR*, cs.CR/0404054, 2004.
5. M. Bond and R. Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, Oct. 2001.
6. K. Borders and A. Prakash. Quantifying information leaks in outbound web traffic. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2009.
7. A. Bortz and D. Boneh. Exposing private information by timing web applications. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 621–628. ACM, 2007.
8. L. Bowyer. Firewall bypass via protocol stenography, 2002. [http://web.archive.org/web/20021207163949/http://networkpenetration.com/protocol\\\_steg.html](http://web.archive.org/web/20021207163949/http://networkpenetration.com/protocol\_steg.html).
9. S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. Oakland, CA, 05/2010 2010. IEEE.
10. CyberiaPC.com. Zenith picture gallery, 2007. <http://zenithpg.sourceforge.net/>.
11. European Network of Excellence (ECRYPT). The Side Channel Cryptanalysis Lounge. Internet: [http://www.crypto.rub.de/en\\_sclounge.html](http://www.crypto.rub.de/en_sclounge.html), Apr. 2010.
12. E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *SIGSAC: 7th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2000.
13. R. A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, Aug. 1983.
14. Z. Kwecka. Application layer covert channel - analysis and detection, 2006. <http://www.buchananweb.co.uk/zk.pdf>.
15. B. W. Lampson. A note on the confinement problem. *ACM*, 16(10):613–615, Oct. 1973.
16. E. W. Myers. An  $O(ND)$  difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
17. Y. Nagami, D. Miyamoto, H. Hazeyama, and Y. Kadobayashi. An independent evaluation of web timing attack and its countermeasure. In *Third International Conference on Availability, Reliability and Security (ARES)*, pages 1319–1324. IEEE Computer Society, 2008.
18. D. of Defense Standard. *Department of Defense Trusted Computer System Evaluation Criteria*, Dec. 1985.