Webpage Resource Protection via Obfuscation and Auto Expiry

Zhuhan Jiang and Jiansheng Huang
School of Computing, Engineering and Mathematics
University of Western Sydney, Sydney, Australia
{z.jiang, j.huang}@uws.edu.au

Abstract—**Content delivery via web pages or sites are becoming increasingly popular due to the effectiveness and versatility of the readily available delivery mechanism, especially in the e-education and training. While the copyright laws are there to protect the ownership and commercial rights of the intelligent properties, the openness of the web architecture often makes it impossible to prevent the content source being misappropriated or incorporated illegitimately elsewhere after some modifications on the downloaded source. We propose here an obfuscation mechanism for the HTML5 to convert a site of raw content into a site of obfuscated pages and images. With the advent of canvas on HTML5 and the AJAX to stop certain unauthorized access, the whole site of documents can be rendered meaningless or useless on both the server and the client side if just a small key part is modified or hidden. Several masquerading algorithms have been proposed for this purpose. The obfuscation will become permanent if a webpage is merely downloaded or even DOM-saved without having all necessary intermediate data or keys tracked by a specialist, before the auto-expiry of such a process, at a cost tantamount or exceeding the reconstruction of the original documents from scratches hence defeating the purpose of piracy. We applied the scheme to the delivery of a university subject by automating the whole process.**

Keywords-Content obfuscation; document ownership protection; e-training; HTML5 and AJAX; document auto-expiry.

I. INTRODUCTION

Web has long since surpassed its original purpose of publishing material on the Internet. It has evolved into a very effective and interactive platform to operate business, create social media, and run educational or training services, to name a few [1]. The largely sharing-by-all paradigm of the web during its inception has been gradually diverted into controlled access and restricted content or media deliveries, especially in e-business and e-education. The question of how much one can safeguard the ownership of the delivered content and to what extent naturally arises and becomes increasingly pertinent. It is generally understood that, if a piece of material is delivered via web to its authorized recipients, the document is practically fully surrendered in that almost all text and images there are at the disposal of the recipients in their original digital format, as long as the recipients have the minimal expertise on the web technologies. This means that these recipients may easily modify the content to reproduce and redistribute the original material. This can be highly undesirable for the protection of intelligent properties, especially when there are powerful web crawlers or site copiers [2]. Due to the common inherited belief that not much can be done in this regard, not much research efforts [3] have been made to seek as much as possible the protection of the delivered material, apart from setting up a few superficial obstacles such as [4] disabling the copy/paste, disallowing printing certain parts of the pages, using data URI scheme, and replacing a portion of a web page by a Javascript (JS) which converts the coded counterpart of the portion, e.g., in base 64, back to the Hyper Text Markup Language (HTML) format. However, these superficial tricks are only effective to the people of no or shallow technical skills.

Before we undertake to investigate how to protect our web source, we have to first establish the level of protection that we seek. If a piece of web content is to be delivered to a client's browser screen, there is no way one can stop the client from taking a picture or a video of the delivered material. Hence, a separate hardcopy or recording of the essential content does not belong to our protection scope. For convenience we refer this as the *knowledge scope*, and web content is thus unprotectable there. Next, we consider the *reproduction scope* in that the web content can be saved outside the original server and utilized to achieve essentially the same browsing experience. Currently, almost all regular web pages can be fully saved and are thus unprotectable in this scope apart from those live stream multimedia objects. Since most web contents are not ideal or practical to be streamed live, we will exclude the consideration of such objects in this work. The last scope we will also look into is the *source scope* in which we will examine whether the source can be saved in a clear and manageable format. A piece of source is considered to be in a readily manageable format if it is close to the original format that is suitable for modification or editing. Our aim in this work is to achieve a reasonable protection in the reproduction scope as well as in the source scope through obfuscation. Because the web delivered material can't be realistically protected in the knowledge scope, then the extent of protection is only limited to the understanding that the efforts required to reproduce the same browsing effect or manageable source are *not less* than the efforts required to start from scratch on the mere basis of "hardcopy" saved in the knowledge scope. Just like the industrial encryption algorithms are often theoretically breakable if there were an infinitely fast computer, but will be nonetheless treated as secure, as long as the current technologies are still way behind the power required to break the encryption, the protection of the web resource is often in comparison to what is required to reconstruct the resource from scratches. In this sense, if an unbearably significant amount of time or effort would be required to break or circumvent a protection, it still serves as a good protection.

It is worth noting at this point that while pure cryptography, with or without the Public Key Infrastructure, will in principle secure the transmission of almost anything digital, it often requires [5] additional support platforms, the measures to secure the keys, as well as the willingness to sacrifice certain performances or flexibilities. This also explains the needs for the obfuscation systems like the one we are proposing here.

This paper is organized as follows. First, Section II describes the basic design principle for the obfuscation of both the images and the text, utilizing HTML5 features, Asynchronous Javascript and XML (AJAX), and time-dependent keys. The automation process of the obfuscation is then outlined in Section III, with an implementation done for the demonstration in Section IV. Section V finally gives a conclusion.

II. THE DESIGN PRINCIPLE AND METHODOLOGIES

A. *The Basic Design Principle*

When a browser renders a page, it first loads a preliminary source of the page and then gradually loads other sources specified in the current or updated page content on a need to basis or recursively. Because additional page or resource loading may be activated by JS after interaction with a client user, the required resources in principle cannot be totally identified before a browsing session is fully completed. This means when JS are properly engineered, a

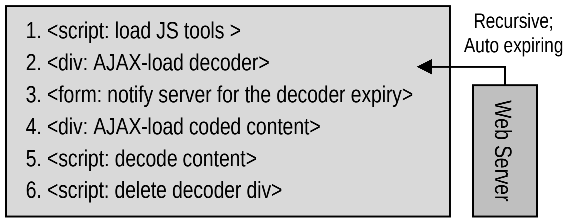

1. <script: load JS tools >
2. <div: AJAX-load decoder>
3. <form: notify server for the decoder expiry>
4. <div: AJAX-load coded content>
5. <script: decode content>
6. <script: delete decoder div>
Recursive;
Auto expiring
Web Server

Figure 1: A simplistic scheme

browser will have no memory of its intermediate interactive page modifications or loading by the time of saving a page, leaving the fully saved version still missing the intermediate components to function independently on its own. Some browsers, such as Chrome, as opposed to IE, will save maximally the page content by using the *resulting* Document Object Model (DOM) at the time of making a copy, and thus store elements additionally loaded through such as AJAX. However, the unavoidable loss of the intermediate execution memory still leaves room to craft the protection or obfuscation of the web source. A simplistic scheme is depicted in Fig. 1 in which a “decoder” in JS is dynamically loaded, directly or recursively, via AJAX, and the decoder expires immediately or automatically after a certain period of time. We note that a direct page saving will miss the decoder, and a manual tracing may hit the expiry time quantum particularly when recursive AJAX loading is utilized. When the same page is reloaded, it may load a different decoder or key valid within a different time quantum. Hence, if a part of the content to is to be dynamically reconstructed at the expired viewing time it will produce an illegible result. This is what we will call *lapse protection* in that the pages are unrecoverable after the site is disabled even though they are downloaded and “saved”. That is, completely recoverable pages must be constructed before the site life expectancy quantum has lapsed.

The coded content does not necessarily imply an encryption, and in many cases it is not encrypted at all. Some browsers can still save partly the images and text in the content. However this can be avoided too to the similar extent of lapse protection. The basic idea is to use JS to generate and rearrange the general text and use canvas in HTML 5 to superpose images and wipe out the loading traces of the component images. Because the JS portions need to be preserved so as to maintain the original interactivity, mixture of JS and plain HTML portion cannot be “flattened” out into just HTML without losing the interactivity. This explains that all browsers save the mixture of JS and text as they are, but the coded form of such a mixture wouldn’t be much more useful compared to a hard camera copy.

B. *Conceal the Images*

Since images constitute an important part of a typical web page, how to conceal the source of the original images deserves a separate consideration. Whenever an image is rendered in a web page, it appears in the form of an image element and will be typically stored as a part of DOM with its hosting page, even though some browsers may not do so when images appear in an AJAX-loaded DIV section. Our strategy comes with the advent of the CANVAS object

1. Replace all image tags of a given page with the canvas tags;
2. AJAX-load JS which is used to load all images into variables of image objects;
3. AJAX-load JS that maps images to canvases;
4. Draw and decode images on the corresponding canvases
5. Remove the element containing the JS loaded in steps 1 and 2

Figure 2: Draw images in canvases

which is meant to support interactive image modifications at the pixel level. Just like JS interactivity in general can’t be flattened into a piece of JS-free text, the dynamic nature of canvas implies that it is not stored state-wise when a page containing a canvas is saved. Hence the simplest strategy to protect images will be to display each image within a canvas, as described in Fig. 2.

However, images will stay in the cache for some time when they are loaded even if loading paths may be removed by JS within a web page. To avoid such cached images to be directly retrieved and made use of, one may load their distorted counterparts instead and use JS and a rectification key to dynamically to convert the images back to the original on the canvases. As the images are meant to be delivered to the recipient for viewing and can be camera-copied anyway, there is not much point to hide all picture details, and a reasonable image distortion will render the source not directly useable by a third party. There are infinitely many ways to distort an image, and we will consider several here in detail.

We start with a simple block-wise permutation of the images, with each block optionally subject to an additional "rescaling". For a given picture image P of resolution $M\times N$ pixels, we carve it up into blocks $\{ B_{i,j}: i=1,..,m, j=1,..,n \}$, or simply $\{ B_k: k=0, ..., K\text{-}1 \}$ with $K=m\cdot n$, of $\lfloor M/m\rfloor \times \lfloor N/n\rfloor$ pixels starting from the top left corner of the image, see Fig.

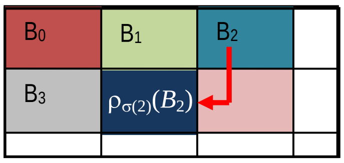


Figure 3: Image blocks

3. Let σ be any permutation of $0,1,.., K\text{-}1$, and ρ_k an invertible mapping that can be used to transform any images. If each image block B_k in P is replaced by $\rho_{k'}(B_{k'})$ where $k'=\sigma(k)$ is the index permuted from k by σ, then the resulting image P' is our distorted image which can be reverted back if one knows σ and all the ρ_k. For simplicity, we have kept intact the potential strips left over on the right and bottum due to the incomplete block partition. For any key ω in the form of a sequence of random characters, we can derive a corresponding permutation σ in the following way: (i) Repeatedly concatenate ω with itself if $|\omega|<K$; (ii) Let $c=\omega(0)$ be the 1st character of string ω, and $d \equiv c \bmod (K)$, and we assign $\sigma(0)=d$, i.e., 0 is permuted to d; (iii) Let $c=\omega[1]$ be the next character in ω, and $d \equiv c \bmod (K\text{–}1)$, set the ordered indices I as $\{0, 1, .., K\text{–}1\}\backslash\{\sigma(0)\}$ in increasing order, then the index $I(d)$ is the index that 1 will be permuted to, i.e.,

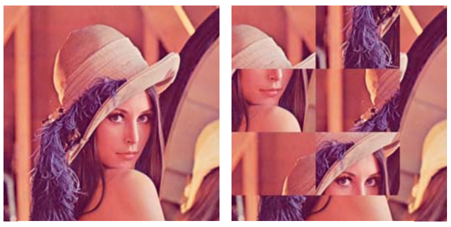


Figure 4: Image block permutation

$\sigma(1)= I(d)$; (iv) Repeat essentially step (iii) to have the permutation σ completely constructed. We have thus shown how a random string can uniquely determine a permutation of K indices. Fig. 4 shows an image block permutation with block size 200×150 pixels and ω=abc.

Next, let us explore what image transformers ρ_k we can design. The simplest transformers will be an identity mapping or an inverse mapping which produces a "negative" image. Although in theory one could decompose any image into a form of combination of two images one of which is pseudo randomly generated pixel-wise, such decomposition would result in massively reduced image compression [6][7] and would thus be impractical. Hence we propose to define an image transformer to be of the form $\rho_k: x_{i,j}\rightarrow x'_{i,j}=x_{i,j}+ \psi_{i,j}$ where image $\Psi=(\psi_{i,j})$ is a smooth image so that any smoothly varying portion of the image $X=(x_{i,j})$ will be mapped to a similarly smooth portion of image X' so that compression will work at a similar scale to the original one. As a simple example, one can use the previously mentioned random key ω to create seeds for a pseudo random number generator, and then generate the pseudo integer constants τ_k and set $\psi_{i,j} =\tau_k$ for all i and j. There are many algorithms to generate pseudo random number, including for instance the Mersenne Twister [8], or the generator by Marsaglia [9]. We note that permutation of image blocks can of course be done in a variety of ways, including the one implemented in [5] which tries to smooth the block borders in addition. In contrast to their work which focuses more on the transmission of images alone, we are more concerned with the speed and efficiency, and will thus have to avoid any algorithms that would lead to the massive increase on the size of the transmitted images.

C. *Obfuscate the Text*

Since the text of an HTML page is delivered in plain to its recipient, the only way to obfuscate the content or make it illegible or unusable is to resort to the client side scripts that would dynamically convert the obfuscated text back to the intended version within a browser, or vice versa. Since the role of JS is to provide client interactivity rather than solely manipulate the text, it is not possible to holistically emulate the execution of all the JS by just the "resulting" text. This thus lends us means to encrypt, hide, obfuscate or camouflage the text. In principle, one may additionally make use of the current visibility of the DOM elements within a browser, using for instance [10] getBoundingClientRect, so that only those elements within the current viewport of the browser will be guaranteed to be dynamically converted back to the intended text or format and some of the other DOM elements will still contain the coded or "incorrect" content.

Assume that a decoders array of algorithms are loaded into the current page as in Fig. 1, then a piece of coded text T may be converted back to the intended format by executing *decode*(T, k), where k is a key, and then having its result written back to the page via the JS function w defined as document.write if possible, and can be achieved via the DOM element editing anyway if necessary. In fact, one can place T into an invisible element with a designated element ID so that a JS can easily further decide whether to convert the coded text. A decoder could be a decrypter, with a key extracted from ω if needed, or could be as simple as a word or letter shuffler. If the coded text T is made to retain the demarcation of the sentences, then one can also use JS to parse T sentence wise and have each sentence decoded by a different decoder sequentially. If the word structure is also preserved or identifiable, then the decoding can be moved towards the word level too.

Due to the nature of our goal, cryptographic encryption is generally an over kill. Instead, an effective design of text scramblers will serve the purpose better in general. Our text scrambler will permute a selected group [11] G of g characters containing at least a-Z, A-Z and 0-9, that is, all the characters from base 64 apart from "+" and "/". Hence, for any string $s \in G^*$, any character $s[k]$ indexed by x in G can be permuted to the character indexed by $x' = \sigma(x+k)$ in G. For notational convenience, we may simply identify the character in G with its index. This is simple to implement and hides away also the original character statistics. We note that additional cryptographic features can also be added at this stage.

D. Document Corruption and Presentation Erosion

Although a typical protection application will have most of the publically transmitted web resource in a "ciphered" or "corrupted" form and the correct content and presentation converted back real-time, the methodology is essentially the same as corrupting or eroding a proper page into something "illegible". To corrupt or obfuscate a web page, we can erode the textual accuracy, image content or correspondence, Cascading Style Sheets (CSS) styling, as well as JS interactivity, to name a few. One may even choose to erode a web page to the extent that is proportional to the time elapsed from the expiry date of the decoding key. There can be countless ways to corrupt or destroy the page content so as to protect the content ownership; we will thus examine a few prominent and effective strategies that can be employed to achieve this goal.

First, global variables will be ideal to represent the expiry status which can be easily retrieved elsewhere using a non-telling name such as $w[x]$ where w contains the `window` object and x contains the string name of a global variable. Additional JS tools, if needed, can be loaded dynamically into the HEAD element of the DOM with callback enforced. Next, one can traverse the DOM tree to scramble the page content of text nodes, and to remove randomly or selectively some of the nodes for embedded or external CSS. If one wants to select only *some* CSS definitions to disable for an external CSS file, the `document.styleSheets` array allows one to do so. As for JS, one can enumerate all the properties and functions for any given object, e.g., `for(var p in obj) { if(typeof obj[p] == 'function') { .. } }` much like playing the role of a name space, to determine which ones are to be modified or deleted for instance. If `obj` is the `window` object, then all global variables and functions can be located or modified. We note that any functions or properties defined via such as `obj.f=function(){..}` or `obj.p='foo'`; can be located this way.

As for the images on a web page, they can be temporarily drawn on an invisible canvas, modified and then saved to replace the original for the display. One can also use JS to distort the ordering of the images within the same page. We have thus shown that there should be no technical obstacles to have a JS code implemented to distort, encrypt or camouflage a web page.

E. Issuing Time-dependent Ticket of a Key

How to create a ticket for a timestamp and a given key of characters so that the ticket gives a random look and can be used to recover the original key if another (new) timestamp not exceeding a specified time difference is passed to it? In other words, if a mapping T=genT(t,k) generates a ticket T from a timestamp t and key k, and a mapping k'=genK(t',T) generates a key k' from a timestamp t' and a ticket T, then $k=k'$ only if t' has not exceeded t by a specified amount. For this purpose, we allocate M bits of which N bits are to cater for the timestamp scope, leaving thus $K=M-N$ bits to represent part of the key. The procedure to generate a ticket is as follows: (i) convert into binary the timestamp t and the key k which is padded bits 0 to make it a multiple of K bits; (ii) insert a block of N random bits after every K bits of the key k to form an expanded key k^*; (iii) collect M bits on the right of timestamp t, padding bits 1 on the left if necessary, to make an expanded timestamp t^*; (iv) XOR t^* and k^* bitwise into T^*; (v) pad bits 0 to T^* on the right to make it a multiple of 6 bits and then convert it into the final ticket T by mapping successively each block of 6 bits into a base 64 character or an equivalent; see Fig.5 for an illustration. The reverse function genK can be similarly constructed. To avoid the use of "+" and "/" characters which have a special meaning in a web page, we used "." and "_" instead. As an example, for M=30, N=15, k="Random", t=1388494800 (1 Jan 2014), we have T=gwUAj_dGtMungJk6WQi3 as one of the valid tickets for the same key and timestamp.

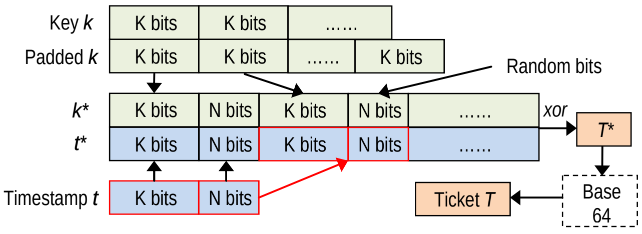


Figure 5: Bitwise mixing key with timestamp

Our experiments show that the generated ticket does appear quite random because of the distributed random bits in k^*, particularly when K is relatively small. For a much larger K, one can also apply an additional pattern permutation to make the appearance more random. This can be regarded as a form of *variation divergence*, and can also be used to masquerade a pattern like a timestamp, such as the bitwise mixing in Fig. 5 if desired, after T^* is obtained there.

III. Automate the Protection Process

In terms of the eventual applications of our proposed webpage source protection scheme, a typical scenario would be that a web designer first creates a web site mostly in a normal manner, embedding additional automation directives within some web pages if necessary. Then, a web server will dynamically create a serving version of the pages that are injected with the content obfuscation mechanism. In this section, we first outline a general framework that can be employed to serve a regular web page *a.html* in a protected mode, and then examine a number of implemental techniques. Even though the original pages and images may be stored in a database as in a typical Content Management System (CMS), we will for simplicity leave them as files behind a firewall, hence not directly accessible to the outside.

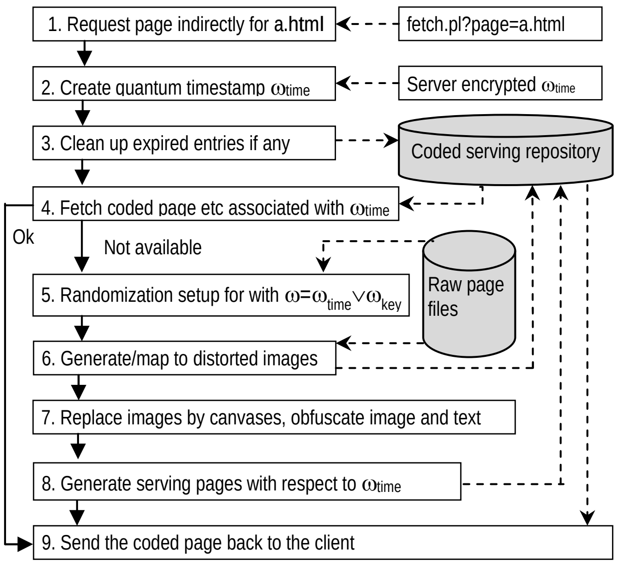

1. Request page indirectly for a.html
fetch.pl?page=a.html
2. Create quantum timestamp ω_{time}
Server encrypted ω_{time}
3. Clean up expired entries if any
Coded serving repository
4. Fetch coded page etc associated with ω_{time}
Ok
Not available
5. Randomization setup for with $\omega=\omega_{time}\vee\omega_{key}$
Raw page files
6. Generate/map to distorted images
7. Replace images by canvases, obfuscate image and text
8. Generate serving pages with respect to ω_{time}
9. Send the coded page back to the client

Figure 6: Work flow of protected document delivery

The workflow of delivering the content of a "raw" page *a.html* that resides behind a firewall to a remote client in a protected mode can be illustrated in Fig. 6. When a request for a page, say, *a.html*, is made in step 1 through a server side script, say, *fetch.pl*, a timestamp ω_{time} is created in step 2 and such timestamps will differ for every quantum of time, say 1 hour or a single day or a month. For each new quantum timestamp we can create a new serving version of the document which will be associated with a different set of algorithms and keys to convert the serving page back to the intended format via JS. The version associated with an outdated timestamp will thus expire and be removed in step 3 at the next appropriate time after which full reconstruction of the pages saved for the outdated timestamp will become impossible. If the coded serving version is available for the current timestamp, then the page will be immediately returned via steps 4 and 9 to the client. Hence repeated access of the same page within the expiration time quantum will be straightforward and may be cached on the client.

If the serving version for the current timestamp is not available, then steps 5-8 will immediately generate one. The *race condition* [12] can be easily avoided if such a creation has to first acquire a lock that would expire in a given period of time such as several seconds. The unprotected raw page files will be archived behind the firewall or protected by the access permissions so that they are not directly accessible by any remote clients but are accessible to certain server scripts. Step 5 will typically accomplish the following tasks:

i) Read in the raw page *a.html*, extract all image element IDs and other element IDs, and assign distinct new IDs to the image elements if they don't have an existing one. For sections of text, wrap them with <span> tags along with distinct new IDs. To facilitate the processing and conversion, processing directives may be inserted in the raw pages to indicate such as code skipping via <!--wdp--skip--> up to <!--wdp--/skip--> and text for obfuscation via <!--wdp--paragraph--> up to <!--wdp--/paragraph-->.

ii) Construct an array TxtId for all the IDs of the text elements that are to be encoded, and associative arrays such as ImgViaId and ImgAlgorithm for proper identification of the images, recovery algorithm, etc.

iii) All the generated supporting files such as the algorithms in JS and distorted images may be kept in subdirectory ω, or each file is prefixed by the "ω_". The ω_{key} can be regarded as an optional seed for the randomization in this Step 5.

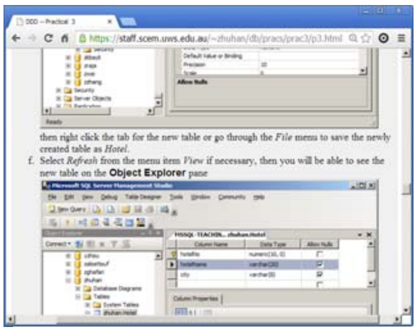
Step 6 will generate distorted images from those original ones dynamically. Such a distortion can also be constructed offline by separate image processing software with the mapping saved in the same directory as the raw page. On top of the images being distorted, the randomization of the image names could further obfuscate the correspondence between the images and their rightful positions in the document. As for the image distortion, it should be dynamically achievable through the use of such as *ImageMagick* package and the *PerlMagic* module.

To replace the images by canvases in step 7, we load all the distorted images into image objects created by JS in a single go as explained in Fig. 2, and then paint them on the respective canvases before deleting those loaded image objects. As for the text coding, we use the *seed* ω to generate a pseudo random list of scrambling algorithms, and sequentially code the *i*-th section of text by the *i*-th algorithm. Combining all these together we can complete the task of step 8. Once the page is loaded through step 9, the decoders will unscramble the text and rectify all the distorted images.

IV. Implementation and Experiments

Although all strategies and techniques studied earlier may be implemented to protect the original webpage source by obfuscation and timed expiry, a selected subset often suffices the main needs while not inflicting too much implementation cost. Our first application is on the web delivery of the practicals for a university database subject. The main procedure is as follows. First, we created the normal web pages for the practicals. We then developed a PERL script extractImages.pl which reads in a regular webpage, say, a.html, and generate another page, say, b.html. When this new page is loaded into a browser, it will generate all the corresponding obfuscated images, appending "-x" to the corresponding image name while leaving the file extension intact. The image conversion is done through the JS we implemented, utilizing the support for the *canvas* element and the existing JS tools such as FileSaver.js and canvas-toBlob.js. Then using another PERL script convertPage.pl we developed for this implementation to convert the raw webpage to a new version to be eventually made available on the web server. In the raw page a.html we inserted <!--wdp--paragraph--> and <!--wdp--/paragraph--> to delimit the body of the page: all the tag-free text portions will be separately and randomly obfuscated while tags and JS and CSS are preserved. In fact, the tag-free text will be first parsed into "sentences" and the obfuscation is independently done at the sentence level.

As an example, a part of the converted page will appear as in Fig. 7 when the page is being served in the normal

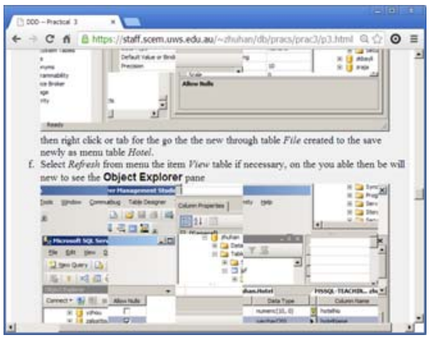

https://staff.scem.uws.edu.au/~zhuhan/db/pracs/prac3/p3.html

then right click the tab for the new table or go through the *File* menu to save the newly created table as *Hotel*.

f. Select *Refresh* from the menu item *View* if necessary, then you will be able to see the new table on the **Object Explorer** pane

Figure 7: Normal view of the page

manner. Fig. 8 shows how it appears when the page is expired. We observe that both the text and images are now distorted or obfuscated. If the page is saved as source code,


https://staff.scem.uws.edu.au/~zhuhan/db/pracs/prac3/p3.html

then right click or tab for the go the the new through table *File* created to the save newly as menu table *Hotel*.

f. Select *Refresh* from menu the item *View* table if necessary, on the you able then be will new to see the **Object Explorer** pane

Figure 8: Expired view of the page

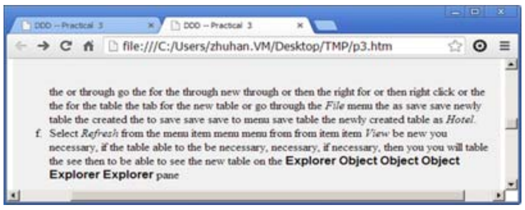
typically of the saving with IE, it will actually save not much. If the page is saved as a complete page, practically dumping the DOM elements as with Chrome, the part displayed on the saved page will appear obfuscated as in Fig. 9. Besides, all interactivity via the JS on the page will cease to function. The code for the 2nd image in Fig. 7 reads

```
<a href='viewimage.php?u=lab3h-x.png'
id='image_lab3h_png'>
<canvas id='canvas_lab3h_png' width=0 height=0
        class=iconwidth _width=450></canvas>
<script>imageOnCanvas('lab3h-x.png',
        'canvas_lab3h_png',128,96);</script></a>
```

that corresponds to the original text <a href=lab3h.png><img lab3h.png</a>, and part of the (auto converted) source code for the above takes the following typical out of order form

```
… <i><span id='sid_82_0'>View</span><script>a('sid_82_0');
</script> </i><span id='sid_83_0'>able new table on you will
the be to then the necessary, if
see</span><script>a('sid_83_0'); </script> <b><span
id='sid_84_0'>Object Explorer</span> …
```

which means the source being served is completely obfuscated in both text and images. When the source code gets into the client's browser, it becomes rectified under proper display styles but otherwise undecipherable as in Fig.


file:///C:/Users/zhuhan.VM/Desktop/TMP/p3.htm

the or through go the for the through new through or then the right for or then right click or the the for the table the tab for the new table or go through the *File* menu the as save save newly table the created the to save save save to menu save table the newly created table as *Hotel*.

f. Select *Refresh* from the menu item menu menu from from item item *View* be new you necessary, if the table able to the be necessary, necessary, if necessary, then you you will table the see then to be able to see the new table on the **Explorer Object Object Object Explorer Explorer** pane

Figure 9: Viewing the saved page

9. Fig. 10 gives a comparison of the same page when being displayed in normal, expired and saved manners respectively. Although Fig. 10 already well shows the broad

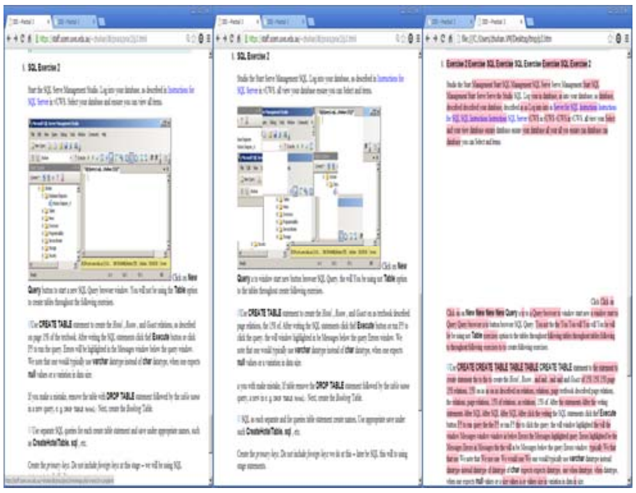
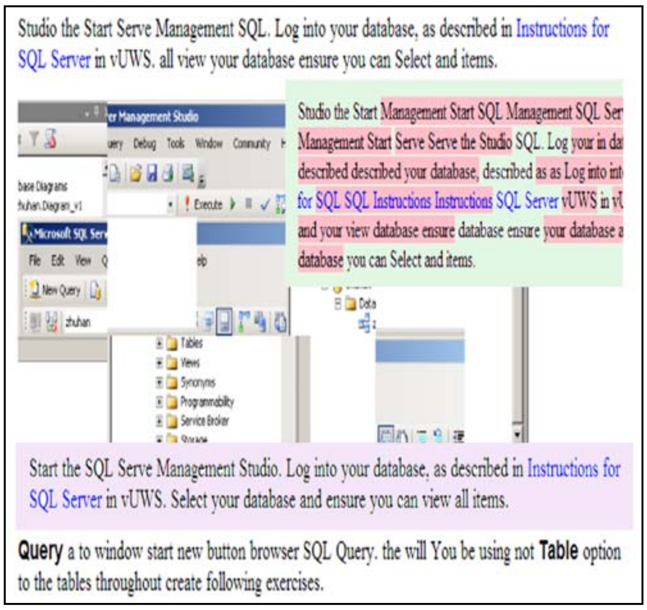


Figure 10: Comparison on normal/expired/saved page view.

differences, we further magnify parts of Fig. 10 in Fig. 11 to see more clearly the details. In Fig. 11, the main background comes from the 2nd picture in Fig. 10, the top left with a light green background comes from the 3rd picture, and the one at the bottom with a purple background extracts the 1st paragraph in the 1st picture of Fig. 10. We note that the extent of the text obfuscation can be controlled and will be somewhat proportional to the expired time. We also note that the image is missing from the saved version because the content on a canvas is not saved when a page is "completely" saved by a browser, and we also deliberately highlighted in pink the random text automatically inserted for the obfuscation.

Since the "decoding" part is to be completed by the client browser, there will be an overhead on the speed of page rendering. Hence, how complex an algorithm is allowed to be must be tied to the allowed performance degradation. Fortunately, the rectification of the deliberately distorted images does not have a visible effect in general, and the complexity of the textual obfuscation can also be controlled by the operating parameters. For the system we



Studio the Start Serve Management SQL. Log into your database, as described in Instructions for SQL Server in vUWS. all view your database ensure you can Select and items.

Studio the Start Management Start SQL Management SQL Ser Management Start Serve Serve the Studio SQL. Log your in da described described your database, described as as Log into int for SQL SQL Instructions Instructions SQL Server vUWS in v and your view database ensure database ensure your database a database you can Select and items.

Start the SQL Serve Management Studio. Log into your database, as described in Instructions for SQL Server in vUWS. Select your database and ensure you can view all items.

Query a to window start new button browser SQL Query. the will You be using not Table option to the tables throughout create following exercises.

Figure 11: Magnified parts of Fig. 10

implemented here, the overhead is observable but easily tolerable, and is in general in the scale of half a second. We however plan to explore more quantitatively the overhead impact in our future work.

V. Conclusion

The protection of web page content has been explored in terms of the reproduction scope and the source scope. We proposed an obfuscation mechanism that protects the page resource on both the server and the client side. On the server side, by removing or altering a simple key on the server, the otherwise fully functional website can be made instantly useless as all the resources there are obfuscated and unrecoverable without the key. On the client side, the web pages will be initially loaded in the distorted format for both the images and the text from the server, and dynamically rectified if the clients' authorization has not expired. If a client saves the "complete page" using a browse saving tool, the saved page remain fully obfuscated at the source level, and will not be able to get dynamically converted into its intended proper format if the deciphering key is not saved, as is the case for all auto-saving, or expired. A client user, having saved the complete pages, will find making use of the saved resource to imitate the original server not any easier than building everything from scratches, according merely to the screenshots of the whole site. This, therefore, defeats the purpose of saving the page resources.

References

[1] F. Greyling, M. Kara, A. Makka, and S. Van Niekert, "IT worked for us: online strategies to facilitate learning in large (undergraduate) classes", Electronic Journal of e-Learning, vol. 6, 2008, pp. 179-188.

[2] HTTrack Web Site Copier, http://www.httrack.com, last accessed on 27 May 2014.

[3] Y. Gao, Y. Zhang, B. Bai, and X. Wang, "Survey of webpage protection system", Computer Engineering, or 计算机工程 as its original name, vol. 30, no. 10, 2004, pp. 113-115.

[4] Web Protection, http://www.wiscocomputing.com/ articles/ protect_web_sites.htm, also http://thenetweb.co.uk/obfuscate-hide-and-obscure-e-mail-addresses-telephone-numbers-and-text, last accessed on 27 May 2014.

[5] A. Poller, M. Steinebach, and H. Liu, "Robust image obfuscation for privacy protection in Web 2.0 applications", Proc. SPIE 8303, Media Watermarking, Security, and Forensics, 2012; doi: 10.1117/12.908587.

[6] K.S. Thyagarajan, Still Image and Video Compression with Matlab, Wiley, 2010.

[7] Z. Jiang, O. de Vel, and B. Litow, "Unification and extension of weighted finite automata applicable to image compression", Theoretical Computer Science A, vol. 302/1-3, 2003, pp. 275-294.

[8] M. Matsumoto, and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", ACM Transactions on Modeling and Computer Simulation, vol. 8, no. 1, 1998, pp. 3-30.

[9] G. Marsaglia, "Seeds for random number generators", Commun. ACM vol. 46, no. 5, 2003, pp. 90-93; also https://groups.google.com /forum/#!msg /comp.lang.c/ qZFQgKRCQGg/rmPkaRHqxOMJ, last accessed on 27 May 2014.

[10] http://stackoverflow.com/questions/123999/how-to-tell-if-a-dom-element-is-visible-in-the-current-viewport, last accessed on 27 May 2014.

[11] J. A. Beachy and W. D. Blair, Abstract Algebra, 3rd edition, Waveland Pr Inc, 2006.

[12] Y. Yu, T. Rodeheffer, and W. Chen, "Race track: efficient detection of data race conditions via adaptive tracking", ACM SIGOPS Operating Systems Review – SOSP'05, vol. 39, issue 5, 2005, pp. 221-234.