Efficiently Crawling CMS Applications

by

Sukwon Oh

A thesis submitted in conformity with the requirements
for the degree of Masters of Computer Science

Department of Computer Science
University of Toronto

Efficiently Crawling CMS Applications

Sukwon Oh

Master of Computer Science

Department of Computer Science
University of Toronto

2016

Abstract

Content Management Systems have been serious driving forces of many web sites ranging from personal blogs, news, and wiki to online shopping sites. As CMS systems power many web sites, importance of writing bug-free CMS systems is growing. Recognizing needs to make CMS systems secure, many automated tools are available. However, a study on well-known eleven scanners identified that they missed many classes of vulnerabilities because they ignored application states. CMS applications have infinite states as new pages are created as users add new contents.

To overcome above-mentioned challenges, we designed and implemented CMSCrawler as a first step toward finding bugs in real-world CMS applications. We build CMSCrawler on top of a browser engine to handle AJAX applications. We also designed a state-space search algorithm that constructs an approximate state space graph of an application and prioritizes pages to equally exercise different parts of application states.

We evaluated CMSCrawler against 4 other scanners and showed that it can generate large number of interesting POST requests compared to other existing automated tools.

Acknowledgments

I would like to thank Professor David Lie for his patience, guidance, deep care for his students and unsparing supports. In addition, I am thankful to my fellow graduate student, especially Beom Heyn Kim for weekly coffee hours with countless interesting discussions. I would like to thank Professor Ashvin Goel for his valuable feedback. I am very thankful to my family for their much needed moral support and help. Finally, I would like to thank University of Toronto and the department of Computer Science for their financial support.

Table of Contents

List of Tables

List of Figures

1 Introduction

Content Management Systems (CMS) have been serious driving forces of many web sites ranging from personal blogs, news, and wiki to online shopping sites. To give statistics, it has been reported that more than 74.6 million web sites depend on WordPress, one of the popular CMS system [1]. The function of CMS systems is to store and organize files while avoiding hand coding. Handing coding refers to editing in underlying representation of content such as documents, instead of on higher-level representation. In terms of programming, hand coding is analogous to hard coding. Most CMS systems avoid hand coding by providing What You See Is What You Get (WYSIWYG) editors, image galleries, and PDF editors and so on. For users without sufficient knowledge of HTML, CSS and JavaScript, such WYSIWYG editors are very attractive because they are able to create quality looking web pages with small efforts.

As CMS systems power many web sites, importance of writing bug-free CMS systems is growing. For example, a WP White Security study found that 73% of all WordPress installations had known vulnerabilities that could be easily detected by automated tools and 170,000 WordPress sites were hacked in 2013 [2].

Recognizing needs to make CMS systems and web applications in general secure, many automated tools have been developed. One of the most popular automated scanners, w3af, uses various of scanning techniques to automatically detect common web security vulnerabilities such as Cross-site scripting (XSS), SQL injection (SQLI) and Cross-site request forgery (CSRF) [3]. However, a study on well-known eleven scanners identified that many of them missed many classes of vulnerabilities because their crawling approach did not correctly model and track the state of an application [4]. A typical web application's state is stored in a database and its behaviour can vary wildly as its state changes. For example, if no state a scanner evaluates has groups present, a vulnerability in membership assignment in a team management application will be overlooked. Therefore, it is important to exercise the application to get into as many different valid application states as possible to avoid overlooking vulnerabilities.

In general, we identify three major challenges to effectively exercising a CMS application. First, many web applications are written in dynamically typed languages such as Python, Ruby and

JavaScript. This makes it difficult to apply techniques such as static analysis because correctly inferring variable types can be hard. Furthermore, due to duck typing [5], function parameter types may change during runtime and applying concolic execution techniques generally result in spending considerable amount of time to understand source code and manually writing test functions for every possible function parameter type. Second, there are many web pages using AJAX, which makes applications event-driven and highly dynamic in nature. As an extreme, it is possible to write an entire application as a single page application (SPA) to provide a desktop application-like experience to users. Building an effective automated tool for modern CMS applications is not possible, ignoring AJAX. Third, CMS applications may have infinite state spaces and effectively exercising them may be difficult. For example, many automated tools that are state-aware perform shallow state space search to avoid state explosion problem but cannot exercise interesting application states [6].

To overcome above-mentioned challenges, we designed and implemented a tool called CMSCrawler as a first step toward finding bugs in real-world CMS applications. To handle first and second challenges, we built CMSCrawler on top of a browser engine that is fully compliant with AJAX and exercise applications on the browser to simulate a real user. To handle third challenge, we designed a state-space search algorithm that constructs an approximate state space graph of an application to exercise interesting application states while minimizing efforts spent in finding new pages.

Our contributions are:

- Designing an efficient state exploration algorithm for CMS applications that finds interesting database states.
- Show how the database states found with CMSCrawler can be used with other techniques to find bugs in CMS applications.

We designed CMSCrawler to be used with other bug finding techniques and it has a different goal compared to other crawlers. Instead of focusing on finding new pages, it focuses on generating interesting HTTP POST requests to find interesting application states. Therefore, we leave the actual bug finding as a future work and instead focus on improving crawling efficiency. We evaluate CMSCrawler on a toy and two CMS applications and show that CMSCrawler can generate more interesting POST requests compared to other existing automated tools.

2 Background

2.1 Document Object Model

Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript are the most foundational web technologies used today and each of them has specific purpose. HTML defines a structure of a page and connects pages together with hyperlinks. CSS is responsible for styling a web page. It defines how a web page must be rendered on screen. CSS properties range from simple properties such as colours, positions, widths, heights to complex properties such as media queries for building a mobile friendly page. While HTML and CSS define static properties of a page, JavaScript makes a page dynamic. JavaScript is a lightweight, interpreted programming language similar to Java. A typical JavaScript code in a page may react to user inputs by calling event listeners and manipulate the page dynamically using DOM API.

To build an efficient web application crawler, it is crucial to understand how modern browsers render web pages. Before a page is rendered in a browser, it remains as a text document written in a combination of HTML, CSS and JavaScript. As in interpreter-based programming languages, a browser parses a page and translates it into some internal data structure representations before rendering on its screen. One such representation is a tree-based model of a page called Document Object Model (DOM). A browser allows manipulating DOM in JavaScript with series of API calls. DOM API includes functions to navigate, add, modify or delete HTML elements.

DOM has been extended several times in last twenty years. DOM Level 1 was recommended by W3C in late 1998 and provided a complete model for an HTML document. DOM Level 2 was recommended in late 2000 and it included event model as well as support for CSS. DOM Level 3 was published in 2004 and added support for XPath and keyboard event handling, as well as an interface for serializing documents as XML. DOM Level 4 is published recently in November 2015.

Although a browser provides DOM API in JavaScript, DOM is not included in the language. In fact, it is possible to manipulate DOM using other programming languages. For example,

popular libraries such as libxml2 and Xerces are both implemented in C++ and supports manipulating DOM.

2.2 AJAX

Asynchronous JavaScript and XML (AJAX) is a set of web development techniques utilizing many web technologies used on the client-side to create asynchronous web applications [7]. With AJAX, applications can send data to and retrieve from a web server in the background without interfering a user, and asynchronously update a page without reloading the entire page. For asynchronous communication, a special JavaScript object, XMLHttpRequest is used. A page may use XMLHttpRequest object to send an AJAX request on receiving a user input and receive responses. Then, it may dynamically update itself using the DOM API.

2.3 Content Management Systems

A content management system (CMS) is a system used to manage the content of a web site [8]. Content management (CM) refers to the administration of digital content from creation to permanent storage or deletion [9]. Digital content includes simple texts, images, documents, and multimedia. CM has several stages including creation, editing, publishing, and deletion of digital content. As an example, WordPress is a CMS application popular for building personal blogs. In WordPress, users can write blog posts using WYSIWYG editors, upload images, and leave comments to others posts.

A CMS application consists of the content management application (CMA) and the content delivery application (CDA) [8]. CMA is a front-end of a CMS application where users manage their contents. For example, popular CMS applications such as WordPress, Drupal, and Wagtail provide admin interfaces (CMA) where a user can create, edit, or delete her contents. CDA is a back-end of a CMS application that compiles and updates the web site with latest user contents. In WordPress, CDA updates the web site with the latest published blog posts while hiding drafts or private posts from other users.

As CMSCrawler tries to exercise application states, it targets CMA part of CMS applications where the actual content management occurs.

3 Related Works

3.1 Open-Source Tools

3.1.1 Crawlers

One popular use of crawlers is by search engine companies like Google to find all pages on the web and index them to return accurate search results.

Another type of crawlers exists and they are usually part of bigger tools called web application vulnerability scanners. Scanners are automated tools for finding security vulnerabilities in web applications. Scanners use crawlers to find many pages in a target application and scan them for any common patterns that may lead to vulnerabilities. Many common vulnerabilities such as XSS, SQLI and CSRF can be found using scanners.

Many scanners download and parse HTML pages for hyperlinks, ignoring any embedded JavaScript code. Then, they click on found hyperlinks to discover any new pages. Since they do not interpret JavaScript, they are usually not effective against AJAX applications.

Open Source Vulnerability Scanners	Handles AJAX
Grabber [10]	Partially
Vega [11]	No
Zed Attack Proxy [12]	No
Wapiti [13]	No
w3af [3]	No
WebScarab [14]	No
Skipfish [15]	No
Ratproxy [16]	Yes
SQLMap [17]	No
Wfuzz [18]	No

Grendel [19]	No
Watcher [20]	Yes
X5S [21]	No
Arachni [22]	Yes

Table 1 Summary of open source web application scanners

Table 1 shows that most open source scanners cannot handle AJAX applications [23]. We anticipate that effectiveness of the scanners will be low on CMS applications.

3.1.2 Browser Automation Tools

In contrary to automated scanners, browser automation tools target application developers and testers. They interact with a real browser and handle AJAX applications better. The most popular automation tool is Selenium [24]. It uses a web driver to interact with all major browsers. To improve testing time, it is possible to use headless browsers such as PhantomJS and HtmlUnit with Selenium.

Selenium API covers wide range of browser operations including history navigation, cookie management, clicks, double clicks, drag and drop, and so on. While it excels in automating a browser, users are required to write tests describing sequence of operations they want to automate. Therefore, it seems difficult to use Selenium other than regression and unit testing purpose.

3.2 Research Solutions

Research community has been active with studying automatic ways in finding security vulnerabilities or bugs and testing web applications. Broadly, we divide them as static or dynamic solutions. Since CMSCrawler takes a dynamic approach, we focus more in describing related works using dynamic approaches.

3.2.1 Finding Vulnerabilities

3.2.1.1 Static Approaches

According to OWASP, the most critical web application vulnerability in 2013 was injection flaws including SQL, OS, and LDAP injections [25]. Injection flaws occur when an application executes interpreters such as database query engines, and OS shells with untrusted data from a malicious user without properly sanitizing it. In fact, OWASP reported that five of top 10 most critical vulnerabilities in 2013 were directly related to improperly sanitizing user inputs. To find such vulnerabilities, many researchers have proposed approaches using static analysis that find possible execution paths without proper user input sanitizations. For example, Pixy is a static analysis tool for PHP that finds vulnerabilities such as SQL injection, XSS, or command injection with flow-sensitive, inter-procedural and context-sensitive dataflow analysis [26]. However, static analysis on web application is often imprecise because many web applications are written in dynamically typed languages. For example, MACE uses static analysis to find access control vulnerabilities in PHP but ignores parts of applications written with object-oriented programming constructs in PHP [27].

Static approaches have been successful in finding taint-based vulnerabilities such as SQL injection and access control vulnerabilities; however, dynamic approaches have been popular in finding other types of vulnerabilities such as logic bugs.

3.2.1.2 Dynamic Approaches

Dynamic approaches interact with the target application to trigger vulnerabilities. One obvious advantage is a low false positive rate compared with static approach. However, with dynamic approaches, only vulnerabilities that are triggered may be found. Therefore, it is important to exercise the target application thoroughly. Furthermore, as applications have infinite state spaces, dynamic approaches often suffer from state explosion problem [28]. For example, QED is a dynamic solution that finds XSS and SQL injection vulnerabilities in Java Servlets [29]. It uses a model checker Java Pathfinder [30], for exercising application states. In addition, it eliminates inputs that are not of interest using static analysis and reduces application state space. Waler is another dynamic solution that finds logic bugs in Java Servlets [31]. As logic bugs are application specific, the authors used an invariant detection tool, Daikon and collected likely invariants from execution traces of target applications. Then, they used Java Pathfinder to

exercise application states and checked that all states satisfy the collected invariants. Both QED and Waler worked on applications that did not use AJAX.

In contrary, AJAX applications are event-driven and testing them requires additional efforts. Several literature has talked about importance of crawling in testing AJAX applications.

3.2.2 Crawling

3.2.2.1 State-Aware Vulnerability Scanner

Doupe, et al. has identified weakness in existing automated scanners [4] and proposed a crawling algorithm to make them more efficient [32]. The authors pointed out those existing tools operated in a point-and-shoot manner to detect common vulnerabilities by crafting random inputs and examining replies for a hint of vulnerabilities. However, an application state may change from a random input received, and some of its pages may never be reachable from the new state. The authors proposed a crawling algorithm to infer application states from output and systematically check every application state. Their evaluation showed that combining their crawling algorithm with existing automated tools found many more vulnerabilities.

Above work suggests that exercising application states is important to detect many web application vulnerabilities or bugs. However, it ignores AJAX and cannot work on AJAX applications.

3.2.2.2 Apollo

Apollo is an automated tool for test generation to detect crashes and malformed HTML on modern PHP applications [33]. It utilizes both concolic execution and explicit-state model checking. Although it is a bug detection tool, significant portion of its algorithm is about crawling. The authors first propose an incomplete solution using concolic execution and a HTML page validator as an oracle, assuming a single page application. Next, they propose a complete solution that uses an explicit-state model checking without the assumption. Apollo queues every application state found during crawling and restores them later to check for errors.

Apollo showed that concolic execution in web applications is an effective way to find a bug. It also showed that keeping track of application states is important when testing web applications. Unfortunately, Apollo ignores AJAX and cannot work on AJAX applications, either.

3.2.2.3 Crawljax

Crawljax is a crawler for AJAX applications [6]. In contrary to above solutions, it interacts with a browser to test an application. The authors recognize that webpages do not have unique URLs in AJAX applications as AJAX operations may update pages partially without changing their URLs. Crawljax captures all such partial changes from AJAX operations and stores them as new states. To detect the partial changes, it uses an algorithm based on edit distance to calculate similarity of two pages and determines if they are different. Furthermore, it remembers elements clicked to trigger AJAX operations to construct a series of clicks to restore states during crawling.

In AJAX applications, all HTML elements with click JavaScript event listeners registered, are clickables. Without instrumenting a browser, it is difficult to know clickable elements on a page [34]. As clicking all elements on a page is not tractable, Crawljax clicks on and <a> elements as a heuristic.

Crawljax is a significant contribution toward testing AJAX applications, as all previous approaches ignored AJAX. However, we have evaluated Crawljax on few CMS applications without success. In a high-level explanation, we have found that Crawljax does little to distinguish AJAX operations that only changes visual appearance of a page with AJAX operations that changes application behaviours. As a result, Crawljax spent most of its time on few pages with many JavaScript event listeners and failed to find all pages.

3.2.2.4 Artemis

Artemis is an automated test generation framework for JavaScript applications [35]. It uses various forms of feedback to generate tests with higher coverage. Feedbacks that use event handler registrations, read-write sets in JavaScript programs, or line coverage in JavaScript programs are used. The feedbacks are used for prioritizing and choosing a set of AJAX operations to execute on a page in the generated tests.

The original version of Artemis consists of a JavaScript engine and a simulated browser environment. It interposes on JavaScript event listener registrations and triggers them to test an application. It is an improvement from Crawljax as Crawljax assumes only and <a>

elements are clickables and randomly tries to click on all of them. Systematically triggering browser events using Crawljax is not possible.

Recent version of Artemis uses WebKit browser engine and does much more. It is capable of symbolic execution on JavaScript included in pages to generate better input values. Although, the original version of Artemis handled multi-page applications, the recent version only works on a single page application (SPA). We have modified Artemis according to the published paper to handle multi-page applications. Our modified Artemis did not perform well against CMS applications and showed similar results to Crawljax. Although Artemis tries to be efficient when exploring a page by prioritizing a set of JavaScript events to trigger, it does little to avoid exploring a state multiple times. In current implementation, it computes a hash of a page's DOM and stores it to a hash table for comparison. With AJAX applications, such approach will consider many pages with partial DOM changes from AJAX as different and dramatically increase the number of pages a crawler needs to explore.

4 Motivation

4.1 Why stateful bugs?

To confirm our hypothesis on existence of bugs that appear only in certain application states, we did a bug study on some of the popular web applications on GitHub. Although many bugs were on improving user interface, we were able to find few bugs on many applications that seemed stateful. For example, a CMS application Wagtail had a bug where a user without an admin or editor roles was continuously redirected to a login page when attempted to login. In a shopping cart application Catridge, a bug where minimum total purchase required to apply discount codes could be ignored was reported. These bugs required applications to be in certain states: Wagtail bug was triggered when a user without admin or editor roles existed and Catridge bug was triggered when items were removed after discount codes were applied.

Application	Stateful Bugs
Wagtail (CMS)	• Setting tags on a page required publishing the page twice. • A user without admin/editor roles could not login.
Catridge (Shopping Cart)	• Removing items after applying discount codes ignored minimum total purchase requirement for the discount codes applied.
MarkUs (Assignment Grading)	• If a student joins a group who submitted after at least one grace day, the student's grace days were not deducted. • If an admin removes a student from one-person group, the student loses ability to view his assignments.
Reviewboard (Code reviewing)	• Review requests that were discarded twice before submitted became private.

Table 2 Example of stateful bugs

Table 2 is the summary of some of the stateful bugs found. We found many of them to be non-trivial to understand and manually setting application states to trigger some of them seemed difficult. Considering infinite state space of these applications, we think triggering these bugs with ignoring application states is like finding a needle in a haystack.

4.2 Large Size of CMS Applications

We also discuss difficulty with crawling real-world CMS applications using Wagtail as an example. Wagtail is a moderate sized CMS application written in 32,000 lines of Python code with many active users. To estimate the minimal efforts required to crawl Wagtail, we have analyzed how many unique pages exist and how they are interconnected. To make the analysis simpler, Wagtail was configured with minimal database contents required to be functional. We have found that Wagtail has at least 40 unique pages and over 1700 hyperlinks. Figure 1 shows simplified graph of persistent pages in Wagtail after an optimization was applied.

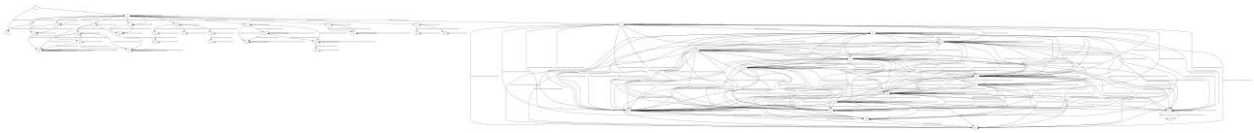

Figure 1 Static Crawling on Wagtail

As Wagtail is not the most complicated CMS application, it suggests that crawling real-world CMS applications needs a different approach. For example, assuming it takes 200 ms to fully load a page on a browser, it will take over five hours to click the hyperlinks just once. If we consider form submissions, other clickable elements, and the fact that new pages may be created during crawling, the number of clicks required explodes dramatically.

5 Design

5.1 Overview

We have designed a crawling algorithm to exercise database states of CMS applications. The crawling algorithm consists of two stages. First stage does the static crawling and builds a navigation graph of a CMS application. Second stage uses the navigation graph to exhaustively exercise the application states.

Static crawling is a lightweight crawling that finds all existing pages without changing database states. As RFC 7231 recommends all state changing requests to use POST methods [36], static crawling avoids sending any requests that use POST methods by avoiding form submissions and triggering browser events. Although, clicking <a> elements send GET requests by default, it may also execute any registered JavaScript event listeners that may send POST requests in AJAX. Fortunately, <a> elements with registered JavaScript event listeners often use placeholder href attributes, "#". Thus, static crawling only clicks on <a> elements with valid href attributes, assuming such links do not invoke AJAX calls.

In CMS applications, as users add new contents, new pages are created. Therefore, it is possible that as a crawler exercises application states, number of pages in the application increases. To find many interesting database states in a reasonable time, a crawler needs to prioritize the new pages. One common approach in prioritizing the new pages calculates similarity of the new pages with previously exercised pages. It drops new pages that are sufficiently similar to previously exercised pages so a crawler can move on to other pages. Many variations exist for calculating similarity of pages. Crawljax calculates edit distance of a new page against previously exercised pages, after removing texts [6]. Artemis calculates hash values of a new page and compares against previously exercised pages [35]. However, we think it is difficult to come up with an algorithm that works in most cases. For example, if the similarity calculation is too strict, most of the new pages will have high priorities and the effect of the prioritization will be minimal. In contrary, if the similarity calculation is too coarse, many of the new pages will be dropped and a crawler will miss many states.

Instead, we use the navigation graph from static crawling to prioritize new pages and show that the prioritization does not have to be precise and still find many interesting database states. The actual prioritization takes place in the second stage.

In second stage, we construct sequence of operations to exercise application states using the navigation graph. In addition, we use a pre-emptive scheduling algorithm to equally exercise different parts of application states.

5.2 Assumptions

We designed CMSCrawler with following assumptions in mind:

- CMS applications consist of multiple pages with some AJAX contents.
- CMS applications store their states in a database.
- CMS applications redirect users to their main pages after successful login.
- CMS applications have admin users with full access to all stored contents.
- Users have permissions for reading and updating database contents.

We do not consider CMS applications that are single-page applications (SPA) as over 75% of all CMS-powered websites consist of multiple pages [2]. Aside from MoinMoin wiki [37], which stores its states as binary files in a local filesystem, we are not aware of any popular web applications that do not use a database. Thus, we safely assume that all CMS applications store their states in a database. Our third and fourth assumptions are from our experience with using some popular CMS applications.

5.3 Static Crawling

As mentioned in Overview, CMS applications may create or delete pages as their states change. One common solution prioritizes the new pages by calculating similarity with previously exercised pages. Many crawlers that we have examined including Crawljax [6] and Arachni [22] simply drop new pages if they are sufficiently similar to one of the previously exercised pages. However, detecting similarity of pages is non-trivial in AJAX applications as pages may change their visual appearances without changing their behaviours. We think it is difficult to come up with an algorithm that works in most cases, without understanding semantics or behaviours of each page.

Instead, we make several observations on CMS applications and propose that a crawler can prioritize the new pages with static crawling. We explain our observations and static crawling below.

5.3.1 Observations

5.3.1.1 First Observation

As explained in Section 2.3, CMS applications consist of the content management application (CMA) and the content delivery application (CDA). Contents include simple texts, images, video, audio, and other multimedia. Content management refers to creating, viewing, editing, or deleting contents. We target CMA part of CMS applications, where the actual content management occurs.

After examining several CMS applications, we propose that it is possible to categorize pages in CMA into the following three categories:

A. Connecting other pages
B. Allows adding new contents
C. Allows viewing, editing, and deleting a content

As CMS applications manage contents, they have pages where users can add new contents. We categorize them into B. In addition, they have pages where users view, edit, or delete an existing content. We categorize them into C. We categorize pages that do not belong in B and C into A.

Categories	Persistent/Ephemeral	Examples in Wagtail
A	Persistent	/admin/; /admin/images/
B	Persistent	/admin/images/multiple/add/
C	Ephemeral	/admin/images/1/

Table 3 Categorization of web pages in CMS applications

Table 3 is an example of categorizing some of Wagtail pages. In the /admin/images/multiple/add/ page, users can upload images. Thus, we categorize the page as B. /admin/images/1/ page is an example of a page created when a user uploads an image. We categorize the page as C, as users

may view, edit or delete the image on the page. Finally, we categorize pages /admin/ and /admin/images/ as A. For example, /admin/ page connects pages such as /admin/images/, /admin/documents/, and /admin/users/. /admin/images/ page connects pages such as /admin/images/multiple/add/ and /admin/images/1/.

Among the three categories, only pages in C directly map to contents in the application. For example, /admin/images/1/ page in Wagtail allows manipulating the image with id 1. As a result, we find that lifecycle of pages in C matches lifecycle of corresponding contents. In contrary, we find pages in A and B are always present. For example, the /admin/images/multiple/add/ page in Wagtail is always available to a user such that the user can add new images. We name pages in category A or B as **persistent pages** and pages in category C as **ephemeral pages.**

Observation 1: CMA consists of persistent and ephemeral pages.

5.3.1.2 Second Observation

We have noticed that in all CMS applications we have examined, the number of content types they support are fixed. For example, Wagtail supports seven content types including texts, images, and documents but does not support uploading a video. Previously, we have suggested that ephemeral pages directly map to contents in CMS applications. Therefore, we propose that ephemeral pages can be grouped by available content types in CMS applications.

Observation 2: Ephemeral pages can be grouped by their content types.

5.3.1.3 Third Observation

A crawler can visit a page by clicking a hyperlink pointing to the page or loading the page's URL directly in a browser. As ephemeral pages are created or deleted during crawling, a crawler may not know their URLs up front. To visit an ephemeral page, a crawler needs to visit a persistent page that has a hyperlink to it. After examining few CMS applications, we observe that ephemeral and persistent pages of same content types are closely connected together.

Ephemeral Pages	Closest Persistent Pages	Clicks Required
/post/6	/posts/	1

/page/1	/pages/	1
/wp-admin/upload.php?item=8	/wp-admin/upload.php	1
/wp-admin/link.php?action=edit&link_id=3	/wp-admin/link-manager.php	1

Table 4 Ephemeral Pages in WordPress

Table 4 shows that ephemeral pages in WordPress are reachable from persistent pages of same content types with a single click. For example, a user can click on a hyperlink from the /posts/ page to visit the /post/6 page. However, the /wp-admin/upload.php page does not have a hyperlink to the /post/6 page. We have observed similar results with other CMS applications. Although, we did not find a case where some ephemeral pages are only reachable from other ephemeral pages, we do not rule out the possibility completely. Thus, we hypothesize that all ephemeral pages are reachable from some persistent pages within few clicks.

Observation 3: Ephemeral pages are reachable from persistent pages of same content type within few clicks.

5.3.2 Proposal

To effectively exercise CMS applications, a crawler needs to be able to do the following:

- Add new contents
- Edit, and delete existing contents

Although, the number of pages where a crawler can add new contents is fixed (persistent pages), the number of pages where a crawler can edit or delete existing contents (ephemeral pages) varies with the stored contents. To handle growing number of ephemeral pages as we crawl, we propose prioritizing ephemeral pages by their groups.

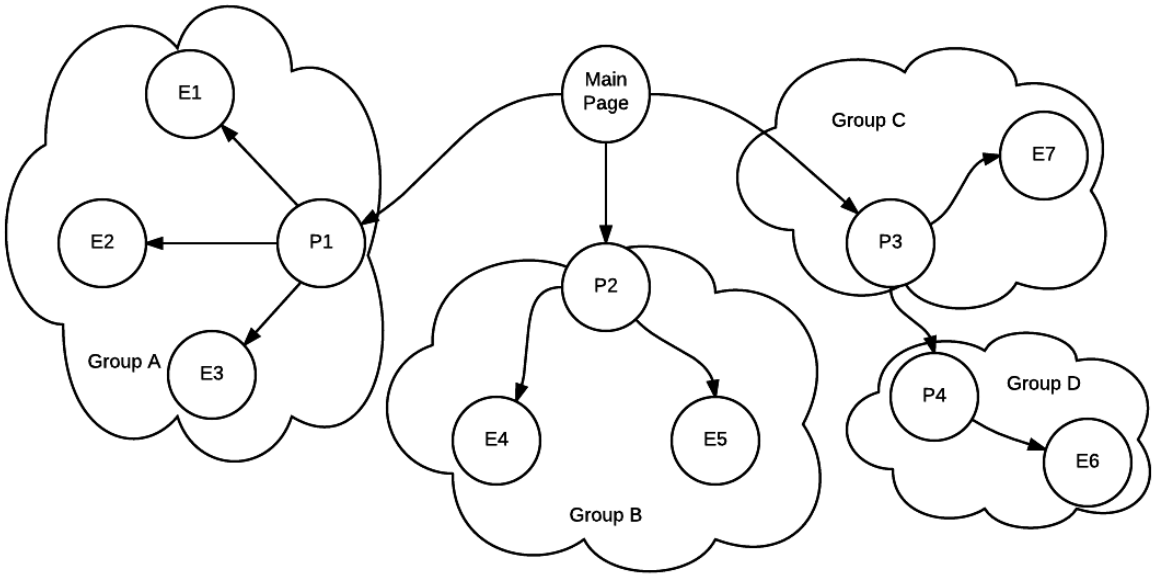

Figure 2 A model of a hypothetical CMS Application

Figure 2 is a model of a hypothetical CMS application that satisfies all our observations. To satisfy observation 1, the application has a set of persistent pages: P1 to P4, and a set of ephemeral pages: E1 to E7. In addition, P3 belongs to both categories A and B as it connects to P4 and users can add new contents. To satisfy observation 2, the application supports four content types: A to D. To satisfy observation 3, pages from E1 to E7 are closely connected to persistent pages with same content types. For example, E1 and P1 have same content types as E1 is created when a user adds a content on P1. Thus, E1 and P1 are grouped together. All ephemeral pages except E6 are grouped accordingly. In contrary, E6 is a special case because it is reachable from both P3 and P4 and both P3 and P4 have same content types. To break a tie, E6 is grouped with P4, as P4 is closer. As the model Figure 2 satisfies all our observations, we think all ephemeral pages in CMS applications can be grouped in a similar manner.

To group ephemeral pages as in Figure 2, we propose **static crawling**. Static crawling is a lightweight crawling where only <a> elements with valid href attributes are clicked. We do not change application states during static crawling, to avoid creating ephemeral pages. Thus, static crawling constructs a graph of all persistent pages. As ephemeral pages are closely connected to persistent pages with same content types, we group ephemeral pages by persistent pages. For example, in Wagtail, ephemeral pages /admin/images/1/, /admin/images/2/, … are grouped by a

persistent page /admin/images/. In contrary, ephemeral pages /admin/documents/1/, /admin/documents/2/, … are grouped by another persistent page /admin/documents/. Later when we exhaustively exercise pages, we prioritize ephemeral pages by groups. We give high priority to ephemeral pages in a group that we have not crawled before to equally exercise all parts of CMS applications.

5.3.3 Algorithm

```
void static_crawl() {
    Q <- Queue()    // execution queue
    Q.add(empty_execution)
    while (!Q.isEmpty()) {
        execution <- Q.pop()
        foreach element_xpath in execution {
            element <- getElement(element_xpath)
            click element
            wait 500 msec
        }
        if (isNewPage()) {
            links <- getElements("a")
            foreach link in links {
                if (isValidLink(link)) {
                    new_execution <- execution + link
                    Q.add(new_execution)
                }
            }
        }
    }
}
```

Figure 3 Pseudocode for Static Crawling

Figure 3 is a simplified pseudocode for the static crawling algorithm. As it is a standard breadth-first search (BFS) algorithm, we skip explaining the code in detail. Instead, we briefly describe `isNewPage()` and `isValidLink()`. `isNewPage()` is an important function that determines if the current page needs to be explored or skipped. Although we assume static crawling does not invoke AJAX requests, we play safe and we look at both URLs and contents of pages to determine if the current page needs to be explored. `isValidLink()` is another important function for avoid changing application states. It filters <a> elements by their href attribute values. First, it filters all <a> elements that points to different hosts. Second, it filters all <a> elements that have “#” as href attribute.

5.4 Navigation Graph Crawling

After static crawling, we generate a graph of persistent pages, called the **navigation graph**. Each node represents a persistent page and each edge represents an XPath expression of an <a> element that needs to be clicked on the source to reach the destination node.

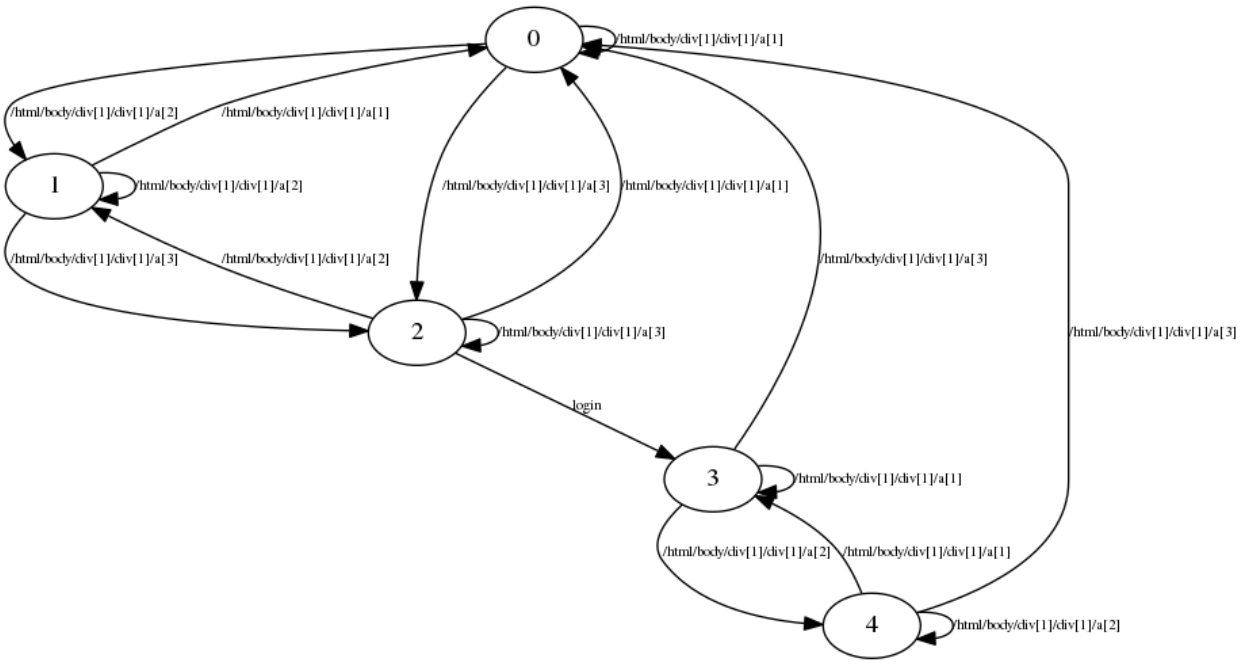

Figure 4 Navigation graph from static crawling minitwit

5.4.1 Crawling Persistent Pages

We crawl persistent pages of a CMS application by exercising each page in the navigation graph. According to 5.3.1.1, persistent pages are categorized into A or B. Thus, as we exercise pages in navigation graph, we will add new contents in various types.

Before exercising persistent pages, we first calculate shortest paths from the root node (Node 0) to all other persistent pages. To visit a persistent page, we construct a list of <a> elements from the shortest path calculated and clicks them one at a time.

Figure 4 is a navigation graph from static crawling a toy application, minitwit. First, we calculate shortest paths from the root node to all persistent pages. In this example, the paths are: 0 → 1, 0 → 2, 0 → 2 → 3, and 0 → 2 → 3 → 4. Second, we visit pages 1, 2, 3, and 4 in a sequence and exercises them by triggering click events on various elements and submitting forms.

5.4.2 Crawling Ephemeral Pages

Crawling ephemeral pages need special attention, as they are not present in the navigation map. We have considered two alternatives:

1. Detect all ephemeral pages created and deleted and update the navigation graph dynamically.
2. Allow crawling ephemeral pages from the nearest persistent pages by giving a depth limit of small amount.

We implemented approach 2, as it is much simpler than approach 1. While we exercise persistent pages, we allow clicking on <a> elements with valid href attributes, to find and exercise ephemeral pages. However, we limit the number of <a> elements clicked to avoid visiting same ephemeral pages many times.

5.5 Pre-emptive Scheduler

Although content types can group all ephemeral pages, each group may have different number of ephemeral pages. We use a simple round-robin pre-emptive scheduler to allocate same amount of time in crawling each group of ephemeral pages. In detail, we assume that each group has equal number of ephemeral pages and give same quantum values.

5.5.1 Problem

As we use a pre-emptive scheduler, a page may have a list of pending operations left before it gets pre-empted. When the pre-empted page runs in the future, the list of pending operations need to be restored to continue from the last run. However, the application state in the future could be different from when the page got pre-empted. Therefore, the restored pending operations may not work correctly.

5.5.2 Proposal

We considered two possibilities to solve the problem:

1. Include the database state when saving the pending operations and restore when the page is switched back.

2. Do not save and restore pending operations during context switch. Instead, increase the quantum for the scheduler to allow the crawler to try more operations during crawling.

We implemented the second approach in CMSCrawler, as we wanted to make CMSCrawler a black-box application as much as possible. Storing database states with pending operations require CMSCrawler to have access to the database. We borrowed an idea from iterative deepening depth-first search (IDDFS) state space search strategy to start with a small quantum and incrementally increase the quantum sizes. IDDFS gives benefits of both breadth-first search (BFS) and depth-first search (DFS) because it finds a shortest path to a goal state and uses much less memory than a standard BFS algorithm.

Although the advantages of IDDFS algorithm do not directly apply to us, we thought IDDFS-based algorithm was suitable for following reasons:

1. Starting with small quantum prioritizes a crawler to crawl more pages and uncover low hanging fruits. (pages with trivial amount of crawling)
2. Incrementing quantum shifts the priority from crawling more pages to spending more time with less number of non-trivial pages to generate interesting requests.

Finally, we mention that incrementing quantum does not guarantee a crawler to make progress and exercise new states. As we do not save and restore application states, it is possible that a crawler need to repeat its previous work to restore the system state. However, we hope that by incrementing the quantum by a sufficient amount, a crawler is likely to progress further in its crawling process and finds more states that are interesting.

6 Implementation

6.1 System Architecture

6.1.1 WebKit Architecture

Before explaining the architecture of CMSCrawler, it is important to understand the architecture of WebKit. WebKit is a popular browser engine that once powered Chrome browser, and currently powers Apple's Safari browser [38]. WebKit ports WebKitGTK+, EFLWebKit, and QtWebKit are available for Linux users.

There are two main releases of WebKit: WebKit1 and WebKit2. Although the difference in the version numbers look small, WebKit's architecture has changed dramatically from WebKit1 to WebKit2.

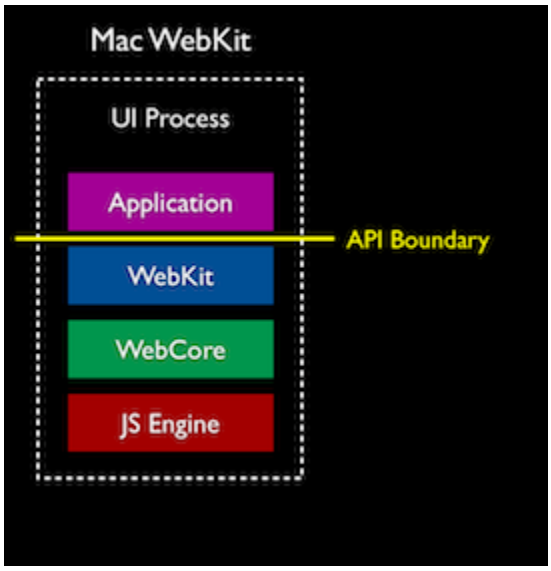

Figure 5 Architecture of WebKit1

In WebKit1, the entire WebKit engine including the JavaScript engine ran as a single process called, UI Process. Internals of the WebKit, such as DOM and the JavaScript engine were accessible from the application using WebKit API.

Although WebKit1's simple architecture is great for building research prototype, it has been deprecated since 2014. Current WebKit-based browsers use WebKit2, which follows Chrome's split-process architecture shown below.

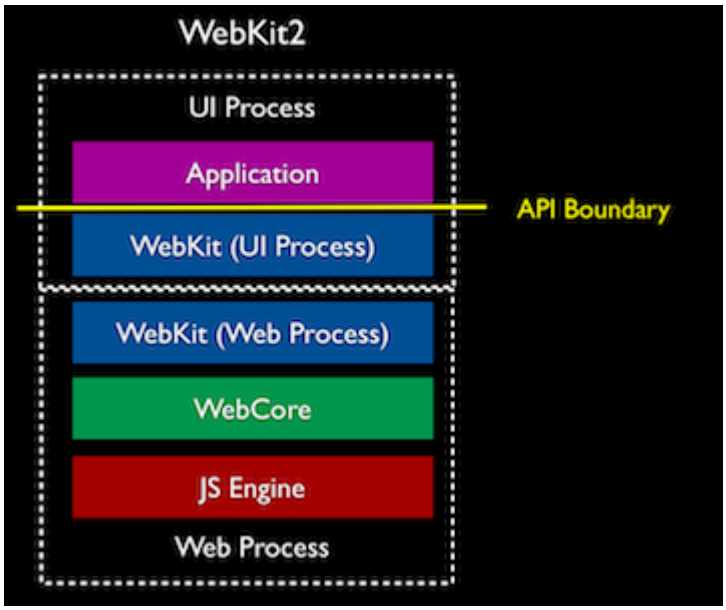

Figure 6 Architecture of WebKit2

WebKit2 runs as multiple processes to isolate its internals and increase security. Applications developed using WebKit runs inside the UI Process and have access to the WebKit API, which includes the function for loading shared libraries as WebKit extensions. The loaded shared libraries run inside the Web Process and have access to the WebKit Extension API. The WebKit Extension API executes inside the Web Process and has access to internals of WebKit including WebCore (DOM) and JavaScript engine. Since the WebKit Extension API is only accessible from the shared libraries, applications that run inside the UI Process communicates with the shared libraries using IPC channels.

6.1.2 CMSCrawler Architecture

Due to the architecture of WebKit2, CMSCrawler consists of two parts. The first part is a simple browser based on the WebKit engine. It is responsible for loading web pages and interacting with the second part. The second part is a WebKit extension with actual crawling algorithm. It runs in the Web Process component of the WebKit engine. Both parts of CMSCrawler interacts with

each other using a message bus system called D-Bus [39]. D-Bus is an IPC mechanism for GNOME and KDE Linux desktop environments.

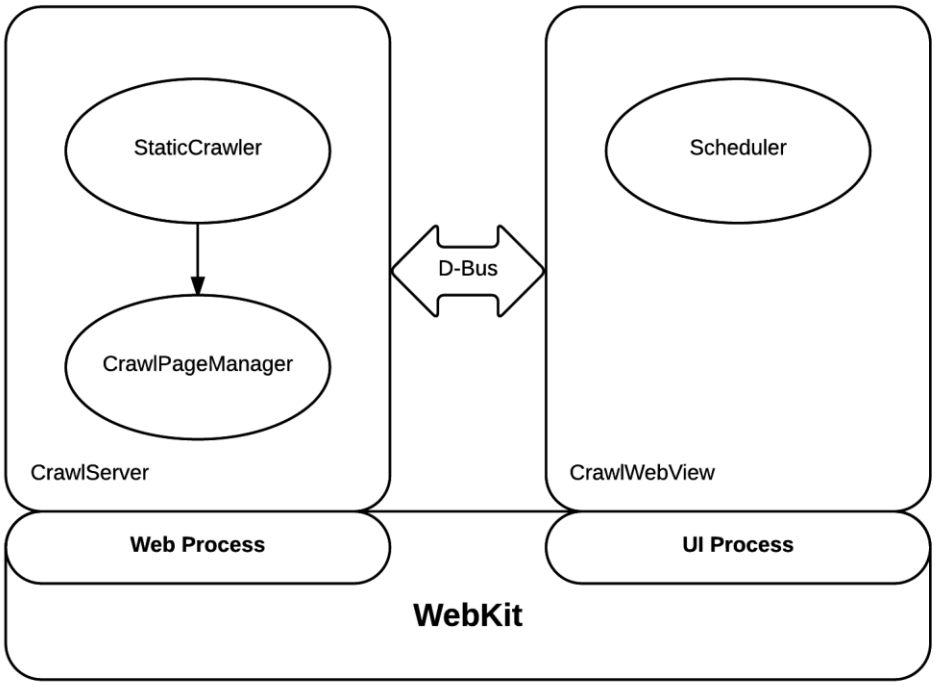

Figure 7 CMSCrawler Architecture

6.1.2.1 Components

Figure 7 shows how all components of CMSCrawler connects together. We describe each component in this section.

CrawlWebView: It is a standard C application that handles browser related functionalities, such as loading the initial web page, clearing the browser cookie, and connecting to the Web Process via D-Bus. It runs as the UI Process.

CrawlServer: It is a WebKit extension with most of crawling logic. It runs as the Web Process and has full access to WebCore and JavaScript engine. It communicates with CrawlWebView using D-Bus.

StaticCrawler: It does a static crawling of an application and constructs the navigation graph. The navigation graph is passed to the CrawlPageManager where the actual crawling happens.

Scheduler: It is the pre-emptive scheduler for scheduling persistent pages. A timer runs on the UI Process to pre-empt the CrawlPageManager that runs on the Web Process. The CrawlPageManager starts and stop the timer on the UI Process using D-Bus.

CrawlPageManager: It is responsible for the navigation graph based crawling algorithm. It constructs shortest paths for all persistent pages in the navigation graph. Then, it schedules each persistent pages and exhaustively exercise application states, until its quantum expires.

6.1.2.2 High-Level Overview

Before running CMSCrawler, we recommend initializing database state with minimal data required for the application to be functional. If the database contains many data, the application may have many ephemeral pages and static crawling will not be efficient.

CMSCrawler works in following steps:

1. CrawlWebView tells CrawlServer to start crawling.
2. CrawlServer starts static crawling.
3. CrawlServer generates a navigation graph.
4. CrawlPageManager calculates shortest paths to all pages using the navigation graph.
5. CrawlPageManager schedules all pages to a scheduling queue.
6. Scheduler selects a page from its queue and starts its timer.
7. CrawlPageManager starts exercising the page.
8. Scheduler pre-empts the CrawlPageManager when the quantum expires.
9. Scheduler selects another page from its queue and starts its timer.
10. When the scheduling queue becomes empty, Scheduler increases quantum and repeats.

6.2 Implementation Detail

CMSCrawler is implemented using WebKitGTK+ 2.10.7 in 3000 lines of Vala. Vala is a C# like language for building GTK applications that is later compiled into C.

6.3 Challenges Faced

6.3.1 Waiting for AJAX completion

A challenge with crawling AJAX applications is determining when AJAX operations are completed. As many JavaScript event listeners read or change the page's DOM dynamically, it is important that the listeners execute after the DOM is fully loaded. For example, jQuery recommends that all jQuery codes be wrapped with an event listener that executes when the current page finishes loading.

Determining if an AJAX operation is completed requires a similar technique. After every operation, it needs to wait until all executed JavaScript event listeners are complete. Unfortunately, it is not possible to know about registered JavaScript event listeners or when all registered listeners finish executing without instrumenting a browser. If a crawler does not wait for all AJAX operations to finish, it may miss some dynamically generated contents.

For example, Wagtail uses JavaScript event listeners in its image uploading page. When a user chooses images, the page sends an AJAX request to Wagtail and new forms are added to the bottom of the page for each uploaded image. The new forms allow the user to change or delete the uploaded images. If a crawler does not wait until all AJAX operations finish, it will miss these forms.

We assume that most JavaScript event listeners will not take a long time to finish. By default, we wait for 1 second after every operation to give sufficient time for JavaScript event listeners to complete. The time out value (1 second) is configurable.

6.3.2 Black Listing

When evaluating CMSCrawler against few CMS applications, we have encountered some problems. With some CMS applications, CMSCrawler successfully deleted essential resources such as the user it was using for crawling and it failed all subsequent login attempts.

We handle the problem by allowing users to blacklist certain forms to prevent CMSCrawler from deleting the essential resources.

6.3.3 File Uploading

HTML supports file uploading via `<input type="file">` element. The element must be placed inside a form element and shows up as a button.

```
<form action="demo_form.asp">
  <input type="file" name="pic" accept="image/*">
  <input type="submit">
</form>
```

Figure 8 Input element for file uploading in HTML

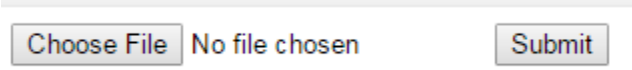

Figure 9 Input element for file uploading

When a button is clicked, a file dialog is opened and the user can select files to upload. This seemingly simple procedure is difficult to implement for several reasons.

First, the `<input type="file">` element had a JavaScript event listener registered in Wagtail and the form was missing a submit button. Instead, clicking the element invoked the registered JavaScript listener function and fired the submit request via AJAX. Manually submitting the form via WebKit's DOM API failed as Wagtail detected that the form submission was not done via AJAX and rejected. To properly handle all such cases, simulating what a real user would do is necessary, which is clicking the `<input type="file">` button.

Second, clicking the `<input type="file">` button could not be simulating from WebKit's DOM API. Manually constructing the JavaScript click event and triggering it to the button does not work. To correctly open the file dialog, the click event must be fired from the underlying OS or its graphical interface such as X Windows.

We implemented file uploading by using the GDK (GIMP Drawing Kit) library, which acts as a wrapper around the low-level functions provided by the GNOME graphics systems. GNOME is

the default graphical environment for Ubuntu OS, and therefore GDK is available in most Ubuntu based OS. GDK library supports querying IO devices attached to the system and intercepting various IO events toward connected devices. To click the button, we first calculated the `<input type="file">` element's location relative to the browser window using the JavaScript function `getBoundingClientRect()`. Then, we created two fake GDK mouse events for pressing and releasing the left mouse button to simulate clicking the button. Finally, we wrote a callback function to automatically select an image from local file system once the file dialog opens.

6.3.4 Confirmation Dialog

Confirmation dialogs are displayed to get confirmation from the user before carrying out critical operations. Although they are important features to prevent user mistakes, they may stop the crawler from working until they are closed.

```
var fakeConfirm = window.confirm;
window.confirm = function() {
  return true;
}
```

Figure 10 Confirmation Dialog Handling Code

To fully make the crawler automatic, we wrote a short JavaScript code and evaluated it after every page is loaded to simulating clicking the "Ok" button.

6.4 Discussion

It is worth mentioning that implementing CMSCrawler as a browser extension is a possibility. Initially, we have considered implementing CMSCrawler as a Chrome extension. We originally thought there are many advantages to the approach. First, Chrome is one of the most widely used browser and most web applications will not break because of browser implementation differences. Second, CMSCrawler can benefit from frequent browser updates and work with latest browser technology. Third, users can install Chrome extensions with few clicks.

Chrome browser comes with very powerful developer tools such as JavaScript console, debugger and profiler. The debugger is controllable with its remote debugging protocol. They support very

powerful features such as looking up on registered event listeners on any DOM elements, setting breakpoints on JavaScript code, and evaluating JavaScript expressions in the context of current page. However, after deeper investigation of the Chrome programming environment, we concluded the remote debugging protocol was not powerful enough for our purpose. Upon some reflection, that while seemingly an attractive implementation option, the Chrome extension API is written with security in mind and was too limiting for our purpose.

7 Evaluation

We have evaluated effectiveness of CMSCrawler in exercising application states by measuring the total number of POST requests it generates. As RFC 7231 recommends all state-changing requests to use POST methods, it is important for a crawler to be able to generate every valid POST request the application accepts to exercise interesting application states.

We compare CMSCrawler against some of popular scanners and crawlers: w3af, Arachni, Crawljax, and Artemis (modified). For applications, we have chosen a toy application: minitwit and two CMS applications: Wagtail, and DjangoCMS.

7.1 Setup

7.1.1 w3af

We use auth plugin to configure login credentials. It accepts configuration parameters to set usernames, passwords, login URLs, and the string for checking current login status.

We have enabled two plugins for testing: auth, crawl.web_spider

7.1.2 Arachni

Arachni also supports a plugin for automated login. Similar to w3af, we specify usernames, passwords, login URLs, and the string for checking current login status.

In addition, we have increased the maximum number of open file limit in Linux to 1000000 to prevent Arachni from running out of file descriptors.

7.1.3 Crawljax

Crawljax does not have a separate login plugin. In contrary, it allows users to specify values to override specific form fields. We have added usernames and passwords for automated login.

In addition, we have configured Crawljax to click on following elements: <a>, , <button>, and <input>. Furthermore, we have configured its depth-first search (DFS) depth to 10 and allowed it to click on a same element multiple times.

7.1.4 Artemis

We have used our modified version for Artemis, as the original Artemis does not handle multi-page applications. Similar to Crawljax, Artemis allows users to specify values to override specific form fields. We have added usernames and passwords for automated login.

7.1.5 CMSCrawler

We have configured login credentials and configured blacklists to void CMSCrawler from breaking applications.

7.2 Application Statistics

Application	Application Type	Lines of Code
Minitwit	Toy version of social media	301
Wagtail	CMS	32445
DjangoCMS	CMS	49945

Table 5 Test Application Sizes

We chose a simple application, minitwit to verify that all crawlers are working correctly. We chose Wagtail and DjangoCMS as they are popular CMS applications written in Django and we have most experience in Django than any other web development frameworks.

7.3 Results

7.3.1 minitwit

Applications	# Unique GET sent	Total # GET sent	# Unique POST sent	Total # POST sent	Running time
w3af	23	52	3	4	4 sec
Arachni	6	27	1	2	1 minute

Crawljax	7	63	3	8	34 sec
Artemis	7	56	3	8	30 sec
CMSCrawler	7	133	3	31	1 minute

Table 6 Result for minitwit

We evaluated crawlers against minitwit first to check if they can handle a very simple application. Excluding Arachni, all scanners were able to generate all POST requests supported by minitwit. We suspect Arachni misses few POST requests as it cannot differentiate the main page before and after a user logs in because it relies on the URL of pages to determine page similarities.

7.3.2 Wagtail

Applications	# Unique GET sent	Total # GET sent	# Unique POST sent	Total # POST sent	Running time
w3af	70	116	2	4	4 sec
Arachni	157	2055	3	9	13 minutes
Crawljax	38	320	2	25	32 sec
Artemis	185	3407	13	801	2 hours
CMSCrawler	236	3140	41	742	40 minutes

Table 7 Result for Wagtail

During our evaluation against Wagtail, we have encountered many problems. w3af could not login as the specified user because it did not include the correct csrftoken in login requests. CSRFToken is a cryptographic token used to protect applications against CSRF attacks. We think the problem occurred because w3af does not parse login pages. Arachni had a problem with the maximum number of file descriptors. Although we have increased the maximum

number of file descriptors limit to 1000000, it ran out available file descriptors. As the total number of HTTP requests it sent is less than 1000000, we suspect the problem is in somewhere else. Crawljax finished quickly, regardless of configuration settings used. We suspect that it has problems with its page similarity calculation algorithm.

7.3.3 DjangoCMS

Applications	# Unique GET sent	Total # GET sent	# Unique POST sent	Total # POST sent	Running time
w3af	54	186	2	3	7 sec
Arachni	146	4849	7	51	12 minutes
Crawljax	32	183	3	6	2 minutes
Artemis	88	19334	27	13770	2 hours
CMSCrawler	47	4869	5	606	7 minutes

Table 8 Result for DjangoCMS

With DjangoCMS, Artemis performed best by a significant margin. DjangoCMS heavily uses iframes, instead of changing the current page's URL and many crawlers failed to detect the iframes. Artemis performed great because we have modified it to differentiate multiple pages loaded on iframes. In contrary, we did not have enough time to implement the similar technique to the CMSCrawler. Thus, CMSCrawler stopped working after 7 minutes because it could not find new pages loaded on separate iframes. We plan to fix this issue.

7.4 Conclusion

Although CMSCrawler did not work as well as we expected in DjangoCMS, we think it is because CMSCrawler does not handle iframes, yet. In addition, we think it still performs significantly better than existing scanners in all other cases. Nevertheless, we feel that we need to add iframe-handling feature to CMSCrawler and evaluates it against more CMS applications to make a proper judgement on the effectiveness of CMSCrawler.

8 Future Work

The most obvious and straightforward way to improve CMSCrawler is to handle more types of JavaScript events supported by major browsers. For example, Artemis handles many JavaScript events by instrumenting its browser engine. In detail, it modifies its WebKit engine to interpose on JavaScript event listener registrations and stores the list of registered events for each page. When it visits a page, it tries triggering registered events on the page. We are planning to take similar approach to handle more JavaScript events.

Although we have targeted CMS applications in this thesis, we predict that similar algorithm may work in general for other web applications. We plan to extend our work for non-CMS applications and evaluate our prediction.

In addition, we have an interesting idea for improving efficiency of crawling AJAX applications that we plan to investigate in the future. Previously, we observed that many AJAX applications use GUI widgets, often build in JavaScript. In addition, we think a user implicitly knows how to use GUI widgets correctly. Thus, we are planning to investigate a reliable way to extract GUI widgets from AJAX pages and formulate correct ways to use GUI widgets. We predict that crawling AJAX applications by GUI widgets will make a crawler to behave more similar to a real user with higher efficiency.

Finally, we want to integrate CMSCrawler with a concolic execution framework to find bugs in CMS applications. One of the main problem with applying concolic execution to programs written in dynamically typed languages has been inferring valid function argument types. We are planning to instrument application to record function argument types while CMSCrawler runs, and generate test cases for all recorded function argument types using concolic execution. Another problem applying concolic execution to programs using database is generating valid database states. We are planning to reuse interesting database states found with CMSCrawler.

9 Conclusion

In this thesis, we described our design of the crawling algorithm for exercising states of CMS applications. Many existing web vulnerability scanners have trouble with the large size of CMS applications. As the number of pages in CMS applications grows as users add contents, we find that many existing scanners exercise only small subset of the application's total state space. To make the matters worse, we find that many page similarity detection algorithm to prioritize new pages do not work correctly in AJAX applications. To handle the growing number of pages in CMS applications, we have made three observations on CMS applications and designed an algorithm that prioritizes new pages according to their content types and equally exercises all parts of the application.

We have implemented our algorithm in a tool called CMSCrawler on top of a WebKit browser engine. We have evaluated CMSCrawler against several other existing scanners and have showed that it performed significantly better in most cases. However, we find that the number of evaluations we have done is small to make a proper judgement on the effectiveness of our algorithm. We plan to continue our evaluation on CMSCrawler and improve its effectiveness. In addition, we plan to integrate the concolic execution framework that we have with CMSCrawler to find bugs in CMS applications.

10 Bibliography

[1] T. Ewer, "14 Surprising Statistics About WordPress Usage," 7 2 2014. [Online]. Available: https://managewp.com/14-surprising-statistics-about-wordpress-usage. [Accessed 13 2 2016].

[2] O. Cassetto, "Why CMS Platforms Are Common Hacking Targets (and what to do about it)," Incapsula, 2014.

[3] w3af, "w3af," 2013. [Online]. Available: w3af.org.

[4] M. C. a. G. V. Adam Doupé, "Why Johnny can't pentest: an analysis of black-box web vulnerability scanners," in *In Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, Berlin, Heidelberg, 2010.

[5] Python Software Foundation, "Glossary," 2016.

[6] A. v. D. a. S. L. Ali Mesbah, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Transactions on the Web*, vol. 6, no. 1, p. 30, 2012.

[7] w3schools.com, "AJAX Introduction," [Online]. Available: http://www.w3schools.com/ajax/ajax_intro.asp. [Accessed 13 2 2016].

[8] M. Rouse, "content management system (CMS)," TechTarget, 1 2011. [Online]. Available: http://searchsoa.techtarget.com/definition/content-management-system. [Accessed 14 2 2016].

[9] M. Rouse, "content management (CM)," TechTarget, 1 2011. [Online]. Available: http://whatis.techtarget.com/definition/content-management-CM. [Accessed 13 2 2016].

[10] R. Gaucher, "Grabber," [Online]. Available: http://rgaucher.info/beta/grabber/. [Accessed 13 2 2016].

[11] Subgraph, "Vega Vulnerability Scanner," [Online]. Available: https://subgraph.com/vega/. [Accessed 13 2 2016].

[12] OWASP, "OWASP Zed Attack Proxy Project," [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. [Accessed 13 2 2016].

[13] N. Surribas, "The web-application vulnerability scanner," [Online]. Available: http://wapiti.sourceforge.net/. [Accessed 13 2 2016].

[14] OWASP, "Category:OWASP WebScarab Project," [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project. [Accessed 13 2 2016].

[15] Google, "skipfish," [Online]. Available: https://code.google.com/archive/p/skipfish/. [Accessed 13 2 2016].

[16] M. Zalewski, "ratproxy," [Online]. Available: https://code.google.com/archive/p/ratproxy/. [Accessed 13 2 2016].

[17] B. D. A. Guimaraes, "Advanced SQL injection to operating system full control," 2009.

[18] Edge-Security, "Wfuzz: The web application Bruteforcer," [Online]. Available: http://www.edge-security.com/wfuzz.php. [Accessed 13 2 2016].

[19] byrnedr, "Grendel-Scan," [Online]. Available: https://sourceforge.net/projects/grendel/. [Accessed 13 2 2016].

[20] R. M. J. D. M. H. J. B. S. Samuel Bucholtz, "Why use the Watcher passive Web-security scanner?," [Online]. Available: https://websecuritytool.codeplex.com/. [Accessed 13 2 2016].

[21] C. Weber, "x5s - XSS and Unicode transformations security testing assistant," [Online]. Available: https://xss.codeplex.com/. [Accessed 13 2 2016].

[22] T. Laskos, "arachni: web application security scanner framework," [Online]. Available: http://www.arachni-scanner.com/. [Accessed 13 2 2016].

[23] INFOSEC, "14 Best Open Source Web Application Vulnerability Scanners," 24 9 2014. [Online]. Available: http://resources.infosecinstitute.com/14-popular-web-application-vulnerability-scanners/. [Accessed 13 2 2016].

[24] Selenium, "SeleniumHQ," [Online]. Available: http://www.seleniumhq.org/. [Accessed 13 2 2016].

[25] OWASP, "OWASP Top Ten Project," 2013. [Online]. Available: https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013. [Accessed 13 2 2016].

[26] N. a. K. C. a. K. E. Jovanovic, "Pixy: a static analysis tool for detecting Web application vulnerabilities," in *IEEE Symposium on Security and Privacy*, 2006.

[27] P. N. a. V. N. V. Maliheh Monshizadeh, "MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications," in *In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[28] W. K. M. N. P. Z. Edmund M. Clarke, "Model Checking and the State Explosion Problem," *Tools for Practical Software Verification,* vol. 7682, pp. 1-30, 2012.

[29] M. M. a. M. S. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in *In Proceedings of the 17th conference on Security symposium*, Berkeley, 2008.

[30] NASA, "JPF," NASA, 30 10 2007. [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf. [Accessed 13 2 2016].

[31] L. C. C. K. a. G. V. Viktoria Felmetsger, "Toward automated detection of logic vulnerabilities in web applications," in *In Proceedings of the 19th USENIX conference on Security*, Berkeley, 2010.

[32] L. C. C. K. a. G. V. Adam Doupé, "Enemy of the state: a state-aware black-box web vulnerability scanner," in *In Proceedings of the 21st USENIX conference on Security symposium*, Berkeley, 2012.

[33] S. A. a. A. K. a. J. D. a. F. Tip, "Finding bugs in web applications using dynamic test generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474-494, 2010.

[34] SpryMedia, "visual event," SpryMedia, 2013. [Online]. Available: http://www.sprymedia.co.uk/article/visual+event+2. [Accessed 14 2 2016].

[35] S. A. a. J. D. a. S. H. J. a. A. M. a. F. Tip, "A Framework for Automated Testing of JavaScript Web Applications," in *Proc. 33rd International Conference on Software Engineering*, 2011.

[36] J. R. R. Fielding, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," 6 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231. [Accessed 13 2 2016].

[37] MoinMoin Wiki, "The MoinMoin Wiki Engine," MoinMoin Wiki, [Online]. Available: https://moinmo.in/. [Accessed 11 2 2016].

[38] P. Irish, "WebKit for Developers," 28 2 2013. [Online]. Available: http://www.paulirish.com/2013/webkit-for-developers/. [Accessed 14 2 2016].

[39] freedesktop.org, "What is D-Bus?," 14 1 2016. [Online]. Available: https://www.freedesktop.org/wiki/Software/dbus/. [Accessed 13 2 2016].