Institutionen för datavetenskap
Department of Computer and Information Science

Master thesis

Link Extraction for Crawling Flash on the Web

by

Daniel Antelius

LIU-IDA/LITH-EX-A--15/11--SE

2015-04-26

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Master Thesis

Link Extraction for Crawling Flash on the Web

by

Daniel Antelius

LIU-IDA/ LITH-EX-A--15/11--SE

2015-04-26

Supervisor: Christoph Kessler
Examiner: Christoph Kessler

Abstract

The set of web pages not reachable using conventional web search engines is usually called the hidden or deep web. One client-side hurdle for crawling the hidden web is Flash files.

This thesis presents a tool for extracting links from Flash files up to version 8 to enable web crawling. The files are both parsed and selectively interpreted to extract links. The purpose of the interpretation is to simulate the normal execution of Flash in the Flash runtime of a web browser. The interpretation is a low level approach that allows the extraction to occur offline and without involving automation of web browsers. A virtual machine is implemented and a set of limitations is chosen to reduce development time and maximize the coverage of interpreted byte code.

Out of a test set of about 3500 randomly sampled Flash files the link extractor found links in 34% of the files. The resulting estimated web search engine coverage improvement is almost 10%.
Keywords: Flash, crawling, spidering, deep web, hidden web, virtual machine, interpretation

Acknowledgments

I would like to thank Picsearch AB (publ) for the opportunity of getting access to the internals of a web search engine. This was invaluable for getting real world test data for the evaluation of the link extractor.

Daniel Antelius
March 2015

Contents

Chapter 1

Introduction

1.1 Background

Crawling is the automated process of visiting hypertext pages recursively on the world wide web. For each page that is visited some or all of the outgoing links are followed to visit new pages. It is a difficult and time-consuming process. Some of the hurdles include the sheer size of the web, infinite dynamic pages (such as calendars) and rich media. Examples of rich media are Adobe Flash, Microsoft Silverlight and interactive HTML5 or JavaScript. Adobe Flash is commonly found on the web [13] embedded on hypertext pages for several different purposes. The embedded Flash used for navigational purposes is the one of interest during crawling.

The easiest way to deal with rich media is to ignore it and only follow links found in the hypertext page. This will lead to poor results when a collection of pages depend on Flash for navigation. To be able to crawl these pages the embedded Flash has to be processed for links.

1.2 Purpose and Goal

The purpose of this thesis is to investigate and implement a means of crawling hypertext pages on the world wide web which utilizes Adobe Flash for navigation. This will be done by sampling a large set of Adobe Flash files, investigating what is required to extract links and then implementing a link extractor for these files. The extractor should from one Flash file produce some or all of the links that are contained in the file. The extractor needs to be able to run on commodity hardware and terminate within one or a few seconds.

The goal is to increase the coverage of web search engines.

1.3 Reader Assumptions

The reader is assumed to be familiar with the World Wide Web and its Hypertext Markup Language. The reader is also assumed to be familiar with the client-side interactive technologies JavaScript and Adobe Flash. An introductory textbook covering the above subjects is Robbins [23].

The thesis also assumes familiarity with language grammars, interpreters and virtual machines. A classic textbook covering these topics is Aho et al. [1].

1.4 Reading Guidelines

- Chapter 2 introduces web search engines and puts them in the context of the problem presented in this thesis.
- Chapter 3 gives an overview of the Flash .swf file format, what capabilities it has and presents a survey which aims to narrow down the feature set that the link extractor would benefit from supporting.
- Chapter 4 is about the design and implementation of the link extractor.
- Chapter 5 presents the final results of the evaluation, details possible improvements and compares the link extractor to other possible solutions.
- Chapter 6 presents recent papers published on this or related topics.
- Chapter 7 summarizes the thesis.

Chapter 2

Search Engines

This chapter introduces search engines for the web and puts them in the context of the problem presented in this thesis. Search engines are described here as they are presented in the initial Google paper [5] from 1998 with some updates from a two part article [11] [12] published in 2006.

2.1 Overview

The operation and functionality of a search engine for the web can be divided in these three steps.

1. Crawling. Crawling is the process of gathering data by visiting many web pages.
2. Indexing. Indexing is the process of building a data structure for fast access by keywords.
3. Searching. Searching is the process of answering search queries.

The following subchapters contain a more thorough explanation of these steps.

2.2 Crawling

A crawl is started from one or several pages, called the seed. Usually a seed with many high quality outgoing links is selected. From this seed, the crawl recursively follows some or all links found on each visited page. The order of traversal and which links are followed is decided by the selected crawling

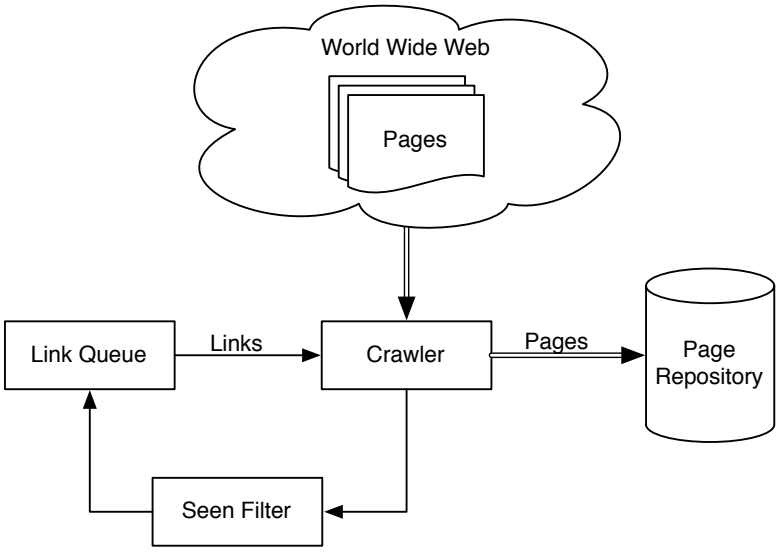

Figure 2.1: Overview of the crawling process.

policy. Each page that is visited and deemed interesting is stored in a page repository for later processing.

Figure 2.1 shows an overview of the crawling process. The simplest form of crawling consists of one queue of links to visit and a fast mechanism for determining if a link has been seen. The crawler performs these steps: Visit the first link in the queue. Extract all the links found. Add each link that has not yet been seen to the queue. Save the page to the page repository. Repeat until the queue is empty. There are however some problems with this simplistic approach. The following three subsections bring up the main problem areas of scalability, coverage and freshness.

2.2.1 Scalability

The number of URLs tracked in a modern web search engine is at least in the tens of billions [2]. The previously mentioned mechanism for determining whether a link has been seen needs to be able to store information about having seen billions of URLs. The queue holding links to be crawled has to have room to store billions of links. To be able to crawl at least a significant portion of the web in a reasonable amount of time crawling has to be parallelized. Retrieving and storing huge amounts of data requires large amounts

of network bandwidth and data storage.

2.2.2 Coverage

The coverage of the spider is decided by a combination of the crawling policy and the technical capabilities of the parser that looks for new links. The policy needs to be such that it minimizes the amount of

Spam content. Artificial material designed by a third party to manipulate search engine results for financial gain.

Duplicate content. Content that can be reached from several different links.

Dynamic content. Dynamic content such as calendars needs to be avoided to avoid crawler traps.

Unwanted content. This could be pages in a foreign language for a domestic search engine.

The crawling policy needs also take politeness into consideration when crawling pages. Reputable crawlers respect the robots exclusion protocol [15]. Small sites can only handle a limited amount of load before their service is degraded. It is also not uncommon that operators of small sites only have limited quota for network traffic. A crawler behaving aggressively might find itself banned from sites which in the end has a negative impact on coverage.

The technical capabilities of the parser that looks for new links and specifically how to handle Flash is what will be investigated in the later chapters.

2.2.3 Freshness

Once a page has been added to the page repository any changes (or removal) made on the web will not be reflected until that page is re-crawled. For the simplistic approach outlined in Section 2.2 this means either restarting with a fresh crawl or re-crawling all or a selected subset of all the visited pages.

2.3 Indexing

Indexing or Information Retrieval is a well researched field. What stands out about web search indexing is the size and non-uniformness of the corpus.

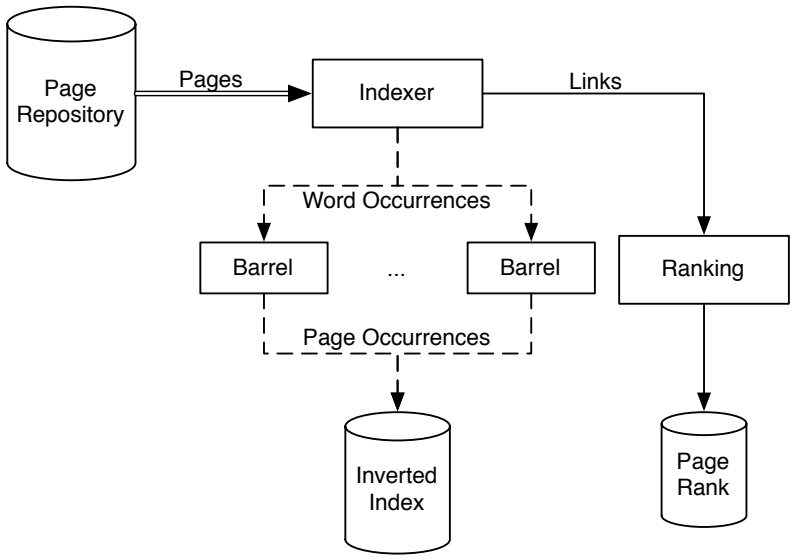

Figure 2.2: Overview of the indexing process.

An overview of the indexing process is shown in Figure 2.2. Very briefly described the indexing process consists of two steps. First the forward index is created. In the context of text web search this is an index of all pages which for each page contains the words bound to that page. The second step consists of inverting the forward index to create the inverted index. In this case the inverted index is from words to pages. Generally some sort of ranking is employed so that words that have many pages are ordered by relevancy in descending order.

A more in-depth view of indexing is given in the following subsections.

2.3.1 Forward Index

All the pages that are stored in the page repository are processed by the indexer. Each page is parsed using an HTML parser and links as well as word occurrences are extracted. The links are stored separately for later ranking purposes. A word occurrence includes the word, the page id and the position of where it was found. These occurrences are stored in the forward index.

2.3.2 Inverted Index

The forward index is inverted so that the word occurrences by page become page occurrences by word. This means that for every encountered word there is a list of pages where this word occurs. Position information is preserved in the inversion.

The inversion is done by distributing the words from the forward index into several barrels. Each barrel holds a consecutive range of words. All barrels are sorted on word with a secondary sort order on page id. A complete inverted index is created by concatenating all barrels.

2.3.3 Ranking

The ranking process assigns a rank to all pages. This rank is usually called the page rank. It is used as a measurement of the quality or value of a page. A simple page rank scheme is to count the number of links pointing to the page. This is the same way papers in academia are measured; by counting references.

A famous refinement of the above mentioned scheme is the Google Page Rank introduced by Brin et al. in [5] and formally defined by Page et al. in [18]. Two refinements are made to how links are valued:

- Normalization. Links are normalized by the number of outgoing links per page. A page with many links will have less valuable links.
- Feedback. The value of a link is adjusted based on the page rank of the linked-from page. A page with high rank will have more valuable links.

The formal definition of the Google Page Rank is:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where $p_1, p_2, \ldots, p_N$ are the pages, $M(p_i)$ is the set of pages that link to p_i, $L(p_j)$ is the number of outgoing links on page p_j, N is the total number of pages and d is a dampening factor.

This page rank forms a probability distribution over all pages (the sum of all page ranks is 1). The Google Page Rank of a page is the probability that a surfer choosing links at random ends up on that page. The dampening factor corresponds to the surfer choosing a page at random (not necessarily from the page the surfer is at).

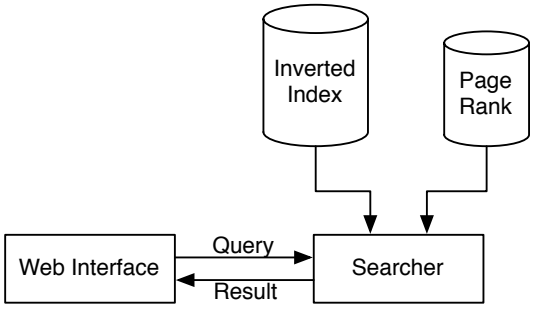

Figure 2.3: Overview of the search process.

The Google Page Rank can be approximated iteratively from all the links that were extracted during indexing. The algorithm typically favors old well linked content over new content. As shown by Cheng et al. in [6] the Google Page Rank is susceptible to tampering. The rank of a page can intentionally be increased by adding links to it.

2.4 Searching

Searching is the process of answering user queries. The inverted index along with the page rank information is combined and processed to provide an ordered result set. The goal is to produce as accurate and relevant result sets as possible within a timely manner. It is safe to assume that the average web search user expects query results within very few seconds. This leads to two main challenges: relevancy and performance.

2.4.1 Overview

The user enters queries to the web front end which are sent verbatim to the searcher. The searcher will do one lookup in the inverted index for each word. Multiple word queries are merged by page id. Any query filter that the user might have requested is used for merging. The searcher now has a list of pages that match the query. The searcher ranks the pages and returns them to the web front end.

An overview of the search process is shown in Figure 2.3.

2.4.2 Relevancy

The information available to the searcher for ranking results of queries is word location(s) and page rank. The word location include information such as where in the page the word occurred and typographical hints of the significance of the word. If the word occurred in the URL, page headings, written in plain text, etc. are all properties that are taken into consideration. For multiword queries locations are analyzed to determine whether the words occur in the same phrase, section or are not close. All these properties and the page rank are combined together to a single scalar. The pages are sorted on this scalar in descending order and sent back to the web interface.

2.4.3 Performance

As mentioned in Section 2.4 a user expects query results within very few seconds. There are a number of tricks web search engines can employ to achieve sub second searching.

- Data Structures. Choosing data structures carefully to keep operations on the large data sets cheap and to minimize disk seeks.
- Limiting the number of displayed results. This means that the searcher can stop processing the result set when it has found enough results.
- Multiple copies of the inverted index. To be able to answer both single and multi word queries quickly two copies of the inverted index can be kept. One copy with each word sorted on relevance for single word queries. Another copy with each word sorted on page id for quickly merging result sets from different words.
- Extensive use of caching.
- Parallelism. As Barroso et al. describes in [4] the reverse index can be sharded. Each shard has a randomly chosen subset of the documents of the full index. The query processing can be performed on each shard in parallel and then the results from each shard can be merged at the end.

Chapter 3

Flash

This chapter gives an overview of the Flash .swf file format and what capabilities it has. Flash files are presented from a binary (file) perspective. In particular we ignore how they are produced and from what source. This chapter only presents Flash files up to but not including version 9.

3.1 Overview

Adobe Flash .swf files are embedded in web pages to provide interactivity and multimedia streaming capabilities. The files can contain vector and bitmap graphics, pre-recorded video and audio clips, script segments and references to other .swf files. The script segment, called ActionScript, has access to library code in the player plugin and can be used to build all sorts of applications. ActionScript can open sockets, files, access input devices, microphones, cameras and more. The typical use case on the web is interactive advertisements, menu systems for navigation or video players.

The web browser loads the .swf file into the Adobe Flash Player plugin and lets the plugin execute the file. The player plugin provides libraries and a virtual machine in which the file is executed. The player is a piece of proprietary software but the .swf file format specification has been released with a permissive license [14].

Important concepts during Flash execution is the timeline and the stage. The timeline is divided into frames. The stage is the main drawing area and is what is shown in the web browser.

3.2 History

The Flash file format has been incrementally extended while maintaining backwards compatibility. New tags and new actions were added mainly to extend functionality. For example, the different available actions went from about ten to more than one hundred, while adding a stack machine, registers, variables, conditional branching and a JavaScript like prototype based inheritance model. For version 9 the ActionScript part was completely rewritten.

3.3 Embedding

The Flash files can be embedded either statically using HTML or dynamically using Javascript. The following are examples of how they can be embedded. Each empty string `"..."` represents a possible Flash URL.

```
<object data="..."><param value="...">
<embed src="...">
<a href="...">
<script>swfobject.embedSWF("...",
```

The first three are HTML variants. Uninteresting tag attributes are not shown. The param tag is only allowed inside the object tag. The script example is for a popular Javascript embedding library. There exists many different such libraries.

3.4 File Structure

Flash files are structured in a streamable format. At the beginning of the file there is a so-called magic file identifier followed by a header. This header contains file length, whether the body of the file is compressed and some other miscellaneous information. The body of the file can be compressed using the well known deflate algorithm [7]. The body is structured into a stream of file chunks called tags. Each tag is prepended with a tag identifier and the tag length. This makes it possible to start processing the tags even before the entire file has been downloaded. It is also possible for a file parser to skip uninteresting or new and unknown tags.

The tags can be categorized as either definition or control tags. Definition tags introduce new shapes, images and other assets. These definitions are assigned a unique identifier and are stored in a key value store called the

dictionary. Control tags create and manipulate instances of these previously defined assets or control the flow of execution. The control and definition tags are generally interleaved and in timeline order.

Tags can contain lists of other tags, some only contain references to the dictionary while others contain entire JPEG images or sound samples. Tags can also as previously mentioned contain ActionScript which are represented as byte code actions to perform during certain events such as at the start of a specific frame on the timeline, when a user clicks a button, etc. Depending on which version of Flash the file is targeting the actions can be of a very simple nature such as pausing the timeline up to fully featured opcodes executing on a stack machine with registers, variables and objects in scope chains.

All parts of the file employ common compression techniques such as bit packing, delta encoding, standard compressed formats and optional extensions with default values. This makes the file format cumbersome to parse but space efficient.

The interested reader is referred to the Flash format specification [14] for the details of all tags and actions.

3.5 Links

The links present in a Flash file can be categorized as one of two types: static or dynamic. Static links can be read out by parsing the file for all tags and actions. Dynamic links requires executing or interpreting actions in the context of a virtual machine.

The tags that can contain static links are Import and DefineEditText. Import is used for including external Flash files (similar to an import statement). DefineEditText is used for displaying a text box or text input in Flash. Optionally, the text box can contain a restricted subset of HTML. This HTML can contain anchor tags with links.

The actions that can contain either static or dynamic links are GetUrl and GetUrl2. The first contains the link as a static string stored with the action. The second pops the virtual machine stack for the link.

3.6 Grammar

To give the reader a better overview of the overall file structure a few grammar rules are presented in Backus-Naur form. The rules are derived from

the Flash format specification [14]. The rules highlight the general parse structure of a Flash file and where links can be found.

⟨*flash-file*⟩ ::= ⟨*magic*⟩ ⟨*header*⟩ ⟨*body*⟩

⟨*body*⟩ ::= ⟨*deflate*⟩ | ⟨*tag-list*⟩

⟨*deflate*⟩ ::= ⟨*tag-list*⟩

⟨*tag-list*⟩ ::= ⟨*tag*⟩ ⟨*tag-list*⟩ | ⟨*end-tag*⟩

⟨*tag*⟩ ::= ⟨*tag-id*⟩ ⟨*tag-length*⟩ ⟨*tag-body*⟩
| ⟨*tag-define-button*⟩ ⟨*action-list*⟩
| ⟨*tag-define-button-2*⟩ ⟨*action-lists*⟩
| ⟨*tag-define-edit-text*⟩ ⟨*html*⟩
| ⟨*tag-define-sprite*⟩ ⟨*tag-list*⟩
| ⟨*tag-do-action*⟩ ⟨*action-list*⟩
| ⟨*tag-do-init-action*⟩ ⟨*action-list*⟩
| ⟨*tag-import*⟩ ⟨*import-url*⟩
| ⟨*tag-place-object*⟩ ⟨*action-list*⟩

⟨*action-lists*⟩ ::= ⟨*action-list*⟩ ⟨*action-lists*⟩ | ⟨*action-list*⟩

⟨*action-list*⟩ ::= ⟨*action*⟩ ⟨*action-list*⟩ | ⟨*end-action*⟩

⟨*action*⟩ ::= ⟨*action-id*⟩
| ⟨*action-id*⟩ ⟨*action-length*⟩ ⟨*action-body*⟩
| ⟨*action-declare-function*⟩ ⟨*action-list*⟩
| ⟨*action-declare-function-2*⟩ ⟨*action-list*⟩
| ⟨*action-get-url*⟩ ⟨*get-url*⟩
| ⟨*action-get-url-2*⟩

3.7 Example File

In order to illustrate Flash file structure and semantics a small file is presented here. The file is one of the smallest files from the test set used in Chapter 5 and contains a single button. In grammar terms the file structure is:

⟨*magic*⟩
⟨*header*⟩

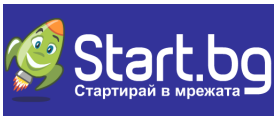

Figure 3.1: Screenshot of example Flash file.

⟨*tag-set-background-color*⟩
⟨*tag-define-shape-3*⟩
⟨*tag-place-object-2*⟩
⟨*tag-define-font-2*⟩
⟨*tag-define-text*⟩
⟨*tag-place-object-2*⟩
⟨*tag-define-shape-3*⟩
⟨*tag-define-button-2*⟩
    ⟨*action-get-url*⟩ ⟨*get-url*⟩
    ⟨*end-action*⟩
⟨*tag-place-object-2*⟩
⟨*tag-show-frame*⟩
⟨*tag-end*⟩

The tags are executed by the Flash runtime in file order. The header contains information about canvas size and the number of frames on the timeline. This file only has one frame. The first tag sets the background color of the canvas. Next a shape containing a vector image is defined and placed. Following is an embedded font and a short text string which is also placed on the canvas. The third shape defined is used as the bounding box for the following button. The button contains a short list of actions that are triggered when a mouse click occurs inside the bounding box. The list of actions contains one action, GetURL, which redirects the user to a new page. The button is placed on the canvas. The final tag, ShowFrame, instructs the Flash runtime to render the first frame.

A screenshot of the example file can be seen in Figure 3.1.

3.8 File Survey

To better understand which Flash features are in common use a survey was performed. The web page repository of the Picsearch web search engine was randomly sampled for about 500 embedded Flash files. These files were downloaded and processed with a parser. The parser emitted information

Name	Seen
DefineButton	2%
DefineButton2	41%
DefineEditText	**30%**
DefineSprite	73%
DoAction	58%
DoInitAction	13%
Import	**0%**
Import2	**0%**
PlaceObject	0%
PlaceObject2	92%
PlaceObject3	0%

Table 3.1: Summarized file survey results of tags. Possible link sources in bold.

Name	Seen
Declare Function	37%
Declare Function2	24%
GetURL	**40%**
GetURL2	**28%**

Table 3.2: Summarized file survey results of actions. Possible link sources in bold.

about Flash version and which tags and actions were present in each file. The tags that can contain links or are required for parsing the entire structure are summarized in Table 3.1, the actions in Table 3.2. Complete tables of all tags and actions are included in Appendix A.

From the distribution of seen tags and actions a few observations can be made. Flash files contain both dynamic and static links in significant numbers. Because dynamic links are only found in the GetURL2 action, the majority of links should be found by parsing and extracting static links. Also based on the relative number of found GetURL2 actions the addition of dynamic link extraction appears worthwhile. It should increase the number of found links noticeably. Actual numbers for a much larger data set is presented in Chapter 5.

Chapter 4

Link Extractor

This chapter presents the design and implementation of the link extractor.

4.1 Design

The work on this thesis was started in late 2006 when only Flash versions up to 9 existed. Because of this and in order to limit the scope of the thesis only Flash files up to and including version 8 are investigated.

The link extraction process can be divided into two distinct steps: parsing and interpreting. The parsing step is comparatively easy and finds all static links. It builds on top of the file survey tool from Section 3.8. Additions are made to store found URLs and all the parsed data for later interpretation. For the GetURL Action, the URL is stored inside the action. For the DefineEditText Tag, an HTML parser is needed. For the Import Tag, the URLs are stored directly inside the tag.

The interpretation step is harder. In order to exactly replicate the execution of Flash files a Web browser with a Flash runtime plugin is required. The web browser is pointed at the web page where the flash file is embedded. Programmatic interaction with the Flash plugin is difficult. A possible options is if the Flash file and browser supports accessibility aids for disabled or impaired users that could be used for accessing the user interface of the Flash file. Another option is automating mouse clicks for exhaustive searching of the rendered user interface. The search would attempt to visit all user interface states in order to find all clickable links. This thesis takes a simplified and lower level approach.

The link extractor will parse and selectively interpret actions in an environment that simulates the Flash runtime environment. The overall strategy

of the interpretation is to cover as much byte code as possible as few times as possible. Two basic assumptions or limitations are made.

1. No external resources are allowed to be fetched.

2. No complex calculations need to take place for URLs to be generated.

The reasoning behind these limitations is scalability and limiting the scope of the implementation.

As a result of these limitations the extractor will run in isolation and not access any external resources. The extractor will not honor any branch instructions or other control instructions other than function calls. Function calls will be restricted to a very shallow depth. Exception handling and other esoteric functionality present at the instruction level is ignored. This results in a machine model with a stack, registers and automatically allocated heap memory. The supported data type is dynamic and can assume values of types such as boolean, integral, string, callable (function object) or object (data and code encapsulation). In addition to the machine the runtime environment contains scopes and constant pools. The scope is used for the local variables and for chaining variable lookups into the scopes of any enclosing tags. The root of the scope chain should contain parts of the ActionScript 1 or 2 standard library.

To simulate normal execution in the Flash runtime environment the byte code is interpreted in two steps. The first step is the timeline interpretation. This step simulates the normal execution path with a user interacting as much as possible with the Flash. As soon as an object is put on the stage all of its event handlers are fired. The second step is coverage interpretation. This step tries to call all previously not called functions. All calls are made with all null parameters, which is identical to the functions being called without parameters.

The overall execution order of the link extractor:

1. Initialization. Interpret all actions contained in all DoInitAction Tags. Add sprites to scope chains from PlaceObject Tags.

2. Fire onLoad events. Interpret actions in onLoad event handlers on previously added sprites.

3. Timeline interpretation. Interpret the actions in all Tags in timeline order, including event handlers.

4. Coverage interpretation. Call all previously defined functions that have not been called.

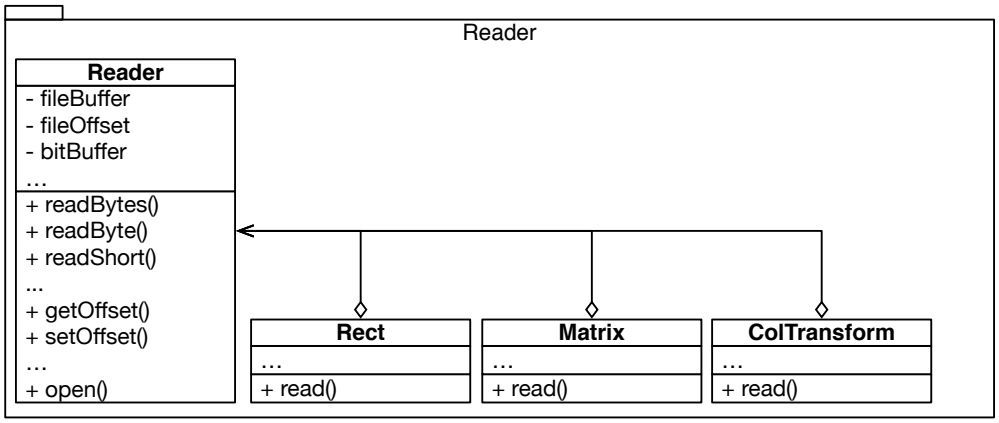

Figure 4.1: UML class diagram of the Reader component.

All GetURL2 action calls will store the URL string internally in the link extractor. At the end of parsing and interpretation all the unique strings will be emitted including information of which source and which stage generated how many URLs.

4.2 Implementation

The implementation can be divided into four major parts: Reader, Interpreter, Tags and Actions. The Reader and Interpreter parts are fairly independent and present interfaces to the file and virtual machine. The Actions and Tags parts both represent the parsed structure of the file and use the Reader for file access and the Interpreter for static relationships. The Actions part represents the ActionScript contained in the tags and drive the Interpreter.

4.2.1 Reader

The Reader performs the actual file opening and reading. It performs the initial file header reading and the optional deflate decompression. The well known Zlib decompression library [8] is utilized. The Reader provides an interface for bit and byte specific reads as well as reading common Flash data structures. The overall control of reading all tags and actions is delegated to the Tags and Actions. All of the classes in the Reader component can be seen in Figure 4.1.

4.2.2 Interpreter

The Interpreter part provides the machine context in which the Tags and Actions are initialized and interpreted. A machine context contains a stack, registers and the currently active scope. Depending on the version of the file and if the Interpreter is inside a function either 4 or 255 registers are available. A machine context also maintains a counter for the current call depth. The Interpreter exposes interfaces and objects to both the Tags and Actions modules to facilitate interpretation. Actual interpreting of bytecode is performed by the Action module on the Interpreter module.

The data type used is dynamic. A single data type called Element is implemented in the link extractor that can take the shape of either: float, int, bool, string, object, null, undefined, function or scope. Element implements very generous type conversions since exceptions were left out of the design. The data types are the same as found in JavaScript with the exception of scope.

The object is an ordered mapping of member names to values. Members are accessed either using index or by member name. Actionscript uses prototype based inheritance, similar to JavaScript. The prototype inheritance of objects is not implemented. As a place holder all objects have a fixed and shared prototype.

Variable scope is implemented using a simple mapping from name to value. To represent lexical scope each enclosing tag has its own scope. These scopes are chained together with the current (local) scope for variable lookups. The root of the scope chain is the shared global scope. Scope is re-used as a datatype when placing Flash specific constructs in the top level scope. For example a sprite or a button from a DefineSprite or DefineButton tag can be stored in a named variable in the global scope.

A link extractor specific part of the Interpreter is the URL store. The store is a container for all found URLs during parsing and interpretation. It categorizes all found unique URLs based on source. The source is either the tag or action name paired with the stage of interpretation. After the link extractor is finished the URL store emits all found unique URLs and counts of sources.

All of the classes of the Interpreter can be seen in Figure 4.2.

4.2.3 Tags

The Tags uses the Reader's interface for reading tags. If the tag is interesting in terms of later interpretation it is read out and its contents are stored. Each

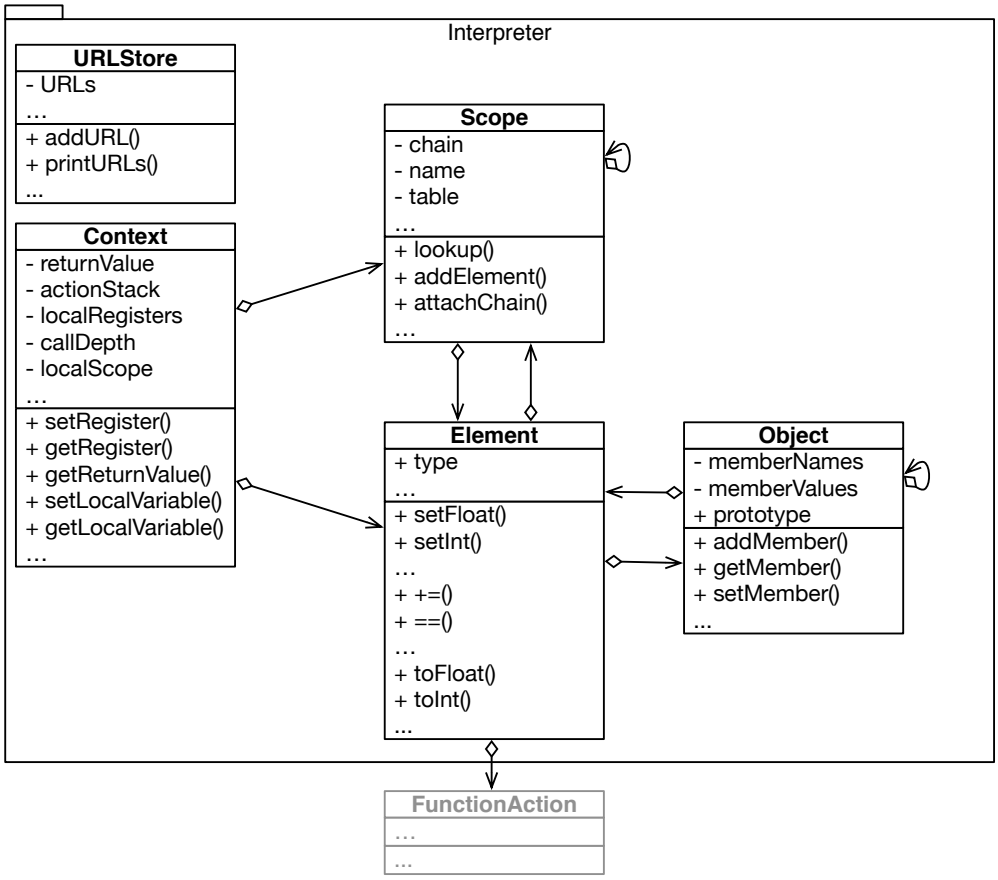

Figure 4.2: UML class diagram of the Interpreter component.

interesting tag is subclassed from the base tag class and implements its own read and interpret methods. The base tag simply discards the contents of the tag and advances the Reader to the next tag.

Tags can in turn contain lists of other tags or actions. These are recursively read out by either the Tags or Actions parts and stored in its enclosing tag as a list of either tags or actions.

If the tag contains static links they are parsed out directly. This requires searching HTML for anchor tags with href attributes.

The tag is also responsible for the interpretation order of the tags or actions it contains. Each step in the interpretation order laid out in Section 4.1 is performed by a method on each tag or action. A Flash file is represented by nested lists of tags and actions. Tags can not be nested inside actions. The interpretation is done by calling the same method on each element in each list in a depth first nested list traversal.

Most of the classes of the Tags module can be seen in Figure 4.3. Similar tags have been stacked on top of each other. External dependencies or associations are drawn in gray.

4.2.4 Actions

The Actions use the Reader's interface for reading actions. In contrast to Tags, no actions are discarded and all actions have an enclosing tag. The base class for an action stores the opcode of the bytecode. An opcode that carries payload data overrides the read method to correctly parse the payload. During the parsing stage links from GetURL are parsed out.

Each action implements its own functionality using the interface of the Interpreter. The base class contains the functionality of all actions without payload. The majority of actions are implemented but some are simplified and others are even ignored. Actions that were found to be very uncommon in the file survey were skipped. Actions that branch or control the timeline are simplified so that the branch is never taken. Actions that perform esoteric or very complex functionality are either skipped or simplified to reduce the scope of the implementation. This is a consequence of the design decision in Section 4.1. The overall approach is to include the basic functionality required for generating simple strings on the stack for the GetURL2 action. Table 4.1 summarizes the status of implemented actions.

Each list of actions can have an optional constant pool. The constant pool is carried in the payload of `ActionConstantPool` and is tracked separately for use by `ActionPush`. `ActionPush` can either push a constant from the pool or push data directly from its payload.

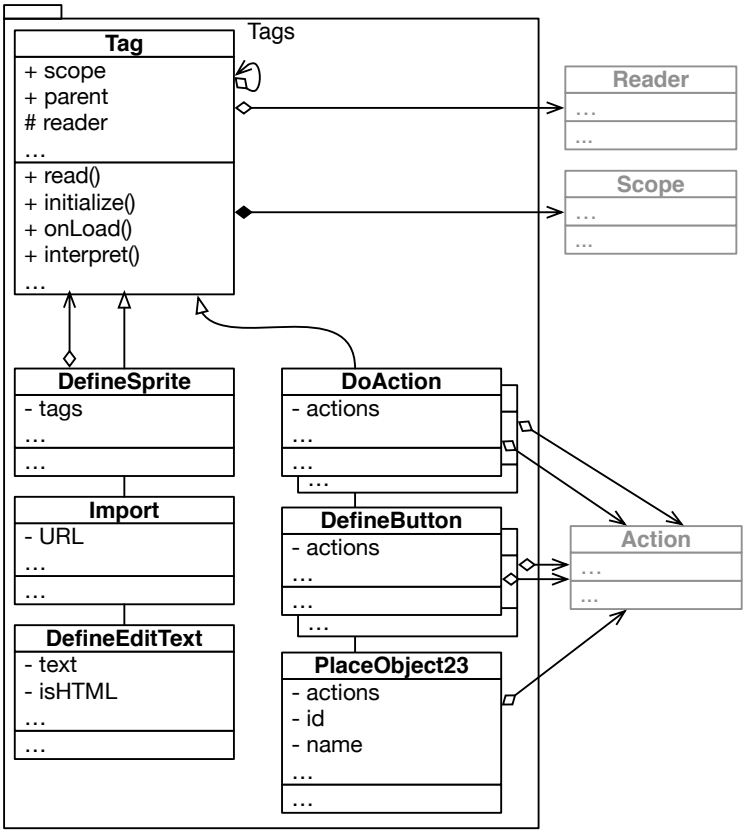

Figure 4.3: UML class diagram overview of the Tags component.

Implemented		Not implemented	Ignored branching	Simplified
Add	RandomNumber	Call	GoToLabel	AddTyped
And	Return	CastOp	GotoFrame	EqualsTyped
AsciiToChar	SetProperty	CloneSprite	GotoFrame2	GetMember
BitAnd	SetVariable	Delete	If	GetTime
BitLShift	StackSwap	Delete2	Jump	LessTyped
BitOr	StoreRegister	EndDrag	NextFrame	SetMember
BitRShift	StrictEquals	Enumerate	Play	ToNumber
BitURShift	StringAdd	Enumerate2	PreviousFrame	ToString
BitXor	StringEquals	Extends	Stop	
CallFunction	StringExtract	ImplementsOp	WaitForFrame	
CallMethod	StringGreater	InitArray	WaitForFrame2	
CharToAscii	StringLength	InitObject		
ConstantPool	StringLess	InstanceOf		
Decrement	Subtract	MultiByteAsciiToChar		
DefineFunction	Trace	MultiByteCharToAscii		
DefineFunction2	TypeOf	MultiByteStringExtract		
DefineLocal		MultiByteStringLength		
DefineLocal2		NewMethod		
Divide		NewObject		
Equals		PushDuplicate		
GetProperty		RemoveSprite		
GetURL		SetTarget		
GetURL2		SetTarget2		
GetVariable		StartDrag		
Greater		StopSound		
Increment		StrictMode		
IntegralPart		TargetPath		
Less		Throw		
Modulo		ToggleQuality		
Multiply		Try		
Not		With		
Or				
Pop				
Push				

Table 4.1: Implementation status of actions. The Action prefix is omitted on all names.

A special class of actions are the actions that define functions, represented by `FunctionAction`. These actions contain, in their data payload, the name of the function, a specification of the function's arguments and the function body as a list of actions. Interpreting the `FunctionAction` defines the function. The `FunctionAction` implements function calls on top of the Interpreter in the method `call()`, including creating the local scope and pushing and popping arguments on the stack. The function is also used by the Interpreter for storing functions and tracked for the coverage interpretation step.

Most of the classes of the Actions module can be seen in Figure 4.4. Similar tags have been stacked on top of each other. External dependencies or associations are drawn in gray.

4.2.5 Example Interpretation

To illustrate the interpretation steps of the link extractor an example file is presented here. A screenshot of the example file can be seen in Figure 4.5. The Flash file presents an animated menu which expands on hover and redirects the user when clicked. The file is mostly a long repetitive sequence of tags defining the shapes of the menu structure. There are two interesting ActionScript snippets, both executed as the menu is constructed.

The first snippet is enclosed in a `DoAction` tag. The actions in this tag are executed in the Flash runtime when the timeline reaches the frame this tag is part of. The link extractor executes these actions when the enclosing tag is reached during the interpretation step of the depth first nested list traversal.

```
ActionConstantPool
...
ActionPush pool#1
> push string<_global>
ActionGetVariable
> pop string<_global>
> push object<globalScope>
ActionPush: pool#25, pool#26
> push string<MicroURL>
> push string<http://www.micrometscientific.co.za/>
ActionSetMember
> pop string<http://www.micrometscientific.co.za/>
> pop string<MicroURL>
> pop object<globalScope>
```

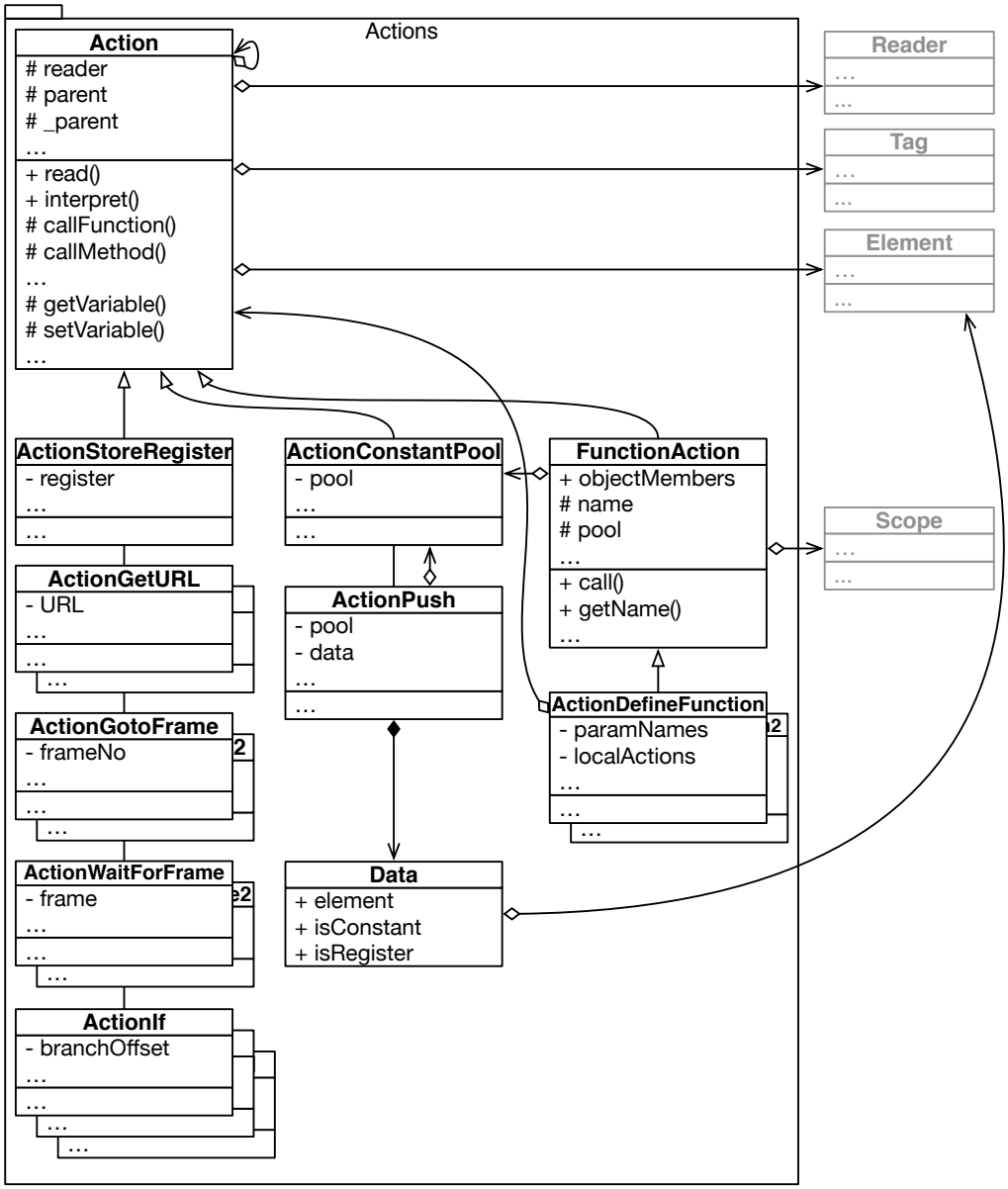

Figure 4.4: UML class diagram of the Actions component.

...

The first snippet defines a constant pool early on in the action list. A few uninteresting actions are not shown. The payload of the following `ActionPush` actions only contain indices for accessing strings in the pool. The strings are used to first access the global scope using the special `_global` identifier and later to save the variable `MicroURL` in the global scope. The variable contains the base URL for the website of the menu.

The second interesting snippet is the actions list for a `DefineButton2` tag. The button is part of the menu structure. In the Flash runtime environment these actions are executed when a user clicks the button. The link extractor executes these actions as soon as the tag is reach during the timeline interpretation step.

```
ActionPush payload
> push: string<_global>
ActionGetVariable
> pop string<_global>
> push object<globalScope>
ActionPush payload
> push string<MicroURL>
ActionGetMember
> pop string<MicroURL>
> pop object<globalScope>
> push string<http://www.micrometscientific.co.za/>
ActionPush payload
> push string<products.aspx?CategoryId=1&SubcategoryId=10>
ActionAddTyped
> pop string<products.aspx?CategoryId=1&SubcategoryId=10>
> pop string<http://www.micrometscientific.co.za/>
> push string<http://www.micrometscientific.co.za/products.
    aspx?CategoryId=1&SubcategoryId=10>
ActionPush payload
> push string<>
ActionGetURL2
> pop string<>
> pop string<http://www.micrometscientific.co.za/products.aspx
    ?CategoryId=1&SubcategoryId=10>
```

The second snippet doesn't use a constant pool. Instead the data that is pushed is stored in the payload of `ActionPush`. appends a path component to the previously saved base URL. The resulting URL is passed to the link

Figure 4.5: Screenshot of example Flash file. The mouse pointer is hovering above "Products".

extractor via `ActionGetURL2`. This action takes two parameters on the stack: target and URL.

Chapter 5

Evaluation

This chapter presents the evaluation of the link extractor and details possible improvements.

5.1 Methodology

This section describes how the link extractor is evaluated.

5.1.1 Page Selection

The link extractor is evaluated using pages from the Picsearch page repository. The Picsearch page repository contains the pages used for the Picsearch image search engine. The search index contains over three billion images [19] and consequently several hundred million pages. Randomly sampling these pages gives a good approximation of how the link extractor performs with real world data. The page data in the used repository was last retrieved or updated in the middle of 2014.

The HTML of the pages is searched to find embedded Flash files. The search uses a handful of regular expressions targeting the embedding techniques presented in Section 3.3. The pages are searched offline using the page repository. Once a possible Flash link is found it is downloaded and inspected. Only one Flash file per page is downloaded. Only pages containing embedded Flash files of version eight or below are selected. About 3500 pages and Flash file pairs are selected at random.

5.1.2 Flash

The link extractor is run on each Flash file. It performs link extraction in isolation as described in Chapter 4. The extractor emits all links it finds, counted and categorized by source and extraction phase. Any problem encountered during parsing or interpretation is also emitted.

5.1.3 Links

The extracted links are interesting primarily in two ways. First, web search engine coverage improvement. A link improves the coverage if it is not previously seen. Second, page ranking. All found links including previously seen links affect the page rank of a page. It is therefore interesting to count both the number of total links and the number of unique links.

To know whether a link is not previously seen is a difficult task because of the number of pages on the web. A simplification is to treat each page in isolation. Measuring the coverage improvement then simply involves comparing the extracted links with the links on the page where the Flash file is embedded. To find all links on the page all of the HTML anchor tags' href attributes are extracted. This is done using the Python library "Beautiful Soup" [22].

The search coverage improvement is measured in both absolute numbers and percent improvement. The percentage is calculated by comparing the number of new unique links from the link extractor with the number of unique links found in the HTML of the page. The percentage is capped to 100%. With E as the set of link extractor links and H as the set of HTML links:

$$\left\lceil \frac{|E \setminus H|}{|H|} \right\rceil \%$$

5.2 Results

This section presents the results of the evaluation.

5.2.1 Parsing

Out of the test set of 3567 Flash files the link extractor failed parsing 1.9% of the files. A failure in the parsing stage results in zero links extracted. Another 11% of the files were parsed but with some errors. Parsing errors do not necessarily result in zero links extracted and the file will continue on to the interpreter stages. 87% of all files were parsed without any errors.

Parse failures	66	1.9%
Parse errors	391	11%
No parse errors	3110	87%
No links found	2283	64%
Links found	1218	34%
Total	3567	

Table 5.1: File parsing success rates.

	Link Sources		Link Count		Unique Link Count	
Parsing	1075	82%	4261	83%	3494	83%
On Load	2	0.2%	5	0.1%	4	0.1%
Timeline	172	13%	734	14%	605	14%
Coverage	66	5.0%	153	3.0%	109	2.6%
Total	1315		5153		4212	

Table 5.2: Link source distribution.

Out of the test set the link extractor found links in 34% of the files. Taking into account the files that failed parsing this results in 64% of the files being successfully processed without finding any links. The parsing results are summarized in Table 5.1.

5.2.2 Link Extraction

As described in Section 4.1 the link extractor works in stages. All stages except for the first interpretation stage (initialization) generated links. The second interpretation stage (on load) only contained two files with five links. Both of these stages can be dismissed as uninteresting.

Of the files classified as containing links 82% contained links found during parsing. As mentioned in Section 3.5 these links are static and were found before any interpretation was performed. The static links made up 83% of the total number of found links. The links found during interpretation were distributed almost five to one between the timeline and coverage interpretation stages. The distribution of sources for the extracted links are summarized in Table 5.2.

It can be noted that the links found during coverage interpretation contained more duplicates than the links found during parsing. The relative frequency was about 50% higher. This is likely a result of repeated calls to the same function. Multiple calls are allowed because differing call parameters or global state could generate different links.

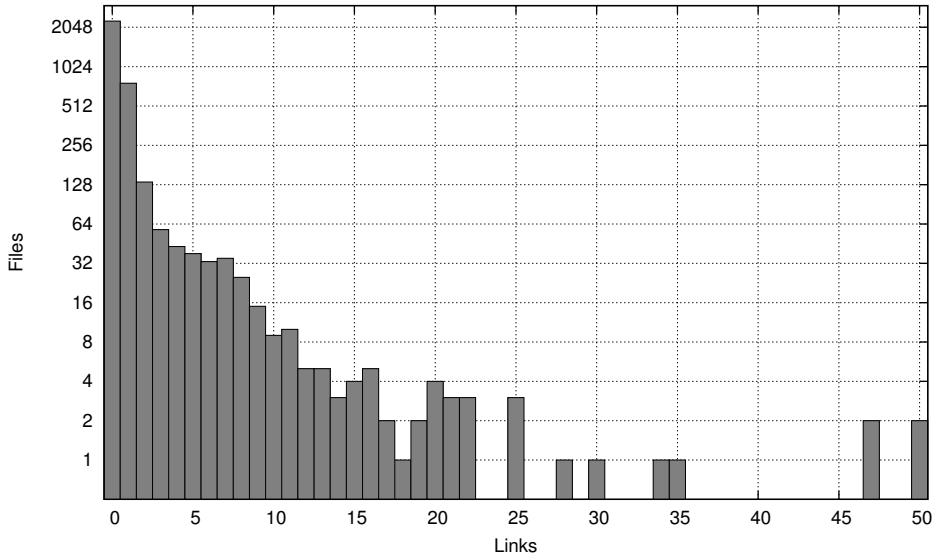

Figure 5.1: Distribution of extracted links per file.

On average 1.4 links were found per successfully processed file. Only counting unique links the average was 1.2 links per file. The distribution of link count per file can be seen in Figure 5.1. The most common number of found links is one. The frequency distribution of found links reduces logarithmically between 2 and 15 to flatten out and reach zero at 20. There are a few outliers with more than twenty links. The maximum number of found links in one file was 200. A complete table is available in Appendix B.

Extracted links can be false positives. For example links constructed using undefined or null data where a string was expected. The link extractor converts this data to either "null" or "undefined" instead of raising exceptions. By looking for these string in links from GetURL2 the number of false positives can be estimated. The estimated false positive percentage for timeline interpretation is 12.7% and for coverage interpretation 33.9%. The high relative number of false positives for the coverage interpretation could be a result of calling not visited functions with all null parameter or reaching dead code.

The total runtime of the link extractor for the entire test set was 1m 50.0s. The average and median execution time per file were 30.9ms and 11.0ms respectively. Minimum and maximum time per file were 5.0ms and 8.61s. On average about half of the processing time was spent parsing and

Minimum	5.0ms
Median	11.0ms
Average	30.9ms
Maximum	8.61s
Total runtime	1m 50.0s

Table 5.3: Runtime performance of the link extractor.

Parsing	52%
On Load	0.70%
Timeline	11%
Coverage	36%

Table 5.4: Average percent time spent in each link extraction stage.

half was spent interpreting. For the slowest file about 95% of the runtime was spent in coverage interpretation. The test was run on commodity hardware with a 2.4GHz Intel Core 2 Duo "Penryn" processor and solid-state storage. The runtime performance is summarized in Table 5.3 and Table 5.4.

5.2.3 Search Coverage

In absolute numbers the simple HTML parsing found 145 642 unique links. Using the HMTL links as a base for uniqueness comparison the link extractor found an additional 3664 unique links. This is a total link count increase of 2.5%. As stated in Section 5.1 the link uniqueness is only checked per page. The link extractor found 483 links that were already found in the HTML link set.

The averaged coverage improvement in percent across the entire test set is 9.8%. The most common case excluding no improvement is 100% improvement. This is when the page contained fewer links than the link extractor found. The 100% coverage improvement occurred on 7.2% of the processed pages. There is also a small cluster around 1% improvement stretching up to 20%. A histogram can be seen in Figure 5.2. A complete table is available in Appendix B.

5.3 Suggestions for Improvement

During file selection almost one in four Flash files were discarded because of the version requirement. The number of found links and the coverage

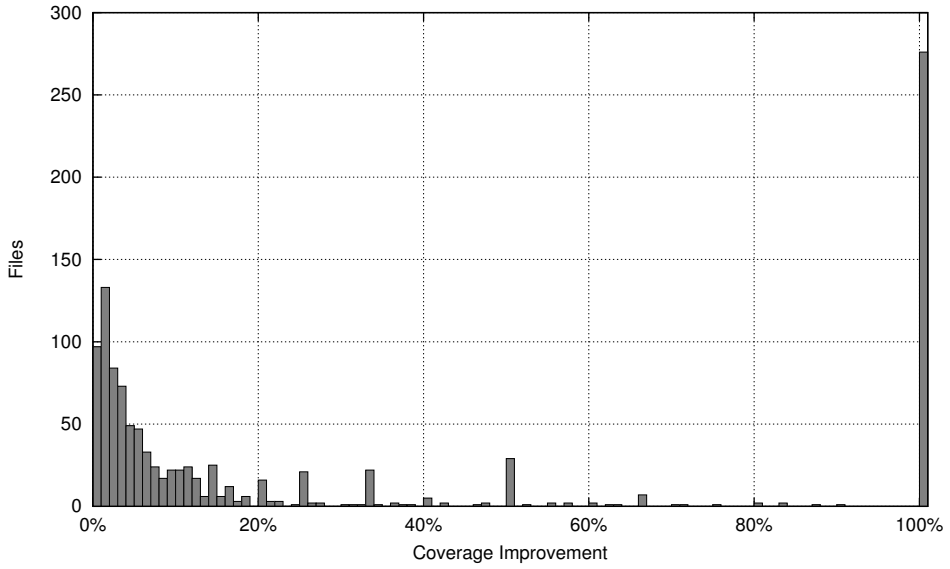

Figure 5.2: Histogram of percent coverage improvement, excluding the files with zero improvement.

improvement would be better if support for versions above 8 was added. However, this requirement did not affect the numbers in the previous section. The new virtual machine used for version 9 and above is available as a joint open source project from Mozilla and Adobe [17]. It could be used as a reference for building a link extractor for version 9 and above.

All of the parser errors and the few parses failures from the test set in the previous section should be investigated. A smaller set of a few hundred files were used during development.

The dynamic links extracted from GetURL2 can have query parameters added to the URL by the Flash runtime. There is a separate setting for enabling this feature in the action payload. When it is enabled all of the current local variables are either appended to the URL as query parameters or sent using an HTTP POST in the body of the request. This is not implemented in the link extractor. Implementing it should improve the quality of the extracted links and might result in a higher unique link count.

Variables can be passed to the Flash files from the embedding page. Either as query parameters on the URL of the Flash file or as so called flashvars in the HTML <object> tag. These values are made available in the root scope of the virtual machine. This variable passing is not implemented.

Implementing it should improve the quality and count of the extracted links.

The analysis of the file survey in Section 3.8 suggested that the dynamic links should make a significant contribution to the found links. During interpretation a majority of the dynamic link sources did not produce usable links. This implies that there is room for improvement in the interpreter parts of the link extractor.

The interpretation strategy used for interpretation was chosen both for limiting the runtime of the link extractor and ease of development. This strategy should be re-evaluated with the goal of improving the number of successfully extracted links. Branching or jumping could be implemented in a new interpreter stage with the aim of increasing the ratio of interpreted bytecode. This would be similar to the current final interpretation step of visiting all previously not visited functions.

The maximum depth level of function calls can be increased. Currently it is limited to a depth of five. Five was chosen as a compromise between the number of found links and the runtime on the test set. Setting it higher increases the runtime of a handful of files by an order of magnitude.

Commonly used standard libraries for ActionScript 1 and 2 could be implemented. Currently there are no libraries implemented. Investigating how many missing lookups there are should be done to determine if this is worthwhile. The implemented libraries or parts of libraries can then be added to the root scope chain.

The prototype inheritance model is not implemented. The current implementation has a shared prototype for all objects. Adding prototype inheritance would be possible with the implementation of parts of the standard library.

Many other ActionScript features are not implemented: exceptions, multibyte strings, full object model, modifying the timeline.

Chapter 6

Related Work

Two related research topics are hidden web crawling and Flash malware detection.

6.1 Client-side Hidden Web Crawling

The hidden web or deep web is usually defined as the part of the web that is dynamically generated. This dynamic generation can be done both at the server side and client-side. The hidden web is the part of the web that is not searchable using the conventional web search engines. Embedded Flash is one of the hurdles of crawling the client-side hidden web. Another similar but more researched hurdle is JavaScript.

Alvarez et al. [3] presents a technique for handling crawling of the client-side deep web. They utilize automated Internet Explorer web browsers to perform the crawl and link extraction. The page and any required JavaScript resources are retrieved and then rendered by these automated browsers, including script execution. The page's document object model (DOM) is inspected for links that are added to a crawl queue. JavaScript events are then triggered on interesting nodes to find more links. Before triggering events the current state of the DOM is saved so that each event can be triggered in isolation on an unmodified DOM. If an event triggers new objects or modification of existing objects a new search is performed on the altered objects. Any new links generated from this search are also extracted and added to the crawl queue.

The paper mentions that the automated browser can be extended to support executing Flash but will have difficulty interacting with the Flash runtime. They give the example of handling a Flash banner that automati-

cally redirects the user to another page.

The search approach used by Alvarez et al. differs from the approach used in the link extractor. The link extractor uses the same machine state without rolling back to the previous state after triggering events on buttons or calling functions in the coverage interpretation step.

Mesbah et al. [16] presents a similar technique. They have written a tool called Crawljax for crawling the client-side hidden web with a focus on single page JavaScript applications. Mesbah takes the same approach as Alvarez in inspecting the DOM and firing events. Both use automation of desktop web browsers. The main difference is the addition of treating all of the page's different user interface states as different pages. The purpose of Crawljax is not limited to crawling. It is also used for only generating the state graphs of web applications for visualization and testing purposes.

The paper presents runtime performance numbers for six different ajax pages on comparable hardware as used in Section 5.2. On the six different pages Crawljax finds between 16 and 148 states in between 14 and 701 seconds. States is what they use to describe separate "ajax pages" within the ajax single page application. On average they find 0.23 links per second. It is important to note when comparing Crawljax with the link extractor that Crawljax performs network requests whereas the link extractor runs offline. The link extractor found 38.3 links per second.

Hammer et al. [10] presents another similar technique. They use WebKit from the QT library [21] for getting access to a slimmed-down version of a desktop browser. This browser is used for analyzing user-submitted comments to newspaper articles. The goal is to get access to the textual representation of the comments. These comments almost always require JavaScript to be loaded and displayed on the page. The full page is fetched and rendered and then the viewport is scrolled down to trigger loading the comments.

All of the above mentioned papers present techniques that work in a similar manner. A desktop web browser is automated and either the rendered page or its document object model is analyzed. Using a desktop browser for crawling is problematic for bigger data sets because it is not as scalable as specialized web crawling software. The crawling in the two first papers is restricted to DOM analysis but the Flash runtime system is not part of the DOM. The crawling in the third paper is restricted to the textual representation of the page and the Flash is not rendered at all.

6.2 Flash Malware Detection

A related research topic is Flash malware detection. Multiple papers have been published on the subject.

Ford et al. [9] presents a tool for detecting malicious Flash advertisements. The tool is called OdoSwiff. They use a combination of static and dynamic analysis. The static analysis parses the tags and actions in the file to detect known malicious techniques. For example, out of bounds ActionScript jumps or malicious code hiding where image data should be stored. The dynamic analysis consists of executing the Flash file and tracing all of the executed actions. The open source Gnash Flash runtime [20] is used for execution and creating the trace. Gnash is limited to the same versions as the link extractor. The trace is then analyzed for malicious behavior.

Odoswiff is very strict in its classification. Any redirect that happens without user interaction results in the file being classified as malicious.

The static analysis step of OdoSwiff is similar to the parsing step of the link extractor. The dynamic analysis step is similar to the link extractor's timeline interpreting step. However, during this step the runtime is allowed full network access.

The OdoSwiff authors mention that the dynamic analysis step can take several minutes because of the extensive trace output. They also mention that the trace output can reach several gigabytes in size. The link extractor only emits the extracted links unless it is in debug mode.

Another interesting technique is their monitoring of the stack to find unused URLs. They describe that it is common for malicious Flash files to build up URLs on the stack that are not used because of for example failing date or country checks. The malicious file is in stealth mode and doesn't do anything malicious. The link extractor doesn't monitor the stack for potential URLs. Only the strings that are actually used as URLs are extracted.

Van Overveldt et al. [25] presents the evolution of Odoswiff: FlashDetect. It only supports Flash versions from 9 in contrast to the link extractor which supports versions under 9. Like the link extractor it is an offline file analyzer. It uses an instrumented version of the open source Lightspark Flash runtime for the dynamic analysis step. The trace generated is limited to only interesting events. The dynamic analysis step is stopped after a timer expires but there is no mention of actual runtime performance.

The authors mention that the Lightspark component can be unreliable and that they have implemented certain features redundantly, in both the static analysis step and in the analysis of the trace output from Lightspark.

The link extractor has no such redundancy as it has not been observed to fail to run its interpreter on any significant number of files.

Van Acker et al. [24] presents another malware detection tool. The tool that is called FlashOver automatically detects cross site scripting vulnerabilities in Flash files. The tool uses a commercial ActionScript decompiler for its static analysis. The static analysis results in a list of potential attack URLs. The dynamic analysis is performed by loading the Flash file from one of these attack URLs using the desktop web browser Firefox with the Flash runtime plugin installed. The browser is connected to an in-memory framebuffer and mouse emulation software. After the Flash file is loaded 10 000 mouse clicks are simulated. The clicks have a chance of triggering injected JavaScript code. The browser is then monitored for any execution of injected code.

The technique of randomly clicking on the rendered Flash file is not very sophisticated. In order to guarantee finding all clickable elements Flashover would have to exhaustively search the rendered user interface. For example, by systematically clicking everywhere on the canvas and for each observed change of the interface perform a new search. The technique of interpretation used by the link extractor will trigger all events on all clickable elements.

The dynamic analysis step of the FlashOver tool is very resource intensive. For the authors' test set of 8441 files they describe using 70 dual core computers in parallel.

The technique that is used for generating the list of possibly exploitable variables could be useful also for the link extractor. It could help the link extractor by preloading variables in the global scope with values from the page where the Flash file was embedded. The same Flash file embedded on different pages could then yield different URL sets.

Chapter 7

Conclusions

The Flash file format was introduced and files from the web were analyzed for which Flash features were in common use. The scalability needs of a web search engine were discussed. With this information a link extractor was designed and implemented that will fit into an existing crawling ecosystem of specialized and scalable tools.

The link extractor tool performed well. It was evaluated against a large test set of about 3500 Flash files. The successful parsing rates are high, 98%. Of the files processed a third contained links. The average runtime of the link extractor per file on commodity hardware was 31ms.

The goal of increasing the coverage of a web search engine was achieved. Compared to normal HTML crawling the link count increased by an estimated 2.5%. The averaged per page coverage increase was estimated to 9.8%.

A weakness is the interpretation stages of the link extractor. The numbers imply that there are more links available than the link extractor found. There are a number of potential improvements listed in Section 5.3 that require more research.

Appendix A

File Survey

ID	Name	Seen	ID	Name	Seen
0	End	100%	42	DefineTextFormat	0%
1	ShowFrame	97%	43	FrameLabel	22%
2	DefineShape	82%	45	SoundStreamHead2	9%
3	FreeCharacter	0%	46	DefineMorphShape	9%
4	PlaceObject	0%	48	DefineFont2	29%
5	RemoveObject	1%	49	GeneratorCommand	0%
6	DefineBits	10%	51	CharacterSet	0%
8	JPEGTables	10%	55	Unknown	3%
9	SetBackgroundColor	98%	56	Export	25%
10	DefineFont	15%	57	Import	0%
11	DefineText	40%	58	ProtectDebug	0%
12	DoAction	58%	59	DoInitAction	13%
13	DefineFontInfo	10%	60	DefineVideoStream	5%
14	DefineSound	11%	61	VideoFrame	2%
15	StartSound	6%	62	DefineFontInfo2	5%
17	DefineButton	2%	63	Unknown	0%
18	SoundStreamHead	2%	64	ProtectDebug2	0%
19	SoundStreamBlock	3%	65	ScriptLimits	3%
20	DefineBitsLossless	8%	66	SetTabIndex	0%
21	DefineBitsJPEG2	31%	69	FileAttributes	40%
22	DefineShape2	31%	70	PlaceObject3	0%
24	Protect	13%	71	Import2	0%
25	PathsArePostscript	5%	72	Unknown	0%
26	PlaceObject2	92%	73	DefineFontAlignZones	19%
28	RemoveObject2	56%	74	CSMTextSettings	10%
32	DefineShape3	38%	75	DefineFont3	19%
33	DefineText2	2%	76	Unknown	9%
34	DefineButton2	41%	77	Metadata	10%
35	DefineBitsJPEG3	15%	78	DefineScalingGrid	1%
36	DefineBitsLossless2	19%	82	Unknown	8%
37	DefineEditText	30%	83	DefineShape4	10%
39	DefineSprite	73%	84	DefineMorphShape3	0%
41	SerialNumber	6%			

Table A.1: File survey results of seen tags.

ID	Name	Seen	ID	Name	Seen
0	End	100%	65	Declare Local Variable	14%
2	Unknown	0%	66	Declare Array	18%
4	Next Frame	3%	67	Declare Object	19%
5	Previous Frame	2%	68	Type Of	12%
6	Play	37%	69	Get Target	3%
7	Stop	61%	70	Enumerate	0%
8	Toggle Quality	0%	71	Add typed	46%
9	Stop Sound	3%	72	Less Than typed	41%
10	Add	1%	73	Equal typed	45%
11	Subtract	41%	74	Number	20%
12	Multiply	44%	75	String	16%
13	Divide	42%	76	Duplicate	31%
14	Equal	0%	77	Swap	2%
15	Less Than	1%	78	Get Member	49%
16	Logical And	5%	79	Set Member	49%
17	Logical Or	2%	80	Increment	35%
18	Logical Not	53%	81	Decrement	22%
19	String Equal	2%	82	Call Method	54%
20	String Length	1%	83	New Method	16%
21	SubString	1%	84	Instance Of	6%
23	Pop	52%	85	Enumerate Object	18%
24	Integral Part	14%	95	Unknown	0%
28	Get Variable	62%	96	And	4%
29	Set Variable	49%	97	Or	2%
32	Set Target dynamic	1%	98	XOr	1%
33	Concatenate Strings	5%	99	Shift Left	1%
34	Get Property	15%	100	Shift Right	3%
35	Set Property	15%	101	Shift Right Unsigned	0%
36	Duplicate Sprite	6%	102	Strict Equal	18%
37	Remove Sprite	4%	103	Greater Than typed	34%
38	Trace	15%	104	String Greater Than typed	0%
39	Start Drag	3%	105	Extends	10%
40	Stop Drag	2%	110	Unknown	0%
41	String Less Than	0%	112	Unknown	0%
42	Throw	2%	120	Unknown	0%
43	Cast Object	4%	129	Goto Frame	34%
44	Implements	3%	131	Get URL	40%
45	FSCommand2	0%	135	Store Register	32%
46	Unknown	0%	136	Declare Dictionary	56%
48	Random	12%	138	Wait For Frame	2%
49	String Length multibyte	0%	139	Set Target	5%
50	Ord	0%	140	Goto Label	11%
51	Chr	0%	141	Wait For Frame dynamic	0%
52	Get Timer	14%	142	Declare Function2	24%
53	SubString multibyte	0%	143	Try	2%
55	Chr multibyte	0%	148	With	8%
58	Delete	22%	150	Push Data	65%
59	Delete All	8%	153	Branch Always	44%
60	Set Local Variable	32%	154	Get URL2	28%
61	Call Function	38%	155	Declare Function	37%
62	Return	28%	157	Branch If True	54%
63	Modulo	15%	158	Call Frame	0%
64	New	38%	159	Goto Expression	7%

Table A.2: File survey results of seen actions.

Appendix B

Link Extractor

Files	Links
0	2283
1	765
2	134
3	58
4	43
5	38
6	33
7	35
8	25
9	15
10	9
11	10
12	5
13	5
14	3
15	4
16	5
17	2
18	1
19	2
20	4
21	3
22	3
25	3
28	1
30	1
34	1
35	1
47	2
50	2
72	1
79	1
141	1
171	1
200	1

Table B.1: Distribution of extracted links per file.

2460	0.00%	1	0.94%	5	1.92%	3	4.92%	1	15.00%
1	0.08%	2	0.95%	1	1.94%	8	5.00%	1	15.15%
1	0.15%	1	0.97%	5	1.96%	1	5.04%	1	15.38%
1	0.16%	1	0.98%	7	2.00%	4	5.26%	3	15.49%
1	0.17%	3	0.99%	1	2.01%	13	5.56%	10	16.67%
1	0.18%	1	1.00%	3	2.04%	1	5.71%	1	17.24%
3	0.25%	1	1.01%	3	2.08%	1	5.77%	1	17.65%
1	0.26%	1	1.03%	2	2.13%	1	5.83%	4	18.18%
1	0.28%	2	1.04%	3	2.17%	11	5.88%	1	18.75%
1	0.29%	2	1.05%	2	2.22%	2	5.97%	16	20.00%
1	0.30%	1	1.08%	7	2.27%	4	6.06%	1	21.21%
3	0.31%	1	1.09%	1	2.30%	8	6.25%	1	21.43%
1	0.32%	2	1.11%	6	2.33%	1	6.45%	2	22.22%
3	0.36%	2	1.14%	9	2.38%	1	6.52%	20	25.00%
1	0.37%	1	1.15%	1	2.41%	2	6.56%	1	26.09%
2	0.38%	1	1.16%	4	2.44%	14	6.67%	1	27.27%
1	0.40%	2	1.18%	1	2.47%	1	6.90%	1	27.78%
2	0.43%	3	1.23%	5	2.50%	10	7.14%	1	31.58%
1	0.44%	1	1.25%	1	2.56%	1	7.41%	1	32.14%
2	0.45%	2	1.27%	1	2.59%	1	7.58%	22	33.33%
2	0.46%	3	1.30%	3	2.63%	10	7.69%	1	34.21%
1	0.48%	2	1.32%	2	2.70%	1	7.89%	2	36.21%
1	0.49%	2	1.33%	2	2.78%	3	8.20%	1	37.50%
2	0.50%	2	1.35%	1	2.80%	10	8.33%	1	38.46%
5	0.51%	4	1.39%	6	2.86%	3	8.70%	5	40.00%
1	0.53%	1	1.41%	7	2.94%	1	8.93%	2	42.86%
1	0.55%	2	1.43%	1	2.99%	14	9.09%	1	46.15%
2	0.56%	1	1.44%	12	3.03%	1	9.38%	1	47.37%
1	0.57%	4	1.45%	1	3.08%	2	9.52%	1	47.39%
1	0.58%	1	1.46%	7	3.12%	1	9.80%	26	50.00%
1	0.59%	1	1.47%	1	3.17%	1	9.84%	1	52.94%
2	0.60%	3	1.49%	3	3.23%	17	10.00%	1	55.56%
1	0.61%	1	1.50%	2	3.28%	1	10.26%	2	57.14%
2	0.62%	6	1.52%	4	3.33%	1	10.39%	2	60.00%
2	0.63%	3	1.54%	3	3.39%	1	10.42%	1	62.50%
1	0.64%	3	1.56%	7	3.45%	1	10.91%	6	66.67%
2	0.66%	2	1.59%	9	3.57%	15	11.11%	1	70.00%
3	0.70%	1	1.60%	6	3.70%	2	11.54%	1	71.43%
4	0.71%	2	1.61%	1	3.73%	4	11.76%	1	75.00%
2	0.72%	7	1.64%	8	3.85%	1	11.86%	2	80.00%
1	0.74%	7	1.67%	1	3.97%	1	12.00%	2	83.33%
1	0.75%	5	1.69%	8	4.00%	14	12.50%	1	87.50%
1	0.79%	4	1.72%	7	4.17%	1	13.11%	1	90.91%
1	0.80%	2	1.75%	8	4.35%	1	13.33%	255	100.00%
2	0.82%	4	1.79%	1	4.44%	1	13.56%		
1	0.84%	1	1.80%	8	4.55%	1	13.64%		
3	0.85%	1	1.81%	2	4.65%	1	13.79%		
1	0.88%	6	1.82%	1	4.69%	1	13.95%		
1	0.89%	2	1.85%	3	4.76%	23	14.29%		
1	0.93%	4	1.89%	2	4.88%	1	14.75%		

Table B.2: Distribution of percentage coverage improvement.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1st edition, 1986.

[2] Jesse Alpert and Nissan Hajaj. Google Official Blog: We knew the web was big... `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`, 2008. Accessed: 2015-03-07.

[3] Manuel Álvarez, Alberto Pan, Juan Raposo, and Ángel Viña. Crawling the client-side hidden web. In *ICWI*, pages 1179–1182, 2004.

[4] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The Google cluster architecture. *Micro, Ieee*, 23(2):22–28, 2003.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[6] Alice Cheng and Eric Friedman. Manipulability of PageRank under sybil strategies. In *First Workshop on the Economics of Networked Systems (NetEcon06)*, pages 75–82. ACM, 2006.

[7] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, The Internet Engineering Task Force, May 1996.

[8] L. Peter Deutsch and Jean-Loup Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950, The Internet Engineering Task Force, May 1996.

[9] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious flash advertisements. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 363–372. IEEE, 2009.

[10] Hugo Hammer, Alfred Bratterud, and Siri Fagernes. Crawling JavaScript websites using WebKit–with application to analysis of hate speech in online discussions. *Norsk informatikkonferanse (NIK)*, 2013.

[11] David Hawking. How things work: Web search engines (part 1). *IEEE Computer*, pages 86–88, June 2006.

[12] David Hawking. How things work: Web search engines (part 2). *IEEE Computer*, pages 88–90, August 2006.

[13] Adobe Systems Inc. Adobe Flash Platform Statistics. `http://www.adobe.com/mena_en/products/flashplatformruntimes/statistics.html`. Accessed: 2015-03-07.

[14] Adobe Systems Incorporated. SWF File Format Specification (version 19). `http://www.adobe.com/devnet/swf.html`, 2012. Accessed: 2015-03-07.

[15] M. Koster. A Method for Web Robots Control. Internet-Draft draft-koster-robots-00.txt, IETF Secretariat, December 1996.

[16] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

[17] Mozilla Tamarin. `https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Tamarin`. Accessed: 2015-03-07.

[18] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999.

[19] Picsearch Image Search. `http://www.picsearch.com/`, 2014. Search engine portal.

[20] GNU Project. Gnash - the gnu flash player. `http://savannah.gnu.org/projects/gnash/`. Accessed: 2015-03-07.

[21] QT Project. Webkit in QT. `http://qt-project.org/doc/qt-4.8/qtwebkit.html`. Accessed: 2015-03-07.

[22] Leonard Richardson. Beautiful Soup Documentation. `http://www.crummy.com/software/BeautifulSoup/bs4/doc/`, 2004–2014. Accessed: 2015-03-07.

[23] Jennifer Niederst Robbins. *Learning Web Design*. O'Reilly Media, fourth edition edition, 2012.

[24] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 12–13. ACM, 2012.

[25] Timon Van Overveldt, Christopher Kruegel, and Giovanni Vigna. FlashDetect: ActionScript 3 malware detection. In *Research in Attacks, Intrusions, and Defenses*, pages 274–293. Springer, 2012.