Kraaler: A User-Perspective Web Crawler

Thomas Kobber Panum*, René Rydhof Hansen†, Jens Myrup Pedersen*
*Department of Electronic Systems, †Department of Computer Science
Aalborg University, Denmark
Emails: tkp@es.aau.dk, rrh@cs.aau.dk, jens@es.aau.dk

***Abstract*—Adaption of technologies being used on the web is changing frequently, requiring applications that interact with the web to continuously change their ability to parse it. This has led most web crawlers to either inherent simplistic parsing capabilities, differentiating from web browsers, or use a web browser with high-level interactions that restricts observable information. We introduce Kraaler, an open source universal web crawler that uses the Chrome Debugging Protocol, enabling the use of the Blink browser engine for parsing, while obtaining protocol-level information. The crawler stores information in a database and on the file system and the implementation has been evaluated in a predictable environment to ensure correctness in the collected data. Additionally, it has been evaluated in a real-world scenario, demonstrating the impact of the parsing capabilities for data collection.**

I. Introduction

Information on the web has reached magnitudes and change at a rate which are infeasible for humans to structure and manage. This has motivated the use of machines for automated solutions capable of interpreting content on the web and represent the interpretation in a structured form. These types of solutions are referred to as web crawlers and consist of applications that systematically visit web servers and parse their content [1]. Web crawlers have been an active research topic within recent decades and were initially motivated by the need for making the web searchable [2]. However, they are now being used in a wider variety of applications, such as gathering data sets for statistical modeling [3], [4].

The typical retrieval process of web crawlers involves exploiting the interlinked structure of Hypertext Markup Language (HTML) documents being collected, by parsing the collected documents for hyperlinks to other documents and consider the found links as documents to be collected. This process stems from a time when web pages included fewer content types and relied less on externally linked dependencies.

However, the number of technologies and content types used on the web has increased drastically over recent decades. This has led to the end user client, the web browser, to become a complex application and include parsers for a variety of content types. The size of the code bases of the three most widely adopted parsers among web browsers (Chromium Blink [5], WebKit [6], Mozilla Gecko [7]) highlight this complexity, and are respectively: 2.9M, 13.9M, and 20.7M lines of code, as of April 2019. This complexity leaves most web crawlers unable to implement parsing capabilities to the same extent, causing a deficit in method web applications are being crawled compared to the user's perspective. This deficit stems from the fact that existing web crawlers do not parse the crawled content to the same extent as a typical browser engine would. Examples of crawlers using web browsers have been seen, but their interaction with the web browsers leaves them unable to access detailed information about the underlying request behavior, e.g. usage of the HTTP protocol and subsequent requests. This leads to information being lost or unavailable for analysis and is a continuous problem as the capabilities of web browsers change over time.

In order to be able to enhance information gained throughout a crawling process, and have it represent the user's perspective, we introduce the universal web crawler, Kraaler. It is a user-perspective web crawler that uses the browser engine of Google Chrome (Chrome), Blink [5], through the use of the Chrome Debugging Protocol (CDP) while obtaining information about parsing and HTTP usage. The contributions of our work can be summarized as:

- Demonstrate a new method for user-perspective web crawling while observing detailed protocol information.
- Present a data structure containing this information while making it efficiently available for behavioral analysis of web applications, such as phishing detection.
- Provide an open-source and extendable implementation.

The implemented crawler is evaluated by exposing it to a predictable environment and a real-world environment, respectively. Each environment provides the ability to validate the correctness of the obtained data and demonstrate the impact of the web browser's parser in a crawling setting. Lastly, examples of the applicability of obtained data for behavioral analysis of web applications is shown.

II. Related Work

Information hosted on web servers is accessed through the Hypertext Transfer Protocol (HTTP) or its encrypted variant Hypertext Transfer Protocol Secure (HTTPS). End users typically use these protocols on a high-level through their own client, the web browser. Machine-based approaches, in the form of web crawlers, tend to typically interact with the protocol directly. Browsers abstract from the numerous underlying requests being sent back and forth between the browser and the web server, when a user interacts with a web page. The order, and to which extent these requests are being sent, are determined by the browser's parsing component, the browser engine.

978-3-903176-17-1 / © 2019 IFIP

Browser engines contained within web browsers impact and define the capabilities of the applications hosted on the internet. They thereby serve both as a delimiter and enabler of technologies used for web applications and affect their respective usage. Namely, the programming language JavaScript was designed to be used for web applications and is now considered one of the most widely adopted programming languages [8], [9].

Two common web browsers, Chrome and Mozilla Firefox (with respective market shares of $\approx 70\%$ and $\approx 10\%$ [10]), are using their own respective browser engines: Blink (a fork of WebKit) and Gecko. These engines are able to interpret and display numerous types of content, markup languages, and programming languages. This ability has developed over time and continues to do so, as the desire for richer web applications keeps persisting.

Web crawlers are applications for systematically gathering information from web servers, and have been an active research topic for decades. The research was initiated by the Mercator web crawler, which went on to become the web crawler of the Alta Vista search engine [2]. Following this, Kumar et al. has surveyed existing work and proposes a five type categorization of web crawlers: Preferential crawlers, hidden web crawlers, mobile crawlers, continuous crawlers, and universal crawlers [1].

Preferential crawlers are conditioning their crawling behavior, such as restricting only gathering from a subset of sources or only gather selective information.

Hidden (or sometimes referred to as *Deep*) web crawlers focus on crawling information that cannot be obtained by just following hyperlinks. Examples of this are web applications that use dynamic input for presenting results, such as search engines that require a search query in order to present a list of results. CrawlJax is a hidden web crawler capable of crawling web applications that rely on JavaScript for creating user interactions [11]. In order to crawl such web applications, PhantomJS is used for instrumenting a web browser to programmatically perform user actions within the browser [12].

Mobile web crawlers constitute a subset of crawlers that use an experimental method of crawling proposed by [13]. This method seeks out to avoid the iterative crawling behavior, of obtaining information across multiple requests for one source, by expecting remote web servers to be able to receive a data specification. The received data specification is then used to initiate a stream of actions locally on the remote server, in order to reduce bandwidth usage in the crawling process.

Continuous crawlers constitute a subset of crawlers that addresses the problem of prioritizing crawling targets, in the setting of restricted resources and crawling targets continuously changing their content.

Universal crawlers are the counterpart to the previously described crawlers, as they are designed for a broader and less specific use case. They are designed to visit any type of content or hyperlink they observe and repeat this process continuously. An example of a universal web crawler is BUbiNG, an open source high-performance distributed web crawler that performs HTML parsing and can store results on distributed storage systems [14]. Apache Nutch is another universal web crawler that has a modular design and has been actively developed for more than a decade. The modular design of Apache Nutch has led researchers to use it as a base, and extend it to new types of web crawlers [15].

A subset of web crawlers set out to extract exact information in a structured manner from a web application, this method is known as *web scraping*. Scrapy is a widely popular web scraping framework developed in the programming language Python [16]. The framework requires the user to be familiar with Python in order to specify which information to extract and the following method. An alternative solution is Portia, a web scraping framework built on Scrapy requiring no competences in programming [17]. The user clicks on information on a web page, and the framework attempts to extrapolate the underlying HTML template, defining a strategy for crawling the desired information. This extrapolation can, however, lead to incorrect identification of the underlying HTML structure, leading to incorrect or false information being extracted.

Kraaler is a universal web crawler that uses the parser of a web browser for interpreting web applications, allowing it to observe HTTP requests initiated by the HTML parser, the JavaScript interpreter, and more. A similar abstract design has been patented by Kraft et al., which describes the use a web browser for gathering dynamic content of web applications and utilize for optical character recognition for interpreting text in images [18]. However, certain design details are undocumented and, to our knowledge, there exists no open source implementation of the described design nor a specification of the data it collects.

III. User-Perspective Crawling

HTTP is a protocol based upon requests that are answered by responses, and when a user enters a URL in their browser, a request of the method `GET` with a couple of key-value pairs in the header is sent towards the host specified in the URL. Most of the existing universal crawlers base their crawling behavior on this, so given a set of URLs, the crawler will perform `GET` requests towards a given URL and retrieve the body of the corresponding HTTP response. This response body, which is often assumed to contain HTML, is typically analyzed for the presence of hyperlinks or URLs that are then added to the pool of known URLs.

Modern web applications consist of multiple content types, spread across multiple URLs referenced to by an HTML document, acting as dependencies for the web application. The browser is expected to visit the HTML document first, and then a process of parsing it starts, leading the browser to perform asynchronous requests for resources that the document references. The parsing process repeats for every response that is received by the browser, creating a recursive parsing process, causing the browser to perform a series of subsequent requests from the initial request of the document.

This recursive request behavior can be seen as an interlinked dependency graph of distributed resources, rooting at an initial

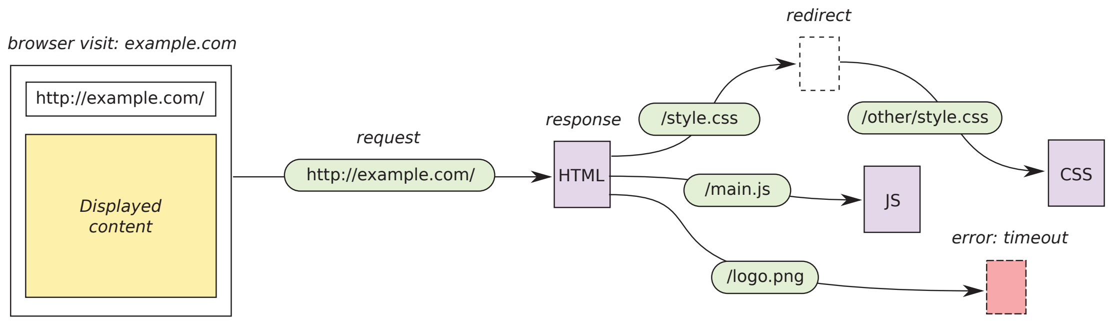

Fig. 1. Series of subsequent requests being performed by a web browser, when visiting a page.

document, as illustrated in Figure 1. The initial document is typically an HTML document and yields the browser to create a Document Object Model (DOM), which is an application programming interface for client-side programming languages to manipulate the received HTML and display the parsed model to the user [19]. Concretely, this allows modern web applications to use JavaScript to manipulate the initial HTML tree of the received document. This enables additional methods of navigating and updating the information of web applications without the user having to navigate to other HTML documents [11]. This is accomplished by allowing JavaScript to programmatically perform asynchronous HTTP requests, Asynchronous JavaScript and XML (AJAX), in order to either send or retrieve information following manipulation of the DOM [20].

These features, in addition to other capabilities of modern browsers not described in this article, have made it increasingly difficult to ensure the features are only available in contexts that meet a minimum security level [21]. Following this, the specification of Secure Contexts was introduced, which is a method for controlling the security level of actions by the browser on behalf of the web application's content. This means that certain aspects of the actions performed by the browser are conducted in an isolated sandbox environment, and certain actions can be restricted. As an example, a document served over a secure and encrypted connection is not allowed to reference other resources served using an insecure and unencrypted protocol. Parsing content of the web applications is an ever-changing process that evolves over time, causing modern browsers and their underlying browser engines to become applications spanning millions of lines of code. The implementation of new browsers engines for parsing content is, therefore, a costly and often infeasible process in the design of a crawler.

Historically, this left designers with the choice of either partial parsing capabilities, or to use high-level instrumentation of a web browser. High-level instrumentation of a web browser typically involves using a web driver, such as Selenium Webdriver, for programmatically controlling a subset of user-based features available in the browser [22]. These features typically include navigating the browser to a certain URL or interacting with the JavaScript shell.

An alternative to the Selenium solution is PhantomJS, which is a headless browser allowing for more information to be extracted from the underlying browser engine [12], [11]. In comparison, it allows for extracting request and response information from HTTP. The project has, however, been suspended, leaving the underlying engine to become obsolete from modern standards of web browsers.

In the context of Kraaler, the web browser Chrome is used as an external component for performing the HTTP requests and parsing of their respective responses during crawling. This choice stems from the fact that Chrome provides a remotely accessible application interface to its underlying browser engine, Blink, denoted Chrome Debugging Protocol (CDP) [23]. CDP allows for reading some of the data structures present in the browser engine during runtime in a structured form, such as detailed information about requests and responses, and sending instructions to the browser. Thus enabling the use of the complex parser contained within the browser engine without missing information contained within the HTTP protocol and other types of low-level information contained within the engine, which is difficult or infeasible in other widely-used browsers.

Information in CDP is exchanged through a series of subject-based channels, e.g. the channel of networking is named `Network`, and subscribing to the same channel allows for receiving events being published on the channel. Throughout this article, events published by CDP will be following the notation of `<channel>.<event>`, so in the case of `Network.requestWillBeSent`, `Network` refers to the channel which sends `requestWillBeSent` events. These events are published to their channel as JSON objects containing information related to the event, e.g. `Network.requestWillBeSent` contains information about a request that Chrome is about to initiate. The entirety of events, across of all channels, can be seen as a structured log of the captured behavior in Chrome. Instructions pushed to channels follow the same notation, e.g. `Page.navigate` will navigate the active window to a certain URL contained in the payload. CDP defines the concept of a page, describing the situation of when the browser navigates to a URL using the address bar. As previously described, this will perform an HTTP request, for which the response is parsed and can lead to subsequent requests being executed as a background activity. Kraaler inherits the concept of page, as illustrated in Figure 1, and defines a page to include: a series of HTTP requests and responses, the appearance of the web application in the browser, the JavaScript shell of the given web application, and other information described in Section V. In addition to the concept of pages, the concept of action is introduced to group a request with its respective response, or the connection error returned by the browser, when trying to perform the request.

IV. Overview

Kraaler has been implemented in the programming language of Go. Go was chosen due to its native support for parallelization primitives and its ability to compile statically-linked binaries for multiple platforms. The code is open-source with a GNU GPLv3 license and accessible in a git repository, hosted on GitHub[1]. Running the implementation depends on Chrome or Docker being available in order to either use or install Chrome.

Internally, Kraaler consists of two components, controller and worker, that are responsible for interacting and orchestrating a set of external components, as illustrated in Figure 2. Controller is responsible for communicating and orchestrating the parallel crawling process, conducted by a pool of workers while publishing their results to the external data stores. Each worker is responsible for conducting crawling tasks and orchestrate their individual instance of Chrome.

A. Controller

The primary role of the controller is to maintain the parallel crawling process. The controller will continuously push tasks to the next available worker in the pool and keep the set of workers occupied with tasks until no new task is available. Tasks become available as a sampler continuously samples URLs from a set of known URLs. Initiating a crawling process thereby requires the set of known URLs to be populated with some amount of URLs. Populating this set is done by parsing a set of domains or URLs to the controller, as it is being instantiated.

A sampler is a module for containing the mechanism for prioritizing which, and when, known URLs should be transformed to tasks. In order for a sampler to conduct this prioritization, it is exposed to the current set of known URLs and timestamp of their most recent crawl, at the time of sampling. Under these conditions, users of Kraaler are capable of implementing new samplers that encapsulate a prioritization strategy suitable for their respective use cases. Currently, there are two samplers implemented in Kraaler, a uniform sampler, and a domain-pair sampler. As tasks are being completed, pages are returned from the pool of workers to the controller, and it starts two actions: storing the information of the page and pushing newly found URLs to the pool of known URLs.

A page's textual and numeric properties are by default stored in a relational database, while the byte-based properties are stored on the file system in a structured form and referenced in the database. Schemes and data structures of the relational database are covered in more detail in Section~V. The byte-based properties include response bodies and screenshots. For certain applications, storing all byte-based information can be too extensive and unnecessary. To address this, Kraaler allows for restricting information stored, through a set of modules that modify a page received from a worker before being stored. Currently, Kraaler only has one module, that deletes response bodies of a certain content type for a page's actions. This leads to the functionality of restricting byte-based information contained in the following data, as empty response bodies are not stored. An example of a use case for this module is to restrict response bodies stored to only be text-based by deleting response bodies for which their content type is not prefixed with `text`. Additional modules can be implemented by users of Kraaler, for further controlling and restricting information being stored in the resulting data set of a crawl.

For obtaining a continuous crawling process, URLs found in the response bodies are by default added to the pool of known URLs. However, if it is desired to have a discrete crawling process or filter certain URLs, it is possible to control which URLs are added to the pool by assigning a URL filter.

B. Worker

The worker component is responsible for controlling the life cycle of a single Chrome instance, while also interacting with its instance through CDP. Problems involved in controlling the life cycle of Chrome is covered in more detail in Section VI. The interaction spans across three CDP channels: `Page`, `Runtime`, and `Network`.

Using the `Page` channel, a worker will use `Page.navigate` for sending instructions to navigate the browser window to a specific URL. The worker will use the event `Page.domContentEventFired` for determining when the initial document has been parsed and loaded. When this event has been received, the worker attempts to capture one or more screenshots through the instruction `Page.captureScreenshot`, for which each capture is conducted a configurable amount of seconds after the DOM is loaded.

The `Runtime` channel is used for the event `Runtime.consoleAPICalled` in order to capture console output from the JavaScript shell of the given page, e.g. debugging messages used by web developers.

Most of the information obtained by Kraaler is from the `Network` channel, which covers a variety of network events. `Network.requestWillBeSent` is an event being published when a page is about to send a HTTP request. The event covers a variety of information, but Kraaler saves the HTTP request included in the event and the initiator of the given request. Initiators defined by CDP are *parser*, *script*, *preload*, *SignedExchange*, and *other*. Kraaler inherits and expands upon this definition by introducing additional initiators, determined by request information, namely: *redirect*

[1]https://github.com/aau-network-security/kraaler

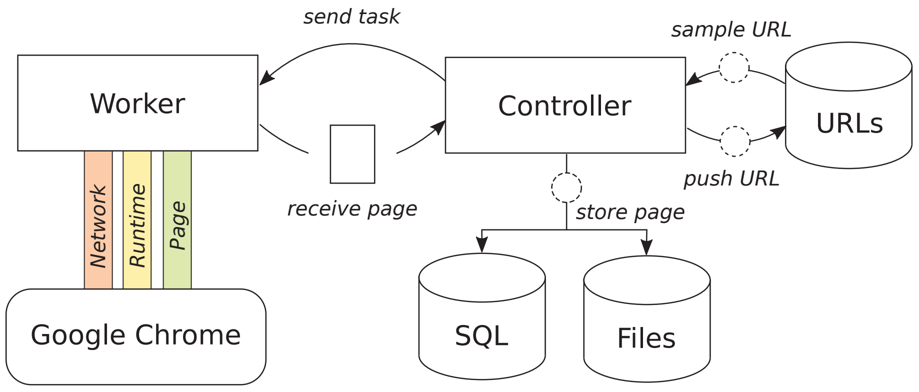

Fig. 2. Component overview of Kraaler.

and *user*. The *redirect* initiator stems from the fact that Kraaler expands a chain of HTTP redirects to become a series of individual requests, while CDP assumes a request is tied to a response body. This means that if a requested resource is located on a certain URL, it will respond with one or more redirects before receiving a response body, then CDP inherit these into the same request, except that by HTTP it is a series of requests. Introduction of the *user* initiator was done to improve clarity of the generic initiator *other*, by marking user-initiated requests, such as the initial HTTP request performed by a page navigating to an URL. For gathering the response of a request, the `Network.responseReceived` event is published, containing the HTTP response for the request. The `Network.responseReceived` event just include the metadata of the response. In order to obtain the response body, the instruction `Network.getResponseBody` is sent.

The events received across these channels are observed between the initial `Page.navigate` instruction and after the `Page.captureScreenshot` instructions has completed. Lastly, the information from these events are saved in the internal data structure of a page, being the result of a crawl.

V. Data Structure

Creating value from a crawling process requires information related to the crawled content to be available for post-processing. In Kraaler, this is conducted by having the controller store the pages it receives in a relational database and on the file system. The choice of relational database in Kraaler is SQLite, which provides efficient reads while the data can be contained in a single file. With data available in a single file, it eases data transfers to other computation environments.

Having two types of data stores allows for separating byte-intensive information, such as screenshots and response bodies, from contextual information. Storing of screenshots in Kraaler is structured in a configurable directory structure using `screenshots/<domain>/<id>.png`, for which `domain` is the domain of hosting the page visualized and `id` being a random unique identifier. Response bodies are stored in a separate directory with the structure of `bodies/<hash>.<ext>`, with `hash` being the SHA256 hash of the body and `ext` being a determined file extension based on the content type of the body. Storing every response body can yield to a substantial amount of data, causing data sets to become overwhelmingly large in size. Kraaler provides a set of modules that can filter or change the behavior of storing response bodies, namely a compression module and a filter module. The compression module allows for using gzip to compress every stored response body to reduce the data set size. Additional measures to reduce the size of the response bodies include using the filter module to reduce the set of saved response bodies to only a certain set of content types.

Contextual information being stored in the database can be seen in Table I. Entities are shown using the following notation:

Some entity $\leftarrow 0..*$ *Parent entity*

In this example the notation reads as: *Parent entity* has zero or more $(0..*)$ *Some entity*.

In order to make the underlying database efficient for analysis purposes, a set of database design principles from online analytical processing (OLAP) has been adapted in the design of the database scheme [24]. In Kraaler, the OLAP snow-flake scheme is used and provides efficient storage in terms of the size used by the database and reading speeds for database operations.

VI. Orchestration

Communication with external components, namely Chrome, is a fundamental part of the crawling process of Kraaler. The CDP service of Chrome is a central dependency that would interrupt the crawling process, in case of its absence. Ensuring the availability of the service is inherently difficult, as its presence can only be observed from an operating system perspective or by interacting with it.

Experience gained from implementing Kraaler made it clear that the service could become unavailable or unusable. This led to a set of scenarios, that were difficult to differentiate from external observations, such as: external web servers being unresponsive, web servers replying slowly, or the instance of Chrome being in an unexpected state.

These scenarios could lead to a worker becoming unable to continue crawling, and drove the design of increased fault

TABLE I
PROPERTIES OF DATA STRUCTURE

	Property	Description
Page	Browser resolution	Resolution used by the browser
	Navigated time	Unix time nanoseconds of when a page is request
	Loaded time	Unix time nanoseconds of when a page's DOM is loaded
	Terminated time	Unix time nanoseconds of when a crawl of page is complete
	Page connection error	Possible connection error of page
Console output $\leftarrow$ 0..$*$ Page	Sequence	Index of the `console.log` message for the page
	Origin	Position of JavaScript call using `console.log`
	Message	Message printed by `console.log`
Screenshot $\leftarrow$ 1..$*$ Page	Time taken	Unix time nanoseconds of when screenshot was captured
	Path	Filesystem path to screenshot file
Action $\leftarrow$ 0..$*$ Page	Parent of action	Possible previous action causing this action
	Request method	Method used for HTTP request of action
	Protocol	Protocol used for the action
	Initiator	The type of initiator initating the action
	Status code	Status code of the action's HTTP response
	Connection error	Possible connection error of the action
Host $\leftarrow$ 1 Action	Apex domain	Domain without subdomains
	Top-level domain	Top-level domain
	IPv4	IPv4 Address of domain being resolved
	Name servers	Authoritative name servers of domain
URL $\leftarrow$ 0..$*$ Action	Scheme	Scheme used within the URL
	User	User field of URL
	Host	Host contained within the URL
	Path	Path of URL
	Fragment	Fragment used in URL
	Query	Query string of URL
Response Header $\leftarrow$ 0..$*$ Action	Key	Key field of response header
	Value	Value field of response header
Request Header $\leftarrow$ 0..$*$ Action	Key	Key field of request header
	Value	Value field of request header
Security Details $\leftarrow$ 0..1 Action	Secure protocol	Secure protocol used by the given action
	Certificate key exchange	Type of key exchange used by action
	Certificate issuer	Issuer of the certificate used in the action
	Certificate cipher	Certificate cipher used for action
	Certificate san list	San list of certificate used by action
	Certificate subject name	Subject name of certificate used for action
	Certificate valid from	Unix time nanoseconds of validation start of cert. for action
	Certificate valid to	Unix time nanoseconds of validation end of cert. for action
Response Body $\leftarrow$ 0..1 Action	Browser MIME type	MIME type of body, determined by the browser
	Worker MIME type	MIME type of body, determined by the worker
	Hash	SHA256 hash of the response body
	Size	Size, in bytes, of response body
	Compressed size	Gzip compressed response body size in bytes
	Path	Path to file containing response body

tolerance. Measures for increasing the fault tolerance, and included in the orchestration of external components, are: adding timeouts for crawls, release of unresponsive resources, isolation of Chrome instance and designing for errors.

Timeouts are a time-based threshold measure for defining and reacting to unexpected behavior in an application. They are typically defined by having a time limit that describes the maximum allowed time a certain process is allowed to be spending for processing. For Kraaler this mechanism is used internally within each worker, timing the process responsible for sending instructions and receiving feedback from CDP. In case of a timeout, a page with an internal connection timeout error is returned to the controller.

Following the case of a timeout, the resources of the asynchronous process, that was unable to complete on time, is not guaranteed to be freed and can be locking resources. In Kraaler, we explicitly ensure to inherit an idiomatic design pattern for solving this, while also restarting the instance of Chrome. This restart is done to ensure that the instance of Chrome is cleared from its potential faulty state, by returning it to the predictable initial state.

Our recommended method of running the external Chrome instances is by letting Kraaler communicate with the Docker daemon. This method allows for spawning Chrome instances in an isolated run time scope while controlling the resources available to them. The isolation ensures that the Chrome instances are unable to interfere with each other, as their run time scope is independent. Kraaler will by default constrain the spawned Chrome instances to 756MB, which from experience has been found sufficient for single page browsing.

Restarting the Chrome instance of a worker can be costly in terms of wasted crawling time, and should only be considered a last resort. In order to reduce the number of unnecessary restarts of the Chrome instance, a set of errors returned by CDP are provided corresponding recover mechanisms. This allows workers to cheaply recover from errors such as CDP connections timeouts, `Page.navigate` timeouts, and more. However, in the case of an error without a defined recover mechanism, the Chrome instance is restarted to ensure no present side effects.

VII. Evaluation

Correctness is an essential attribute for a data set in order to be used for future analyses. In the setting of Kraaler, the data is collected by observing external web servers, for which their underlying design and behavior is unknown. Thereby in order to increase confidence in collected data being correctly crawled and stored, this functionality is evaluated against known web servers, for the sake of predictability. A set of automated end-to-end tests are designed, for which each individual test hosts a web server with distinct behavior. The hosted web servers are then crawled by Kraaler, following observations of changes in the database and file system. These changes are then compared against an expected change, to ensure the expected behavior of the implementation.

The set of test cases does, however, not validate the value of the browser engine's parsing capabilities in a real-world setting, as the tested web application might not be representative of that population. In order to validate the value, Kraaler was set out to crawl a uniform sample of Alexa Top 1M, while being restricted from pushing new URLs to the pool of URLs. This crawl was conducted on a single machine running eight worker instances and resulted in 8156 pages and 331561 actions. The influence of the browser engine's parsing capabilities, in terms of HTTP requests being initiated by the browser engine, can be seen in Table II. User-oriented initiations, i.e. those started by `Page.navigate`, are filtered out to focus only on the ones conducted by the browser engine.

TABLE II
Overview of non-user initiators of actions from Alexa Top 1M page crawls.

Initiator	Page actions μ (σ)	Body size (kB) μ (σ)
parser ($n = 266819$)	34.34 (25.31)	41.04 (125.77)
script ($n = 46131$)	5.94 (6.39)	46.77 (96.53)
redirect ($n = 4512$)	0.58 (0.96)	107.00 (161.40)
other ($n = 2399$)	0.31 (1.62)	24.96 (83.15)

In comparison to a naive crawler, one which just fetches a given response body without parsing it, a substantial amount of requests would not be conducted. The initiators, *parser* ($\mu = 34.34$) and *script* ($\mu = 5.94$), account for a significant amount of additional requests compared to a naive crawler. These initiated requests are a representation of the sum of the parsing capabilities of the browser engine, and can be expressed as an upper bound measure for requests initiated by parsing for user-perspective web crawling. The total size of the response bodies for these subsequent HTTP requests represent 95.8% of the total amount of bytes for response bodies in the crawl, leaving the root documents to 4.2%.

Browsing the size of subsequent response bodies of pages, in relation to their content type, can indicate the amount of information being carried by certain technologies. The amount of information being carried by certain types of technologies can suggest the importance of certain language parsers in a crawler setting. This information, carried across pages, for the ten most frequently used MIME types across pages, is illustrated using a cumulative distribution function in Figure 3. JavaScript with its three MIME types (application/x-javascript, application/javascript, text/javascript) is responsible for a large degree of the bytes across the pages crawled. We suspect it might be due to popular domains using complex user interfaces and relies on JavaScript-based front-end frameworks, which take up a certain byte volume.

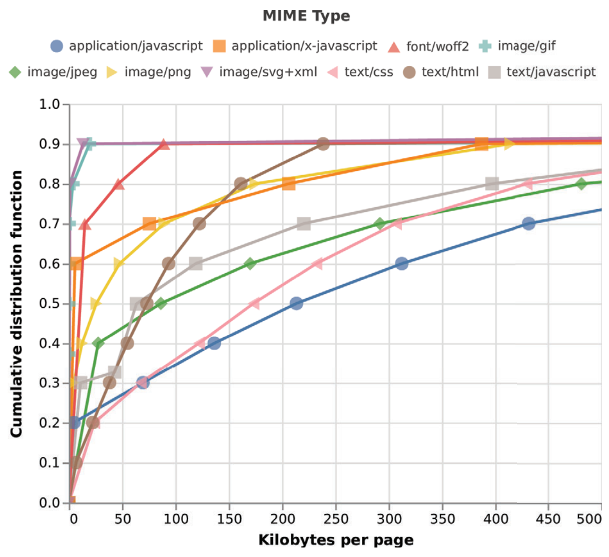

Fig. 3. Cumulative distribution function of bytes per page for the ten most frequent content types.

The usage of technologies, in terms of request frequency and byte volume, vary significantly, e.g. *parser* actions $\sigma = 25.31$, and *parser* bytes $\sigma = 125.77$. This might suggest that pages rely on a distinct set of linked dependencies, due to the present variance in amount actions and their respective byte size for pages. This variance could also prove useful for identification of segments that carry meaning for a given problem domain that attempts to classify web applications. As an example, identification of web applications hosting malicious activities could be accomplished under the assumption that malicious web applications differentiate in their interlinked structure and usage of certain dependencies. However, this example and other applicabilities of the data set are for future research to examine, in addition to exploring the protocol-based information also contained within the data set.

VIII. Conclusion

In this article, we have presented the design of a crawler, named Kraaler, that uses the Chrome Debugging Protocol for parallelized crawling, using a modern browser engine, while obtaining detailed information from the HTTP protocol usage. Follow this, a design for storing this detailed information was covered, allowing the stored data efficiently read and available for data analysis.

The challenges of interacting with external components, such as a web browser, was presented in addition to the measures we have taken in order to solve them and provide orchestration. Throughout the implementation of our solution, the methods used for determining its correctness has been described.

Following the impact of the obtained parsing capabilities, from the browser engine, for the request frequency and size of the information collected throughout a crawling process. Demonstrating that the HTML parser ($\mu = 34.34$) and the JavaScript interpreter ($\mu = 5.94$) accounted for a significant amount subsequent HTTP requests for a page visit, for a subset of crawled Alexa Top 1M domains. The size of these response bodies, in terms of bytes, accounted for 95.8% of the total size of response bodies that was crawled.

We suggest that data sets collected using Kraaler could potentially be used for a variety of applications that seek to conduct statistical analysis of web applications.

IX. Acknowledgements

With the Chrome Debugging Protocol being essential for this work, our gratitude towards the Chromium Developer Team cannot be understated. A great appreciation should be sent to Mathias Frederiksson, for his wonderful CDP Go package. I (Thomas) would like to thank my first-born daughter, Joanna, for the ability to be born in a timely fashion to cheer for her dad throughout his long working days, required to complete this article. Likewise, her caring mother, Maria, for indulgently accept my absence.

References

[1] Manish Kumar, Rajesh Bhatia, and Dhavleesh Rattan. A survey of web crawlers for information retrieval. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2017.

[2] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[3] Marco Vieira, Nuno Antunes, and Henrique Madeira. Using web security scanners to detect vulnerabilities in web services. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, page nil, 6 2009.

[4] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[5] Google. Chromium Blink. https://chromium.googlesource.com/chromium/blink, 2013.

[6] KDE. WebKit. https://svn.webkit.org/repository/webkit/, 1998.

[7] Mozilla. Mozilla Gecko. https://hg.mozilla.org/mozilla-central/, 1998.

[8] StackOverflow. Developer Survey Results 2018. https://insights.stackoverflow.com/survey/2018 Last accessed on 2019-13-04, 2018.

[9] GitHub. GitHub Octoverse. https://octoverse.github.com/projects Last accessed on 2019-13-04, 2018.

[10] Statista. Global market share held by leading desktop internet browsers from January 2015 to December 2018. https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/ Last accessed on 2019-13-04, 2018.

[11] Arie van Deursen, Ali Mesbah, and Alex Nederlof. Crawl-based analysis of web applications: Prospects and challenges. *Science of Computer Programming*, 97(nil):173–180, 2015.

[12] Ariya Hidayat. PhantomJS - Scriptable Headless Browser. http://phantomjs.org/, 2010.

[13] Joachim Hammer and Jan Fiedler. Using mobile crawlers to search the web efficiently. *International Journal of Computer and Information Science*, 1:36–58, 2000.

[14] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing. In *Proceedings of the 23rd International Conference on World Wide Web - WWW '14 Companion*, page nil, - 2014.

[15] Clement de Groc. Babouk: Focused web crawling for corpus compilation and automatic terminology extraction. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, page nil, 8 2011.

[16] Mydeco. Scrapy: a fast high-level web crawling & scraping framework for Python. https://scrapy.org/ Last accessed on 2019-13-04, 2008.

[17] Scrapinghub. Portia: Visual scraping for Scrapy. https://github.com/scrapinghub/portia Last accessed on 2019-13-04, 2014.

[18] Reiner Kraft and Jussi P. Myllymaki. System and method for enhanced browser-based web crawling, 2000.

[19] Ian Jacobs, Chris Wilson, Jonathan Robie, Mike Champion, Arnaud Le Hors, Robert S Sutor, Scott Isaacson, Steven B Byrne, Gavin Nicol, and Lauren Wood. Document object model (DOM) level 1. W3C recommendation, W3C, October 1998. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

[20] Julian Aubourg, Anne van Kesteren, Hallvord Steen, and Jungkee Song. XMLHttpRequest level 1. WD not longer in development, W3C, October 2016. https://www.w3.org/TR/2016/NOTE-XMLHttpRequest-20161006/.

[21] Mike West. Secure contexts. Candidate recommendation, W3C, September 2016. https://www.w3.org/TR/2016/CR-secure-contexts-20160915/.

[22] Sagar Shivaji Salunke. *Selenium Webdriver in Java: Learn With Examples*. CreateSpace Independent Publishing Platform, USA, 1st edition, 2014.

[23] Chrome Debugging Protocol. https://chromedevtools.github.io/devtools-protocol/, 2019. [Online; accessed 28-March-2019].

[24] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.