ARC: Protecting against HTTP Parameter Pollution Attacks Using Application Request Caches

Elias Athanasopoulos[1], Vasileios P. Kemerlis[1], Michalis Polychronakis[1], and Evangelos P. Markatos[2]

[1] Department of Computer Science
Columbia University, New York, NY, USA
{elathan,vpk,mikepo}@cs.columbia.edu
[2] Institute of Computer Science
Foundation for Research and Technology – Hellas
markatos@ics.forth.gr

Abstract. HTTP Parameter Pollution (HPP) vulnerabilities allow attackers to exploit web applications by manipulating the query parameters of the requested URLs. In this paper, we present Application Request Cache (ARC), a framework for protecting web applications against HPP exploitation. ARC hosts all benign URL schemas, which act as generators of the complete functional set of URLs that compose the application's logic. For each incoming request, ARC exports the URL, extracts the associated schema, and searches for it in the set of already known benign schemas. In case the schema is not found, the request is rejected, and the event is recorded.
ARC can be transparently integrated with existing web applications without any modifications to the server and client code. It is implemented in Google's Go language and uses efficient data structures for storing the URL schemas, imposing negligible computational overhead on the web application server. When running on a 4-core Linux server, ARC can process hundreds of thousands of URL requests per second. A typical URL resolution is in the scale of microseconds.

Keywords: HPP, Web Security.

1 Introduction

Web applications are experiencing a variety of highly sophisticated attacks that stem from many different sources. Some of them exist due to fundamental design choices of the web platform [5], while others rise due to faulty browser implementations [4,20]. Some of them are based on deceiving users by creating specially crafted visual conditions [10,15], and others emanate from the complexity and the wide use of web applications in many different systems [27,7]. HTTP Parameter Pollution (HPP) is a recently discovered technique for exploiting web applications. HPP can be considered as an *injection* attack that targets URLs;

one of the fundamental concepts of the web platform [33]. Web browsers communicate with web applications through HTTP requests and responses, which reference resources using URLs. This communication can be polluted by injecting parameters in the HTTP stream. These *injected* parameters form URLs, which if served, instruct the application to perform actions that were originally not part of the application's design. Thus, the control flow of the web application is altered according to an attacker's need.

To illustrate the attack in a short example (in Section 2 we give a detailed presentation of HPP and, more particularly, in Section 2.1 we discuss a formal threat model), consider an e-store application taking two arguments, namely a product identifier and an action, which affects the product state. A combination of a product identifier and the action *purchase* results in ordering a product. The product identifier and the action must be attached as parameters in a URL, which in turn is communicated to the application through the construction of an HTTP request. If the attacker manages to *pollute* the request with extra parameters, then the control flow of the application may change in numerous ways. The simplest manifestation of the vulnerability is for the attacker to inject a particular parameter multiple times. In case the parameter that carries out the product identifier is duplicated, then many different control flows can take place, depending on the parameter occurrence (first, last, or a combination of) that the application will give significance while the URL is parsed.

About 1,499 of 5,000 highly ranked in Alexa.com web sites are considered vulnerable to HPP exploitation according to the methodology outlined by Balduzzi *et al.* [3]. In this paper, we propose Application Request Cache (ARC), a framework that can *protect* web applications from HPP exploitation. ARC does not detect HPP vulnerabilities, although it can record HPP exploitation attempts. It is deployed at server side and works completely transparently. A web application can be protected, using ARC, from HPP exploitation by simply incorporating ARC in the application server. Note that clients need no further modifications. In contrast to PAPAS [3], which currently is the only available methodology for discovering HPP vulnerabilities, ARC aims at protecting the web application without auditing. ARC assumes that the web application *is* vulnerable and tries to protect it from being exploited. To this respect, ARC and PAPAS can be combined. The former as a protection layer and the latter as a periodic auditor.

ARC is based on the following fundamental concept. Each web application is characterized by a set of URL schemas, which act as generators of the complete functional set of URLs that compose the application's logic. A URL schema is extracted by a URL by masking out all variables that are assigned to the URL's parameters. Each control flow is triggered by having the application serving a URL, which stems from a particular URL schema. ARC collects all schemas taken from benign requests during a training phase.[3] At production time, for each incoming request, ARC extracts the URL and its schema, and searches for

[3] Notice that the term "cache" is frequently used to describe temporary storage that holds recently or frequently used elements for improving performance. In this work,

it in a set of already known benign schemas. In case the schema is not found, the request is rejected and the event is recorded. An incoming *polluted* URL will have no schema stored in ARC and thus will be rejected. This methodology cannot only prevent HPP, but also certain types of XSS [11], where JavaScript is attached to HTTP parameters [37].

ARC is fast. Our prototype is developed using Google's Go, a very efficient programming language for constructing system tools. ARC stores all cached schemas in carefully selected data structures, which are implemented using `maps` and `slices`, as provided by Go. ARC also takes advantage of the multiprocessing features of Go, goroutines and `channels`. In a 4-core Linux server, ARC can process hundreds of thousands of URLs per second. A typical request resolution takes no more than a few microseconds.

Contributions. This paper contributes the following:

- We define a formal threat model for HPP; a new class of vulnerability targeting web applications.
- We design ARC, a framework that can efficiently protect web applications from HPP exploitation. The framework can be applied transparently in any application server. The web application and the available clients need no modifications.
- We implement and evaluate an ARC prototype. We implement ARC in Go, a fast strong typed C-like language by Google. ARC running on a 4-core Linux server, with 4 concurrently running goroutines, can process hundreds of thousands of URL requests per second. Memory requirements, in terms of RSS, from application to application increase linearly with the size of different URL schemas.

2 HTTP Parameter Pollution

Web sites have evolved from simple, mostly-static document repositories to complex, multi-tier applications. Although different organizational paradigms are possible (*e.g.,* 3-, 4-, and n-tier), modern web applications incorporate a mixture of technologies that are typically grouped into two parts: the *application* part and *presentation* part. The former runs on the server and consists of server-side code written in PHP, Perl, Java, ASP.NET, or even C/C++, whereas the latter is rendered by the client, *i.e.,* the web browser, and is made up of (D)HTML, JavaScript, Flash, *etc.* The two parts communicate over TCP using the HTTP protocol in a request-response manner. A typical form of communication involves a request issued from a web browser, for accessing a resource provided by the web application, using a request path defined very precisely in a URL [6]. The web browser issues an HTTP request, which embeds the URL describing the location of the resource, and if the web server can serve the request, it does so by returning the result in the form of an HTTP response. Otherwise, an error is

we use the term "cache" to refer to storage that holds a set of benign URL schemas, which can generate all possible URLs that can be safely served by a web application.

returned, again as an HTTP response. The following simplified URL shows an example of an on-line purchase.

```
http://www.e-store.com/purchase?item_id=42
```
(1)

The communication channel between the server and the client that is used for exchanging URLs can be attacked, affecting both the confidentiality and integrity of the application. An attacker can eavesdrop the communication and steal confidential information (*e.g.,* credit card numbers or account credentials), or modify a request issued by the client, before reaching the server, and hence break the integrity of the communication. Such attacks can be easily prevented, by forcing the web application to communicate with the client over HTTPS [17]. Nevertheless, carefully crafted injection attacks can still happen, even when HTTPS is in use. For instance, an attacker can lure an unsuspecting user to click on a hyperlink that targets a URL embedding some JavaScript code. Upon clicking the link, a request from the victim's browser is sent to a server. This request embeds JavaScript code, which, if not sanitized correctly by the web server, exists in the response and will be executed in the victim's browser. This is called Cross-Site Scripting (XSS) *reflection* attack. HPP is yet another injection technique for attacking web applications [21]. Instead of pushing JavaScript code in URLs, the attacker is polluting the URL by injecting her own combination of HTTP parameters. Consider the following URL that has the same HTTP parameter (*i.e.,* `item_id`) encoded twice.

```
http://www.e-store.com/purchase?item_id=6&item_id=42
```
(2)

The result of processing this request depends on the web application's logic. There are three possible scenarios. If the application consumes the first (from left to right) occurrence of `item_id`, then the item with `id` 6 is purchased. On the other hand, if the application consumes the second occurrence of `item_id`, then the item with `id` 42 is purchased. Finally, it is possible that the application considers both values, or a concatenation of them, as a valid `id`. In that case, both items or item 642 (or 426) are purchased. This ambiguity in processing URL parameters is the core weakness behind HPP. The attacker is taking advantage that there is no standardized way of processing URL parameters, in order to exploit a web application by altering its the control flow.

To a large extent, HPP attacks are manifested by duplicating URL encoded parameters. However, it is also possible to launch an HPP attack without injecting the same parameter multiple times, but by constructing URLs that the web application does not handle correctly.

```
http://www.e-store.com/purchase?item_id=42&action=empty_basket
```
(3)

Normally, the request shown in URL 3 results in purchasing item 42. However, due to the high complexity of modern web applications, each incoming request

is processed by a series of scripts. Hence, the script chain of the imaginary web application may host a script for which the `action` parameter is significant. If such a script is executed, then the basket holding user products will be emptied.

Running Example. Suppose that Alice is the victim, e-store is an electronic commerce application, vulnerable to HPP, and Bob is the attacker, who runs his own web site. Bob's goal is to force Alice buying a different product than the one she originally intended to. Additionally, Bob has no access to the e-store and has not compromised Alice's host machine or her browser. However, Bob can lure Alice into visiting his site. Bob's site presents some offers that can be purchased from the e-store. The web application of e-store has an entrance page, which shows all items per category, in the following form:

```
<a href='http://www.e-store.com/show?category=1'>Show cat 1<a/>
<a href='http://www.e-store.com/show?category=2'>Show cat 2<a/>
...
<a href='http://www.e-store.com/show?category=9'>Show cat 9<a/>
```

Upon clicking one of the above links, the e-store application extracts the `category` parameter, and concatenates it with the purchase action and a list of available `ids` (`item_id`) for the selected category. Note that e-shop erroneously trusts `category` and does not verify it for validity before processing it.

```
<a href='http://www.e-store.com/purchase?category=7&item_id=1'>Buy item 1<a/>
<a href='http://www.e-store.com/purchase?category=7&item_id=2'>Buy item 2<a/>
...
<a href='http://www.e-store.com/purchase?category=7&item_id=99'>Buy item 99<a/>
```

Now, Bob is creating his own entrance page with offers that can be purchased from e-store and lures Alice to visiting his site. Bob's site has the following form:

```
<a href='http://www.e-store.com/show?category=1%26item_id=42'>Go to offer 1<a/>
<a href='http://www.e-store.com/show?category=2%26item_id=42'>Go to offer 2<a/>
...
<a href='http://www.e-store.com/show?category=9%26item_id=42'>Go to offer 9<a/>
```

Alice clicks one of the above hyperlinks and the e-store application extracts the `category` parameter, which in our case is `<number>%26item_id=42`, and performs the concatenation. The result is shown below (notice that `%26` has been compiled to `''&''`).

```
<a href='http://www.e-store.com/purchase?category=7&item_id=42&item_id=1'>Buy item 1<a/>
<a href='http://www.e-store.com/purchase?category=7&item_id=42&item_id=2'>Buy item 2<a/>
...
<a href='http://www.e-store.com/purchase?category=7&item_id=42&item_id=99'>Buy item 99<a/>
```

Assuming that the e-store application gives significance to the first parameter (from left to right) while parsing a URL, the product with identifier 42 will be purchased no matter which hyperlink Alice clicks.

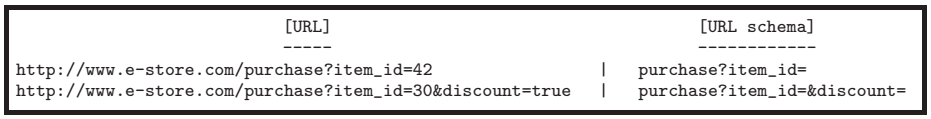

Fig. 1. Examples of URLs (left) and their respective URL schemas (right). A URL schema expresses a family of HTTP requests that act as descriptors of valid control flows.

2.1 Formal Threat Model

We now define a formal threat model for HPP vulnerabilities. A is a web application, and u_i is used for denoting any URL schema that has the following form: `action?`p_1`=&`p_2`=&...&`p_N`=`. The schema is composed of an action and a set of parameters that can take arbitrary values. *URL schemas express families of HTTP requests that are served by the web application and act as descriptors of valid control flows.* Figure 1, illustrates a set of URLs with their respective URL schemas.

$U_a = \{u_1, u_2, ..., u_n\}$ is a set that contains all benign URL schemas that A can handle. This means that for each incoming URL in U_a, a well defined control flow f takes place, according to the application's logic. More formally:

$$\forall u \in U_a \longrightarrow f \in F_L$$

$F_L = \{f_1, f_2, ..., f_N\}$ contains the control flows that can be handled safely by the web application. We denote as F_c the set of *all* possible control paths of A. Apparently, $F_L \subseteq F_c$ and $F_h = F_c - F_L$ is the set of all control flows that A can reach, but not initially programmed to execute.

We define the set $U_{hpp} = \{v_1, v_2, ..., v_N\}$ that contains all URL schemas that can initiate a control flow $f \in F_h$. Ideally, we want A to *reject* all incoming v for which the following relationship holds:

$$\forall v \in U_{hpp} \longrightarrow f \in F_h.$$

Notice that flows in F_h may have arbitrary consequences and force the web application to produce undesired results.

2.2 Extreme Cases

We have defined HPP as a technique that is based on the creation of URLs embedding a combination of legitimate, yet unexpected, HTTP parameters, which can drive a web application to an undesired state. So far, we have discussed only the case where a *combination* of parameters is not handled (sanitized) correctly. However, it is possible that HPP can be carried out using the following techniques, depending always on the complexity of the web application.

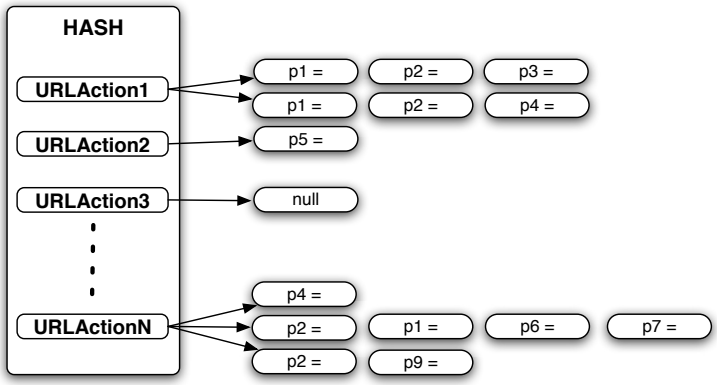

Fig. 2. The data structures used by ARC. A hash table, which holds references to linked lists hosting the set of the parameters of each schema. Each entry in the table has been produced by hashing the action part of a URL schema.

Parameter Sequence. An attacker may carefully construct URLs that contain valid HTTP parameters, but in a non-expected order. Depending on the complexity of a web application, it might be possible that the URLs trigger a series of server-side scripts, which if executed in a non-expected order, a surprising result occurs. Note that ARC can be configured so that it can protect against such attacks (see Section 3).

Parameter Values. An attacker may carefully construct URLs that have an expected sequence of HTTP parameters, but with erroneous values. This case is hard to prevent, since it stems from unsafe input sanitization. ARC is based on URL schemas, which have already masked out all values, and tries to prevent parameter *injection*. We believe that with minor modifications, ARC might be able to handle such scenarios, but it needs significant effort and knowledge of the web application's internals by the developer.

3 Application Request Cache

An ARC is a cache that stores all possible URL schemas supported by a web application's logic. Recall that a URL schema is characterized by an action and a set of parameters. Each parameter is not bounded by a specific range of values. URL schemas express generators of HTTP requests served by the web application and they act as descriptors of valid control flows. A URL schema describes a series of different control flows. For example, consider the following URL schema taken from the running example of this paper:

```
www.e-store.com/process-item?item_id=&action=
```

The schema is characterized by an action, in our example "`process-item`", and a set of two parameters: {`item_id, action`}. New control flows are created

depending on the value each parameter of the set takes. If "`delete`" is assigned to "`action`" the product corresponding to a given "`item_id`" will be erased. If "`show`" is assigned to "`action`" the product corresponding to a given "`item_id`" will be rendered in the user's browser. ARC aims at collecting and maintaining all benign URL schemas supported by a web application. An ARC-enabled web application, checks every incoming HTTP request to verify if a benign URL schema for the particular request is already stored in the ARC. In the case there is no available schema, ARC does not forward the HTTP request to the application server, and the event is logged. There are two crucial things for the transparent and efficient operation of the system. First, the collection of URL schemas must take place in a controlled environment and in an automated fashion. Second, upon the schemas' cache has been built, ARC must resolve each incoming HTTP request as fast as possible.

Data Collection - Training. ARC needs to know in advance all valid URL schemas supported by the web application. Thus, ARC needs initially to be trained, while the web application is running in non-hostile environment and is receiving only legitimate traffic. It is common for many anomaly detection systems to require an initial training phase [30,25,29]. While in training phase, ARC passively monitors all web traffic received by the web application, filters out all URLs and extracts all URL schemas. These URL schemas are the generators of the complete set of legitimate HTTP requests the web application can serve without becoming HPP exploitable. Training is particularly easy for large companies, which perform extensive beta-testing prior publishing their applications in the wild. Passively monitoring a web application while it is being developed can produce the complete set of allowed schemas, since developers are used to test every new feature they implement. Training is also easy for applications that are based on frameworks for providing blog, forum, or other web services. This is because the application must be monitored *once* for extracting all URL schemas. The same cache can be used by all application instances.

Another option is to use a crawler or scanner for extracting all possible URLs the application provides. However, modern applications use dynamic interfaces implemented in AJAX [12], which many times perform requests towards the application server asynchronously using JavaScript. These requests cannot be easily captured by a crawler. However, today, there are efforts towards sophisticated crawlers that can handle the complexity and the dynamic nature of Web2.0 applications with rich interfaces. One such effort is Crawljax [22], which has been used by researchers for extracting the user interface of Web2.0 applications [8]. Finally, notice, that many frameworks assume that all URLs an application can handle is known [26,16,2] (see discussion in Section 6).

Data Structures. The data structures used by ARC is a hash table and a collection of linked lists. Each schema is stored in the cache in the following way. First, the *action* part of the schema is hashed. In the case there is no entry in the hash table with the same key, a new hash node is inserted at the index, which is equal to the key. Otherwise, a pointer of the currently occupied index of the hash table is fetched. This pointer holds references to linked lists, which

host the set of the parameters of each schema. In the case that there is no list hosting the parameters of the new schema, a new one is created and a reference is assigned to the hash index. The data structures are schematically depicted in Figure 2. Observe the hash table that stores each action (from 1 to N), which is noted with `URLActionX`. Each hash entry stores pointers towards a series of linked lists. In the example ARC of Figure 2, `URLAction1` stores two pointers, meaning that this entry describes two different URL schemas, each of them described by three different parameters. In the same fashion, `URLAction2` stores one pointer towards a list that contains a single parameter, and `URLAction3` stores a pointer towards `null`, meaning the particular schema takes no parameters. Finally, `URLActionN` stores three pointers towards three lists, containing 1, 4, and 2 parameters, respectively.

Search Algorithm. It is trivial now to derive the search algorithm and its complexity, since we have analyzed the data structures employed by ARC in the previous part. For each incoming HTTP GET or POST request the URL schema is derived by parsing the request line. The action part (the part before the character "`?`") and the set of parameters (all left parts of expressions "`par=var`" delimited with each other by the character "`&`") are derived in this step. We assume that URLs follow the specification [6]. ARC can be extended to use a custom URL schema, for web applications that do not follow the specification, since ARC runs purely at the server side and, thus, can co-operate with the application server. We do not account for parsing operations in complexity, since all requests have to be parsed by the application server, no matter if ARC is enabled or not. When a URL schema is derived, the action part is hashed and is looked-up in the ARC table. The complexity of this operation is $O(1)$. Now, the set with the parameters of the schema has to be checked against all sets already stored with this action. We define as URL action density, ρ, the ratio of unique actions over all possible URL schemas. For example, a web application that supports 1,000 URL schemas and those include 100 unique actions, has $\rho = 0.1$. The density reversed approximates how many schemas are associated with a particular action, or how many lists are associated with each hash bucket. Assuming that an input schema has a number of parameters, N, then the complexity of the search is $O(\frac{N}{\rho})$. Thus, the complexity of the complete algorithm is $O(1) + O(\frac{N}{\rho}) \simeq O(N)$. Thus, the search algorithm has linear complexity with the number of parameters of each input schema.

Optimizations. We can substitute the linked lists with trees, in order to reduce the search time required for scanning the lists. The optimized version can reduce the search time and, thus, increase the URL throughput (see Section 4). However, security must be sacrificed, since cases described in Section 2.2 cannot be handled correctly. Thus, for the rest of this paper, we discuss and evaluate only the unoptimized ARC. A second approach is to use DFAs for searching the cache. Consider, for example, that each URL can be represented by a string, whose characters are selected from a space defined by all the different parameters, which can occur in all collected URL schemas. Although, a DFA has linear complexity in search, in practice, implementing regular expressions that can contain all the

thousands of URL parameters used by a large web application is not considered trivial, due to intrinsic constraints of current off-the-shelf implementations. For example, PCRE [4] has a hard limit for the maximum size of a regular expression, and it needs special recompilation for changing this. Finally, we can use a single hash table for speeding up search. For each incoming schema, we can concatenate the action part and all parameters and feed the result to a hashing function. By definition, this will speed up the search to $O(1)$, no-matter the length of the URL parameters of each incoming schema. However, large web applications, using many URLs, will experience hash collisions, which can be resolved by incorporating linked lists. Thus for large web applications, this approach is, essentially, identical to the approach we follow for building ARC.

3.1 Implementation

We implemented an ARC prototype in Go [19]. Go is a programming language created by Google for fast system development. We created two versions, one single-threaded and one that utilizes 4 threads. In the world of Go, the term *goroutine* is used, instead of thread. Goroutines cannot be used standalone. There is no way for a goroutine to complete and communicate the result to the rest of the program, unless a channel is used. Thus, for the rest of this paper, we will refer to the single-threaded version as *single-channel ARC* and to the multi-threaded one as *4-channel ARC*. As far as the data structures are concerned, we use `maps` for implementing the hash and `slices` for implementing the linked lists. `Maps` and `slices` are standard data structures provided by Go. A `map` represents a relation of two data types, one serving as the key and one as the data holder. On the other hand, `slices` are similar to C arrays, but their size can be modified at run-time. The ARC implementation works as follows. First, it builds the cache by reading a collection of already stored URLs in the disk. It forms the cache (see Figure 2), which is maintained in memory (we evaluate the system's memory footprint in Section 4). For each incoming HTTP request the application server extracts the URL (and the POST parameters, if it is required) and forwards it to the ARC. The URL schema is extracted and the ARC looks up in the available cache for its existence. If the schema exists, the parsed form of the URL is forwarded to the application server, otherwise the incoming request is dropped and the event is logged. We implemented ARC and the application server in Go. However, with minimum changes, ARC can cooperate with any modular application server.

4 Evaluation

All experiments are carried out using artificially created traces. In this way, we are able to create large collections with thousands of URL schemas, in order to stress our implementation as much as possible. Initially, we create three different

[4] `http://www.pcre.org`

Web Application	URLs	Min Par.	Max Par.	ρ
Small	1,000	5	12	0.01
Medium	10,000	7	15	0.001
Heavy	100,000	12	20	0.001

Table 1. Properties of URL sets used in evaluation. Each set is characterized by 4 properties: (1) the amount of URLs the set includes, (2) the minimum number of parameters a random URL of the set may include, (3) the maximum number of parameters a random URL of the set may include, and (4) the density ρ of the set.

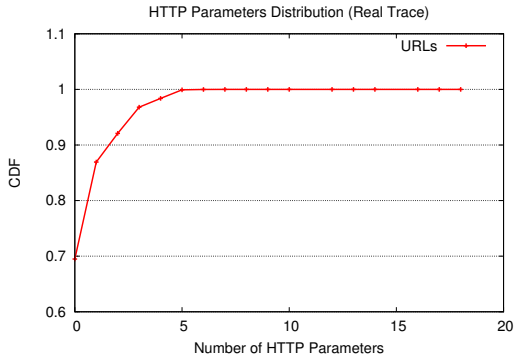

Fig. 3. Cumulative distribution function of HTTP parameters, as collected from a real-world trace, including HTTP/HTTPS traffic for the phpBB and phpMyAdmin applications. The plot depicts 1 million URLs sampled from a trace containing over 50 millions of captured URLs.

URL sets. The set is composed by URLs that are formed by a random action part and by a set of random strings representing URL parameters. Each parameter is a random string of size between 6 and 16 characters. Each set is characterized by 4 properties: (1) the amount of URLs the set includes, (2) the minimum number of parameters a random URL of the set may include, (3) the maximum number of parameters a random URL of the set may include, and (4) the density ρ of the set. Recall from Section 3, that ρ is defined as the ratio of unique actions over all possible URL schemas. Thus, we create three URL collections, each one representing a different web application. The first set contains 1,000 URLs, each one having 5 to 12 parameters, with $\rho = 0.01$. We will further refer to this set as *Small Application*. The second set contains 10,000 URLs, each one having 7 to 15 parameters, with $\rho = 0.001$. We will further refer to this set as *Medium Application*. Finally, the third set contains 100,000 URLs, each one having 12 to 20 parameters, with $\rho = 0.001$. We will further refer to this set as *Heavy Application*. We summarize all these details in Table 1.

The characteristics of the artificially created traces are based on real-world evidence. We monitored two well-known web applications, phpBB and phpMyAdmin, and managed to collect over 50 millions of URLs. We then analyzed

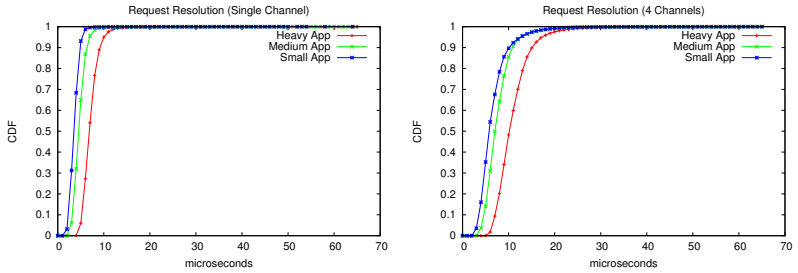

Fig. 4. Cumulative distribution function of all measured resolutions, for both the single-channel and the 4-channel version of the ARC, and for all different web applications. The majority of all request resolutions, about 98%, are completed in less than 10 microseconds.

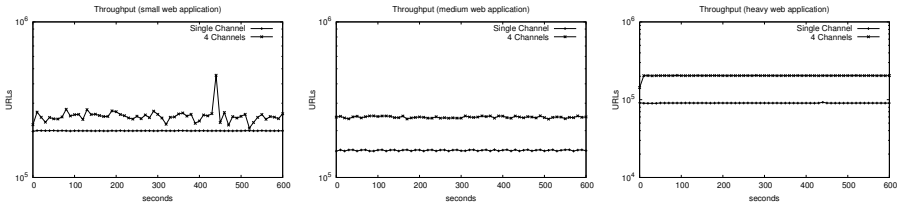

Fig. 5. Resolved requests per second for the small, medium, and heavy application, respectively. The 4-channel ARC significantly outperforms the single-channel one, serving hundreds of thousands requests per second, in all applications.

a sample of 1 million URLs and measured the number of HTTP parameters per HTTP GET/POST request. We plot the CDF in Figure 3. Notice, that the majority of HTTP requests include less than 5 different parameters, and there were not recorded HTTP requests containing more than 18 parameters. The tree different URL sets are precomputed and stored to files on disk. For each experiment, ARC loads the URLs, exports the schemas, and creates the caches as we described in Section 3 (see Figure 2). All information is maintained in memory. As far as the hardware setup is concerned, all experiments run in a Linux server, equipped with i7/2.93 GHz (4-cores) and 4 GB RAM.

4.1 Request Resolution

We are interested to identify the average time it takes for ARC to process one singe request. We run the ARC with one of the three URL sets, which correspond to a particular web application (small, medium, and heavy). We forward 1,000,000 URL requests towards ARC, after it has loaded all URLs and has built all data structures. All requests are taken randomly from the initial file that hosts the artificially created URLs. For each request we measure the time needed by ARC to find the URL schema that corresponds to the incoming URL request.

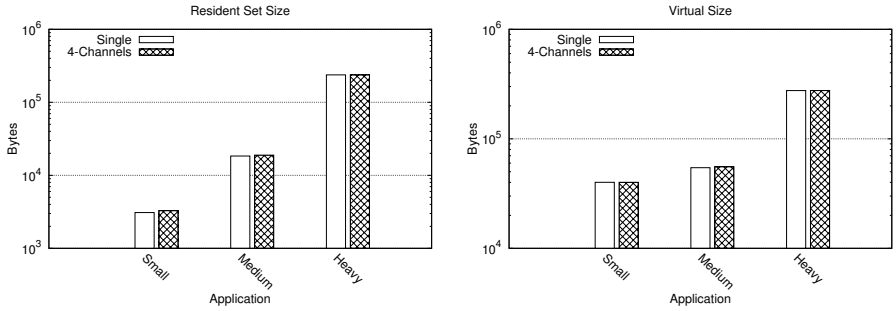

Fig. 6. Resident Set Size (RSS) and Virtual Size (VSIZE), both as reported by `ps(1)`, while running ARC for each one of the three applications. Notice, that both, 4-channel and single-channel, versions have similar memory requirements. Also, memory requirements, in terms of RSS, from application to application increase linearly.

The search time includes parsing the initial URL. We perform all measurements with the `Nanoseconds()` function, which is contained in the `time` package.

In Figure 4 we plot the CDF of all measured resolutions, for both the single-channel and the 4-channel ARC, and for all different web applications. It is important to highlight the following. First, the majority of all request resolutions, about 98%, are completed in less than 10 microseconds. We consider that the performance is enough for not causing significant overhead to an application server, even in configurations that are based on commodity hardware. Second, the requests for the heavy application seem to be resolved a little bit slower than the medium one, and the requests of the medium one seem to be resolved a little bit slower than the small one. This is reasonable, since the heavy application is characterized by URLs that have more parameters that the ones of the medium and of the small ones. This has two consequences: (1) the parsing time is longer (recall, that we account for parsing in every search operation), and (2) the lists' size is larger or, more formally, N is larger (recall the complexity of the search algorithm, $O(N)$, presented in Section 3). Finally, notice that the 4-channel ARC behaves worse than the single-channel ARC (all CDFs are shifted to the right, in the right plot of Figure 4). Initially, this seems to be counterintuitive. However, it is not. The 4-channel version has the additional overhead of managing and context-switching the 4 goroutines. This affects slightly the performance of each request resolution. Nevertheless, the overall performance of the 4-channel version significantly outperforms the single-channel version, since the 1,000,000 requests are completed in shorter time. We quantify this in the following part.

4.2 Request Throughput

We configure ARC to run with each one of the three different applications for 600 seconds. We record how many requests ARC can resolve per second for the small, medium and heavy application, respectively. We run all experiments for

```
package hello

import (
    "arc"
    ....
)

func init() {
    http.HandleFunc("/", handler)
    arc_stats = arc.Init()
}

func handler(w http.ResponseWriter, r *http.Request) {
    if (arc.FilterURL(r.URL.RawPath) == true) {
      fmt.Fprint(w, deliver_page(r))
    } else {
      fmt.Fprint(w, deliver_error("URL is not supported."))
    }
}
```

Fig. 7. An example web application written in Go, for running over Google's AppEngine, which incorporates ARC. Some functions are omitted for presentation purposes. Notice, that ARC integrates seamlessly with the rest of the code.

both, 4-channel and the single-channel, ARC implementations. We present the results in Figure 5. Notice, that the 4-channel ARC significantly outperforms the single-channel one in all applications. Observe, that the 4-channel ARC can serve hundreds of thousands requests per second. This is to be expected, because the 4-channel ARC takes advantage of all 4 cores of the server. Thus, a typical request resolution maybe slightly faster for the single-channel ARC, but the overall throughput is much greater for the 4-channel ARC.

4.3 Memory Footprint

ARC stores all information (*i.e.,* all URL schemas) in memory for fastest access. The more the distinct URL schemas a web application has, the more the memory the ARC needs. In Figure 6 we plot the Resident Set Size (RSS) and the Virtual Size (VSIZE), both as reported by `ps(1)`, while running ARC for each one of the three applications. Notice that both versions (*i.e.,* 4-channel and single-channel) have similar memory requirements. This is to be expected, since both versions maintain memory in exactly the same way. Notice, also, that the memory requirements, in terms of RSS, from application to application increase linearly. Recall from Table 1, that the size of complexity, in terms of URL schemas, for each application increases by one order of magnitude.

5 Case Study

Google AppEngine [9] is a platform for deploying web applications. Recently, Google announced an SDK for building web applications in Go. Although, it is

still experimental, it seems the ideal application server for incorporating ARC into. Notice, that ARC can be enabled in any application server as an external CGI script. A typical web application written in Go is composed as a package. There are many official Go packages for managing HTTP requests and URLs, which can be easily imported in the main package, which serves as the core of the application. Next, there is an initialization routine which assigns handlers for URLs matching a specific pattern, and, finally, there is a series of handlers that can serve incoming requests. Enabling ARC for an AppEngine application is trivial. In Figure 7 we present the skeleton of an example web application written in Go for running over Google's AppEngine. Some functions have been omitted for presentation reasons. There are three basic steps needed for enabling ARC. First, the `arc` package must be imported (line 4). Second, `arc.Init()` must be called for initializing the cache (line 10). This function reads all available URL schemas from a text file and organizes them to data structures in memory (see Section 3 for the description of the data structures used). Finally, a check is applied to the core request handler for filtering out all incoming URLs that are not compatible with any of the available stored schemas. This check is performed using the `arc.Filter()` function, which takes as a parameter the incoming URL in raw format (line 14) and returns a boolean value (`true` if the URL compiles to a valid schema, `false` otherwise).

6 Related Work

HPP is originally discovered by Luca Carettoni and Stefano di Paola in 2009 [21]. The most relevant research to ARC is PAPAS [3], which aims at detecting HPP vulnerabilities through a black-box scanning technique. In this respect, PAPAS and ARC are different, since ARC aims at preventing exploitation through HPP; ARC assumes that the application *is* vulnerable. Nevertheless, the two technologies can be combined. A web application, which rapidly changes, can use ARC for protection and occasionally scanned for new HPP vulnerabilities. HPP Finder [1] is a Chrome extension that scans web pages in real-time for detecting potential HPP exploits. Thus, the extension aims at protecting the end user from vulnerable (to HPP) web applications. However, HPP Finder has limited scope. It can identify only hyperlinks and forms that include a particular parameter multiple times. As we have already discussed in Section 2, HPP is a broader class of vulnerabilities that can be manifested when a parameter occurs multiple times in an HTTP request. Moreover, HPP Finder has many false positives, especially in pages with radio buttons. Therefore, HPP Finder is not considered a complete solution against HPP exploitation, but rather a precaution.

There are many frameworks for detecting and preventing XSS [18,23,14,28,26,32]. Robertson and Vigna [26] attempt to introduce structure in the web documents served by a web application, for taking advantage of it and detect potential injections. The framework needs a map of all URLs that the application supports in advance. In their context, this is called a `RouteMap` and it is similar to the `routes` package present in popular web development frameworks, such as

Rails [16] and Pylons [2]. ARC needs also all URLs supported by a web application, in order to extract all possible URL schemas. However, ARC does not assume that this information is known (we have listed techniques in Section 3 for collecting URLs). Researchers have developed generic techniques for covering web exploitation [25,30]. These techniques share a common property with ARC; they are also based on a training phase for collecting features that characterize the benign behavior of the web application. These proposals are more generic, and, thus, suffer from false positives. ARC, on the other hand, is practical and focuses on HPP only. Web exploitation is not only XSS. HPP is among the recently discovered highly sophisticated techniques for attacking a web application [5,7,27,20,4,15,36]. To that end, many academic efforts aim at applying security concepts from operating systems to the web platform [34,35,13,24,31].

7 Conclusion

HTTP Parameter Pollution (HPP) is a recently discovered technique for exploiting web applications. Since web applications communicate with browsers using HTTP requests and responses, the communication can be polluted by injecting parameters that alter the control flow of the web application according to an attacker's need. In this paper, we constructed a formal threat model for HPP and we proposed Application Response Caches (ARC), a framework that can prevent HPP exploitation in web applications. ARC can be transparently enabled in an application server, without further modifications to the web application and to the clients. We implemented a single-channel and a 4-channel ARC prototype using Google's Go language. ARC running on a 4-core Linux server, with 4 concurrently running goroutines, can process hundreds of thousands of URL requests per second. A typical URL resolution is in the scale of microseconds. Memory requirements, in terms of RSS, from application to application increase linearly with the size of different URL schemas.

Acknowledgements. This work was supported in part by the FP7 project SysSec, the FP7-PEOPLE-2010-IOF project XHUNTER, and the FP7-PEOPLE-2009-IOF project MALCODE, funded by the European Commission under Grant Agreements No. 257007, No. 273765, and No. 254116, respectively, and by DARPA through Contract FA8650-11-C-7190. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the European Commission, US Government, or DARPA.

References

1. Athanasopoulos, E.: HPP Finder (2011), `http://www.ics.forth.gr/~elathan/extra/hpp/index.html`
2. B. Bangert and J. Gardner: The Pylons Project `http://pylonsproject.org`, last visited on July 2011.

3. Balduzzi, M., Gimenez, C., Balzarotti, D., Kirda, E.: Automated discovery of parameter pollution vulnerabilities in web applications. In: Proceedings of the 18th Network and Distributed System Security Symposium (2011)
4. Barth, A., Caballero, J., Song, D.: Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In: Proceedings of the 30th IEEE Symposium on Security & Privacy. Oakland, CA (May 2009)
5. Barth, A., Jackson, C., Mitchell, J.C.: Robust Defenses for Cross-Site Request Forgery. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS) (2008), http://crypto.stanford.edu/websec/csrf/csrf.pdf
6. Berners-Lee, T., Masinter, L., McCahill, M.: RFC 1738: Uniform Resource Locators (URL) (1994), http://www.ietf.org/rfc/rfc1738.txt
7. Bojinov, H., Bursztein, E., Boneh, D.: XCS: Cross Channel Scripting and Its Impact on Web Applications. In: CCS '09: Proceedings of the 16th ACM conference on Computer and communications security. pp. 420–431. ACM, New York, NY, USA (2009)
8. Chapman, P., Evans, D.: Automated Black-Box Detection of Side-Channel Vulnerabilities in Web Applications. In: Proceedings of the 18th ACM conference on Computer and Communications Security. pp. 263–274. CCS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2046707.2046737
9. Ciurana, E.: Developing with Google AppEngine. Springer (2009)
10. Dhamija, R., Tygar, J., Hearst, M.: Why Phishing Works. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 581–590. ACM New York, NY, USA (2006)
11. Fogie, S., Grossman, J., Hansen, R., Rager, A., Petkov, P.: XSS Attacks: Cross Site Scripting Exploits and Defense. Syngress Publishing (2007)
12. Garrett, J., et al.: Ajax: A New Approach to Web Applications. Adaptive path 18 (2005)
13. Grier, C., Tang, S., King, S.: Secure Web Browsing with the OP Web Browser. In: Security and Privacy, 2008. pp. 402–416. IEEE (2008)
14. Gundy, M.V., Chen, H.: Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA (Feb 8-11, 2009)
15. Hansen, R., Grossman, J.: Clickjacking (2008), technical Report, SecTheory, http://www.sectheory.com/clickjacking.htm
16. Hansson, D.H. and others: Ruby on Rails http://www.rubyonrails.org, last visited on July 2011.
17. Jackson, C., Barth, A.: Forcehttps: Protecting High-security Web Sites from Network Attacks. In: Proceeding of the 17th International Conference on World Wide Web. pp. 525–534. WWW '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1367497.1367569
18. Jim, T., Swamy, N., Hicks, M.: Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In: WWW '07: Proceedings of the 16th international conference on World Wide Web. pp. 601–610. ACM, New York, NY, USA (2007)
19. John P. Baugh: Go Programming (June 2010), iSBN: 1453636676
20. Lin-Shung, H., Zack, W., Chris, E., Collin, J.: Protecting Browsers from Cross-Origin CSS Attacks. In: CCS 10: Proceedings of the 17th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA (2010)
21. Luca Carettoni and Stefano di Paola: HTTP Parameter Pollution (2009), https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf

22. Mesbah, A., Bozdag, E., Deursen, A.v.: Crawling AJAX by Inferring User Interface State Changes. In: Proceedings of the 2008 Eighth International Conference on Web Engineering. pp. 122–134. ICWE '08, IEEE Computer Society, Washington, DC, USA (2008), http://dx.doi.org/10.1109/ICWE.2008.24
23. Nadji, Y., Saxena, P., Song, D.: Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA (Feb 8-11, 2009)
24. Reis, C., Gribble, S.: Isolating web programs in modern browser architectures. In: Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys). pp. 219–232. ACM (2009)
25. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.: Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In: Proceeding of the Network and Distributed System Security Symposium (NDSS). San Diego, CA (February 2006)
26. Robertson, W., Vigna, G.: Static Enforcement of Web Application Integrity Through Strong Typing. In: Proceedings of the 18th USENIX Security Symposium. Montreal, Quebec (August 2009)
27. Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)
28. Sekar, R.: An Efficient Black-box Technique for Defeating Web Application Attacks. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA (Feb 8-11, 2009)
29. Sommer, R., Paxson, V.: Outside the closed world: On using machine learning for network intrusion detection. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 305–316. SP '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/SP.2010.25
30. Song, Y., Keromytis, A., Stolfo, S.: Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS) (2009)
31. Tang, S., Mai, H., King, S.: Trust and Protection in the Illinois Browser Operating System. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI). USENIX (2010)
32. Ter Louw, M., Venkatakrishnan, V.: Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In: Proceedings of the 30th IEEE Symposium on Security & Privacy. Oakland, CA (May 2009)
33. Tim Berners-Lee: Tim Berners-Lee on the WorldWideWeb project. USENET post. (1991), http://groups.google.com/group/alt.hypertext/tree/browse_frm/thread/7824e490ea164c06/f61c1ef93d2a8398
34. Wang, H.J., Fan, X., Howell, J., Jackson, C.: Protection and Communication Abstractions for Web Browsers in MashupOS. In: Bressoud, T.C., Kaashoek, M.F. (eds.) SOSP. pp. 1–16. ACM (2007)
35. Wang, H.J., Grier, C., Moshchuk, A., King, S.T., Choudhury, P., Venter, H.: The Multi-Principal OS Construction of the Gazelle Web Browser. In: Proceedings of the 18th USENIX Security Symposium. Montreal, Canada (August 2009)
36. Weinberg, Z., Chen, E., Jayaraman, P., Jackson, C.: I Still Know What You Visited Last Summer. In: Proceedings of the 32th IEEE Symposium on Security & Privacy. Oakland, CA (May 2011)
37. XSSed.com: XSS exploit in key example., http://xssed.com/mirror/33541/