



CorbFuzz: Checking Browser Security Policies with Fuzzing

Chaofan Shou, İsmet Burak Kadron, Qi Su, and Tefvik Bultan

University of California, Santa Barbara

{shou, kadron, qisu, bultan}@cs.ucsb.edu

Abstract—Browsers use security policies to block malicious behaviors. Cross-Origin Read Blocking (CORB) is a browser security policy for preventing side-channel attacks such as Spectre. We propose a web browser security policy fuzzer called CorbFuzz for checking CORB and similar policies. In implementing a security policy, the browser only has access to HTTP requests and responses, and takes policy actions based solely on those interactions. In checking the browser security policies, CorbFuzz uses a policy oracle that tracks the web application behavior and infers the desired policy action based on the web application state. By comparing the policy oracle with the browser behavior, CorbFuzz detects weaknesses in browser security policies. CorbFuzz checks the web browser policy by fuzzing a set of web applications where the state-related queries are symbolically evaluated for increased coverage and automation. CorbFuzz collects type information from database queries and branch conditions in order to prevent the generation of inconsistent data values during fuzzing. We evaluated CorbFuzz on CORB implementations of Chromium and Webkit, and Opaque Response Blocking (ORB) policy implementation of Firefox using web applications collected from GitHub. We found three classes of weaknesses in Chromium's implementation of CORB.

I. INTRODUCTION

Web browsers allow users to do various things, such as streaming videos or accessing bank accounts. A malicious website should not be able to access sensitive information about a web application user, for example a bank account page. Unfortunately, due to vulnerabilities like cross-site script inclusion [1], cross-site scripting [2], Spectre [3], and Melt-down [4], malicious websites can access sensitive information that they should not have access to. Because of the aforementioned threats, browsers have adopted an increasing number of security policies like Cross-Origin Read Blocking (CORB) policy [5] that they use to protect sensitive data. The goal of the CORB policy is to prevent cross-origin access to confidential data.

In order to determine if a behavior is malicious or not, a browser security policy has to infer properties about the web application that is being used. Yet, given that a browser does not have access to web applications' internal state, nor its codebase, it cannot precisely determine the properties of the web applications. Instead, security policy implementations use the information browsers have access to, like HTTP responses

and requests, to infer properties of web applications and decide to take a policy action according to those properties.

In this paper, we focus on CORB as a browser security policy because it is one of the most important policies for protecting cross-origin resources. CORB aims to identify and block all cross-origin loads of confidential response content. However, browsers can not determine whether a specific response is confidential without inspecting the state of the web application. Since the browser cannot do that, the CORB policy implementations examine the responses instead and use information inside responses and heuristics that reflect the expected behavior to determine whether the content is confidential.

These heuristic approaches need to be tested comprehensively in order to look for scenarios where they fail to protect sensitive information. A fully automated testing approach would enable browser security policy developers to identify weaknesses in existing policies and to quickly evaluate policy modifications.

We developed a fuzzing technique to check browser security policies. Given a browser and a security policy, we use a set of open-source web applications to look for weaknesses in the security policy implementation of that browser. We use the open-source web applications as fuzzing targets, and our fuzzer creates requests for each of them, intending to achieve as much coverage as possible. By exploring a variety of web applications and covering as many behaviors as possible for each web application, our fuzzer tests a large set of scenarios for the browser security policy implementation.

In order to identify weaknesses in the browser policy implementation, we define a reference implementation of the security policy by tracking the web application states and utilize it as an oracle. The oracle is more accurate than the browser policy implementation since during fuzzing, the oracle has access to all internal information of the web application and properties of each response. Our fuzzer compares the decisions made by the oracle to the security policy implementation of the browser and reports any differences, which correspond to a weakness in the browser security policy implementation.

Most web applications typically access session data, cookies, and data store [6]. These web applications are called data-dependent. Fuzzing a data-dependent web application requires manually setting up these data sources (e.g., populating a database [7], [8]). However, given that we need to use a

This material is based on research supported by NSF under Grants CCF-1901098 and CCF-1817242.

set of web applications during fuzzing, it is not practical to manually set up the data store and session values for each web application. Thus, we propose a runtime for data-dependent web applications that enables us to automate the process. Instead of manually setting up data sources for all fuzzing targets, our runtime automatically synthesizes data store, sessions and cookies. This approach not only removes the requirement for manually setting up an environment for a web application, but also allows our fuzzer to easily mutate the data store, leading to higher coverage.

The runtime we propose generates SMT constraints for database queries and sessions or cookies usage. The SMT constraints for a database query encode the SQL statement, and we use an SMT solver to generate data values consistent with the query. The SMT constraints generated for sessions and cookies are used to check the feasibility of execution paths of the web application.

Using this approach, we have implemented a fuzzer focusing on the CORB policy, which we call CorbFuzz. While CorbFuzz is optimized for CORB analysis, it can be easily extended to support other policies by defining corresponding oracles. Additionally, for our prototype, we restrict our scope to PHP applications. Our approach can be extended to support web applications developed in different programming languages, including Python or NodeJS, by providing simple instrumentation for the target language as discussed in Section IV.

We evaluate the implementation of CORB policy in both Chromium and Webkit. CorbFuzz did not find any policy violations in Webkit and shows that CORB implementation in Webkit is robust. In Chromium, on the other hand, CorbFuzz identifies three types of code patterns that can enable attackers to bypass CORB protection. Furthermore, we modified CorbFuzz to check a sibling policy by Firefox called Opaque Response Blocking (ORB).

In this paper we present the following research contributions:

- *Browser Policy Fuzzer:* We propose a new fuzzer, CorbFuzz, for checking browser security policies. CorbFuzz is guided by web application code coverage and uses a policy oracle to identify weaknesses in browser security policies. It is fully automated and can be easily applied after each change in policy implementation.
- *Data Synthesis:* To tackle fuzzing environment setup for data-dependent applications, we propose a runtime that synthesizes and mutates the data when required. Our data synthesis approach uses SMT encoding and constraint solving to ensure consistency of data generated for database queries and sessions or cookies usage.
- *Empirical Evaluation:* We used CorbFuzz to check the CORB implementation of Chromium and Webkit. We also checked a sibling policy ORB for Firefox. We fuzzed these policies using responses of PHP web applications that we obtained from GitHub. Using CorbFuzz, we discovered three code patterns that expose weaknesses in the CORB implementation of Chromium. One of these

code patterns has been previously documented, and the Chromium team patched the policy weakness caused by another code pattern we discovered after our report.

The paper is structured as follows. In Section II, we present the background on browser precautions. In Section III, we present our fuzzing framework. In Section IV, we discuss how we synthesize the data and bypass authentication for web applications. In Section V, we evaluate CorbFuzz and describe the detected CORB weaknesses by our tool. In Section VI, we present the related work. In Section VII, we conclude the paper.

II. BACKGROUND

In this section, we provide the background information on Site Isolation and Cross-Origin Read Blocking policy.

A. Site Isolation and Information Leakage

Browser information leakage has gained increasing exposure in the last few years. According to the Same-Origin Policy (SOP) [9], one of the fundamental rules in browsers, documents from different origins cannot interact with each other. However, many exploits have been discovered to conduct cross-origin content leak [10]–[15]. Additionally, the discovery of cache-related side-channel vulnerabilities like Spectre [16] and Meltdown [4] worsen the information leaks.

Site Isolation policy [17], [18] has been proposed to counter cross-origin content leaks. Such a policy is also known as “one site per process” policy. Namely, a browser should ensure that documents from different origins are rendered and executed in their own respective sandbox. Such an effort reduces the chance of success of cache side-channel attacks and makes most cross-origin information leakage vulnerabilities in browsers no longer exploitable.

B. Cross-Origin Read Blocking

While Site Isolation policy removes the possibility of documents in different origins interacting with each other directly, there are still ways to inject documents from different origins via interfaces provided by browsers. A possible approach is to include the documents from different origins as resources required by the webpage. Some examples have been provided below, for which the first line is to load an endpoint as an image, and the second line is to load it as a script.

```

<script src="//a.com/secret"></script>
```

In addition, other browser JavaScript interfaces could be used to pass partial sensitive information from one origin to another. A famous example is CVE-2020-6442 [19]. The vulnerability is that by loading two cross-origin documents into the cache, it is possible to calculate the difference of sizes between two documents by calculating the increase in the size of the cache. The size leakage technique could be easily exploited to deduce the preference and the visiting history of users.

All these interactions make Site Isolation policy no longer effective. While blocking all cross-origin requests could solve

the issue, existing websites legitimately utilizing cross-origin resources would similarly be affected by such an approach. Thus, Cross-Origin Read Blocking (CORB) policy has been proposed. It aims to prevent HTTP responses from being loaded into contexts at different origins if the information is deemed confidential. The authors have claimed that this could effectively reduce potential dubious cross-origin resource fetches. Previous examples of XSSi attacks or the CVE-2020-6442 vulnerability become ineffective when a browser implements the CORB policy [20].

A simplified version of the CORB policy implementation in Chromium is shown in Procedure 1. This code is executed as soon as a response is received by the browser. It performs a few initial checks, including whether the scheme is HTTP(S). If these checks are not violated, the response is allowed to be loaded into a context in a different origin (i.e., not blocked). The procedure returns NULL if the response is blocked.

The CORB policy authors defined a set of response MIME types likely related to secrets, namely protected MIME types. The response having Content Type header value as a protected MIME type is blocked. For instance, responses with Content Type headers related to images would not be blocked, yet responses with Content Type headers related to JSON are blocked because web developers commonly use JSON serialized responses to conduct communication between frontend and backend.

Chromium team took a different approach to implement CORB. Instead of strictly following the policy documented at W3C [21], the team added extra measures to confirm the MIME types by inspecting the response content [22]. This measure is known as “confirmation sniffing”. They claimed that this could effectively reduce false positives (i.e., reduce the cases when a legitimate response is blocked), thus increasing the compatibility of Chromium with more web applications [5]. For instance, as seen in Lines 6 and 7 in Procedure 1, if the response MIME type is related to JSON, which is in the protected MIME type list, but the content in the response is an image, not a JSON, then Chromium follows the property of the content and does not block. On the other hand, Webkit strictly follows the policy and blocks the response since it does not have such a measure [23].

III. BROWSER POLICY FUZZING

In this section we present CorbFuzz, which is a fuzzing technique for checking browser security policies.

A. Fuzzing Algorithm

We present the architecture of CorbFuzz in Figure 1 and its algorithm in Procedure 2. CorbFuzz is a distributed and multi-threaded fuzzer that loops over all given web applications and calls CORBFUZZTESTONE, which conducts coverage-guided fuzzing for the given web application individually.

Initially, CORBFUZZTESTONE creates multiple instances of the application runtime instrumented with data synthesis discussed in Section-IV. We define the runtime to be a function $P : (\text{URL}, \text{Seed}) \rightarrow \{\text{Metrics}, \text{R}, \text{M}\}$, where Seed

Procedure 1 Partial CORB Implementation in Chromium

```

1: procedure CORBCHECK(Response)
2:   if Response.Scheme  $\notin$  {HTTP, HTTPS} then
3:     return Response
4:   mime  $\leftarrow$  Response.ContentType
5:   if mime  $\in$  ProtectedMimeTypes then
6:     if mime  $\in$  JSON  $\wedge \neg$  IsJSON(Response) then
7:       return Response
8:     if mime  $\in$  XML  $\wedge \neg$  IsXML(Response) then
9:       return Response
10:  else
11:    return NULL
12:  return Response

```

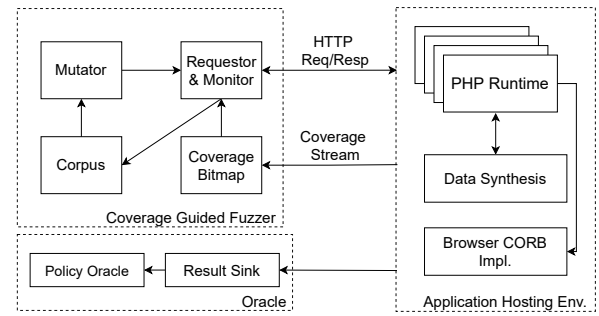


Fig. 1. CorbFuzz Architecture

is an identifier mapping to a state of the web application (i.e., database, cookies, and sessions), Metrics represents the coverage metrics, R represents the resource queried by the web application, and M maps each CORB implementation to its decision on whether to block the response. Analogous to a pipeline, the HTTP requests are first passed to the runtime hosting web application and HTTP responses generated are then served as inputs for different CORB implementations.

In CORBFUZZTESTONE, a bitmap is created so as to record the coverage (Procedure 2, Line 6). CORBFUZZTESTONE additionally declares a result sink (Line 7) for storing the information required by the oracle. The details of the oracle are elaborated in the following sections. A corpus (Line 8) is also defined as a list of pairs, where each pair contains the URL of the request and the seed. Additionally, a list is declared (Line 9) for storing the URLs extracted from the HTTP response (e.g., href values and API calls). During fuzzing in CORBFUZZTESTONE, it randomly selects an input from either corpus or unvisited links extracted (Line 9). The input contains a request URL and potentially a seed mapping to a state. Then, CORBFUZZTESTONE mutates and sends the corresponding HTTP request and seed to the runtime (Lines 10, 12). If the input leads to increased coverage, it is added to the corpus (Lines 14-15).

After the fuzzing terminates (i.e., ShouldTerminate() returns true), the oracle aggregates the information in result sink and provides a decision for each HTTP response. These decisions

Procedure 2 CorbFuzz Algorithm

```
1: procedure CORBFUZZ(TestBench)
2:   for WebApplication  $\in$  TestBench do
3:     CORBFUZZTESTONE(WebApplication)
4:   procedure CORBFUZZTESTONE(WebApplication)
5:      $P \leftarrow \text{DataSynthesis}(\text{WebApplication})$ 
6:     CovBitMap  $\leftarrow \text{BitMap}()$ 
7:     ResultSink  $\leftarrow \text{HashTable}()$ 
8:     Corpus  $\leftarrow \text{List}[\text{Pair}]()$ 
9:     NewURL  $\leftarrow \text{List}()$ 
10:    Visited  $\leftarrow \text{Set}()$ 
11:    for  $\neg \text{ShouldTerminate}()$  do
12:      U, Seed  $\leftarrow (\text{Corpus} \cup (\text{NewURL} - \text{Visited})).\text{Pop}()$ 
13:      U, Seed  $\leftarrow \text{Mutate}(\text{U}, \text{Seed})$ 
14:      Visited  $\stackrel{+}{\leftarrow} \text{U}$ 
15:      Metrics, R, M  $\leftarrow P(\text{U}, \text{Seed})$ 
16:      NewURL  $\stackrel{+}{\leftarrow} \text{ExtractLinks}()$ 
17:      if IsNewCoverage(CovBitMap, Metrics) then
18:        Corpus  $\stackrel{+}{\leftarrow} \text{U}, \text{Seed}$ 
19:        CovBitMap  $\stackrel{+}{\leftarrow} \text{Metrics}$ 
20:      if IsUniqueResponse() then
21:        ResultSink  $\stackrel{+}{\leftarrow} (\text{R}, \text{M})$ 
22:    RunOracle(ResultSink)
```

are compared with the browser decisions to identify potential weaknesses.

B. Policy Oracle

To define a policy oracle (i.e., RunOracle function in Procedure 2, Line 22), we need to categorize the response as confidential or non-confidential by evaluating resource accesses. We limit the scope of resources to be only provided by the database for this work. We use a method similar to Pellegrino et al. [24] which deduces confidentiality of a resource by observing resource access frequency. After fuzzing terminates for each web application, we aggregate and count the number of resources accessed by each database query executed while handling each request. In our implementation, we use the average number of accesses as our threshold. If any query uses resources that have a frequency below the threshold, the oracle infers that the query is accessing a confidential resource, of which the response should be blocked, and checks whether the CORB implementation blocked it.

The granularity of the resource impacts the result of the oracle. For example, if each resource is considered as a table, oracle is more likely to decide to block the response than if each individual resource is considered as a row in the table. Hence, using a coarse-grain resource definition is more likely to produce false positives. We designed two types of oracles with different resource granularity. To reduce false positives, one oracle considers each unique row (i.e., query constraints) to be a resource, and to reduce false negatives, the other oracle considers each table to be a resource.

C. Coverage Metrics

In CORBFUZZTESTONE, rather than focusing on test coverage for the CORB function of the browser implementation, fuzzing is guided by test coverage for the given web application. While the coverage information of CORB function may enhance the fuzzing in regards to efficiency, it would not be useful since CORB function is a small piece of code in both WebKit and Chromium. Thus, it is easy to achieve high test coverage for the CORB function while focusing on achieving high test coverage for the given web application. Additionally, we are evaluating the policy for different code patterns. Focusing on what CORB is able to handle would not lead to identifications of potential weaknesses in the implementation.

IV. DATA SYNTHESIS

In this section, we discuss our data synthesis techniques that enable us to handle data-dependent web applications automatically during fuzzing (i.e., during the execution of CORBFUZZTESTONE), without the need for manual set up of fuzzing targets. Instead of querying the database, the data synthesis approach translates the query to constraints and generates the respective data. Additionally, the data synthesis approach generates results for comparisons involving sessions or cookies so as to achieve higher test coverage and bypass authentication.

The data synthesis workflow depends on a seed that is generated and tracked by CORBFUZZTESTONE, first mentioned in Procedure 2, Line 13. A seed is a 32-bit integer sampled from a uniform distribution over $[0, 2^{32} - 1]$. There is a bijection between the seed and the state of the database and a weak bijection between the seed and the cookies or sessions.

A. Query Constraint Extraction

We first discuss the handling of database queries. The results generated for a specific database query are constrained by three measures: row count, table architecture, and constraint that describes the resulting rows and columns from the query. Most open-source web applications either do not include a table schema or require laborious work to set up the tables. Thus, we assume that the table schema is not given, and the generation of the database query result is run without the knowledge of the table architecture. For these, we respectively define three functions: MAXROW, FIELDS, CONSTRAINT. The input of all these functions is a relational algebra expression translated from the query.

MAXROW provides an estimation of the maximum rows of the query result. It is implemented by considering the set operators and LIMIT.

To reconstruct the table schema, the data synthesis approach learns from the query by observing the field names used inside it. We define the FIELDS function, which produces a set of pairs representing fields returned by the query. The first part of the pair indicates the table name, and the second part is the name of the field. The function is implemented by tracking the rename, projection, and select operators. In the case that

```

1 <?php
2 $conn = mysqli_connect (...);
3
4 $res = $conn->query(
5     "SELECT_*_FROM_A_WHERE_A.c_=_1"
6 );
7
8 $x = $res->fetch_assoc();
9
10 $a = $x["a"];
11
12 if ($a == 0) echo 1;

```

Fig. 2. Example of PHP Application Database Call

a wildcard projection (i.e., asterisk) is used, the function only returns the fields used throughout the relational algebra expression, which could be a subset of fields returned if the query is executed on the correct table schema. The missing fields are addressed by FIELD procedure in Procedure 3.

To ensure that the response generated using data synthesis can be reproduced in the web application with real database settings, we additionally extract the constraints from the query and generate a consistent result that conforms to these constraints. For this, we define CONSTRAINTS function, which outputs all the row-based and column-based constraints in the relational algebra expression for the SMT solver. We utilize a subset of translation rules proposed by Veanes et al. [25]. Note that this function also assigns types to fields if the field is compared with a concrete value in the select operator or returned by set functions like *COUNT*.

We provide an example for the query in Line 5 of Figure 2 as input. The relational algebra expression for the query is $\text{SELECT}_{A.c=1}(A)$. Since there is no *LIMIT* operation inside the query, the MAXROW outputs that the maximum line is infinite. The FIELDS function produces a set with one pair: $\{\langle A, c \rangle\}$. The CONSTRAINTS function translates the condition in the select operator to the SMT formula: $(= A\#c\ 1)$ and assigns $\langle A, c \rangle$ to be of integer type.

The crucial procedures for the generation workflow are presented in Procedure 3. Before fuzzing starts in CORBFUZZTESTONE (Line 5, Procedure 2), the INITIALIZATION procedure is executed. This procedure initiates three global hashtables for caching. These are preserved throughout the runtime lifecycle and synchronized throughout all runtimes (since we use multi-threaded distributed fuzzing, this is necessary).

When a query is sent to the database, and the web application is waiting for the response, ADD procedure replaces the original code for sending the query and receiving the response from the database. ADD procedure takes two arguments: the query and the seed. If the cache contains the previous solution for the query and the seed, the cached result is returned. Otherwise, the query is parsed into relational algebra to extract constraints, fields, and maximum length (as mentioned before), and an empty hashtable is returned. The hashtable, regardless of whether there is a cache hit, is tracked and used by the web

application as the output of the database query.

Procedure 3 Database Query Result Generation Algorithm

```

1: procedure INITIALIZATION
2:   ConcreteResults  $\leftarrow$  HashTable()
3:   Types  $\leftarrow$  HashTable()
4:   Cache  $\leftarrow$  HashTable()
5: procedure ADD(Query, Seed)
6:   if Cache(Query) = NULL then
7:     ra  $\leftarrow$  Parse(Query)
8:     Cache(Query).L  $\leftarrow$  MAXROW(ra)
9:     Cache(Query).F  $\leftarrow$  FIELDS(ra)
10:    Cache(Query).C  $\leftarrow$  CONSTRAINTS(ra)
11:   results  $\leftarrow$  ConcreteResults(Query, Seed)
12:   return Tracked(results)
13: procedure FIELD(Query, Seed, Name)
14:   cache  $\leftarrow$  Cache(Query)
15:   r  $\leftarrow$  ConcreteResults(Query, Seed)
16:   if Name  $\notin$  cache.F then
17:     Types(Query, Name)  $\leftarrow$   $\tau$ .AssignWeight(0)
18:   r.F  $\leftarrow$  cache.F  $\stackrel{+}{\leftarrow}$  Name
19:   for field  $\in$  r.F do
20:     if field.Type = NULL then
21:        $\tau \leftarrow$  Types(Query, field)
22:       field.Type  $\leftarrow$  Sample( $\tau$ )
23:   len  $\leftarrow$  Seed % cache.L
24:   r  $\leftarrow$  Solve(cache.C  $\cup$   $\neg$  cache.Solved(r.F, len), len)
25:   if r = UNSAT then return Abort()
26:   cache.Solved(r.F, len)  $\stackrel{+}{\leftarrow}$  r
27:   return Tracked(r(Name))
28: procedure NOTIFY(Query, Name, IType)
29:    $\tau \leftarrow$  Type(Query)(Name)
30:    $\tau(IType) \leftarrow \tau(IType) + \text{Weight}$ 

```

If the tracked hashtable is searched in the later executions of the web application and the searched key corresponds to NULL value, the FIELD procedure is called. In addition to the query and the seed, this procedure takes an additional argument: the name of the field (i.e., the key of the hashtable that the application is searching for). FIELD procedure assumes that the web application code is correct and the queried field must exist. Under the circumstance that this specific field name is not inferred from the SQL query statement (e.g., a wildcard select), FIELD appends the field name to the global cache so that in the future, for this query, this specific field would be considered. Before solving the constraints generated from the evaluation of the query, FIELD first probabilistically selects a type from all possible data types for each field, which is discussed in Section B. The number of rows is generated using the seed value. To avoid generating an identical result, FIELD appends constraints stating that the result to be solved should not be equal to previously generated results under the same condition (i.e., same type and same amount of row). If the solver concludes these constraints could derive no result (i.e., UNSAT), the web application immediately returns an internal

error to abort the data synthesis workflow. However, this case rarely happens in our experiments because constraints for SQL queries are very permissive. The returned value of `FIELD` procedure is also tracked for type inference purposes, which is described in the following section.

We demonstrate an example for the workflow over the PHP application code listed in Figure 2. Before the execution of any code, as soon as the runtime starts, the `INITIALIZATION` procedure is called. Then, on Line 2, the code calls `mysqli_connect` to establish a connection to MySQL database. Inside the runtime, this function is replaced with a dummy method that always acts as if there is a successful connection. Then, the code is executed to send a query to MySQL database (Line 4) and wait for the response (Line 8). Instead of sending the query, the runtime calls the `ADD` procedure. Suppose we are using a new seed, the procedure would evaluate the query and return a traced empty hashtable. On Line 10, the hashtable is searched with a key `a`. Since the hashtable is empty, the key points to `NULL` value. Instead of returning `NULL`, the `FIELD` procedure is called to solve for all the fields, including the field searched by the application.

B. Type Inference

The knowledge of field names is not enough to generate the data. Correct type of each field is also required for generating a consistent result. Note that for types here, we are not referring to the actual type of a concrete value. Instead, we are referring to the inherent types. The inherent type is the same after type juggling. Suppose an integer is cast as string in the application, we do not record this as string but instead as integer. Indeed, all fields in the result from the call `mysqli_query` are cast as string, regardless of what the type is attributed to each of them in the table schema. Yet, they are directly used as their inherent type throughout the execution in web applications, which is made possible by type juggling. Hence, for data synthesis purposes, we need to infer the inherent types but not the actual types.

We consider type information crucial because an inaccurate type makes web applications prone to producing errors and unrealistic responses. For instance, deserializing an integer or integer-like object would inevitably lead to errors. Another example is that using a string as an index for an integer-indexed array does not lead to error but breaks the original logic of the web application. This situation is unwanted in this context because it produces a spurious response that is not reproducible in an actual run of the web application using the real table schema.

In the query, we could gain type information for fields when the operations processing or generating the field are known and the argument or return types are well-defined. This is because type juggling in SQL would lead to an error or warning. For instance, a comparison between a field and an integer would help us conclude that the field type must be integer. However, it is impossible to infer all types from evaluating queries. Thus, we additionally infer the type of fields by the information during the execution. Specifically, data synthesis

runtime collects type information via two methods. First, if the field encounters the binary comparison operand, CorbFuzz records the type of the concrete value it is comparing to. Second, CorbFuzz tracks the internal functions that the field is served as an argument. Internal functions typically have a clear definition of the types of each argument. For simplicity, CorbFuzz ignores corner cases like comparison between two fields and passing to an internal function supporting all types. Future work may leverage Hindley-Milner algorithm [26] to construct a more fine-grained typing system.

Still, the runtime analysis is not enough for inferring types of all fields. Some of them may not be passed to an internal function or used in comparisons. Additionally, comparison between variables of different types is allowed, and it is impossible to deduce the inherent type of a concrete value. These factors mean there is a possibility that a different type is used against the compared variable. To accommodate these cases, we define a domain of types (τ) for each field and assign a weight to any type $t \in \tau$. At initialization, each t is set with an initial weight and increased whenever it matches inference (e.g., passed to an internal function), which we refer to as a type hint, after the generation of the result. If query analysis has already assigned a type, then the type would have infinite weight in τ . Before constraint solving is initiated, CorbFuzz conducts a probabilistic sampling from τ for each field based on weights assigned to types (the probability of a type to be chosen is proportional to the weight of the type). Due to probabilistic selection, a variety of types are explored during fuzzing. Here, we assume that if a type for a variable is not intended, then this incorrect type used would lead to either errors or no effect on analysis. In general, CorbFuzz tries to increase the likelihood that a correct type will be used.

In Procedure 3, `FIELD` procedure conducts a probabilistic sampling over the τ for each field (Line 19-22). In our implementation, we utilize A-res algorithm [27]. `NOTIFY` procedure is called when the tracked value returned by `FIELD` procedure is used in internal functions or for comparison. The type hint is then used to increase the weight for that type in τ . In our implementation, we only let τ include integers, strings, and booleans. Type hints for types that are not in τ are ignored.

In the example provided in Figure 2 Line 10, after the `FIELD` procedure ends, `$a` is assigned the generated value that is tracked. On Line 12, the tracked value is compared to an integer. The `NOTIFY` procedure is called, adding weight for the integer type for the field `a` in global hashtable `Types`.

C. Authentication Bypass Workflow

Cookies and sessions are commonly leveraged by web applications to make HTTP requests stateful [6], [28], allowing for the implementation of authentication. Both of them could be represented as a hashtable. We observed that there could be a significant increase in coverage for a web application if cookies or sessions are properly set (e.g., an authentication token presents for a specific field). It is because complex logic inside web applications tends to be reached after the request

```

1 <?php
2 session_start();
3
4 if (isset($_SESSION["is_auth"]))
5     echo $_SESSION["welcome_message"];

```

Fig. 3. Example of PHP Application Session Usage

presents to be authenticated or authorized. Usually, cookies or sessions keys and values are compared to a constant or a result from the database. Therefore, using a fuzzer to explore cookies and sessions is largely ineffective since there is a huge search space for the keys and values.

To better explore behaviors of web applications, we generate decisions for comparison operations that involve sessions or cookies. Still, we conduct concolic execution and record the constraints for the decisions made to check whether all decisions made are satisfiable. CorbFuzz treats each item in session or cookie as a pair of symbolic variables: $\langle \phi, \alpha \rangle$, where ϕ is the gated boolean symbolic variable that shows whether the item is defined and α represents the symbolic variable for the item. This method is inspired by hybrid fuzzing but differs from it. The runtime only solves the constraint one time when necessary. That is, if an item of cookies or sessions has not been passed to an operation that does not have an SMT formula translation available, the value would never be generated.

The reason we do not generate the data as soon as it is used is largely due to the use cases of sessions and cookies. They are used in multiple or nested branches, but most of the time, their concrete value would not be evaluated. Additionally, there are very few internal functions that commonly use sessions or cookies as arguments. We have implemented only basic arithmetic and `isset` [29] internal call with the translation of the SMT formula. Still, most requests in our experiment do not require generating the concrete value of sessions and cookies.

We have shown the crucial components for the workflow in Procedure 4 for session, which is identical for cookie. Similar to the previous workflow for database, there is also an initialization procedure that creates a global hashtable for caching. Specifically, `GC` is for storing the mapping between seed and the sessions. Additionally, there is a `START` procedure, which is called before each HTTP request is handled and the variable declared only survives during the lifecycle of that request. The procedure creates a copy of the seed and declares a hashtable for saving the constraints for each session item used during the request.

When an item of sessions is compared with a concrete value, the `DO` procedure is used before evocation of the original comparison handler. CorbFuzz first checks whether there is already a cached item for the given seed (Line 7). If there is a cache hit, then the item is assigned a concrete value, and the internal implementation of the comparison operation is executed. Otherwise, CorbFuzz checks whether the comparison

Procedure 4 Session Generation Algorithm

```

1: procedure INITIALIZATION
2:   GC  $\leftarrow$  HashTable()
3: procedure START
4:   RCache  $\leftarrow$  HashTable()
5:   NewSeed  $\leftarrow$  Copy(Seed)
6: procedure DO(Name, Opline)
7:   if GC(Seed, Name) then
8:     Session(Name)  $\leftarrow$  GC(Seed, Name)
9:     return Next()
10:  if Opline.Operand  $\in$  ImplementedOp then
11:    decision  $\leftarrow$  NewSeed & 1
12:    ShiftRight(NewSeed)
13:    cons  $\leftarrow$  ToConstraint(Opline, decision)
14:    if  $\neg$  IsSAT(RCache(Name)  $\cup$  cons) then
15:      return DO(Name, Opline)
16:    RCache(Name)  $\stackrel{+}{\leftarrow}$  cons
17:    return decision
18:  else
19:    solved  $\leftarrow$  Subset(GC(*, Name))
20:    cons  $\leftarrow$  RCache(Name)  $\cup$  solved
21:    if  $\neg$  IsSAT(cons) then return Abort()
22:    GC(Seed, Name)  $\leftarrow$  Solve(cons)
23:    return DO(Name, Opline)

```

operation is implemented (Line 10) so that it could convert the decision of the operation to a constraint. If so, a decision is generated from the seed, and the constraint for performing this decision is appended to the constraints over that item (Lines 11-13). An SMT solver is then used to check whether the constraint is satisfiable (Line 14). If it is not satisfiable, then the procedure recursively consumes the seed until there is a decision that could be satisfied. Our implementation assumes there are at most 32 decisions since we are using a 32-bit seed. In our experiments, the maximum consumption is only 11 bits in a specific request. The decision is then returned, and the internal implementation of the comparison is ignored. As for the corner case that a session item is compared to another session item, we treat this comparison as an unimplemented operation for one side and then apply the workflow to the other side. When the operation is not implemented, then a concrete value is generated by solving the constraint for that item. To ensure the uniqueness of the concrete value generated, the solver tries to avoid generating already solved values stored in the global cache for that field name. To reduce UNSAT cases, `DO` only selects a random subset of the stored values in the cache and removes them from consideration as the result of solving (Line 19). Note that by doing so, we do not create a strict bijection here between the seed and the sessions. Same sessions may map to multiple seeds. This is because the constraints here are not permissive.

For PHP code listed in Figure 3, when it executes until Line 4, CorbFuzz first declares a pair of symbolic variables $\langle \phi_0, \alpha_0 \rangle$ and makes a decision for the unary comparison

isset based on the seed. Suppose the seed indicates the decision is to return true, then the constraint $\phi_0 = \text{true}$ is added to the set of constraints for the `$_SESSION["is_auth"]`. Note that this session item is not used later, so its concrete value is never generated. Then, on Line 5, another session item is used. We have not implemented echo function and the value of `$_SESSION["welcome_message"]` is generated with respect to its constraints (i.e., no constraint in this context).

In certain cases, the data stored in the cookies or sessions may be subject to decryption or deserialization in web applications. Before a decryption or decoding function is executed with input from cookies or sessions, the workflow must synthesize the concrete value. This situation is undesirable because authentication cannot be bypassed. A more general version of this issue, which is that symbolic execution fails to model a comprehensive list of syscalls, also plagues hybrid fuzzing [30], and no solution has been proposed so far. We discuss a potential ad-hoc solution and future work to address this problem in Section V-B.

D. Adapting to Other Programming Languages

The aforementioned methods target PHP applications. However, they can be extended to other programming languages that are widely used for web application development. Specifically, the data synthesis workflows embedded with type inference can be applied to other programming languages supporting type juggling, like Perl or JavaScript/NodeJS, or using a dynamic type system, like Ruby and Python. For statically typed programming languages, like Golang or Java, the type inference component would not be needed, but the data synthesis workflows can similarly be adapted.

V. IMPLEMENTATION & EVALUATION

We have implemented the coverage-guided fuzzer and oracle for CorbFuzz in Python with 900 lines of code (LoC) for fuzzing web applications written in PHP. Unlike existing web application fuzzers that only consider responses related to PHP, CorbFuzz considers all responses after a web page is loaded, including images, CSS, and RPC communications. The data synthesis workflow is implemented as an external module with 500 LoC in C and 1200 LoC in NodeJs for PHP. PHP 7.4 has been instrumented to support the workflow and provide branching information for coverage evaluation. To allow for fair evaluation on data synthesis effectiveness, we implement two baseline workflows by removing components inside CorbFuzz.

In the following subsections, we address the following research questions;

- RQ1.** Is data synthesis workflow generating consistent data?
- RQ2.** Can data synthesis workflow increase test coverage?
- RQ3.** Can CorbFuzz detect bugs in implementation in existing browsers?

LoC Range	Number of Applications	Average LoC
Less than 1K	15	476.9
Between 1K and 10K	15	3022.5
Between 10K and 100K	6	43075.5
More than 100K	3	250875.5

TABLE I
TOTAL LOC STATISTICS FOR FUZZING TARGETS

A. Experimental Setup

1) *Environment*: We evaluate CorbFuzz on two Intel Xeon Phi 7210 (64 cores) nodes. Both nodes use Ubuntu 20.04 with one node running NGINX [31] for serving web application on the instrumented PHP environment and other node running the coverage-guided fuzzer.

2) *Targets*: We evaluate CorbFuzz with two popular web browsers: Chromium and WebKit (Safari). Chromium has already added CORB into its current stable release. We implement a test harness based on the Chromium shared library containing the CORB implementation. For WebKit, the developers have created a pull request for CORB implementation but it has not yet been merged into the main branch. Since its implementation is relatively simple and straightforward, we directly translate it into Python to implement a test harness for CORB implementation of Webkit.

3) *Web Applications*: Web applications are fuzzed to provide responses as input for browser policy test harnesses. We crawled 300 repositories containing PHP code from GitHub between March 2nd, 2021 and April 10th, 2021. The repositories are filtered out if they do not contain `index.php` or `index.html`. For simplicity, we do not consider applications that require downloading dependencies with Composer [32], a dependency management tool. The number of remaining applications is 58 with varying LoCs. We fuzz the policies with these 58 applications but for the sake of evaluation of data synthesis effectiveness, we only use 39 of them, for which CorbFuzz reports existence of branches and utilization of databases. The statistics of these applications are presented in Table I.

B. Data Synthesis Effectiveness

To evaluate the data synthesis approach and address **RQ1**, we ran CorbFuzz with and without type inference for three minutes¹ with each web application. We compared the percentage of correct type generations by tracking whether the generated value matches the type of (1) a concrete value when compared to it; (2) the defined argument when used to call internal function. Figures 4 and 5 demonstrate the percentage of correct generations for comparison statements and internal function calls respectively. Figure 4 shows that for 10 applications, CorbFuzz generates the correct type for comparisons more often than CorbFuzz without type inference with 17% more data generations with correct type on average. Figure 5 shows that for 11 applications, CorbFuzz generates the correct type for internal function calls more often than CorbFuzz

¹Due to the randomness feature of the fuzzer, we ran each experiment five times and take maximum value (e.g., edge coverage).

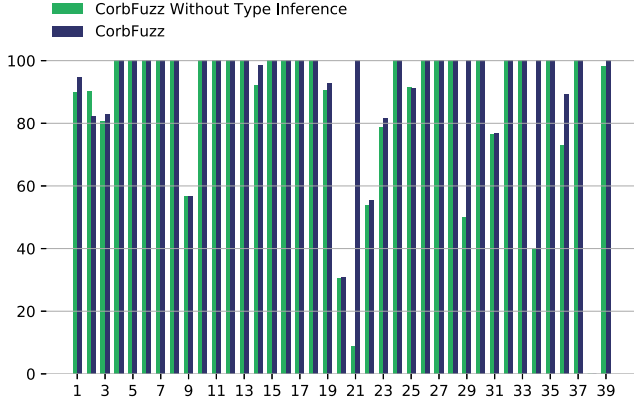


Fig. 4. The percentage of correct type generation for comparison statements for CorbFuzz and CorbFuzz without Type Inference for 3 minutes of fuzzing. X axis denotes the web application ID.

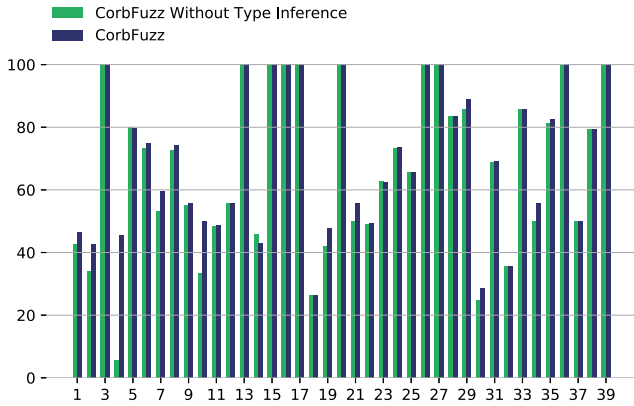


Fig. 5. The percentage of correct type generation for internal function calls for CorbFuzz and CorbFuzz without Type Inference for 3 minutes of fuzzing. X axis denotes the web application ID.

without type inference with 5% more data generations with correct type on average.

On some applications, CorbFuzz has little improvement on the accuracy of type generation because in the results we show, we consider all type violations. However, many of these type violations are due to developers using type juggling and not due to data synthesis. Therefore, these violations cannot be removed by improving type inference in data synthesis.

We also evaluated the impact of data synthesis on fuzzing effectiveness and address **RQ2**. Figure 6 demonstrates the edge coverage difference, in terms of percentages, of edge counts between CorbFuzz without any type inference and authentication bypass, and CorbFuzz. For this evaluation, we only chose applications containing more than one branch. Figure 6 shows that for almost all applications, the inclusion of type inference and authentication bypass improves coverage. The average number of edges covered is 16.2 edges for CorbFuzz without type inference and authentication bypass

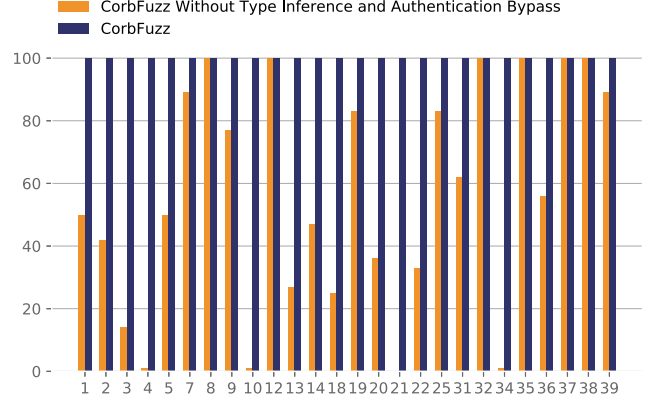


Fig. 6. The percentage of edges covered for CorbFuzz without Type Inference and Authentication Bypass in 3 minutes of fuzzing against edges covered by CorbFuzz. X axis denotes the web application ID.

and 27.5 for CorbFuzz. These results demonstrate that with the inclusion of type inference and authentication bypass, we can cover on average 70% more edges, which shows the effectiveness of our data synthesis approach.

The number of edges covered is low for some applications because these applications (e.g., WordPress) save and use structural, encrypted, or serialized data from the database or cookie. The data synthesis workflow is unaware of the structural property of any field. Therefore, it generates a large amount of data that can not be deserialized or decrypted by the web application, so CorbFuzz fails to explore these web applications. However, we recognize that this can be prevented by enlarging the domain of type τ defined. By considering the common structural properties as types (e.g., JSON type) and instrumenting deserialization libraries to provide type hints, future work could implement an approach that is able to further improve web application coverage.

C. Detected CORB Weaknesses

In evaluation of **RQ3**, we have discovered three common classes of code patterns, which are discussed in following sections, that cause the CORB implementation in Chromium to not function as expected. One of the cases has been filed in the Chromium bug tracker before our discovery by a Chromium developer and is still in discussion². We have reported another case³, which has later been resolved by a patch in the CORB component of Chromium⁴.

All the weaknesses discovered are due to the novel fuzzing approach we present in this paper. The weaknesses are not present in every web application but only in a few of them. Fuzzing a single web application would likely not lead to the discovery of any of the weaknesses, while fuzzing multiple web applications without the data synthesis would require

²<https://crbug.com/795470>

³<https://crbug.com/1148397>

⁴<https://chromium-review.googlesource.com/c/chromium/src/+2596879/>

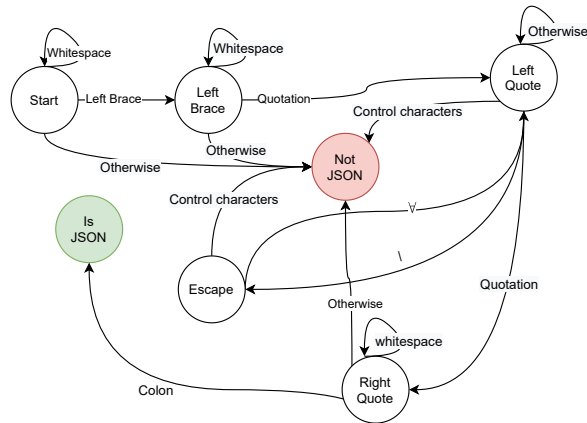


Fig. 7. Finite State Machine for Validating JSON

setting up the database and seeding the tables for numerous web applications, which is infeasible. Data synthesis allows us to effortlessly fuzz multiple web applications by symbolically evaluating database queries. Additionally, all the web application states that lead to the aforementioned weaknesses are under some extent of authentication or authorization. Without a manual definition of the login method for a web application, the fuzzer would not discover these weaknesses. Yet, manually defining authentication or authorization method for a considerable number of applications is tedious and unrealistic. In contrast to the manual approach, the data synthesis approach we present automatically generates appropriate sessions and cookies items, which allows exploration using authorized requests.

Serialized Array as JSON Response. In Chromium, if the response MIME type is related to JSON, CORB would check the response content to learn whether it is indeed JSON. A finite state machine (FSM) conducts such a check. As illustrated in Figure 7, the FSM does not comprehensively parse the response content to perform the check. Instead, it only checks whether the content has a left brace at the beginning and has matching quotes for the first key to determine if the content is JSON.

As permissive as it is, such a check would not identify a serialized array in JSON format, which is considered a JSON object inside the JavaScript runtime of Chromium. Indeed, the latest JSON specification (RFC 8259 [33]) refers this to be a different type from JSON object known as JSON array. For instance, for a simple response as `[1, 2, 3]`, a JSON array, JSON check in CORB implementation would first look for the left brace. Yet, the first character is left bracket, which makes the FSM classify the content as not JSON. However, `fetch`, `XMLDocument`, and `JSON.parse` APIs in JavaScript runtime parse the content into a JavaScript object without warning.

Sending JSON arrays as responses is commonly seen in web application APIs. The responses of these APIs would likely carry sensitive information. Thus, we consider catching JSON

array for JSON MIME type in CORB implementation to be a reasonable patch.

Malformed JSON Response. It is not uncommon for web application developer to adopt the following code pattern, where `$var` represents any variable the attacker can control (i.e., a tainted variable), which could be achieved through methods including URL manipulation and security-unrelated CSRF [34].

```

1 <?php
2 header('Content-Type: application/json');
3 echo "{\"$var\": \"\$secret\"}";

```

This code pattern does not leverage the existing serialization library. Instead, it produces serialized objects by direct string concatenation and manipulation. If the attacker is able to control at least one character in the first key of a JSON object, they would be able to bypass the CORB check by making that key contain a control character. According to JSON specification, control characters (U+0000 through U+001F) inside key and value of JSON object should be escaped (i.e., append a reverse solidus before the control character). Similarly, the JSON verifier inside CORB implementation in Chromium follows this pattern and rejects all JSON objects with unescaped control characters on the first key.

Consider the PHP code shown above. If we set `$var` to be the control character `\u0017`, the resulting response becomes `{"\u0017": "[SECRET]", [MORE SECRETS]}`. The JSON checker FSM enters the state “Left Quote” after encountering the first and second characters. It then compares character `\u0017` to control character range and identifies it as an unescaped control character, misclassifying the response as not JSON.

We consider this weakness should be addressed as the existence of such a code pattern is not negligible. We have reported this to the Chromium team, and it has been fixed by removing the check for control character inside the JSON checker.

The discovery of this weakness is only possible if the fuzzer can mutate the database state efficiently. In all the cases that lead to this malformed JSON response, the variable as the key of the JSON array is retrieved from the database. CorbFuzz symbolically evaluates the database queries and synthesizes the concrete value for that variable, which allows the efficient exploration of the domain of this variable (i.e., exploring interesting values for UTF-8 character). In contrast, a conventional fuzzing approach has to mutate the database state by sending requests to the web application or by resetting the database, leading to significantly larger search spaces and high timing overhead for mutating the content of the variable derived from the database.

Confirmation Sniffing. In most web applications, warnings and errors in plaintext or HTML are directly prepended to the response. For PHP, a warning in HTML is generated whenever an undefined behavior happens. If a malicious actor is able to trigger an undefined behavior in responses that are checked with confirmation sniffing, then CORB in Chromium could

be bypassed since the responses start with data that is not of their MIME type.

This weakness, including all previous weaknesses, could be considered as the side effect of increase in permissiveness caused by confirmation sniffing. We consider that confirmation sniffing is harming the effectiveness of the CORB implementation in Chromium. Future work, on the other hand, could work on testing the contribution of confirmation sniffing on compatibility and conclude whether confirmation sniffing is indeed redundant.

D. Fuzzer Flexibility

We have constructed an oracle for ORB and test the proposed implementation. Our fuzzer is unable to discover any weakness of ORB. It is because ORB applies a whitelist approach to block requests yet CORB uses a blacklist, which means ORB is much less permissive than CORB. Future work could apply similar approach to evaluate its compatibility.

VI. RELATED WORK

Coverage-guided Fuzzing. Coverage-guided fuzzing has been used to find bugs in different types of programs, such as virtual machines [35], web browsers [36], [37], network functions [38], [39], and operating systems [40], [41]. The state-of-the-art implementations are AFL [42] and libFuzzer [43]. In this paper, we leveraged coverage-guided fuzzing to explore responses from web applications for browser security policy checking. Yet, our approach is not using the coverage of the browser but is instead guided by the coverage of the web applications. Our approach also conducts a series of coverage-guided fuzzing with different targets instead of fuzzing an individual program.

Browser Fuzzing. Domato [44], Dharma [45], and Freedom [36] are all specialized fuzzers used to discover memory-related vulnerabilities and assertion violations in DOM implementation of browsers. They generate structural data that contain valid HTML, CSS, and DOM-related JavaScript for browsers to render. Fuzzilli [37] and Jsfunfuzz [46] are fuzzers for discovering vulnerabilities in JavaScript engines, which utilize a similar approach to generating structural data. Our work is different from all these approaches since the oracle of CorbFuzz is defined based on the property of the web applications, and CorbFuzz does not generate the test cases but instead utilizes web applications' responses. Roy et al. [47] fuzzes web applications and supplies responses to browsers to detect visual inconsistencies between browsers. It is similar to our work in the sense that both works treat web applications and browsers together as a black box. Unlike their work which focuses on testing web applications, our work focuses on testing security policies in browsers. We also do not cross-reference between browsers but use an oracle instead.

Web Application Testing. Alshahwan et al. [7] and Biagiola et al. [8] propose search-based approaches to testing web applications. Both works use metaheuristic approaches such as genetic algorithms to explore and generate different inputs to extensively test web applications. Different from our work,

[7] requires the input types and login information. [8] requires Page Objects to be provided to test the web application. Our work instead avoids manual analysis through data synthesis. Elbaum et al. [48] proposes that web application testing should mutate the sessions and provides a few mutation techniques that could help achieve better coverage. Data synthesis in our work is different than the work of Elbaum et al. since we do not mutate the sessions but instead symbolically evaluate or generate them. Apollo [49] and Wassermann et al. [50] leverage concolic testing to increase coverage of web applications and to discover vulnerabilities. Session generation workflow in our data synthesis approach is utilizing concolic execution, but it is fundamentally different than the concept of concolic testing.

VII. CONCLUSION

We have created a browser policy fuzzer CorbFuzz which uses web application responses to fuzz the browser security policies. To avoid setting up the web applications manually, we proposed a web application runtime that synthesizes data. The resources queried by the web application are either generated or symbolically represented. We have shown that the data synthesis approach not only generates consistent data but also increases test coverage for web applications. We have evaluated CorbFuzz on CORB implementations of Chromium and WebKit as well as ORB proposal for Firefox. By fuzzing with 58 applications, we discovered three classes of weaknesses in CORB implementation of Chromium.

REFERENCES

- [1] V. Hailperin, "Cross-Site Script Inclusion." [Online]. Available: <https://www.scip.ch/en/?labs.20160414>
- [2] J. Grossman, S. Fogie, R. Hansen, A. Rager, and P. D. Petkov, *XSS attacks: cross site scripting exploits and defense*. Syngress, 2007.
- [3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher et al., "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.
- [4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, 2018, pp. 973–990. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>
- [5] "Cross-Origin Read Blocking (CORB)." [Online]. Available: https://chromium.googlesource.com/chromium/src/+master/services/network/cross_origin_read_blocking_explainer.md
- [6] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of web application testing—a survey of recent advances," *Information Systems*, vol. 43, pp. 20–54, 2014. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0306437914000271>
- [7] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 3–12.
- [8] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*. Springer, 2017, pp. 18–32.
- [9] J. Schwenk, M. Niemietz, and C. Mainka, "Same-origin policy: Evaluation in modern browsers," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 713–727. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk>

- [10] B. Gulmezoglu, A. Zankl, T. Eisenbarth, and B. Sunar, "Perfweb: How to violate web privacy with hardware performance events," *Computer Security – ESORICS 2017*, p. 80–97, 2017.
- [11] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," *2012 IEEE Symposium on Security and Privacy*, 2012.
- [12] H. Kim, S. Lee, and J. Kim, "Inferring browser activity and status through remote monitoring of storage usage," *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.
- [13] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," *2014 IEEE Symposium on Security and Privacy*, 2014.
- [14] R. Spreitzer, S. Griesmayr, T. Korak, and S. Mangard, "Exploiting data-usage statistics for website fingerprinting attacks on android," *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016.
- [15] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.
- [16] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: <http://arxiv.org/abs/1801.01203>
- [17] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 1661–1678.
- [18] "Site Isolation - The Chromium Projects." [Online]. Available: <https://www.chromium.org/Home/chromium-security/site-isolation>
- [19] "CVE - CVE-2020-6442." [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6442>
- [20] "Issue 1013906: Security: expose stored (in cache) cross-site response's size." [Online]. Available: <https://bugs.chromium.org/p/chromium/issues/detail?id=1013906>
- [21] "Fetch Standard." [Online]. Available: <https://fetch.spec.whatwg.org/#corb>
- [22] "MIME Sniffing Standard." [Online]. Available: <https://mimesniff.spec.whatwg.org/>
- [23] "185331 – Cross-Origin Read Blocking (CORB)." [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=185331
- [24] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting csrf with dynamic analysis and property graphs," 2017.
- [25] M. Veanes, J. d. Halleux, N. Tillmann, and P. de Halleux, "Qex: Symbolic sql query explorer," Tech. Rep. MSR-TR-2009-2015, October 2009, updated January 2010. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/qex-symbolic-sql-query-explorer/>
- [26] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dec. 1978. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/0022000078900144>
- [27] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, Mar. 1985. [Online]. Available: <https://dl.acm.org/doi/10.1145/3147.3165>
- [28] M. Johns, S. Lekies, B. Braun, and B. Flesch, "Betterauth: Web authentication revisited," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 169–178. [Online]. Available: <https://doi.org/10.1145/2420950.2420977>
- [29] "isset - php manual." [Online]. Available: <https://www.php.net/manual/en/function.isset.php>
- [30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, vol. 16, 2016, pp. 1–16.
- [31] "NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy." [Online]. Available: <https://www.nginx.com/>
- [32] "Composer." [Online]. Available: <https://getcomposer.org/>
- [33] "The javascript object notation (json) data interchange format." [Online]. Available: <https://tools.ietf.org/html/rfc8259>
- [34] A. Shankar (D1r3Wolf), "Chaining No impact(N/A) Bugs to get High impact." [Online]. Available: <https://blog.d1r3wolf.com/2020/04/chaining-no-impactna-bugs-to-get-high.html>
- [35] T. Brennan, S. Saha, and T. Bultan, "Jvm fuzzing for jit-induced side-channel detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1011–1023. [Online]. Available: <https://doi.org/10.1145/3377811.3380432>
- [36] W. Xu, S. Park, and T. Kim, "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event USA: ACM, Oct. 2020, pp. 971–986. [Online]. Available: <https://dl.acm.org/doi/10.1145/3372297.3423340>
- [37] "Fuzzilli - A JavaScript Engine Fuzzer," Apr. 2021, original-date: 2019-03-20T15:32:47Z. [Online]. Available: <https://github.com/googleprojectzero/fuzzilli>
- [38] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, ser. NetAI'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–7. [Online]. Available: <https://doi.org/10.1145/3341216.3342206>
- [39] C. Shou, "Porkfuzz: Testing stateful software-defined network applications with property graphs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1660–1662. [Online]. Available: <https://doi.org/10.1145/3468264.3473487>
- [40] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville Ontario Canada: ACM, Oct. 2019, pp. 147–161. [Online]. Available: <https://dl.acm.org/doi/10.1145/3341301.3359662>
- [41] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: <https://www.ndss-symposium.org/wp-content/uploads/2020/02/24018.pdf>
- [42] lcamtuf, "American Fuzzy Lop." [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- [43] K. Serebryany, "libFuzzer, a library for coverage-guided fuzz testing," *LLVM project*, 2015.
- [44] "Domato - DOM fuzzer," Apr. 2021, original-date: 2017-09-21T15:28:59Z. [Online]. Available: <https://github.com/googleprojectzero/domato>
- [45] "Dharma - Generation-based, context-free grammar fuzzer." Apr. 2021, original-date: 2015-03-25T17:56:23Z. [Online]. Available: <https://github.com/MozillaSecurity/dharma>
- [46] "A collection of fuzzers in a harness for testing the SpiderMonkey JavaScript engine." Apr. 2021, original-date: 2015-07-08T01:05:26Z. [Online]. Available: <https://github.com/MozillaSecurity/funfuzz>
- [47] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-pert: a web application testing tool for cross-browser inconsistency detection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 417–420.
- [48] S. Elbaum, S. Karre, and G. Rothermel, "Improving web application testing with user session data," in *25th International Conference on Software Engineering*, 2003. *Proceedings.*, 2003, pp. 49–59.
- [49] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 261–272. [Online]. Available: <https://doi.org/10.1145/1390630.1390662>
- [50] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 249–260. [Online]. Available: <https://doi.org/10.1145/1390630.1390661>