CorbFuzz: Checking Browser Security Policies with Fuzzing

Anonymous Author(s)

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 persistent layer 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) policies on web applications collected from GitHub and found three classes of weaknesses in Chromium's implementation of CORB.

ACM Reference Format:
Anonymous Author(s). 2021. CorbFuzz: Checking Browser Security Policies with Fuzzing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Web browsers allow users to do a variety of 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 [23], cross-site scripting [21], Spectre [31], and Meltdown [35], malicious websites can access sensitive information that they should not have access to. Due to the aforementioned threats, browsers have adopted an increasing number of security policies like Cross-Origin Read Blocking (CORB) Policy [3] 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 application's internal state, nor its codebase, it cannot precisely determine properties of the web application. 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. 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 coverage-guided 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 with the goal of achieving 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 [34]. These web applications are called data-dependent. Fuzzing a data-dependent web application requires manually setting up these data sources, for example populating a database [16, 18]. However, given that we need to use a 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 an execution environment for data-dependent web applications that enables us to automate the process. Instead of manually setting up data sources for all fuzzing targets, our tool automatically synthesizes data store, sessions, and cookies. This approach not only removes the requirement of web applications on its environment but also allows our fuzzer to easily mutate the data, leading to higher coverage.

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The execution environment we propose generates SMT constraints for database queries and sessions or cookies usage. The SMT constraints for a database query encodes the SQL statement and we use an SMT solver to generate data values that are consistent with the query. Those SMT constraints generated for sessions and cookies are used to check feasibility of execution paths of the web application like in symbolic execution.

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 in different languages by providing a simple instrumentation for the target language as discussed in Section 3.

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 before deployment. 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 an execution environment 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 execution.
- *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 2, we present the background on browser precautions. In Section 3, we discuss how we synthesize the data for web applications. In Section 4, we present our fuzzing framework. In Section 5, we evaluate CorbFuzz and describe the detected CORB weaknesses by our tool. In Section 6, we present the related work. In Section 7, we conclude the paper.

2 BACKGROUND

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

2.1 Site Isolation and Information Leakage

Browser information leakage has gained increasing exposure in the last few years. According to the Same-Origin Policy (SOP) [40], one of the fundamental rules in browsers, documents from different origins cannot interact with each other. However, more and more methods have been discovered to conduct cross-origin content leak [22, 24, 26, 27, 33, 43]. Additionally, the discovery of cache-related side-channel vulnerabilities like Spectre[30] and Meltdown[35] worsen the information leaks.

Site Isolation policy [11, 38] has been proposed to counter cross-origin content leak. 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 attack and makes most cross-origin information leakage vulnerabilities in browser no longer exploitable.

2.2 Cross-Origin Read Blocking

While Site Isolation policy removes the possibility of documents in different origins interacting with each other directly, there are still existing ways to inject documents from different origins via interfaces provided by browsers. A possible method 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 second line is to load it as a script.

```
<img src="//a.com/secret" />
<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 [4][7]. 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 the size of the cache. The size leakage technique could be easily exploited to deduce preference and visiting history of users.

All these interactions make Site Isolation 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 this method. 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.

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 that are likely related to secrets, namely protected MIME types. For instance, MIME types related to images would not be blocked, yet MIME types related to JSON are blocked as web developers commonly use JSON serialized response 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[5], the team added extra measures to confirm the MIME types by inspecting the response [9]. 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 [3]. For instance, as seen in Lines 6&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 [1].

Procedure 1 Partial CORB Implementation in Chromium

```
 1: procedure CorbCheck(Response)
 2:     if Response.Scheme ∉ {HTTP, HTTPS} then
 3:         return Response
 4:     mime ← Response.MimeType
 5:     if mime ∈ ProtectedMimeTypes then
 6:         if mime ∈ JSON ∧¬ IsJSON(Response) then
 7:             return Response
 8:         if mime ∈ XML ∧¬ IsXML(Response) then
 9:             return Response
10:     else
11:         return NULL
12:     return Response
```

3 DATA SYNTHESIS

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

3.1 Query Constraint Extraction

We first discuss handling of database queries. The results generated by CorbFuzz 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 table schema or require laborious work to set up the tables. Thus, we assume that 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 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*.

```
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;
```

Figure 1: Example of PHP Application Database Call

To reconstruct the table schema, CorbFuzz 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 function is implemented by tracking the rename, projection, and select operators. The first part of the pair indicates the table name, and the second part is the name of the field. Note that our implementation generates and returns all fields involved in the execution of the query in contrast to what the database with the actual table schema would return. This approach reduces the complexity of implementation and is justified by our assumptions that the developer would never use a field in the application that does not exist. In the case that a wildcard projection (i.e., asterisk) is used, the function only returns the fields used throughout the relational algebra, which could be a subset of fields returned if it is executed on the correct table schema. The missing fields are addressed by Field procedure.

To ensure the response could 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 for the SMT solver. We utilize a subset of translation rules proposed by Veanes et al. [44]. 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 functions with query in Line 5 of Figure 1 as input. The relational algebra of the query is $\text{SELECT}_{\text{A.c} = 1}(\text{A})$. 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: $< \text{A}, \text{c} >$. The Constraints function translates the condition in the select operator to the SMT formula: `(= A#c 1)` and assign $< \text{A}, \text{c} >$ to be of integer type.

The query result generation depends on a seed that is generated and tracked by the fuzzer. The seed is a 32-bit integer sampled from a uniform distribution over $[0, 2^{32} - 1]$ and there is a bijection between the seed and the state of database. The crucial procedures for the generation workflow are presented in Procedure 2. Before execution of a web application, the Initialization procedure is executed after the initialization of the runtime. This procedure initiates a set of 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 as the result. The returned hashtable, regardless of whether there is a cache hit, is tracked.

Procedure 2 Database Query Result Generation Algorithm

```
1:  procedure Initialization
2:      ConcreteResults ← HashTable()
3:      Types ← HashTable()
4:      Cache ← HashTable()
5:  procedure Add(Query, Seed)
6:      if Cache(Query) = NULL then
7:          ra ← Parse(Query)
8:          Cache(Query).L ← MaxRow(ra)
9:          Cache(Query).F ← Fields(ra)
10:         Cache(Query).C ← Constraints(ra)
11:     results ← ConcreteResults(Query, Seed)
12:     return Tracked(results)
13: procedure Field(Query, Seed, Name)
14:     cache ← Cache(Query)
15:     r ← ConcreteResults(Query, Seed)
16:     if Name ∉ cache.F then
17:         r.F ← cache.F $\overset{+}{\leftarrow}$ Name
18:         Types(Query, Name) ← $\tau$.AssignWeight(0)
19:     for field ∈ r.F do
20:         if field.Type = NULL then
21:             $\tau$ ← Types(Query, field)
22:             field.Type ← Sample($\tau$)
23:     len ← Seed % cache.L
24:     result ← Solve(r.Q ∪¬ cache.Solved(r.F, len), r.F, len)
25:     if result = UNSAT then return Abort()
26:     cache.Solved(r.F, len) $\overset{+}{\leftarrow}$ result
27:     return result(Name)
28: procedure Notify(Query, Name, IType)
29:     $\tau$ ← Type(Query)(Name)
30:     $\tau$(IType) ← $\tau$(IType) + Weight
```

If the tracked hashtable is searched in the later executions of the application and the searched key corresponds to NULL, 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), CorbFuzz 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 previous evaluation of the query, CorbFuzz 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, CorbFuzz appends constraints stating that the result should not be equal to previously generated results if they have same type and same amount of row count as the current context. If 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 has not happened in our experiments because constraints for SQL queries are very permissive.

We demonstrate an example for the workflow over the PHP application code listed in Figure 1. Before the execution of any code, as soon as PHP runtime starts, the Initialization procedure is called. Then, on Line 2, the code calls `mysqli_connect` to establish 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 would refer to NULL. Instead of returning NULL, the Field procedure is called to solve for all the fields, including the field searched by the application.

3.2 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 through out 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 to be 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 error. 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 which is not reproducible in an actual run of the web application using the actual table schema.

In the query, we could gain type information for fields when the operations processing or generating the field is known and the argument types are well-defined. This is because type juggling in SQL would lead to an error or warning. For instance, comparison between a field and an integer would help us conclude that the type of the field 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, CorbFuzz 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 clear definition of 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 [36] to construct a more fine-grained typing system.

Still, concolic execution is not enough for inferring types of all symbolic variables. Some of them may not be passed to an internal function or used in comparisons. Additionally, comparison between variables of different types are 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 for these cases, we define a domain of types (τ) for each symbolic variable 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 after the generation of result by concolic execution, which we refer as type hints. If query analysis has already assigned a type, then the type would have infinite weight in τ. Before constraint solving for generating result is initiated, the synthesizer conducts a probabilistic sampling from τ for each symbolic variable 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 2, Field procedure conducts a probabilistic sampling over the τ for each field (Line 19-22). In our implementation, we utilize A-res algorithm [45]. Result returned by Field procedure is always tracked. Notify procedure is called when the tracked value is used in internal functions or for comparison. The type hint is used to increase weight for that type in τ. In our implementation, we only let τ to include integers, strings, and booleans. Type hints for types that are not in τ are ignored.

In the example provided in Figure 1 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`.

3.3 Authentication Bypass Workflow

Cookies and sessions are commonly leveraged by web applications to make HTTP requests stateful [25, 34], 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 set properly (e.g. an authentication token presents for a specific field). It is because complex logics inside web applications tend to be

```php
<?php
session_start();

if (isset($_SESSION["is_auth"]))
    echo $_SESSION["welcome_message"];
```

Figure 2: Example of PHP Application Session Usage

reached after the request presents to be authenticated or authorized. Usually, the cookie or session 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 with session or cookie involved. The method is inspired by hybrid fuzzing but different from it. Instead of solving constraint for the path, the fuzzer only solves the constraint one time when necessary. That is, if an item of cookies or sessions have not been passed to operation that does not have a SMT formula translation available, the value would be never generated. Still, we record the constraints for the decisions we made by treating each item in sessions or cookies a pair of symbolic variable: $< \phi, \alpha >$, where ϕ is the gated boolean symbolic variable that shows whether the item is defined and α represents the value of the item[1].

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` [6] internal call with translation of SMT formula but most requests do not require generating the concrete value of sessions and cookies. Note that we do not define a strict bijection here between seed and the session. A specific session state may map to multiple seeds. This is because the constraints here are not permissive.

We have shown the crucial components for the workflow in Procedure 3 for sessions, which is identical for cookies. Similar to previous workflow for database, there is also an initialization procedure which creates a global hashtable for caching. Specifically, `GC` is for storing the mapping between seed and the session state. Additionally, there is a Start procedure, which is called before the HTTP requests is handled and the variable declared only survive during the lifecycle of that request. The procedure creates a copy of seed and declare a hashtable for saving the constraints for each during the request.

When an item of the session is compared with a concrete value, the Do procedure is used. This procedure functions as a middleware over the comparison operation. CorbFuzz first checks whether there is already cache for the item compared in regards to the seed. 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 operation is implemented so that it could convert the decision of the operation to a constraint. If so, a decision is generated from the seed and the

[1]We used an identical approach as previous to infer types. For conciseness, we do not list the operations for type here.

Procedure 3 Session Generation Algorithm

```
1: procedure Initialization
2:     GC ← HashTable()
3: procedure Start
4:     RCache ← HashTable()
5:     NewSeed ← Copy(Seed)
6: procedure Do(Name, Opline)
7:     if GC(Seed, Name) then
8:         Session(Name) ← GC(Seed, Name)
9:         return Next()
10:    if Opline.Operand ∈ ImplementedOp then
11:        decision ← NewSeed & 1
12:        ShiftRight(NewSeed)
13:        c ← ToConstraint(Opline, decision)
14:        if ¬ IsSAT(RCache(Name) ∪ c) then
15:            return Do(Name, Opline)
16:        RCache(Name) $\overset{+}{\leftarrow}$ c
17:        return decision
18:    else
19:        c ← RCache(Name) ∪¬ GC(*, Name).RRemove()
20:        if c = UNSAT then return Abort()
21:        GC(Seed, Name) ← Solve(c)
22:        return Do(Name, Opline)
```

constraints for performing this decision are appended to the constraints over that item. An SMT solver is then used to check whether the constraint is satisfiable. 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 then is 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 (i.e. generate the concrete value) 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 using already solved values stored in global cache for that field name. To reduce UNSAT cases, each stored value in cache is randomly removed from consideration. As mentioned previously, by doing so, we are unable to achieve uniqueness.

For PHP code listed in Figure 2, when it executes until Line 4, CorbFuzz first declares a pair of symbolic variables $< \phi_0, \alpha_0 >$ 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).

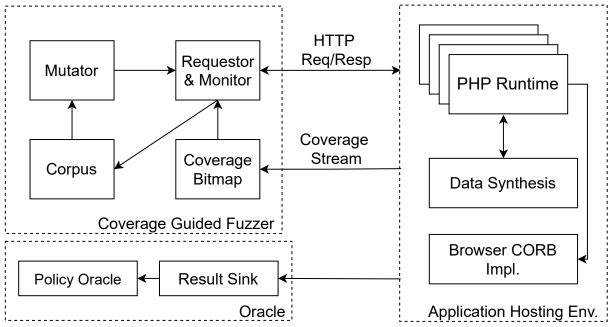

Figure 3: CorbFuzz Architecture

4 BROWSER POLICY FUZZING

In this section we present a coverage-guided fuzzing method called CorbFuzz to check browser security policies. Coverage-guided fuzzing is a method for testing programs to find bugs and security vulnerabilities. It generates new inputs by mutating an element in corpus to test the program. If a mutated input leads to increase in the code coverage of the software, it would be added to corpus for future mutation.

4.1 Fuzzing Algorithm

Procedure 4 CorbFuzz Algorithm

```
1: procedure CorbFuzz(WebApplication)
2:     $P$ ← DataSynthesis(WebApplication)
3:     CovBitMap ← BitMap()
4:     ResultSink ← HashTable()
5:     NewURL ← List()
6:     Visited ← Set()
7:     for ¬ShouldTerminate() do
8:         U, Seed ← (Corpus ∪ (NewURL - Visited)).Pop()
9:         U, Seed ← Mutate(U, Seed)
10:        Visited $\overset{+}{\leftarrow}$ U
11:        Metrics, R, M ← $P$(U, Seed)
12:        NewURL $\overset{+}{\leftarrow}$ ExtractLinks()
13:        if IsNewCoverage(CovBitMap, Metrics) then
14:            Corpus $\overset{+}{\leftarrow}$ U, Seed
15:            CovBitMap $\overset{+}{\leftarrow}$ Metrics
16:            ResultSink $\overset{+}{\leftarrow}$ (R, M)
17:    AnalyzeResult(ResultSink)
```

We present the architecture of CorbFuzz in Figure 3 and its algorithm in Procedure 4. CorbFuzz is a multi-threaded fuzzer and it supports distributed fuzzing. Initially, the fuzzer creates multiple instances of the application runtime which is instrumented as discussed in Section-3. We define the application hosting environment to be a function $P : (\text{URL}, \text{Seed}) \rightarrow \{\text{Metrics}, \text{R}, \text{M}\}$, where 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.

On Line 3 of Procedure 4, a bitmap is created so as to record the coverage. CorbFuzz additionally declares a result sink (Line 4) for storing the inputs for the oracle. The details of these are elaborated in the following sections. During fuzzing, the fuzzer randomly selects a pair of inputs, which are a request URL and potentially a seed, to mutate and send the corresponding HTTP request to the hosting environment (Line 7). If the input leads to increased test coverage, it is added to the corpus. The information required by the oracle is stored in the result sink. Additionally, a list is declared (Line 5) for storing the URLs extracted from the HTTP response (e.g. href values and API calls). Note that this list is different from the corpus. As previously mentioned, an input in corpus contains an URL as well as a seed for describing the state of the persistent layer. The element in the list is instead only the URL.

After the fuzzer terminates, the oracle aggregates the information in result sink and provides a decision for each HTTP response. These decisions are compared with the browser decisions to identify the potential weakness.

4.2 Policy Oracle

To define a policy oracle, we need to categorize the resource accesses as confidential or non-confidential and as we described in Section 2.2, CORB should block any HTTP response containing confidential information. We limit the scope of resource to be only provided by the database for this work. We use a method similar to Pellegrino et al. [37] which deduces confidentiality of a resource by observing resource access frequency. After fuzzer terminates, we aggregate and count the number of resource accesses done by each database query executed during handling each request. In our implementation, we use the average number of accesses as our threshold. If any query uses resources that have frequency below the threshold, we infer that the query is accessing a confidential resource, which should be blocked, and check if the CORB implementation blocked it.

Granularity of the resource impacts the result of the oracle. For example, if the resource is considered as a table, oracle is more likely to block the response than if the resource is considered as a row in that table. Hence, coarse grain resources are 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.

4.3 Coverage Metrics

Rather than utilizing test coverage of CORB function, we collect test coverage from the web application. That is, we are feeding inputs that could contribute to the coverage of the web application. While the coverage information of CORB function may enhance the fuzzing in regards to efficiency, this is largely useless. It is because CORB function is a small piece of code in both WebKit and Chromium. Thus, it is extremely easy to achieve high coverage of CORB function during attempts to achieve high test coverage in web application. Additionally, we are evaluating the policy for different code patterns. Focusing on what CORB is able to handle would not lead to identification of potential weaknesses in the implementation.

LoC Range	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 1: Total LoC Statistics for fuzzing targets

5 IMPLEMENTATION & EVALUATION

We have implemented CorbFuzz in Python in 500 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 responses containing images, CSS, and RPC calls. The fuzzer is able to fuzz a web application and check security policies. 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 workflow by removing components inside the previously implemented fuzzer.

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?

5.1 Experimental Setup

5.1.1 Environment. We evaluate CorbFuzz on two Intel Xeon Phi 7210 (64 cores) nodes. Both nodes use Ubuntu 20.04 with one node running NGINX[10] for serving web application on the instrumented PHP environment and other node running the coverage-guided fuzzer.

5.1.2 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 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 Webkit's CORB implementation.

5.1.3 Web Applications. Web applications are fuzzed to provide response as input for browser test harnesses. We crawled 300 repositories using PHP 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[2], a dependency management tool. The count of remaining applications are 58 with varying LoCs. We fuzz the policies with these 58 applications but for the sake of evaluation of data synthesis

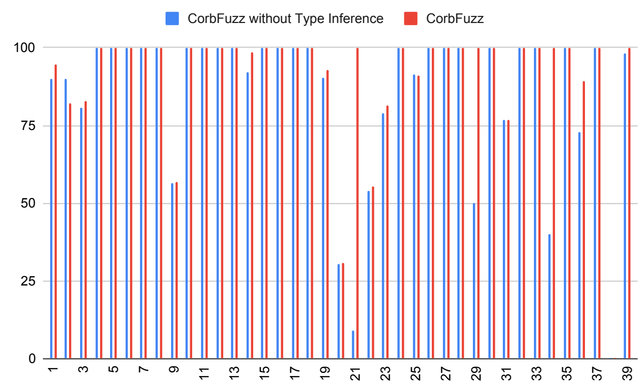

Figure 4: The percentage of generating correct types for comparison statements for CorbFuzz and CorbFuzz without Type Inference for 3 minutes of runtime. X axis denotes the web application number.

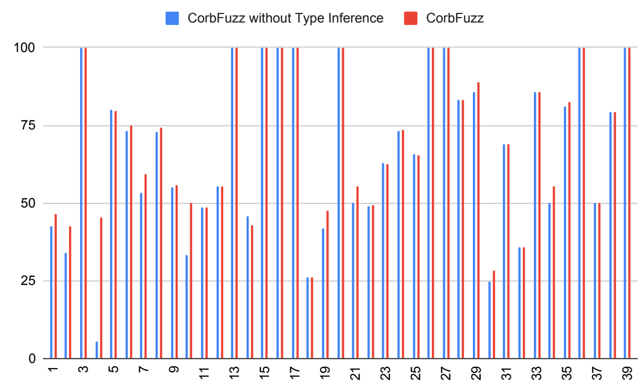

Figure 5: The percentage of generating correct types for internal function calls for CorbFuzz and CorbFuzz without Type Inference for 3 minutes of runtime. X axis denotes the web application number.

effectiveness, we only use 39 of them, for which CorbFuzz reports existence of branches or utilization of databases. The statistics of these applications are presented in Table 1.

5.2 Data Synthesis Effectiveness

To evaluate the data synthesis approach and address **RQ1**, we ran CorbFuzz with and without type inference for 3 minutes with each web application and compared the percentage of correct data generations for comparisons (where the generated variable is compared to a fixed value, e.g an integer) and internal function calls (where the arguments have defined types). 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 in average. Figure 5 shows that for 11 applications, CorbFuzz generates the correct type for internal function calls more often than CorbFuzz without type inference with 5% more data generations in correct type in average.

On some applications, CorbFuzz has little improvement on 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 they are not due to data synthesis. Therefore they cannot be removed by improving type inference in data synthesis.

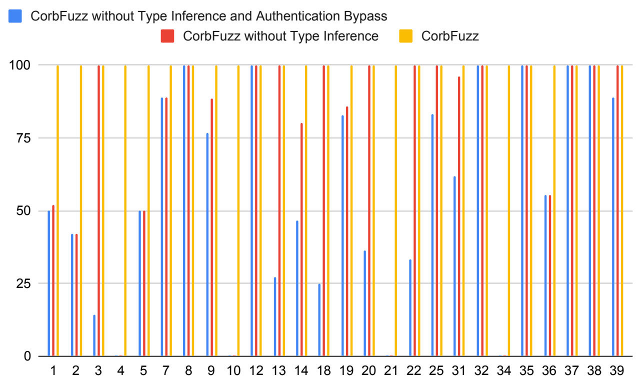

Figure 6: The percentage of edges covered for CorbFuzz, CorbFuzz without Type Inference and CorbFuzz without Type Inference and Authentication Bypass in 3 minutes of runtime against edges covered by CorbFuzz. X axis denotes the web application number.

We are also evaluating the impact of data synthesis on fuzzing effectiveness and address **RQ2**. Figure 6 demonstrates the edge coverage difference in terms of percentages or actual edge counts between CorbFuzz without any type inference or authentication bypass, CorbFuzz without type inference and CorbFuzz. For this evaluation, we only chose applications containing more than one branch as some applications just present the data obtained from the database without any branching involved. 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, 19.0 edges for CorbFuzz without type inference and 27.5 for CorbFuzz. These results demonstrate that with the inclusion of type inference and authentication bypass, we can cover in average 70% more edges and with the inclusion of just type inference, we can cover in average 45% more edges which shows the effectiveness of our data synthesis.

The number of edges covered is low for some applications because these applications save and use structural or serialized data from the database. 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 by the web application and fails to explore these web application. 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 instrument deserialization libraries to provide

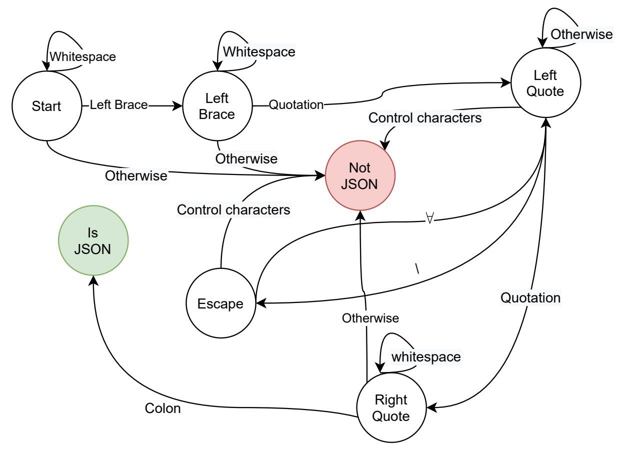

Figure 7: Finite State Machine for Validating JSON

type hints, future work could implement an approach that is able improve web application coverage.

5.3 Detected CORB Weaknesses

We have discovered three 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 discussion[2]. We have reported another one[3], which has later been resolved by a patch in the CORB component[4].

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[8]) 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 array 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[42].

```php
<?php
header('Content-Type:␣application/json');
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 the first key of a JSON object, they would be able to bypass the CORB check by making that key as 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 would become `{"\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.

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 reduction 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.

5.4 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.

6 RELATED WORK

Coverage-guided Fuzzing. Coverage-guided fuzzing have been used to find bugs in many programs such as virtual machines [19],

[2]https://crbug.com/anonymized
[3]https://crbug.com/anonymized
[4]https://chromium-review.googlesource.com/c/chromium/src/+/anonymized/

web browsers [15, 47], and operating systems[28, 29]. The state-of-the-art implementations are AFL [32] and libFuzzer [41]. In this paper we used coverage-guided fuzzing for browser security policy checking. Yet, our approach is not using the coverage of the browser, but instead guided by the coverage of the web applications.

Browser Fuzzing. Domato [14], Dharma [13], and FreeDom [47] 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[15] and Jsfunfuzz[12] 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 utilize web applications responses. Roy et al. [39] fuzzes web applications and supply responses to browsers to detect visual inconsistencies between browsers. It is similar to our work in the sense that both works treat web application and browsers together as black box. Unlike their work which focuses on testing web applications, our work focuses on testing security policy in browsers. We also do not cross reference between browsers but using an oracle defined based on web applications instead.

Web Application Testing. Alshahswan et al. [16] and Biagiola et al. [18] 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, [16] requires the input types and login information. [18] requires Page Objects to be provided to test the web application. Our work instead avoid manual analysis through data synthesis. Elbaum et al. [20] proposes the web application testing should mutate the sessions and provides a few mutation technique that could help achieve better coverage. Data synthesis in our work is different from Elbaum et al. since we do not mutate the sessions but instead symbolically evaluate or generate them. Apollo [17] and Wassermann et al. [46] leverage both concolic execution and fuzzing (i.e. hybrid fuzzing) to increase edge coverage of web applications and discover their vulnerabilities. Session generation workflow in data synthesis is utilizing a hybrid method but it is fundamentally different from the concept of hybrid fuzzing.

7 CONCLUSION

We have created a browser policy fuzzer CorbFuzz which uses web application response to fuzz the browser security policies. To avoid setting up the web applications manually, we proposed a web application hosting environment that synthesizes data. The resources queried by the web application are either generated or symbolically represented. We have shown that the data synthesis approach does not only generate 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 them with 58 applications, we discovered three classes of weaknesses in implementation in Chromium.

REFERENCES

[1] [n.d.]. 185331 – Cross-Origin Read Blocking (CORB). https://bugs.webkit.org/show_bug.cgi?id=185331

[2] [n.d.]. Composer. https://getcomposer.org/

[3] [n.d.]. Cross-Origin Read Blocking (CORB). https://chromium.googlesource.com/chromium/src/+/master/services/network/cross_origin_read_blocking_explainer.md

[4] [n.d.]. CVE - CVE-2020-6442. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6442

[5] [n.d.]. Fetch Standard. https://fetch.spec.whatwg.org/#corb

[6] [n.d.]. isset - PHP Manual. https://www.php.net/manual/en/function.isset.php

[7] [n.d.]. Issue 1013906: Security: expose stored (in cache) cross-site response's size. https://bugs.chromium.org/p/chromium/issues/detail?id=1013906

[8] [n.d.]. The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc8259

[9] [n.d.]. MIME Sniffing Standard. https://mimesniff.spec.whatwg.org/

[10] [n.d.]. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/

[11] [n.d.]. Site Isolation - The Chromium Projects. https://www.chromium.org/Home/chromium-security/site-isolation

[12] 2021. A collection of fuzzers in a harness for testing the SpiderMonkey JavaScript engine. https://github.com/MozillaSecurity/funfuzz original-date: 2015-07-08T01:05:26Z.

[13] 2021. Dharma - Generation-based, context-free grammar fuzzer. https://github.com/MozillaSecurity/dharma original-date: 2015-03-25T17:56:23Z.

[14] 2021. Domato - DOM fuzzer. https://github.com/googleprojectzero/domato original-date: 2017-09-21T15:28:59Z.

[15] 2021. Fuzzilli - A JavaScript Engine Fuzzer. https://github.com/googleprojectzero/fuzzilli original-date: 2019-03-20T15:32:47Z.

[16] Nadia Alshahwan and Mark Harman. 2011. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 3–12.

[17] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. 2008. Finding Bugs in Dynamic Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/1390630.1390662

[18] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. 2017. Search based path and input data generation for web application testing. In *International Symposium on Search Based Software Engineering*. Springer, 18–32.

[19] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1011–1023. https://doi.org/10.1145/3377811.3380432

[20] S. Elbaum, S. Karre, and G. Rothermel. 2003. Improving web application testing with user session data. In *25th International Conference on Software Engineering, 2003. Proceedings.* 49–59. https://doi.org/10.1109/ICSE.2003.1201187

[21] Jeremiah Grossman, Seth Fogie, Robert Hansen, Anton Rager, and Petko D Petkov. 2007. *XSS attacks: cross site scripting exploits and defense.* Syngress.

[22] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. 2017. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. *Computer Security – ESORICS 2017* (2017), 80–97. https://doi.org/10.1007/978-3-319-66399-9_5

[23] Veit Hailperin. [n.d.]. Cross-Site Script Inclusion. https://www.scip.ch/en/?labs.20160414

[24] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. *2012 IEEE Symposium on Security and Privacy* (2012). https://doi.org/10.1109/sp.2012.19

[25] Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. 2012. BetterAuth: Web Authentication Revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/2420950.2420977

[26] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage. *Proceedings 2021 Network and Distributed System Security Symposium* (2021). https://doi.org/10.14722/ndss.2021.23104

[27] Hyungsub Kim, Sangho Lee, and Jong Kim. 2016. Inferring browser activity and status through remote monitoring of storage usage. *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016). https://doi.org/10.1145/2991079.2991080

[28] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings 2020 Network and Distributed System Security Symposium.* Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2020.24018

[29] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, Huntsville Ontario Canada, 147–161. https://doi.org/10.1145/3341301.3359662

[30] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[32] lcamtuf. [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/

[33] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. *2014 IEEE Symposium on Security and Privacy* (2014). https://doi.org/10.1109/sp.2014.9

[34] Yuan-Fang Li, Paramjit K. Das, and David L. Dowe. 2014. Two decades of Web application testing—A survey of recent advances. *Information Systems* 43 (2014), 20–54. https://doi.org/10.1016/j.is.2014.02.001

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* 973–990.

[36] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[37] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. arXiv:1708.08786 [cs.CR]

[38] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1661–1678.

[39] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2014. X-PERT: a web application testing tool for cross-browser inconsistency detection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 417–420.

[40] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-Origin Policy: Evaluation in Modern Browsers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 713–727. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk

[41] K Serebryany. 2015. libFuzzer, a library for coverage-guided fuzz testing. *LLVM project* (2015).

[42] Ajay Shankar (D1r3Wolf). [n.d.]. Chaining No impact(N/A) Bugs to get High impact. https://blog.d1r3wolf.com/2020/04/chaning-no-impactna-bugs-to-get-high.html

[43] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. 2016. 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). https://doi.org/10.1145/2939918.2939922

[44] Margus Veanes, Jonathan de Halleux, Nikolai Tillmann, and Peli de Halleux. 2009. *Qex: Symbolic SQL Query Explorer*. Technical Report MSR-TR-2009-2015. https://www.microsoft.com/en-us/research/publication/qex-symbolic-sql-query-explorer/ Updated January 2010.

[45] Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Software* 11, 1 (March 1985), 37–57. https://doi.org/10.1145/3147.3165

[46] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. 2008. Dynamic Test Input Generation for Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 249–260. https://doi.org/10.1145/1390630.1390661

[47] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event USA, 971–986. https://doi.org/10.1145/3372297.3423340