MINERVA: Browser API Fuzzing with Dynamic Mod-Ref Analysis

Chijin Zhou
Tsinghua University
Beijing, China

Quan Zhang
Tsinghua University
Beijing, China

Mingzhe Wang
Tsinghua University
Beijing, China

Lihua Guo
Tsinghua University
Beijing, China

Jie Liang*
Tsinghua University
Beijing, China

Zhe Liu
NUAA
Nanjing, China

Mathias Payer
EPFL
Lausanne, Switzerland

Yu Jiang*
Tsinghua University
Beijing, China

ABSTRACT

Browser APIs are essential to the modern web experience. Due to their large number and complexity, they vastly expand the attack surface of browsers. To detect vulnerabilities in these APIs, fuzzers generate test cases with a large amount of random API invocations. However, the massive search space formed by arbitrary API combinations hinders their effectiveness: since randomly-picked API invocations unlikely interfere with each other (i.e., compute on partially shared data), few interesting API interactions are explored. Consequently, reducing the search space by revealing inter-API relations is a major challenge in browser fuzzing.

We propose MINERVA, an efficient browser fuzzer for browser API bug detection. The key idea is to leverage API interference relations to reduce redundancy and improve coverage. MINERVA consists of two modules: *dynamic mod-ref analysis* and *guided code generation*. Before fuzzing starts, the *dynamic mod-ref analysis* module builds an API interference graph. It first automatically identifies individual browser APIs from the browser's code base. Next, it instruments the browser to dynamically collect mod-ref relations between APIs. During fuzzing, the *guided code generation* module synthesizes highly-relevant API invocations guided by the mod-ref relations. We evaluate MINERVA on three mainstream browsers, i.e. Safari, FireFox, and Chromium. Compared to state-of-the-art fuzzers, MINERVA improves edge coverage by 19.63% to 229.62% and finds 2x to 3x more unique bugs. Besides, MINERVA has discovered 35 previously-unknown bugs out of which 20 have been fixed with 5 CVEs assigned and acknowledged by browser vendors.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Software security engineering**.

KEYWORDS

browser security, interface fuzzing, dynamic analysis

*Yu Jiang and Jie Liang are the corresponding authors.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549107

ACM Reference Format:
Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. MINERVA: Browser API Fuzzing with Dynamic Mod-Ref Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549107

1 INTRODUCTION

Modern browsers export thousands of built-in APIs, allowing web applications to access and manipulate a browsers' inner state flexibly, e.g., accessing an element through *doc.getElementById()*; changing styles through *el.setAttribute()*. The flexibility of browser APIs, however, becomes an essential part of the attack surface. An attacker can craft a sequence of API invocations to exploit browser security bugs, resulting in privacy leakage or even remote code execution.

While browser vendors focused on discovering bugs [3, 15, 27, 29], less attention has been paid to the security of browser APIs. For example, conventional fuzzing works are commonly focused on browser submodules, including third-party libraries [9, 13, 14] and JavaScript engines [16, 17, 31, 34]; consequently, no browser APIs are covered. Recent work [35, 44, 53] focuses on fuzzing the rendering engine of browsers where only a limited set of browser APIs are carefully tested.

For browser testing, the quality of a test case highly depends on its browser API invocations: higher interaction between its API invocations translates to higher coverage. However, thousands of APIs offer near infinite combinations. Enumerating them all is infeasible, limiting fuzzer effectiveness. To mitigate it, existing fuzzers consider only a carefully-selected subset of APIs and use manually-labeled semantics for assisting their pruning heuristics. As a result, existing approaches cannot thoroughly explore the state space behind browser API invocations. First, to reduce the search space, a number of APIs are considered "meaningless" thus manually pruned. With incomplete API coverage, bugs hidden in them will never be discovered. Second, manually-labeled semantics only identify a small amount of inter-API relationships. For example, the semantics of some existing works [35, 44] annotate SVG attributes or CSS properties to allow related APIs to establish data flow at generation time. The annotations heavily rely on domain knowledge from experts, thus only a few important components are concerned currently. To our best knowledge, no known testing approach can scale to the full set of browser APIs.

To expand the number of tested browser APIs, the vast search space can be reduced by removing mutually exclusive API combinations. Every API manipulates or accesses a certain part of the browser's internal data during its execution, by which an API invocation may interfere with others. Putting two non-interfering APIs into a test case is unlikely to trigger any interesting behavior because there is no interaction between them. For example, according to our observation, more than 60% of API invocation pairs in each test case generated by Domato [44] do not access any common resources, leading to a large number of redundant executions during the fuzzing campaign. Therefore, it is important to consider interference between APIs in fuzzing. Once the relations are built, fuzzers can prune vast unnecessary API combinations and generate highly-relevant API invocations within a single test case, massively improving fuzzing performance.

However, it is challenging to capture interference relations and further facilitate input generation. First, we cannot simply identify the relations through interface description: an API may influence another one even though the two APIs are type-independent. For example, the API `Node::appendChild(Node)` can impact the API `HTMLInputElement::setSelectionRange(long,long)`, because the latter will resize the input element and trigger repainting on this page, therefore, every element that was appended to the current page will be checked for reposition during repainting. Hence, the behaviors of `setSelectionRange` interfere with by `appendChild`, even though they neither belong to the same object nor have any relationship in parameter types. Second, our input generation strategy should not only consider interference relations but also conform to semantic correctness. If the concrete parameters do not meet the API requirement, e.g., type hierarchy or argument count mismatch, then the invocation will be rejected before reaching the backend logic, which also renders fuzzing inefficient. Therefore, the generation strategy should ensure both semantic correctness and relevance to each other.

We propose Minerva for efficient browser API bug detection. Leveraging API interference relations, Minerva generates APIs with correct semantics and relevant combinations, producing test cases *higher in coverage* and *dependent in API*. It consists of two main modules: *dynamic mod-ref analysis* and *guided code generation*. Before fuzzing starts, the *dynamic mod-ref analysis* module builds an accurate API interference graph. It first automatically identifies individual browser APIs from a browser's code base. Next, it instruments the browser to dynamically collect mod-ref relations between APIs. In essence, it detects that interface I_i is impacted by I_j if there is a trace where I_i reads a resource after I_j modifies it. During fuzzing, the *guided code generation* module synthesizes API invocation sequences guided by the mod-ref relations. Based on the relations, it assigns weights to APIs to prioritize the ones that are highly relevant to the previous invocations at every selection stage. Besides, it also leverages API specifications to ensure generated test cases are semantically-correct.

We implement the prototype of Minerva and evaluate its performance on recent versions of three mainstream browsers, i.e. Safari, FireFox, and Chromium. Compared to state-of-the-art fuzzers, Minerva improves edge coverage by 19.63% to 229.62% and finds 2x to 3x more unique bugs. In addition, Minerva has discovered 35 previously-unknown bugs on the three browsers; 20 have been fixed with 5 CVEs assigned so far.

In summary, this paper makes the following contributions:

- We identify an insufficiency in browser testing practices and propose a new approach to generate highly relevant API invocations for browser API bug detection. Our approach leverages mod-ref relations between APIs, so that the generated test cases are higher in coverage and less redundant in semantics.
- We design and implement Minerva. It consists of (1) a dynamic mod-ref analysis module to collect APIs' memory access information, and (2) a code generation module that uses API mod-ref relations for highly-relevant API invocations synthesis. The tool will be available at https://github.com/ChijinZ/Minerva.
- We evaluate Minerva on Safari, Firefox, and Chromium. Minerva achieved higher coverage and triggered more unique bugs than the state-of-the-art fuzzers. It has detected 35 previously-unknown bugs where 5 CVEs were assigned so far.

2 BACKGROUND

2.1 Browser API and Interface Description

Modern web browsers provide thousands of built-in browser APIs (a.k.a Web APIs) to support complex operations for manipulating or accessing browsers' inner state, allowing developers to build all kinds of web applications. Figure 1 demonstrates how browser APIs are used in real-world scenarios. When an API is invoked, the JavaScript engine first parses the statement and dispatches the invocation to the corresponding binding code. Then the binding code translates data representations between JavaScript and native code (mostly C/C++), and finally invokes backend functions for manipulating or accessing the browser's inner states.

Interface descriptions are necessary for developers to write API invocations. Generally, every browser has its own interface description in its code base. The description is in WebIDL [46] format and contains every API declaration, including interface name and type properties of its object, return value, and parameters.

2.2 Browser Fuzzing

As one of the most complicated applications, browsers naturally become an attractive target to security researchers and attackers. Fuzzing browser's sub-components such as third-party libraries [13, 14] and JavaScript engines [16, 17, 31, 34] gained significant traction

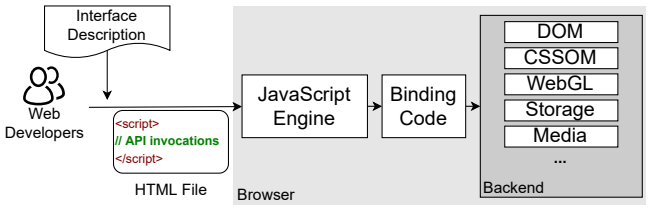
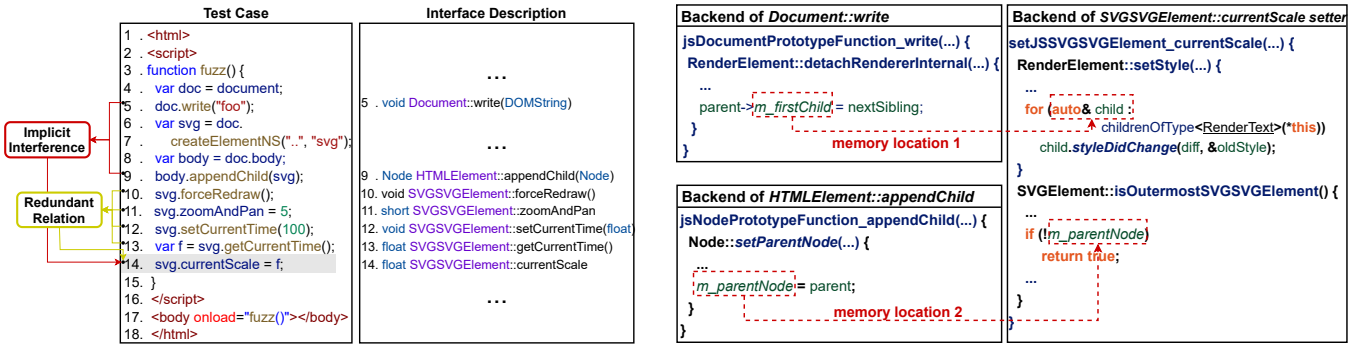


Figure 1: Demonstration of browser API usage. Browser API is exported by browsers for manipulating or accessing browser backend.



(a) A test case and its corresponding interface descriptions

(b) Memory access interference hidden in browser's backend logic

Figure 2: An example for implicit interference relation on WebKit. The logic of `SVGSVGElement::currentScale` is interfered with by *Document::write* and *HTMLElement::appendChild*, although they seem irrelevant from interface description view.

in academic research as well as in industry. Nevertheless, less attention has been paid to *whole browser fuzzing*. The lack of end-to-end fuzzing prohibits finding some vulnerabilities [45].

A common approach of industrial whole browser fuzzers [35, 36, 44] is to take static grammars that describe API specifications and semantics to generate HTML test cases. For each test case generation instance, they iteratively select a rule from the static grammars to instantiate, and generate helper code for it to avoid syntactic or semantic errors in the meanwhile. In addition to the conventional approach, FREEDOM [53], a state-of-the-art DOM fuzzer, improves fuzzing effectiveness by classifying browser APIs into various fuzzing operations on the basis of their functionalities. Based on that, it designs several generation and mutation strategies for maintaining context information. Recently, FAVOCADO [10] is proposed for finding vulnerabilities in binding code. It extracts all API specifications of each browser, and tries to leverage the type property of APIs to generate test cases that trigger fewer syntactic and semantic errors.

3 MOTIVATION

The dilemma of existing browser fuzzers is that their effectiveness cannot be guaranteed once the test surface is increased. On the one hand, fuzzing is **incomplete** without considering the full list of APIs. Those pre-pruned APIs may contain bugs while the fuzzers will never find them. On the other hand, if the full list of APIs is considered, it is inevitable to lead to **inefficient** test cases because of the huge search space.

One potential solution is to consider *interference relations between APIs* during fuzzing. Every API manipulates or accesses a certain part of the browser's internal state during its execution. An API execution may interfere with the ones of other APIs by modifying common internal data. Orthogonally, an API is irrelevant to another one if there is no possibility for them to access shared data. Putting two irrelevant APIs into a test case will not trigger any interesting behavior because there is no interaction between them. Therefore, it is important for fuzzing to consider interference relations between APIs. Once the interference relations are available, fuzzers can exclude vast unnecessary API combinations and only generate highly-relevant API invocations within a single test case, largely improving fuzzing performance.

We observed that the interference relation is generally hidden behind the browser's backend instead of exposed by interface description, making the collection of it challenging. We define **implicit interference** as the relation between two APIs that are irrelevant at the interface description level but have interference relation during execution. Figure 2a presents a representative test case. The test case includes a sequence of API invocations for appending a SVG element to the current document's body (Line 8-9) and manipulating the internal data of this SVG element (Line 10-14). In this example, we focus on the last API invocation, i.e. setting `currentScale` of a SVG element to a floating-point value, and investigate which statement interferes with it. We notice that it is possible that two APIs have interference relations even though they are irrelevant from the interface description view. For example, the logic of the last invocation largely depends on the `doc.write` and `body.appendChild` invocations. Figure 2b is the proof of memory access interference between them. When executing the last invocation, the browser first delegates the execution to the corresponding backend function `setJSSVGSVGElement_currentScale`. The backend function is responsible to set the `currentScale` property of the SVG object and repaint the page if necessary. During its repainting, the SVG object checks the style of each `RenderText` child node of its parent node to know which rendering objects need to be repainted. In this process, the backend function reads memory location 2 for checking if the parent node exists (in the `isOutermostSVGSVGElement` function), and also reads memory location 1 for accessing child nodes of parent node (in the `setStyle` function). The two memory locations are modified by the `doc.write` and `body.appendChild` invocations. The `doc.write` invocation adds a node to the current `document`, during which it modifies the `m_firstChild` variable on memory location 1. On the other hand, the `body.appendChild` invocation modifies the parent node of the SVG object on memory location 2. These implicit interference relations are ignored by existing works.

Besides the implicit interference missed in test cases, we also found that redundant relation widely exists in test cases. We define **redundant relation** as the relation between two APIs that are only relevant at the interface description level but never interfere

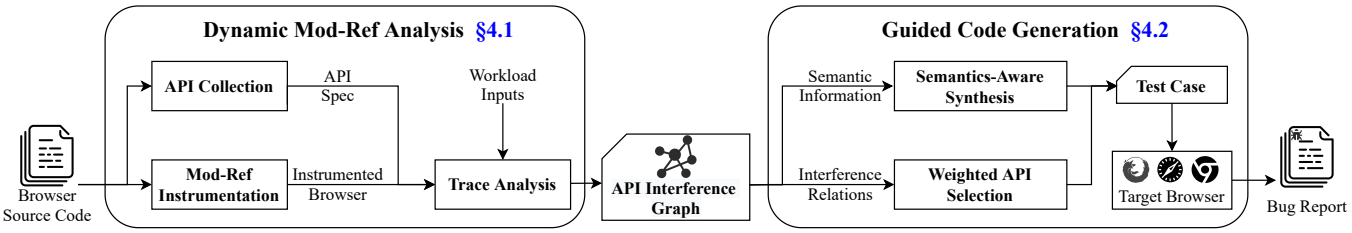


Figure 3: The overall design of Minerva. During the preparation phase, Minerva performs dynamic mod-ref analysis by executing workload inputs and analyzing memory access of each API, and then builds an API interference graph. During the fuzzing phase, guided by the graph, Minerva prioritizes APIs with high relevance to the previously selected ones in every selection stage, and meanwhile preserves semantic correctness when test case synthesis.

with each other's state during execution. In Figure 2a, among all the statements above the last invocation, those SVG operations (Line 10-13) are likely to interfere with it because they are manipulating the same SVG object. Counterintuitively, every execution of those operations did not affect the execution of setting the `currentScale`. For example, the `setCurrentTime` invocation only manipulates the time-related variables of the SVG object in the browser's backend logic, which is isolated to all the memory accesses of the `currentScale` execution. Even though the return value of `getCurrentTime` is used to set the `currentScale`, there is still no interference between them from memory access perspective. Putting independent APIs together absolutely makes the generated test case redundant and reduces fuzzing effectiveness.

In summary, we observe that redundant relation and implicit interference are widely present in API combinations while existing work does not take them into account. The absence of considering API relations damages fuzzing effectiveness because putting non-interfering APIs into a test case will not trigger any interesting behavior. This motivates us to design an automatic browser fuzzing technique that reveals interference relations between browser APIs and facilitate high-quality input generation for more efficient bug detection.

4 DESIGN

Our browser API fuzzer Minerva infers API relations for effective input generation. As shown in Figure 3, Minerva consists of two main modules: a *dynamic mod-ref analysis* to build an accurate API interference graph and a *guided code generation* for generating higher-coverage and API-dependent test cases. (1) During the preparation phase, the *dynamic mod-ref analysis* performs instrumentation on the target browser and then visits some websites or generated test cases for trace collection. After that, it analyzes the collected traces and API specifications to build a directed graph $G = (N, E)$, where N is the set of APIs and $E = \{(u, v) | u, v \in N \wedge u \neq v \wedge hasModRef(u, v)\}$. Each edge denotes that its source node API interferes with the target node API. Next, Minerva leverages the interference graph to generate test cases. (2) During the fuzzing phase, for each test case generation, the *guided code generation* iteratively selects an API to instantiate and meanwhile generates a series of helper code for satisfying the semantic dependencies of this API. It maintains a choice model on the basis of the interference graph so that in every selection stage its algorithm will prioritize APIs with high relevance to the previously selected ones. Besides the API selection, it also adopts a custom synthesis strategy to ensure the generated helper codes are semantically-correct.

The advantage of Minerva over the state-of-the-art counterparts is that Minerva generates less-redundant test cases without extra manual effort. That brings two major benefits. First, Minerva can test the full list of APIs instead of a manually-selected subset. Existing fuzzers avoid considering all APIs because of the huge search space formed by API combinations. By contrast, Minerva leverages interference relation to automatically reduce the search space, making the full APIs fuzzing viable. Second, Minerva is able to generate high-quality test cases. Even though existing fuzzers integrate human knowledge to label semantic information, the test cases they generated still contain many redundant API invocations. The API invocations of each test case generated by Minerva, on the other hand, are highly relevant to each other thanks to the interference relations guidance.

4.1 Dynamic Mod-Ref Analysis

The goal of dynamic mod-ref analysis is to build an API interference graph to facilitate high-quality code generation. Each node of the graph refers to an API, and each edge denotes that its source node API interferes with its target node. We define the interference relation used in Minerva as follows:

Definition 4.1. Interference Relation. An API I_i interferes with another API I_j if the invocation of I_i modifies a memory location that is used by the invocation of I_j, i.e., part of the state of I_j.

To build the graph, we must first collect all API specifications from each browser to form the nodes, and then collect traces of browser's executions to construct the interference edges. Three sub-components are responsible for this task: (1) the *API Collection* component for converting WebIDL files collected each browser to a context-free grammar; (2) the *Mod-Ref Instrumentation* component for instrumenting every modification/reference memory location of browser's memory objects; (3) the *Trace Analysis* component for building the API interference graph by analyzing traces.

API Collection. Minerva relies on an initial API specification. First, Minerva requires all APIs to form the node set of the interference graph. Besides, although API specification does not reveal implicit relations between APIs, its type property information allows Minerva to preserve semantic correctness during input generation. Therefore, the specification is required to keep the same semantic information with the corresponding API's declaration, meanwhile

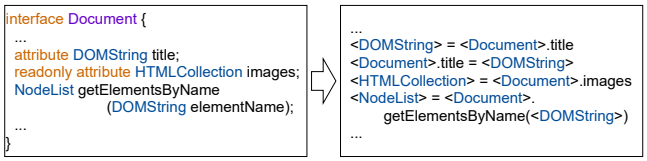


Figure 4: An example of conversion from WebIDL files to the custom context-free grammar. The conversion preserves original semantic information. The ones in angle brackets are non-terminal symbols.

needs to be easily integrated with input generation. To meet this end, MINERVA collects WebIDL files from the code base of the target browser and then converts them to a simplified context-free grammar (CFG), which will be used to assist code generation later. Figure 4 illustrates the general idea of the conversion. The code in the left rectangle is in a format of WebIDL [46], which is similar to header files in C/C++ and describes interfaces that are intended to be implemented in web browsers. The code in the right rectangle is a custom CFG. Each line of the CFG is a production rule in which the right-side non-terminal symbols can be expanded recursively. A non-terminal symbol is regarded as a placeholder for a variable of the corresponding type when code generation. When dealing with `interface` of WebIDL, MINERVA follows the following principles to make the conversion:

- For each read/write member variable, the conversion generates two production rules, one for getting its value, another for setting a value to it.
- For each read-only member variable, the conversion generates a production rule for getting its value.
- For each member function, the conversion generates a production rule in which the left-side non-terminal symbol is the function's return type, and the right-side is a concatenation of object type name, function name and parameter type names.

Mod-Ref Instrumentation. During this stage, MINERVA performs instrumentation to a browser. The goal of this instrumentation is to monitor memory operations, i.e., store and load, of each API during execution, and reveal implicit interference relations between APIs. However, it is challenging because instrumenting the browser on all memory access locations results in unacceptable overhead. During testing, we observed over 10x slowdown and unaffordable overhead in memory for each execution when tracing all memory operations. To mitigate the overhead, we design a custom instrumentation strategy based on two observations: (1) There are many background threads that are responsible for input-irrelevant tasks, e.g. communication to other processes. The behaviors of background threads thus are irrelevant to the browser API execution. (2) Only memory objects that are shared between APIs should be considered. The memory operations of local variables are unnecessary to track because it will never interfere with other APIs.

To meet this end, we perform dataflow analysis to trace the memory operations in which the operator pointer flows from the shared memory objects. We model program's memory behaviors in the same way as Andersen's pointer analysis [2] does, i.e., categorizing pointer operations into four types: address-of (`p=&o`), copy (`q=p`), load (`q=*p`) and store (`*p=q`). To enable field-sensitive analysis, we also consider offsets (`q=p.x`). The pseudocode for instrumentation is described in Algorithm 1. Before performing instrumentation, MINERVA obtains all functions that can be reached from API entries (Line 3). There are a lot of functions irrelevant to input in the browser and thus unreachable from API entries. Such functions should be excluded from our analysis. Next, it performs dataflow analysis to trace memory operations of each concerned memory object (Line 4-8). Note that MINERVA only traces the memory objects that can be shared between APIs. Local memory objects are thus excluded. During the dataflow analysis, MINERVA recursively traces all pointer operations related to the current memory object. If a pointer operation loads from/store to the current object, it will be regarded as a reference/modification instrumentation point (Line 13-15). If an object is offset or copied from the current object, MINERVA will recursively trace its dataflow (Line 16-18). If an object takes the address of the current object, then MINERVA will find every load-site of this object and recursively traces its point-to object's dataflow (Line 19-23). In this way, MINERVA can instrument all memory locations flowed from concerned memory objects.

Trace Analysis. After instrumentation, a browser is prepared to execute workload inputs for trace collection. Every trace contains memory access information of its API invocations, including the API invocation sequence and a set of modification/reference addresses for each invocation.

Figure 5 gives a step by step example of revealing implicit relation between APIs for building the API interference graph. First, various workload inputs, e.g., off-the-shelf websites or generated test cases, are executed by an instrumented browser. During every execution, a trace that contains memory access information of its API invocations is emitted. For workload input 1, we can see that the address `0x7f5dde2ea798` is modified by the 12th API invocation and used by the 27th API invocation. At the same time, the memory access sizes of them are both 64, which means that both APIs most likely access the same object. This is a proof that the 12th invocation interferes with the 27th. Therefore, we

Algorithm 1: Mod-Ref instrumentation algorithm

```
   Input  : Program P and a set of APIs I
   Output : Instrumented program Q
1  Function ModRefInstrumentation(P):
2      Q = P.copy()
3      F = getAPIsReachableFuncs(Q, I)
4      foreach function f in F do
5          foreach memory object obj in f do
6              instrumentWithDataflow(obj, Q)
7          end
8      end
9      return Q
10 Function instrumentWithDataflow(obj, Q):
11     if visited(obj) then
12         return
13     foreach load or store instruction Inst uses obj do
14         Q.instrumentOn(Inst)
15     end
16     foreach object x is offset or copied from obj do
17         instrumentWithDataflow(x, Q)
18     end
19     foreach object x takes address of obj do
20         foreach object y loaded from x do
21             instrumentWithDataflow(y, Q)
22         end
23     end
```

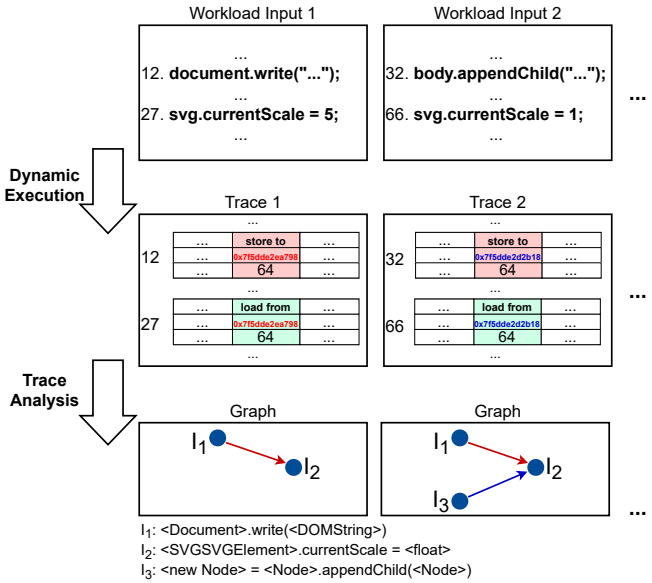


Figure 5: An example of building API interference graph. The instrumented browser executes workload inputs and records corresponding traces. The trace analysis adds an edge to the graph once there is a mod-ref pair in the traces.

can add an edge from the `<Document>.write(<DOMString>)` node to the `<SVGSVGElement>.currentScale=<float>` node. Then the second workload input is processed in the same way. New edges are added to the graph as they are discovered, continously revealing new implicit interface relations.

4.2 Guided Input Generation

The API interference graph can facilitate effective code generation in many ways. First, nodes of the graph include all API declarations, which can help Minerva not only cover all possible APIs but also combine them in a semantically-correct way. Second, edges of the graph indicate that an API can interfere with another one. Minerva can use them to generate less-redundant test cases.

The key idea of Minerva is to prioritize the APIs that are highly relevant to the previous invocations at every selection stage. To meet this end, Minerva maintains a choice model for weighted sampling based on the API interference graph. In addition to API selection, Minerva has a synthesis component for guaranteeing semantic correctness. Once an API is selected, Minerva instantiates it with concrete parameters. The synthesis component is responsible for generating necessary helper invocations in order to make the concrete parameters semantically-correct during the instantiation. Benefiting from weighted API selection and semantics aware synthesis, Minerva can generate test cases that are likely to trigger deep logic behind interaction of browser's APIs.

Weighted API Selection. The goal of our selection strategy is to reduce the redundancy in test cases. Besides, it should also avoid introducing much extra overhead because low throughput damages fuzzing effectiveness. Therefore, we adopt weighted random sampling [11], a light-weight but effective approach, to meet this end. Our intuition is that, when n API invocations have been

Algorithm 2: Test case generation with weighted selection

```
   Input  : API interference graph Graph
   Output : Invocation list List
1  Function generateTestCase(Graph):
2      List ← []
       // model for weighted API selection
3      Model = initialization()
       // context for storing created variables
4      ctx = initialization()
5      while List.size() < MAXLINE do
6          Node = weightedSelection(Model, Graph)
           // synthesize the invocation and helper code
7          Code = synthesizeInvocations(Node, Graph, ctx)
8          updateWeights(Model, Code, Graph)
9          List.append(Code)
10     end
11     return List
12 Function updateWeights(Model, Code, Graph):
13     foreach invoked API I in Code do
14         foreach sink node s in Graph[I] do
15             Model.weights[s] += 1
16         end
17     end
```

synthesized, there should be a higher probability to select those APIs that can be interfered with by the n previous invocations. In this way, the invocations in a generated test case are likely to be influenced by each other.

Algorithm 2 shows how the selection algorithm works when generating test cases. During the process, Minerva maintains a choice model for API selection (Line 3). This model tracks a weight value (initialized to 1) for every API. When generating a test case, Minerva iteratively selects an API based on the weights, synthesizes invocations of this API and updates weights according to synthesized invocations (Line 5-10). The `synthesizeInvocations` function is used to synthesize invocations as well as helper code for semantic correctness. We will detail it later. The `updateWeights` function is used to update weights for relevant APIs. During the updating, for each invoked API in the list of synthesized invocations, Minerva finds all sink nodes of it in the API interference graph and then updates their weights (Line 12-17). In the next iteration, it is likely to select the APIs with high relevance to the previous invocations because their weights are higher than others.

Semantics Aware Invocation Synthesis. The goal of our invocation synthesis is to synthesize semantically-correct invocations for a selected API node. Thanks to the rich semantics from nodes of the interference graph, Minerva can know every API's type property, i.e., types of its object, return value, and parameters. Algorithm 3 details the synthesis workflow. Note that Minerva should not only synthesize an invocation of the selected API, instead, it should also synthesize helper invocations for providing concrete parameters for the selected API. For example, if the `<Element> = <Document>.getElementById(<DOMString>)` production rule is selected, the generation component is responsible for looking for a `Document` variable and a `DOMString` variable. If previous invocations have created them, Minerva will be likely to use them to meet the end. Otherwise, Minerva will invoke some APIs to construct them for guaranteeing semantic correctness. Note that the synthesizer randomly constructs variables to add more instances for each type even though they are created before (Line 6).

Algorithm 3: Semantics aware invocation synthesis

```
Input  : API Node for instantiation Node
         API interference graph Graph
         Context ctx
Output : Invocation list List
Function synthesizeInvocations(Node, Graph, ctx):
    List ← []
    Instance = Node.instantiate()
    Spec = Graph.getAPISpec()
    foreach nonterminal symbol S of Node do
        if ctx.containsVariable(S) and not randomlyDropout() then
            // reuse created variable
            Var = ctx.get(S)
            Instance.add(Var)
        else
            // generate semantically-correct code
            Var, HelperCode=generateHelperCode(S, Spec)
            ctx.update(HelperCode)
            List.append(HelperCode)
            Instance.add(Var)
        end
    end
    List.append(Instance)
    return List
```

5 IMPLEMENTATION

As Figure 3 shows, MINERVA consists of two main modules: a dynamic mod-ref analysis and a guided code generation.

The dynamic mod-ref analysis includes API collection, mod-ref instrumentation, and trace analysis. The API collection is implemented with roughly 600 lines of Python code. It uses off-the-shelf tools [19, 47] to parse WebIDL files to ASTs, and then convert them to context-free grammar. The mod-ref instrumentation is implemented on the top of LLVM 12 [24] with roughly 700 lines of C++ code. It traces the dataflow of every concerned memory object on the top of Link Time Optimizations (LTO). The trace analysis is implemented with roughly 1,900 lines of Rust code. During the trace analysis, a server is spawned for collecting trace information from instrumented browsers. When an instrumented browser is executing a workload input, the runtime of instrumentation tracks its trace information into a block of shared memory at the same time. After the browser is closed, the server will collect its trace and build the API interference graph as algorithm 5 shows. The guided code generation is implemented with roughly 1,000 lines of Python code on the top of DOMATO [44]. The algorithm 2 and algorithm 3 are implemented as third-party extenions for DOMATO.

In addition to MINERVA, we also implemented a browser fuzzing framework with roughly 2,000 lines of Python code to automate mainstream browsers for fuzzing. Its browser automation is based on Selenium [37]. For fair evaluation, it also includes glue code that adapts other existing fuzzers to this framework.

6 EVALUATION

We evaluate MINERVA on three common browsers, i.e. Safari, FireFox, and Chromium. Our goal is to understand MINERVA's bug-finding capability as well as how it compares to other state-of-the-art browser fuzzers. Our evaluation addresses the following research questions:

- **RQ1**: How well does MINERVA perform compared to other state-of-the-art browser fuzzers (Section 6.1)?
- **RQ2**: Does the API mod-ref graph effectively enable MINERVA to generate less-redundant test cases (Section 6.2)?
- **RQ3**: Can MINERVA uncover critical bugs in production-level browsers (Section 6.3)?

Experiment Setup. We perform our evaluation on an AMD EPYC 7742 CPU (2.25GHz) with 64 cores running Ubuntu 20.04. To fuzz our browser targets without a connected display, we use X virtual frame buffer (Xvfb)—a common setup in graphical application testing. All experiments are conducted on the same hardware and repeated five times, following fuzzing evaluation best practices [21] and in line with other fuzzing research [25, 34, 51, 56].

Graph Building. To collect the API mod-ref graph for MINERVA, we use non-relation-guided MINERVA to generate workload inputs for each mod-ref-instrumented browser for 8 hours. We also conducted experiments about the impact of different settings to build graphs, including different time limits and different types of workload inputs, to fuzzing effectiveness. We will discuss our results and corresponding settings in Section 6.2. The number of collected APIs, i.e. nodes of the graph, is 4,630.

Table 1: Features of browser fuzzers using in evaluation.

Fuzzer	Year	Focus	API Spec	API Relation
DOMATO	2017	DOM APIs	hand-written	hand-written
FREEDOM	2020	DOM APIs	hand-written	hand-written
FAVOCADO	2021	JS bindings	automatic	type property
MINERVA	2021	all APIs	automatic	mod-ref analysis

Benchmarks. We target three common browsers: Safari, FireFox, and Chromium. Since Safari cannot be run in a Linux system, we use WebKitGTK [43], a full-featured port of Safari's rendering engine, as an alternative. The versions are WebKitGTK 2.32.3, FireFox 95.0, and Chromium 94.0. For comparison, we use three state-of-the-art browser fuzzers, i.e. DOMATO [44], FREEDOM [53] and FAVOCADO [10] in our evaluation, see Table 1 for their features. Note that all fuzzers used in our evaluation are generation-based and *do not need any initial seeds* for fuzzing.

Metrics. We use three metrics to evaluate each fuzzer: edge coverage, number of unique bugs, and average compactness of test cases. To our knowledge, there is no metric to evaluate how invocations of a test case are relevant to each other, but it is important to evaluate it to see if our mod-ref analysis is useful. Therefore, we define compactness of a test case as the proportion of memory-related invocation pairs to all invocations pairs within a test case. Formally, if a test case contains a sequence of n invocations $[I_1, I_2, ..., I_n]$, then the compactness is defined as

$$compactness = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} R(i,j)}{n * (n-1)/2} \quad (1)$$

where $R(i, j)$ equals 1 if I_i and I_j have a mod-ref relation during runtime; otherwise, $R(i, j)$ equals to 0. The coverage and unique bug metrics are widely used in fuzzing evaluation [9, 17, 21, 34, 53]. We use SanitizerCoverage (SanCov) [23] to collect edge coverage. Note that we only collect coverage from rendering-related modules, i.e. `libwebkit` of WebKit, `libxul` of FireFox, and `libblink` of Chromium, which are responsible for the main backend logic. Other modules, e.g. networking libraries or JavaScript engines, are no need to profiled since they are not our concerned. To get unique bugs, the crashes that each fuzzer found are deduplicated by the

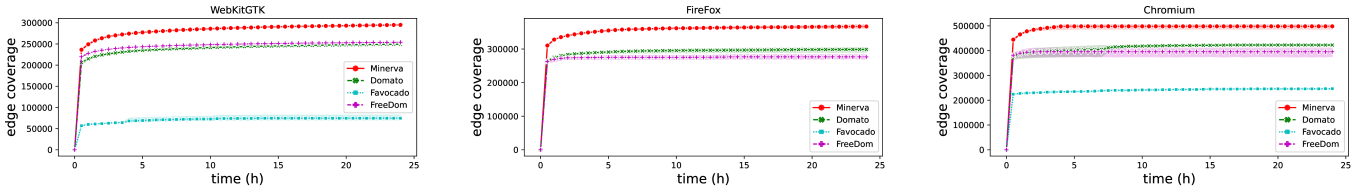


Figure 6: The growth trend of edge coverage over 5 runs in 24 hours. Displayed are the median and the 95% confidence intervals.

root cause of ASan [38] report. Each browser is compiled into two versions: one is instrumented by SanCov and ASan for collecting edge coverage and crashes; another is instrumented by mod-ref instrumentation for collecting each test case's compactness.

6.1 Comparison with Existing Fuzzers

We compare Minerva to three state-of-the-art fuzzers in different aspects to investigate its strength and weakness. For a fair compactness comparison, the number of lines for individual test cases is limited to 1,000. This setting does not affect the results of edge coverage and unique bug according to our experiments. Favocado does not support FireFox, so we omit the experiment.

Edge Coverage. Figure 6 shows the coverage growth over time of each fuzzer. The coverage trend is in line with other evaluation [53]. Minerva achieves higher coverage compared to others in the same amount of time. On average, Minerva improves over Domato, FreeDom and Favocado by 19.63%, 24.90% and 229.62%, respectively. The reason why Minerva outperforms Domato and FreeDom is that Minerva takes the API relations into consideration. On the one hand, the test case generated by Minerva is semantically-correct and able to cover complete test surface. On the other hand, with the help of its API interference graph, Minerva is capable of exploring deep logic between APIs. Although Favocado also generates semantically-correct test cases, its type-based relations introduce redundant relations between APIs as we discuss in Section 3. Besides, its conservative generation strategy only choose a few interface objects in each test case, hindering the fuzzer from state space exploration.

Average Compactness. The compactness of a test case is defined as the proportion of memory-related invocation pairs to all invocations pairs within a test case. A test case with high compactness indicates that the invocations within it are highly API-dependent. Table 2 shows compactness improvement of test cases generated by each fuzzer. Favocado only invokes a few APIs in each of its test case, so we exclude it in this table. For average improvement, we run Mann-Whitney U Tests and the p-values are all <0.05, indicating statistical significance. We can see that Minerva generates more API-dependent invocations within a test case compared to Domato, FreeDom. On average, Minerva outperforms Domato, FreeDom by 39.18% and 68.67%, respectively. The reason of the improvement is that Minerva leverages API interference relations to generate highly-relevant invocations. Although other fuzzers are also aware of API relations, they cannot capture implicit mod-ref relationships. For example, Domato uses manually-labeled parameter relationships to identify that an API may share the same input with the another. However, some mod-ref relationships can exist between two APIs even though they are type-independent.

Table 2: Compactness improvement of Minerva compared to other fuzzers over 5 runs in 24 hours.

Browser	vs. Domato			vs. FreeDom		
	min-impr	avg-impr	max-impr	min-impr	avg-impr	max-impr
WebKitGTK	+40.09%	+52.70%	+93.89%	+41.61%	+76.21%	+109.44%
FireFox	+12.22%	+25.99%	+49.48%	+63.39%	+103.09%	+171.24%
Chromium	+31.31%	+38.85%	+49.72%	+16.65%	+26.70%	+44.87%

Unique Bugs. We deduplicate each crash by its root cause reported by ASan. On FireFox and Chromium, all fuzzers did not trigger any ASan-reported bug within 24 hours. Figure 7 is a Venn diagram to demonstrate the overlapping relations among bugs found by each fuzzer on WebKitGTK. Favocado is omitted because it does not find any crashes in our experiments. We can see that all but one of the unique bugs can be found by Minerva. Compared to Domato, Minerva not only covers all the three unique bugs of it, but also finds 6 more other unique bugs. Compared to FreeDom, Minerva finds 6 other unique bugs.

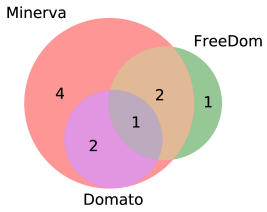


Figure 7: The overlapping relations of the unique bugs found by each fuzzer on WebKitGTK in 24 hours.

Answer for RQ1. Minerva achieves higher coverage as well as higher test case compactness compared to existing browser fuzzers. This means that Minerva can test browser APIs both broadly and deeply, and thus discovers more bugs than others.

6.2 Effectiveness of Redundancy Reduction

We compare different settings of Minerva to systematically understand how mod-ref relations impact fuzzing performance. We first compare Minerva to the one without mod-ref guidance, namely Minerva^0, to see if mod-ref relations really can make test cases less redundant. Second, we compare two different graph building strategies, i.e. visiting generated test cases or real-world websites, to figure out which one is more suitable to learn API relations. Moreover, we also investigate how the amount of API relations impact fuzzing performance.

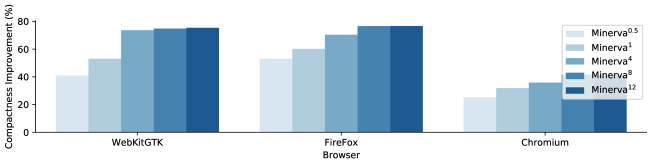


Figure 8: The average compactness improvements of different time settings compared to the non-relation guided MINERVA. MINERVA^h denotes a MINERVA instance guided by h-hours of building mod-ref relations.

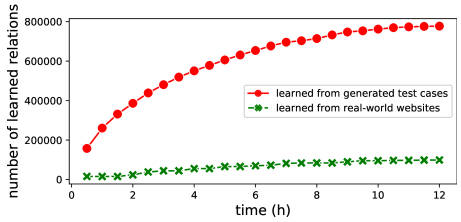


Figure 9: The number of mod-ref API pairs learned from different workload input types over 12 hours on WebKitGTK.

Different Settings of Graph Building. We conduct experiments to understand the contribution of mod-ref relations to fuzzing. Besides, we also investigate how the time of building graphs impacts MINERVA's effectiveness. In figure 8, we calculate the average compactness improvements of different MINERVA^h compared to MINERVA^0. The superscript h on MINERVA^h identifies that we use h hours to build mod-ref relations. For example, MINERVA^1 means that we run the build graph process for 1 hour and then use MINERVA guided by this graph to generate test cases for our experiments. Moreover, the superscript 0 means no graph building process is executed, so the MINERVA^0 runs without any mod-ref guidance. The MINERVA^8 is the fuzzer we use in Section 6.1.

We can see that all the graph-guided MINERVA significantly improve the compactness of MINERVA^0 by 25.25% to 75.39%. This result indicates that the mod-ref relations helps fuzzer generate test cases with higher compactness. With those test cases that contain the higher correlated invocations, the fuzzer is more likely to explore the unexpected states of browsers. Besides, as the time spent in building graph grows, MINERVA achieves higher compactness of test cases. However, there is no significant difference between MINERVA^8 and MINERVA^{12}. This is because most of the relations are found in the first 8 hours.

Moreover, we also investigate the overhead that the mod-ref relation guidance introduces to our fuzzer. On average, MINERVA^0 and MINERVA spend 0.070 and 0.099 seconds, respectively, in generating a test case. The difference is negligible because browser's execution takes most of fuzzing time (roughly 0.1 to 10 executions/second). Browser fuzzing suffers from low throughput [28], so all end-to-end browser fuzzers focus on generating more meaningful test cases rather than boosting the speed of test case generation. Hence, we believe that the mod-ref relation guidance helps our fuzzer generate less redundant test cases at negligible overhead.

Different Workload Types of Graph Building. To collect workload inputs, we adopt two straight-forward approaches: visiting real-world websites or fuzzer-generated test cases. Here we conduct experiments to investigate which type of workload input can bring more mod-ref information. For automatically visiting real-world websites, we write a script to let the instrumented browser visit Alexa top 500 websites [1] and randomly click the links of those websites to jump to other web pages. For visiting generated test cases, we use non-relation-guided MINERVA to generate workloads.

Figure 9 shows the number growth of relations learned from each type. Compare to real-world websites, the generated test cases introduce much more API relation information. In the 12-hour graph building process, visiting real-world websites brings 98691 mod-ref API pairs, which are much less than visiting generated test cases does. This is because the APIs used by real-world websites are not abundant: most of them are requesting resource from outsides, setting styles to DOM elements or attaching event listeners to elements. In contrast, visiting generated test cases can provide more diverse APIs. Covering diverse APIs is important because those browser APIs that are seldom used by website developers could be meaningful to vulnerability discovery. For example, the methods of `FontFace` are seldom used, because most websites do not need to change font face during page loading. However, the internal states of `FontFace` are complicated: once a font face is changed, all elements that use this font are notified to repaint. Its complexity makes it worthwhile to be covered. MINERVA discovers a heap-buffer-overflow vulnerability in the APIs related to `FontFace` on Safari. We will provide a detailed case study for it in Section 6.3.

Impact of the Amount of Mod-Ref Relations. During 24-hours of fuzzing, 99.94% mod-ref relations are used at least once, showing that our guided input generation can thoroughly exploit collected relations. Besides, according to Figure 8 and Figure 9, Minerva's effectiveness and the amount of relations have a positive correlation. The fuzzer guided by the 12-hour analysis (777,268 relations) improves the ones guided by the 0.5-hour analysis (157,590 relations) and the 1-hour analysis (260,812 relations) by 24.40% and 14.64%, respectively.

Answer for RQ2. The API mod-ref relations are helpful to reduce redundancy. Guided by more abundant mod-ref relations, the fuzzer generates more compact test cases.

6.3 Discovering Unknown Browser Bugs

To evaluate the ability of MINERVA in bug discovery, we intermittently run MINERVA for finding bugs in three mainstream browsers for a month. Table 3 shows a summary of all the bugs that MINERVA found in Safari (WebKit), FireFox and Chromium. In total, MINERVA found 35 bugs across diverse browser components, including Font, Iframe, CoordinatedGraphics. 26 bugs have been confirmed out of which 20 bugs have been fixed. Five of the bugs are assigned CVEs and acknowledged by vendors because of their severe security consequences. Two of the CVEs are rated as high impact (8.8/10 CVSS score) by the National Vulnerability Database [42]. Since all the browsers have been heavily tested for several years [13, 14], we believe that the result shows that MINERVA is able to discover previously-unknown bugs in diverse components of browsers.

Case Study. MINERVA identified a heap-buffer-overflow vulnerability (ID 16 in table 3), which may allow attackers to execute code

Table 3: Previously unknown bugs detected by MINERVA.

ID	Browser	Description	Component	status
1	Safari (WebKit)	null dereference	Touch	fixed
2	Safari (WebKit)	heap use after free	SVG	CVE-2021-45482
3	Safari (WebKit)	heap use after free	CoordinatedGraphics	CVE-2021-45483
4	Safari (WebKit)	null dereference	Accessibility	fixed
5	Safari (WebKit)	null dereference	CoordinatedGraphics	fixed
6	Safari (WebKit)	memory corruption	Font	fixed
7	Safari (WebKit)	null dereference	Paint	confirmed
8	Safari (WebKit)	null dereference	Style	fixed
9	Safari (WebKit)	null dereference	FrameView	fixed
10	Safari (WebKit)	null dereference	Paint	confirmed
11	Safari (WebKit)	out of memory	ImageBuffer	CVE-2021-45481
12	Safari (WebKit)	null dereference	Font	fixed
13	Safari (WebKit)	null dereference	FrameLoader	fixed
14	Safari (WebKit)	null dereference	Canvas	reported
15	Safari (WebKit)	heap use after free	Iframe	CVE-2021-30936
16	Safari (WebKit)	heap buffer overflow	Font	CVE-2021-30889
17	Safari (WebKit)	null dereference	Audio	fixed
18	Safari (WebKit)	memory corruption	Paint	reported
19	Safari (WebKit)	heap use after free	IndexedDB	fixed
20	Chromium	assertion failure	Paint	fixed
21	Chromium	assertion failure	Canvas	reported
22	Chromium	assertion failure	Notifications	reported
23	Chromium	assertion failure	WebRTC	reported
24	Chromium	out of memory	Paint	confirmed
25	Chromium	assertion failure	Paint	fixed
26	Chromium	assertion failure	Paint	fixed
27	Chromium	assertion failure	CaptureFromElement	confirmed
28	Chromium	assertion failure	Mojo	confirmed
29	Chromium	assertion failure	Layout	reported
30	FireFox	assertion failure	Dom:Workers	fixed
31	FireFox	assertion failure	SVG	fixed
32	FireFox	assertion failure	Panning and Zooming	reported
33	FireFox	assertion failure	Panning and Zooming	reported
34	FireFox	out of memory	Graphics:WebRender	comfirmed
35	FireFox	assertion failure	Web Painting	reported

Listing 1: CVE-2021-30889 on Safari detected by MINERVA. The code snippet triggers a heap buffer overflow when setting the `family` variable of a `FontFace` object.

```
<html>
<head>
<script>
function jsfuzzer() {
    var svgvar00001 = document.getElementById("svgvar00001");
    var htmlvar00015 = document.getElementById("htmlvar00015");
    svgvar00001.appendChild(htmlvar00015);
    var var00120 = htmlvar00015.firstChild;
    htmlvar00015.removeChild(var00120);
    var var00240 = document.fonts;
    var var00241 = new FontFace("foo", null);
    var var00239 = var00240.add(var00241);
    var00241.family = "bar"; // <- heap buffer overflow
}
</script>
</head>
<body onload=jsfuzzer()>
<svg id="svgvar00001">
    <font id="htmlvar00015">
    <font-face font-family="monospace">
    </font-face>
    </font>
</svg>
</body>
</html>
```

remotely. Listing 1 shows a PoC code snippet generated by MINERVA. The code snippet involves multiple interface objects, including `SVGSVGElement` (`svgvar00001`), `FontFaceSet` (`htmlvar00015`, `var00239` and `var00240`) and `FontFace` (`var00120` and `var00241`). The PoC first gets `SVGSVGElement` and `FontFaceSet` objects from current document, and then appends and removes their child (Line 5-9). Next, it gets a `FontFaceSet` object from current document and adds a new `FontFace` object to the set (Line 10-12). Finally, it sets the `family` variable of the `FontFace` object to a string, leading to a heap buffer overflow (Line 13). Although the setting statement seems very simple and bug-free, its execution is very complicated. Once a `FontFace` is changed, all elements that use this font are notified to repaint. States of multiple internal render objects are accessed and modified during the execution, and a unchecked illegal memory access leads to the overflow. Although a `SVGSVGElement` object seems irrelevant to a `FontFace` object, APIs of the former still interferes with that of the latter.

The implicit relations mentioned above can be captured by MINERVA, and thus it is able to take little time to trigger the bug. MINERVA discovers it within 14 hours. By contrast, we continuously ran the three state-of-the-art fuzzers for 1 week and they did not trigger this bug. This bug was assigned CVE-2021-30889 by Apple Inc. with acknowledgement and has been fixed since Safari 15.1.

Answer for RQ3. MINERVA discovers high-severity real-world bugs in production-level browsers. The API mod-ref graph significantly improves the efficiency of bug detection.

7 DISCUSSION

We now discuss limitations of MINERVA and future directions for testing browser APIs.

Overhead of Instrumentation. Instrumenting load and store instructions for tracing memory access brings considerable overhead. Although we have mitigated the overhead, the instrumented browser is, on average, 3x slower than the non-instrumented version. However, MINERVA unnecessarily traces every execution if there are no newly-discovered relations. MINERVA adopts an offline analysis and thus the overhead is acceptable: once a graph is built, MINERVA switches to ASan-instrumented browsers for vulnerability detection. We plan to further eliminate unnecessary instrumentation points and support online analysis in the future.

Over- and Under-Approximation of API Relations. Our dynamic approach for building API dependence graphs is unlikely to result in over-approximation. The only situation MINERVA cannot deal with is if a memory address is freed and reallocated for another usage. In this case, our analysis will determine that the API modifying the former memory address interferes with the one that uses the latter memory address. Note that this situation rarely happens in practice. This limitation is shared with most dynamic bug detection tools, e.g. ASan [38]. We mitigate it by not only comparing memory address, but also considering the operation size of load and store instructions. Besides, our instrumentation only traces the memory objects that can be shared between APIs. Local memory objects are thus excluded.

Under-approximation, i.e. missing relations, may reduce the chance of testing some APIs during fuzzing campaigns of MINERVA. We investigate the coverage difference between MINERVA and DOMATO to see if there are missed relations. As Table 4 shows, about 3% to 4% edges are covered by DOMATO while not covered by MINERVA. However, the learned relations brings significant extra coverage (~ 20%), showing that MINERVA deeply explores the API relations even though it overlooks a very small part of them.

Imprecise Information of API Description. From our observation, though API descriptions provide a way to generate semantically-correct test cases, its imprecise type property could

Table 4: Difference set of MINERVA coverage set (M) and DOMATO coverage set (D). Each coverage set is the union of five-time 24-hour results.

	$size_{M-D}$	$size_{D-M}$	$size_M$	$size_D$
WebKitGTK	60369 (19.57%)	12457 (4.04%)	308460	260548
FireFox	85875 (22.25%)	12895 (3.34%)	385935	312955
Chromium	115751 (21.82%)	20824 (3.93%)	530422	435495

prevent fuzzers from digging into deep logic. For example, the `setAttribute(<DOMString>,<DOMString>)` only shows that it accepts two string variables as arguments, however, the magic string of each parameter could lead to totally different handle logic. We mitigate it by reusing labeled attribute information from DOMATO. Recently, SyzGen [8] recovers interfaces of close-source MacOS Drivers by iterative refinement of syscall knowledge. We plan to integrate the refinement strategy in the future.

Relations across Multiple API Invocations. Our approach currently focuses on the relation between two APIs. However, inter-API relations could be more complex, e.g., an API can interfere with the other one only if an intermediate API is invoked. Such conditional relations are challenging to model. We note that this case is rare in browsers and only a few APIs, e.g., `Node.appendChild`, may play such an intermediate role. These intermediate APIs are naturally prioritized at our selection stages since they access a large amount of shared data for changing the browser's state. Thus, our approach implicitly handles such complex relations. We plan to use a n-gram model to further reveal multiple inter-API relations.

8 RELATED WORK

Generic Fuzzing. Fuzzing has been proven to be a practical technique on bug detection. To enhence fuzzing effectiveness, a large number of security researchers proposed optimizations from different angles, e.g. boosting coverage tracing [49, 50, 55], improving seed selection strategy [5, 18, 33] or mutation strategy [6, 26], leveraging taint analysis [7, 12, 22] or symbolic execution [32, 40, 54]. A common limitation of these fuzzers is the lack of ability to handle highly-structured inputs. To meet this end, some extensions [4, 39, 48] are proposed to leverage context-free grammar to assist their tree-based mutation. However, none of them can be directly applied to browser API fuzzing because they focus on syntactic correctness while lack support for semantic validity.

Browser Fuzzing. The most relevant works are DOM fuzzers [35, 36, 44, 53]. Domato [44], the most successful industrial DOM fuzzer, leverages hand-written grammar and semantics to generate DOM API invocations. FreeDom [53] classifies APIs into various fuzzing operations based on their functionalities and designs several generation and mutation strategies for maintaining context information. Favocado [10], on the other hand, focuses on fuzzing binding code. *Different from them, MINERVA automatically builds API relations for API-dependent invocation generation while others only consider a small amount of selected API relations.* In addition, fuzzing JavaScript engine has become an active research area recently and many techniques are proposed to preserve semantic validity during their mutation [16, 17, 31]. These works are orthogonal to MINERVA since they aim at different fuzzing targets.

Interface Analysis for Fuzzing. Several static interface analysis techniques [8, 20, 30] are proposed to enhance fuzzing effectiveness from different angles. FuzzGen [20] statically analyzes data dependence of test cases for fuzz driver synthesis. SyzGen [8] recovers interfaces of close-source MacOS Drivers by iterative refinement of syscall knowledge. Moonshine [30] leverage static analysis for detecting dependencies across syscalls. Different from them, MINERVA focuses on identifying interference relations between browser APIs through a novel dynamic mod-ref analysis. Currently, no off-the-shelf technique can meet this end. The static approaches are unlikely precise due to the complexity of browsers. For example, JIT may prevent static analysis from digging into back end logic. *By contrast, MINERVA can analyze the internal states of the browsers during every browser API invocation, producing more accurate relations.*

Some OS kernel fuzzers [41, 52] also rely on dynamic interface analysis for search space reduction. For example, Healer [41] analyzes coverage impact of each syscall-pair for its generation. Krace [52] leverages thread interleaving behaviors as alias coverage to guide its generation. However, *none of these fuzzers can be applied to browser fuzzing because of the non-deterministic behaviors of browsers.* The state of OS kernels is quite *stable* when executing each syscall, i.e. only a specific module is activated and all behaviors are input-related; while a browser has multiple processes and each spawns many background threads that concurrently deal with input-unrelated tasks, e.g. communication to other processes or resource requests from the network. As a result, the coverage and control flow could differ in browsers even when the same invocation is executed, rendering their interface analysis ineffective, no matter for syscall coverage impact or thread interleaving behaviors.

9 CONCLUSION

MINERVA is an efficient browser fuzzer for browser API bug detection. Our key idea is to leverage API interference relations to reduce redundancy and improve coverage. Our fuzzer consists of two modules: *dynamic mod-ref analysis* for building API interference relations and *guided code generation* for synthesizing high-relevant test cases. We evaluate its performance on three mainstream browsers, i.e. Safari, FireFox, and Chromium. Compared to state-of-the-art fuzzers such as DOMATO, FREEDOM, or FAVOCADO, MINERVA improves edge coverage by 19.63%, 24.90% and 229.62% on average, respectively. Furthermore, MINERVA finds 2x to 3x more unique bugs compared to other fuzzers. In addition, MINERVA has discovered 35 previously-unknown bugs; 20 have been fixed with 5 CVEs assigned and acknowledged by vendors so far.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706200, No2021QY0604), ERC under the H2020 research grant 850868, and SNSF grant number PCEGP2-186974.

REFERENCES

[1] alexa. 2021. The top 500 sites on the web. https://www.alexa.com/topsites. (Online; visited on December 20, 2021).

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. Citeseer.

[3] Apple. 2021. Apple security updates. https://support.apple.com/en-us/HT201222. (Online; visited on December 20, 2021).

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[5] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 678–689. https://doi.org/10.1145/3368089.3409748

[6] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 725–741. https://doi.org/10.1109/SP.2015.50

[7] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725. https://doi.org/10.1109/SP.2018.00046

[8] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 749–763. https://doi.org/10.1145/3460120.3484564

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1967–1983. https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[10] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.

[11] Pavlos S Efraimidis and Paul G Spirakis. 2006. Weighted random sampling with a reservoir. *Inform. Process. Lett.* 97, 5 (2006), 181–185.

[12] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[13] Google. 2016. ClusterFuzz. https://google.github.io/clusterfuzz/. (Online; visited on December 20, 2021).

[14] Google. 2016. OSSFuzz. https://github.com/google/oss-fuzz. (Online; visited on December 20, 2021).

[15] Google. 2021. Chrome Vulnerability Reward Program Rules. hhttps://bughunters.google.com/about/rules/5745167867576320. (Online; visited on December 20, 2021).

[16] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[17] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2229–2242. https://doi.org/10.1145/3460120.3484823

[18] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed selection for successful fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 230–243. https://doi.org/10.1145/3460319.3464795

[19] ihgazni2. 2019. webidl2-mozilla-experimental. https://www.npmjs.com/package/webidl2-mozilla-experimental/v/1.0.5. (Online; visited on December 20, 2021).

[20] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2271–2287. https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[22] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*. 154–170.

[23] llvm team. 2021. Clang 13 documentation: SANITIZERCOVERAGE. https://clang.llvm.org/docs/SanitizerCoverage.html. (Online; visited on December 20, 2021).

[24] llvm team. 2021. The LLVM Compiler Infrastructure. https://llvm.org/. (Online; visited on December 20, 2021).

[25] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function Code Aware Fuzz Testing of ICS Protocol. *ACM Trans. Embed. Comput. Syst.* 18, 5s (2019), 93:1–93:22. https://doi.org/10.1145/3358227

[26] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[27] Microsoft. 2021. Microsoft Edge Bounty Program. https://www.microsoft.com/en-us/msrc/bounty-new-edge. (Online; visited on December 20, 2021).

[28] mozilla. 2021. Fuzzing. https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html. (Online; visited on December 20, 2021).

[29] Mozilla. 2021. Security Bug Bounty Program. https://www.mozilla.org/en-US/security/bug-bounty/. (Online; visited on December 20, 2021).

[30] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[31] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[32] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[33] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[34] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2795–2809. https://www.usenix.org/conference/usenixsecurity21/presentation/salls

[35] Mozilla Security. 2015. dharma: Generation-based, context-free grammar fuzzer. https://github.com/MozillaSecurity/dharma. (Online; visited on December 20, 2021).

[36] Mozilla Security. 2016. Avalanche. https://github.com/MozillaSecurity/avalanche. (Online; visited on December 20, 2021).

[37] selenium team. 2021. Selenium automates browsers. That's it! https://www.selenium.dev/. (Online; visited on December 20, 2021).

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[39] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 244–256. https://doi.org/10.1145/3460319.3464814

[40] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[41] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 344–358. https://doi.org/10.1145/3477132.3483547

[42] NVD team. 2021. NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov/. (Online; visited on December 20, 2021).

[43] WebKitGTK team. 2021. WebKitGTK. https://webkitgtk.org/. (Online; visited on December 20, 2021).

[44] the Project Zero team at Google. 2017. Domato: A DOM fuzzer. https://github.com/googleprojectzero/domato. (Online; visited on December 20, 2021).

[45] the Project Zero team at Google. 2021. This shouldn't have happened: A vulnerability postmortem. https://googleprojectzero.blogspot.com/2021/12/this-shouldnt-have-happened.html. (Online; visited on December 20, 2021).

[46] W3C. 2021. WebIDL Level 1. https://webidl.spec.whatwg.org/. (Online; visited on December 20, 2021).

[47] w3c. 2021. WebIDL parser. https://github.com/w3c/webidl2.js. (Online; visited on December 20, 2021).

[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[49] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 147–159. https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

[50] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1010–1024.

[51] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: Detect Runtime Errors in Time-Series Databases With Hybrid Input Synthesis. In *ISSTA '22: 31th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, South Korea, July 18–22, 2022*.

[52] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[53] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 971–986. https://doi.org/10.1145/3372297.3423340

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[55] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 858–870. https://doi.org/10.1145/3324884.3416572

[56] Feilong Zuo, Zhengxiong Luo, Junze Yu, Zhe Liu, and Yu Jiang. 2021. PAVFuzz: State-Sensitive Fuzz Testing of Protocols in Autonomous Vehicles. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 823–828. https://doi.org/10.1109/DAC18074.2021.9586321