Automated Exploit Generation for Node.js Packages

FILIPE MARQUES, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
MAFALDA FERREIRA, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
ANDRÉ NASCIMENTO, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
MIGUEL COIMBRA, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
NUNO SANTOS, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
LIMIN JIA, Carnegie Mellon University, USA
JOSÉ FRAGOSO SANTOS, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal

The Node.js ecosystem, with its growing popularity and increasing exposure to security vulnerabilities, has a pressing need for more effective security analysis tools. To reduce false positives, recent works on detecting vulnerabilities in Node.js packages have developed synthesis algorithms to generate proof-of-concept exploits. However, these tools focus mainly on vulnerabilities that can be triggered by a single direct call to an exported function of the analyzed package, failing to generate exploits that require more complex interactions.

In this paper, we present Explode.js, the first tool capable of synthesizing exploits that include complex call sequences to trigger vulnerabilities in Node.js packages. By combining static analysis and symbolic execution, Explode.js generates functional exploits that confirm the existence of command, code injection, prototype pollution, and path traversal vulnerabilities, effectively eliminating false positives. The results of evaluating Explode.js on two state-of-the-art datasets of Node.js packages with confirmed vulnerabilities show that it generates significantly more exploits than its main competitor tools. Furthermore, when applied to real-world Node.js packages, Explode.js uncovered 44 zero-day vulnerabilities, with 4 new CVEs.

CCS Concepts: • **Security and privacy** → **Software security engineering**; *Domain-specific security and privacy architectures*; Malware and its mitigation; • **Software and its engineering** → *Software verification and validation*; • **Theory of computation** → *Logic and verification*; • **Applied computing** → *System testing and validation.*

Additional Key Words and Phrases: Automatic Vulnerability Detection, Symbolic Execution, Exploit Generation, Node.js Security

ACM Reference Format:
Filipe Marques, Mafalda Ferreira, André Nascimento, Miguel Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2025. Automated Exploit Generation for Node.js Packages. *Proc. ACM Program. Lang.* 9, PLDI, Article 201 (June 2025), 26 pages. https://doi.org/10.1145/3729304

1 Introduction

Exploit generation is an effective approach for confirming security vulnerabilities, with various techniques now available for a range of programming languages, including C/C++ [14], Python [68], and JavaScript (JS) [15, 36]. A key appeal of exploit generation lies in its potential to address the

Authors' Contact Information: Filipe Marques, filipe.s.marques@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal; Mafalda Ferreira, mafalda.baptista@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal; André Nascimento, andreffnascimento@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal; Miguel Coimbra, miguel.e.coimbra@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal; Nuno Santos, nuno.m.santos@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal; Limin Jia, liminjia@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; José Fragoso Santos, jose.fragoso@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal.

high false positive rates common in static analysis and other vulnerability detection tools [10]. False positives place a considerable burden on developers, who must manually verify flagged vulnerabilities, often without clear indicators of actual risk [10, 70]. By automatically generating a functional exploit, developers can quickly determine whether a vulnerability poses a real risk, reducing the need for manual confirmation and accelerating the resolution of security issues.

Exploit generation tools are valuable for detecting security vulnerabilities in Node.js, a JS runtime built on top of the V8 engine [32] that enables server-side execution of JS code. The open nature of the Node.js platform [56], where packages often import numerous others without security guarantees, makes it rife with security vulnerabilities [44, 49, 61]. However, synthesizing workable exploits to confirm such vulnerabilities is challenging due to exploit generation depending heavily on the syntax and semantics of the target language, making techniques from other languages unsuitable for JS. The unique intricacies of the Node.js ecosystem complicate this task further: it is common for vulnerabilities to be deeply nested within Node.js packages. As a result, to trigger a vulnerability, the exploit may need to call several APIs in a specific order, use callbacks, and supply APIs with inputs of the correct type and structure.

Existing exploit generation tools for Node.js packages [15, 16, 36] have focused exclusively on *single-interaction vulnerabilities*, i.e., vulnerabilities that can be exploited by calling a single exported function of the vulnerable package. NodeMedic-Fine [15] implements dynamic taint analysis to detect injection vulnerabilities, executing the package using fuzzer-generated inputs and producing taint provenance graphs, which are then used to synthesize exploits. A shortcoming of this approach is that its effectiveness heavily depends on the coverage of the used fuzzer. FAST [36] builds an enriched control flow graph (CFG) based on a coarse-grained abstract JS semantics, and traverses it to collect the constraints that guide execution from exported functions to sensitive sinks. It does not, however, generate constraints that guarantee the occurrence of observable, attacker-controlled side effects. As a result, some candidate exploits synthesized by FAST fail to execute due to the imprecisions in the used semantics, while others fail to produce the desired side effects.

We present Explode.js, the first exploit generation tool for Node.js packages that can automatically synthesize functional exploits for *multi-interaction vulnerabilities*—vulnerabilities that require a series of interactions with the vulnerable package to be exploited. Given a vulnerable package, Explode.js can identify code injection, command injection, prototype pollution, and path traversal vulnerabilities, and build a JS program that is guaranteed to produce observable side effects, thereby confirming the existence of the vulnerability and eliminating false positives.

Explode.js leverages a novel exploit generation algorithm that combines static analysis with symbolic execution (SE) in a two-stage pipeline. First, Explode.js computes an *exploit template*, consisting of a chain of calls to the functions of the package, where the arguments to each call are symbolic values. The call chain, if provided with the right arguments, could lead to the execution of a targeted sensitive sink with attacker-controlled inputs. In this stage, Explode.js determines which functions need to be called to reach the targeted sensitive sink, the order in which they should be called, and the type of their corresponding arguments. We organize this information in an intermediate structure called *vulnerable interaction scheme* (VIS), from which the tool subsequently creates the exploit template. In the second stage, Explode.js symbolically executes the exploit template, aiming to find a control path to the vulnerable sink. If such a path is found, the path constraints identified by the symbolic execution engine are extended with additional constraints to ensure that the arguments passed to the sink result in an attacker-controlled effect. Then, Explode.js uses a Satisfiability Modulo Theories (SMT) solver to generate concrete inputs that satisfy these constraints. If such inputs are found, a functional exploit is generated by replacing the symbolic variables in the exploit template with the corresponding concrete values.

Our implementation of Explode.js leverages Graph.js [23] to generate a graph-based representation of the program, which serves as the foundation for creating the VIS and subsequent exploit templates. Additionally, we implemented a custom-made SE engine, called ECMA-Symb, to symbolically execute the generated exploit templates. While there are several SE engines for JavaScript [24–26, 57], we cannot use them directly, as they operate on a file-by-file basis. To use them, we would have to transform the analyzed package, along with its dependencies, into a single file, using a bundler such as Webpack [67], Parcel [18], or esbuild [21]. This approach would lead to significantly slower SE time (§6.5), as it increases exponentially with the size of the code being analyzed. Instead, the SE engine of Explode.js offers native support for lazy values [39], allowing for the execution of the package under analysis without access to the code of its dependencies.

To evaluate Explode.js, we first compare its exploit-generation effectiveness against FAST and NodeMedic-Fine using two state-of-the-art curated datasets of Node.js packages with reported vulnerabilities [9, 10], showing that Explode.js finds 4.02× more exploits than FAST and 6.61× more exploits than NodeMedic-Fine. Next, we apply Explode.js to real-world Node.js applications, where it finds 44 new zero-day security vulnerabilities, for which 4 new CVEs have been assigned. Finally, we compare the number of exploits generated with and without VISes and lazy values, obtaining results that indicate that these two techniques are essential for the effectiveness of Explode.js.

2 Overview

We use the example of a vulnerable Node.js package (§2.1) to describe how Explode.js synthesizes a functional exploit for the package's vulnerability (§2.2).

2.1 Motivating Example

The code in Fig. 1a depicts an example vulnerable Node.js package that allows clients to upload data to a remote host. This example reflects vulnerabilities of similar nature and complexity found in real-world packages. This package exports a `FileTransfer` class that manages file uploads by providing methods for preparing and executing the transfer. The `uploadData` method (lines 23-35) receives the data to be uploaded, stores it in a file if the `useTempFiles` flag is enabled, and returns a new object with a `run` property. This property contains a callback that references the `uploadFile` function (lines 4-14), which, in turn, performs the transfer using the `rsync` command. This function first checks that the provided file does not use more disk space than allowed (line 5), in which case it builds the command string and passes it to `execSync` to be executed. This example contains an exploitable command injection vulnerability (see Fig. 1d) that allows attacker-controlled inputs to reach the `execSync` function (line 10), potentially executing unintended shell commands. Automatically generating a working exploit for this example poses two key challenges:

Challenge 1: Generating a Source-to-Sink Call Chain. The `uploadFile` function, where the `execSync` call is located, is not directly accessible to package clients. To reach this sink, a client must execute a specific sequence of function calls, first creating a new `FileTransfer` instance and then invoking the `uploadData` method on that instance. This method returns an object containing a `run` callback that will call `uploadFile`. Automatically identifying this chain of function calls is challenging, as Node.js packages often expose functions whose inputs and outputs have diverse types, including objects and callbacks.

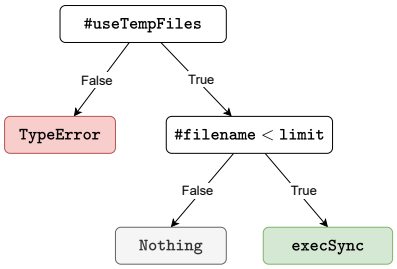
Challenge 2: Crafting an Effectful Payload. Even when knowing a call chain that reaches the `execSync` sink, an exploit must provide a specific payload that will trigger unintended side effects. This includes forming a syntactically valid shell command that results in malicious behavior, such as, for example, deleting all local files by appending "`; rm -f *`" to the command string for `rsync` (line 6). Generating inputs for such exploits requires a thorough exploration of all executions paths.

```
const { execSync } = require("child_process")
const fs = require("fs-extra");

function uploadFile(filename, limit, userid,
↪  userdir, host) {
  if (fs.statSync(filename).size < limit) {
    var command = "rsync -av"
      + filename + " "
      + userid + "@" + host + ":"
      + userdir;
    return execSync(command);
  }
  console.log("File too big");
  return null;
}

module.exports = class FileTransfer {
  constructor(options) {
    this.host = options.host
    this.limit = 1 * 1024 * 1024 * 1024 // 1 GiB
    this.useTempFiles = options.useTempFiles || false
  }

  uploadData(data) {
    if (this.useTempFiles && data != "") {
      tmpfile = `/tmp/${Date.now()}`
      fs.writeFileSync(tmpfile, data)
      return {
        run: (user) =>
          uploadFile(tmpfile, this.limit, user.id,
          user.dstDir, this.host)
        };
      }
     ...
    return null;
  }
}
```

(a) JavaScript code of the file uploads module.

```
let FileTransfer = require("file-transfer");
let options = { host : symStr()
              , useTempFiles: symBool() };
// VIS Call #1: New<FileTransfer>
let m = new FileTransfer(options);
let data = symStr();
// VIS Call #2: Mthd<uploadData>
let result = m.uploadData(data);
let user = { id: symStr(), dstDir: symStr() };
// VIS Call #3: Mthd<run>
result.run(user)
```

(b) Generated exploit template with symbolic inputs.



(c) Symbolic execution tree of the exploit template.

```
let FileTransfer = require("file-transfer");
let m = new FileTransfer({ host: "foobar.baz",
↪  useTempFiles: true });
let data = "sRegTD";
let res = m.uploadData(data);
res.run({ id: "foo", dstDir: "bar; touch pwned.txt" });
```

(d) Synthetically generated exploit.

Fig. 1. Exploit synthesis for a Node.js package with a command injection vulnerability.

2.2 Explode.js in Action

Explode.js combines static analysis with SE in a two-stage pipeline to solve both challenges described above.

Stage 1: Generating Exploit Templates. The static analysis component of Explode.js first produces a VIS, that is, an intermediate structure that includes the chain of function calls needed to reach the sink together with their respective argument types. While a package may have multiple VISes, our example has a single one, shown below, where each line is a function call and the arrows indicate that each function should be applied to the return value of the previous call.

Call #1: *New*⟨`FileTransfer`, `options : { host : string; useTempFiles : boolean }`⟩
Call #2: ⟶ *Mthd*⟨`uploadData`, `data : string`⟩
Call #3: ⟶ *Mthd*⟨`run`, `user: { id : string; dstDir : string}`⟩

This VIS specifies that, to activate the vulnerability in the `run` method, the exploit acting as the package client must first call the `FileTransfer` constructor, which is exported by the package and expects an input object with properties `host` and `useTempFiles` respectively of types string and boolean (see Fig. 1a, lines 17-21). Next, the exploit must call the method `uploadData` on the object returned by the constructor with a string argument. Finally, the `run` method must be called on the object returned by `uploadData`, with an input object with properties `id` and `dstDir` of type string. In the VIS, we use the keywords *New* and *Mthd* to differentiate method calls from constructor calls.

Currently, Explode.js only computes linear VISes, where the i-th function to be called is obtained from the previous $(i-1)$-th call in the sequence. We discuss the impact of this limitation in §7.

After creating the VIS, we transform it into an *exploit template*, that is, a symbolic driver program that calls each function contained in the VIS using symbolic values of the appropriate type. Fig. 1b shows the exploit template generated for our example. Lines 5, 8, and 11 capture the VIS call chain, while Lines 2, 6, and 9 initialize the symbolic values passed to each call, using the primitives `symStr()` and `symBool()` for creating symbolic strings and booleans, respectively.

Stage 2: Resolving Exploit Templates. Having found an exploit template capable of reaching the vulnerable sink, the next step is to replace the symbolic inputs in the template with concrete values that produce unintended side effects. To this end, we use SE to find a control path from the template's source to the sensitive sink. If found, we extend the set of collected *path constraints* with *well-formedness constraints* that ensure that the sink input satisfies its corresponding syntactic rules; for instance, the input of a code injection sink must be a valid JS statement. Then, we use an SMT solver to search for concrete inputs that satisfy all the constraints; if such inputs are found, we inject them into the exploit template to produce a concrete exploit.

Returning to our example, Fig. 1c shows the simplified SE tree generated by Explode.js. The top-level node represents the first conditional encountered during SE, which appears on line 24 of the `uploadData` function. When `useTempFiles` is false, this function returns `null`, leading to a type error on line 11 of the exploit template, which attempts a null dereference (i.e., `null.run(user)`). Continuing along the branch where `useTempFiles` is true, the second conditional appears on line 5 of the `uploadFile` function. This function checks that the size of the file to be uploaded is below the specified threshold using the external function `statSync` of the `fs-extra` package. To support such calls, Explode.js uses a new form of *lazy initialization* [39], creating a fresh symbolic value that represents the file size. When this size is below the `limit` threshold, the program reaches the sensitive sink with the *symbolic payload*:

```
"rsync -av /tmp/1730296434234 " + id + "@" + host + ":" + dstDir
```

and the *path constraint* `useTempFiles` $\wedge$ `statSync(filename).size` $<$ `limit`. Importantly, the symbolic payload is obtained by concatenating a series of strings, of which `id`, `host`, and `dstDir` are attacker-controlled. However, to prove that the sink is exploitable, we must find concrete values for the symbolic variables that direct the execution along the identified control path and produce a valid sink payload containing an attack. To this end, we extend the generated path constraint with the well-formedness constraint:

ValidAttack(`"rsync -av /tmp/1730296434234 " + id + "@" + host + ":" + dstDir`)

that guarantees that the sink payload is a valid sequence of shell commands and contains an attack.

Finally, Explode.js queries an SMT solver for a model that satisfies the generated constraints, obtaining, for instance, the following concrete values for the symbolic variables in the template:

```
host = "foobar", useTempFiles = true, id = "foo", dstDir = "bar; touch pwned.txt \#"
```

Explode.js then replaces the symbolic variables in the template with the obtained concrete values, generating the concrete exploit given in Fig. 1d. The symbolic variables not appearing in the generated model (e.g., `data`) are assigned random values of the appropriate type. The execution of the generated exploit leads to a call to the sensitive sink `execSync` with the string:

```
"rsync -av /tmp/1731165952 foo@foobar:bar; touch pwned.txt \#"
```

resulting in the creation of the file `pwned.txt` and confirming the vulnerability.

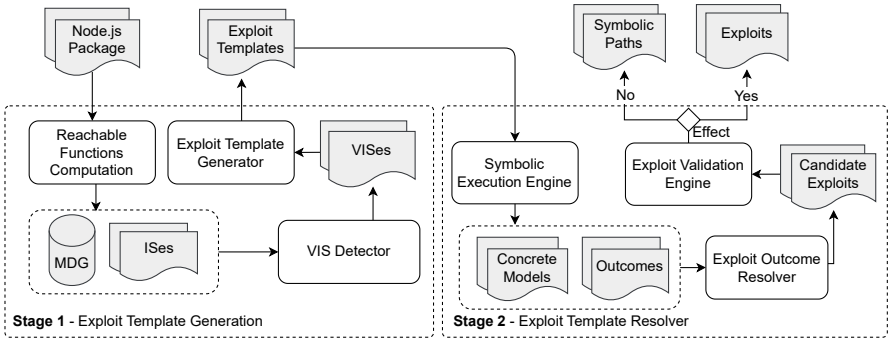


Fig. 2. Explode.js pipeline for vulnerability detection, exploit generation, and validation.

3 Explode.js

Fig. 2 presents the architecture of Explode.js. The tool takes a Node.js package as input and outputs a list of vulnerability reports, each corresponding to an identified vulnerability within the code. These reports include the control path leading to the vulnerable sink and, if generated, a concrete proof-of-concept (PoC) exploit that confirms the existence of the vulnerability. If Explode.js is unable to synthesize a concrete exploit, only the vulnerable control path is reported. Currently, Explode.js is able to detect and exploit path traversal (CWE-22), command injection (CWE-78), code injection (CWE-94) and prototype pollution (CWE-1321) vulnerabilities.

Explode.js is organized into several modules that implement its two main execution stages, as introduced in §2.2. Stage 1, which is responsible for generating exploit templates for each potential vulnerability identified in the code, operates through the following three modules:

(1) **Reachable Functions Computation:** This module identifies all attacker-controlled functions in the given Node.js package, i.e., functions that an attacker can directly call by interacting with the package. Each attacker-controlled function is associated with an *interaction scheme*, outlining the interactions required to obtain and execute the function. We describe the algorithm for identifying attacker-controlled functions and computing their interaction schemes in §4.1.
(2) **Vulnerable Interaction Scheme (VIS) Detector:** This module determines, for each sensitive sink, whether that sink is reachable from an attacker-controlled function with attacker-controlled inputs. To this end, it leverages Graph.js [23] to build a Multiversion Dependency Graph (MDG) of the package under analysis and executes sink-specific vulnerability-detection queries on the constructed graph to identify vulnerable data flows connecting attacker-controlled inputs to sensitive sinks. The interaction schemes associated with the functions in which such data flows originate are referred to as *vulnerable interaction schemes*.
(3) **Exploit Template Generator:** This module compiles each generated VIS into an exploit template that invokes the functions in the VIS in the correct order with symbolic arguments of the appropriate type. To determine the argument types, we implement an intra-procedural, flow-insensitive type analysis inspired by [35]. We describe the compilation of VISes in §4.2.

Taking each exploit template as input, Stage 2 attempts to identify a vulnerable control path and synthesize a concrete exploit. This process is achieved by the following modules:

(1) **Symbolic Execution Engine (ECMA-Symb):** This module employs ECMA-Symb, a custom-made SE engine developed specifically for Node.js, to symbolically execute exploit templates in order to identify control paths from the template's source to the targeted sensitive sink. For each path, it produces an *exploit outcome* comprising the type of the reached sink, the symbolic payload passed to the sink, and the generated path constraints. To avoid analyzing all external packages, we employ a novel form of *lazy initialization* [39], which models the

results of function invocations on third-party objects as special symbolic values whose behavior adapts according to the contexts in which they are used. If a lazy value flows into a sensitive sink, the tool reports a potentially vulnerable exploit outcome, which it then tries to concretize into a vulnerable exploit in the next two stages. This concretization may, however, fail because lazy values over-approximate the behavior of their associated external packages. In such cases, developers should provide symbolic summaries [55] capturing the precise behavior of the imported package. We formalize the symbolic semantics underpinning ECMA-Symb in §5.1.

(2) **Exploit Outcome Resolver:** Receiving as input an exploit outcome together with the corresponding exploit template, this module starts by constructing an SMT formula to find concrete inputs that direct the execution along the identified control path and that produce a valid sink payload containing an attack. If the solver finds a satisfying assignment, the obtained values are substituted into the exploit template, producing a candidate exploit. For the SMT solver, we use Z3 [17]. Our exploit resolution algorithm is formally described in §5.2.

(3) **Exploit Validation Engine:** Since our lazy initialization mechanism does not fully capture the behavior of imported packages, the candidate exploit may fail to produce the expected result. To prevent Explode.js from reporting false positives, this module executes the generated exploit in Node.js and checks if its execution produces an observable effect, reporting only those exploits for which such an effect is confirmed. The type of observable effect depends on the vulnerability; for instance, for code and command injection exploits, the engine checks if a file with a predetermined name is created in the local directory. If the effect is observed, the exploit is considered valid and presented to the user; otherwise, it is discarded, and only the potentially vulnerable control path is shown.

4 Exploit Template Generation

This section defines VISes, formalizes their construction algorithm and correctness criterion (§4.1), and describes our algorithm for converting VISes into exploit templates (§4.2).

4.1 Vulnerable Interaction Schemes (VISes)

The first step of Explode.js's core algorithm is to identify the attacker controlled functions of the given package, i.e., the functions that an attacker can directly call via one or more interactions with the package. To this end, we introduce the notion of *interaction scheme*, $\sigma \in \Sigma$. We say that σ is an interaction scheme for function f if it comprises a sequence of calls to the functions, methods, and constructors of the given package that ends with a direct call to f. Essentially, an interaction scheme describes how to obtain and execute a function from the package, starting from the exported object and progressing through a sequence of interactions with the objects and functions returned by the package. A limitation of our approach is that it only generates linear interaction schemes, i.e., schemes where calling a function requires it to be either exported by the package or returned by the previous call within the scheme. Nevertheless, our experiments indicate that these schemes are sufficiently expressive to identify most vulnerabilities in real-world Node.js packages.

Formally, we define an interaction scheme as a sequence $c_1 \triangleright \ldots \triangleright c_n$ of call actions $c \in C$, where a call action consists of a pair (f, ϕ), abbreviated as f_ϕ, comprising a function identifier $f \in \mathcal{F}$ and a call type $\phi \in \Phi$. Call types are given by the grammar $\phi \in \Phi ::= \mathit{call} \mid \mathit{new} \mid \mathit{mthd}(p)$, where *call* denotes a function call, *new* a constructor invocation, and $\mathit{mthd}(p)$ a method call on property p. To simplify the notation, we omit the name of the property holding the method when it coincides with the method's identifier, writing f_{mthd} instead of $f_{\mathit{mthd}(f)}$. For instance, the interaction sequence for the arrow function `run` (line 28) of Fig. 1a is:

$$\texttt{FileTransfer}_{\mathit{new}} \triangleright \texttt{getRemoteUploadData}_{\mathit{mthd}} \triangleright \texttt{run}_{\mathit{mthd}}$$

Given a package P, we say that $\sigma \in \Sigma$ is a valid interaction scheme for a function $f \in \mathcal{F}$, if it consists of a sequence of call actions that, starting from the package's exported object, leads to the execution of f. We formalize this definition below.

Definition 4.1 (Valid Interaction Scheme). An interaction scheme $\sigma \in \Sigma$ for a function $f \in \mathcal{F}$ is said to be valid for a package P, written $P \vdash \sigma \gg f$, if it can be derived using the following rules:

$$\textsc{FunExported}\ \frac{f \in P.\text{exported} \qquad \phi = P.\text{exportType}(f)}{P \vdash f_\phi \gg f} \qquad \textsc{FunReturned}\ \frac{(g, \phi) \in P.\text{returners}(f) \qquad P \vdash \sigma \gg g}{P \vdash \sigma \triangleright f_\phi \gg f}$$

There are two cases to consider. [FunExported] If a function f is exported by P, its interaction scheme σ consists only of the call action of f. The call type ϕ depends on how f is exported: if it is exported within an object, it will be a method call; if it is exported directly, it can be either a constructor call or a function call. We assume that directly-exported functions that use the keyword `this` should be called as constructors. [FunReturned] If a function f is returned by a function g for which there is an interaction scheme σ, then the interaction scheme of f is obtained by concatenating σ with the call action of f. The call type ϕ depends on how f is returned by g. In the definition, we use $(g, \phi) \in P.\text{returners}(f)$ to mean that f may be returned by g with the call type ϕ. We have implemented a custom-made static analysis to compute this information.

```
1  Function FindPaths(P, f):
2      W ← [ (⟨⟩, f) ]
3      R ← []
4      while W ≠ [] do
5          (σ, h) ← W.pop()
6          if P.isExported(h) then
7              φ ← P.exportType(h)
8              R.push(h_φ ▷ σ)
9          foreach (g, φ) ∈ P.returners(h) do
10             if g ∉ σ then
11                 σ' ← h_φ ▷ σ
12                 W.push(σ', g)
13     return R
```

Algorithm 2: Interaction schemes leading to function f in package P.

Algorithm 2 describes a deterministic procedure for computing the valid interaction schemes for a function f from package P. The algorithm computes the interaction schemes backwards, starting from the target function and tracing back to the package's exported functions. To this end, it maintains a work list $\mathcal{W}$ of pairs of the form (σ, g), where σ is a candidate scheme that reaches f starting from the return of g. At each iteration of the main loop, the algorithm proceeds as follows. It pops the first pair (σ, h) from $\mathcal{W}$ and checks if h is an exported function. If it is, the algorithm computes the call type ϕ of h, and pushes the complete sequence to the result list $\mathcal{R}$. Subsequently, the algorithm identifies all functions g that return h and are not yet present in the current candidate scheme σ. For each such function, it adds a new candidate scheme to $\mathcal{W}$, extending σ, with a call to h of type ϕ. The following theorem establishes the correctness of Algorithm 2, ensuring the validity of all computed schemes.

Theorem 4.2. $\sigma \in \textsc{FindPaths}(P, f) \implies P \vdash \sigma \gg f$

After finding all the attacker-controlled functions and their associated interaction schemes, Explode.js leverages Graph.js [23] to identify potentially vulnerable data flows connecting the inputs of these functions to sensitive sinks. To achieve this, we have implemented a set of Graph.js queries [23] that look for vulnerable data flows corresponding to our targeted vulnerabilities. The interaction schemes corresponding to attacker-controlled functions for which such data flows exist are referred to as *vulnerable interaction schemes*.

$$\textsc{Scheme Compiler}\quad \frac{r_0 = P.\text{exportedObject} \qquad AC_P(r_{i-1}, (f_i, \phi_i)) = \langle s_i,\ r_i\rangle\ |_{i=1}^{n}}{SC_P\big((f_1, \phi_1),\ \ldots,\ (f_n, \phi_n)\big) = s_1\ ;\ \ldots\ ;\ s_n}$$

$$\textsc{Call Action Compiler (Function Call)}\quad \frac{P.\text{paramTypes}(f) = [\tau_1,\ \ldots,\ \tau_n] \qquad TC(\tau_i) = \langle s_i,\ x_i\rangle\ |_{i=1}^{n} \qquad r\ \textit{fresh} \qquad s = s_1\ ;\ \ldots\ ;\ s_n\ ;\ r \coloneqq g(x_i,\ \ldots,\ x_n)}{AC_P(g, f_{call}) = \langle s,\ r\rangle}$$

$$\textsc{Type Compiler (Number)}\quad \frac{x\ \textit{fresh} \qquad s = x \coloneqq \texttt{symNum}()}{TC(\texttt{number}) = \langle s,\ x\rangle}$$

$$\textsc{Type Compiler (Object)}\quad \frac{TC(\tau_i) = \langle s_i,\ x_i\rangle\ |_{x=1}^{n} \qquad x\ \textit{fresh} \qquad s = s_1;\ \ldots;\ s_n\ ;\ x \coloneqq \{p_i : \tau_i\ |_{i=1}^{n}\}}{TC(\{p_1 : \tau_1,\ \ldots\ p_n : \tau_n\}) = \langle s,\ x\rangle}$$

Fig. 3. Exploit Template Generation Compilers.

4.2 Exploit Templates

We transform a vulnerable interaction scheme into an *exploit template* by traversing the scheme's call actions and creating, for each call action, a call statement with symbolic arguments of the appropriate type. Argument types are inferred using a straightforward flow-insensitive intra-procedural analysis. Types $\tau \in \mathcal{T}$ are given by the grammar:

$$\tau ::= \texttt{number} \mid \texttt{string} \mid \texttt{boolean} \mid \{p_1 : \tau_1,\ \ldots,\ p_n : \tau_n\}$$

where `number`, `string`, and `boolean` respectively denote the number, string and boolean primitive types, whereas $\{p_1 : \tau_1, \ldots, p_n : \tau_n\}$ denotes the object type that associates each property p_i with type τ_i. For instance, the type of the constructor's argument `options`, appearing in line 17 of Example 1a, is { `host` : `string`, `useTempFiles` : `boolean` }, signifying that the constructor expects as input an object with a property `host` of type `string` and a property `useTempFiles` of type `boolean`. The property `host` is inferred to be of type string since it is used in the string concatenation operation in line 8. Analogously, the property `useTempFiles` is inferred to be of type boolean because it is used in the logical binary expression of line 24.

We formalize exploit template generation with the help of three compilation functions: **(i)** the *interaction scheme compiler SC* that given a package P and an interaction scheme σ generates the corresponding exploit template in the form of a symbolic test; **(ii)** the *call action compiler AC* that converts a call action c into a symbolic statement that performs the corresponding call with symbolic arguments of the appropriate types; and **(iii)** the *type compiler TC* that, given a type τ, generates a statement that creates a symbolic value of that type. Fig. 3 depicts a subset of the compilation rules. Using the algorithm captured by the rules, Explode.js generates the exploit template shown in Fig. 1b for the vulnerable interaction scheme of Fig. 1a. In the following, we explain the compilation rules in a top-down manner.

Compiling an interaction scheme (Scheme Compiler) involves compiling each of its call actions (f_i, ϕ_i) using the call action auxiliary compiler, *AC*. The first call action is compiled to a call statement on the package's exported object and all subsequent call actions are compiled to call statements on the result r_{i-1} of the preceding call. To compile call actions of type *call* (Call Action Compiler - Function Call), the action compiler *AC* first determines the types $\tau_1,\ \ldots,\ \tau_n$ of the function's parameters, using its type analyzer. Then, it uses the auxiliary type compiler, *TC*, to translate each type τ_i to a statement s_i that assigns a fresh symbolic value of type τ_i to x_i. The action compiler then generates a call statement to the function g returned by the preceding call action, using the symbolic variables $x_1,\ \ldots,\ x_n$ as arguments. The type compiler *TC* is more straightforward. For primitive types, it simply generates a fresh symbolic value of the appropriate type; for object types, it recursively compiles each property type τ_i to a new value, and then constructs a new object with the compiled properties.

5 Exploit Template Resolver

To symbolically execute the code of a package without stepping into the code of its dependencies, Explode.js supports lazy values, which are special symbolic values whose behavior adapts according to the contexts in which they are used. This section formalizes our new SE engine for JS focusing on its new aspects (§5.1) and describes our algorithm for generating concrete exploits (§5.2).

5.1 Symbolic Semantics of Lazy Values

In the following, we formalize the symbolic semantics underpinning Explode.js for the core of JS given below.

Core JavaScript Syntax

$$
\begin{array}{l}
e \in \mathcal{E}xpr ::= v \in \mathcal{V} \mid x \in \mathcal{X} \\
s \in \mathcal{S}tmt ::= x := e \mid x := e_1 \oplus e_2 \mid x := \{\} \mid x := e.p \mid e_1.p := e_2 \mid x := e_1[e_2] \mid e_1[e_2] := e_3 \\
\qquad\qquad \mid x := f(e_1, \ldots, e_n) \mid \texttt{return}\ e \mid \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \mid \texttt{while}\ (e)\ \{s\} \mid s_1; s_2 \mid \hat{s} \\
\hat{s} \in \widehat{\mathcal{S}tmt} ::= x := t.\texttt{sval}() \mid \texttt{assume}\ e \mid \texttt{assert}\ e
\end{array}
$$

Expressions $e \in \mathcal{E}xpr$ include literal values and program variables. Statements $s \in \mathcal{S}tmt$ include simple assignments, binary operations, object allocations, static/dynamic property lookups and assignments, function calls, the `return`, `if`, and `while` statements, sequencing, and symbolic statements. Symbolic statements $\hat{s} \in \widehat{\mathcal{S}tmt}$ are used for creating and reasoning about symbolic values: we use $x := t.\texttt{sval}()$ for creating a fresh symbolic variable of type t, `assume` e for adding the constraint e to the current path constraint, and `assert` e for checking if the constraint e is implied by the current path constraint. *Symbolic values* are given by: $\hat{v} \in \widehat{\mathcal{V}} ::= v \mid \hat{x} \mid \nu \mid \hat{v}_1 \oplus \hat{v}_2$, which include concrete values $v \in \mathcal{V}$, symbolic variables $\hat{x} \in \widehat{\mathcal{X}}$, lazy values $\nu \in \mathcal{N}$, and binary operations on symbolic values.

We formalize the symbolic semantics of the core JS statements using a semantic judgment of the form: $\langle h, \rho, \pi, s\rangle \rightsquigarrow \langle h', \rho', \pi', \omega\rangle$ meaning that the symbolic evaluation of the statement s in input configuration $\langle h, \rho, \pi\rangle$ results in the output configuration $\langle h', \rho', \pi', \omega\rangle$. An input configuration includes: **(1)** a *symbolic heap* $h \in \widehat{\mathcal{H}eap} : \mathcal{L} \times \mathcal{S}tr \rightharpoonup \widehat{\mathcal{V}}$ mapping pairs of locations $l \in \mathcal{L}$ and properties $p \in \mathcal{S}tr$ to symbolic values; **(2)** a *symbolic store* $\rho \in \widehat{\mathcal{S}tore} : \mathcal{X} \rightharpoonup \widehat{\mathcal{V}}$ mapping program variables to symbolic values; and **(3)** a *path constraint* π keeping track of all the constraints on which the current execution has branched so far. Output configurations $\langle h, \rho, \pi, \omega\rangle$ are similar to input configurations, except that they additionally include a *computation outcome* ω that captures the flow of execution. Computation outcomes are given by the grammar:

$$\omega ::= \texttt{Cont} \mid \texttt{CMI}(\pi, f, \hat{v}) \mid \texttt{CDI}(\pi, f, \hat{v}) \mid \texttt{PP}(\pi, l, p, \hat{v}) \mid \texttt{AsrtFail}$$

where: **(1)** the empty continuation outcome `Cont` denotes that the execution may proceed to the next statement; **(2)** the command injection outcome $\texttt{CMI}(\pi, f, \hat{v})$ denotes that the execution generates a command injection vulnerability with path constraint π, sink type f, and sink payload $\hat{v}$; **(3)** the code injection outcome $\texttt{CDI}(\pi, f, \hat{v})$ denotes that the execution generates a code injection vulnerability, with path constraint π, sink type f, and sink payload $\hat{v}$; **(4)** the prototype pollution outcome $\texttt{PP}(\pi, l, p, \hat{v})$ denotes that the execution generates a prototype pollution vulnerability with path constraint π, built-in object l, property p, and value $\hat{v}$; and, **(5)** the failed assertion outcome `AsrtFail` denotes that the execution resulted in an assertion failure. Fig. 4 presents a subset of the symbolic semantic rules. For clarity, we omit the outcome element ω when the rules produce the empty continuation outcome `Cont`.

$$\textsc{Binary Operation}\quad \frac{[\![e_1]\!]_\rho = \hat{v}_1 \qquad [\![e_2]\!]_\rho = \hat{v}_2 \qquad \mathtt{type}(\oplus) = (\tau_1, \tau_2) \qquad \hat{v}_1', \pi_1 = \mathsf{conv}(\hat{v}_1, \tau_1) \qquad \hat{v}_2', \pi_2 = \mathsf{conv}(\hat{v}_2, \tau_2) \qquad \rho' = \rho[x \mapsto \hat{v}_1' \oplus \hat{v}_2'] \qquad \pi' = \pi \wedge \pi_1 \wedge \pi_2}{\langle h, \rho, \pi, x := e_1 \oplus e_2 \rangle \rightsquigarrow \langle h, \rho', \pi' \rangle}$$

$$\textsc{Lazy Static Property Lookup}\quad \frac{[\![e]\!]_\rho = v \qquad l, v' \; \mathit{fresh} \qquad h' = h \uplus (l, p) \mapsto v' \qquad \rho' = \rho[x \mapsto v'] \qquad (\pi' = \pi \wedge v = l) \; \mathcal{S}at}{\langle h, \rho, \pi, x := e.p \rangle \rightsquigarrow \langle h', \rho', \pi' \rangle}$$

$$\textsc{Lazy Static Property Assignment}\quad \frac{[\![e_1]\!]_\rho = v \qquad [\![e_2]\!]_\rho = \hat{v} \qquad l \; \mathit{fresh} \qquad h' = h \uplus (l, p) \mapsto \hat{v} \qquad (\pi' = \pi \wedge v = l) \; \mathcal{S}at}{\langle h, \rho, \pi, e_1.p := e_2 \rangle \rightsquigarrow \langle h', \rho, \pi' \rangle}$$

$$\textsc{If-then-else (True)}\quad \frac{[\![e]\!]_\rho = \hat{v} \qquad \hat{v}', \pi_{\hat{v}'} = \mathsf{conv}(\hat{v}, \mathtt{Boolean}) \qquad (\pi' = \pi \wedge \pi_{\hat{v}'} \wedge \hat{v}') \; \mathcal{S}at}{\langle h, \rho, \pi, \mathtt{if}\ (e)\ \{s_1\}\ \mathtt{else}\ \{s_2\} \rangle \rightsquigarrow \langle h, \rho, \pi', s_1 \rangle}$$

$$\textsc{Lazy Symbolic Function Call}\quad \frac{[\![f]\!]_\rho = v \qquad [\![e_i]\!]_\rho = \hat{v}_i \;|_{i=1}^{n} \qquad v' \mathit{fresh} \qquad \rho' = \rho[x \mapsto v'] \qquad \pi' = \pi \wedge v(\hat{v}_1, \ldots, \hat{v}_i) = v'}{\langle h, \rho, \pi, x := f(e_1, \ldots, e_n) \rangle \rightsquigarrow \langle h, \rho', \pi' \rangle}$$

$$\textsc{Command Injection}\quad \frac{f \in \mathit{Sinks}.\mathtt{CMI} \qquad [\![e]\!]_\rho = \hat{v} \qquad \mathtt{isSymbolic}(\hat{v}, \pi)}{\langle h, \rho, \pi, f(e) \rangle \rightsquigarrow \langle h, \rho, \pi, \mathtt{CMI}(\pi, f, \hat{v}) \rangle}$$

$$\textsc{Code Injection}\quad \frac{f \in \mathit{Sinks}.\mathtt{CDI} \qquad [\![e]\!]_\rho = \hat{v} \qquad \mathtt{isSymbolic}(\hat{v}, \pi)}{\langle h, \rho, \pi, f(e) \rangle \rightsquigarrow \langle h, \rho, \pi, \mathtt{CDI}(\pi, f, \hat{v}) \rangle}$$

$$\textsc{Prototype Pollution}\quad \frac{[\![e_1]\!]_\rho = l \qquad \mathtt{isPrototypeChainModifiable}(l, p, \hat{v}) \qquad [\![e_2]\!]_\rho = \hat{v} \qquad \mathtt{isSymbolic}(\hat{v}, \pi)}{\langle h, \rho, \pi, e_1.p := e_2 \rangle \rightsquigarrow \langle h, \rho, \pi, \mathtt{PP}(\pi, l, p, \hat{v}) \rangle}$$

Fig. 4. Symbolic Semantics of Core JS (fragment): $\langle h, \rho, \pi, \hat{s} \rangle \rightsquigarrow \langle h', \rho', \pi', \omega \rangle$.

Binary Operation. To evaluate a binary operation $x := e_1 \oplus e_2$, we first evaluate e_1 and e_2 in the current store ρ, obtaining the symbolic values $\hat{v}_1$ and $\hat{v}_2$. Then, we check that the types of these values are compatible with the operand types of the binary operator $\oplus$, performing any necessary conversions, which yields the symbolic values $\hat{v}_1'$ and $\hat{v}_2'$ and the additional path constraints π_1 and π_2. Finally, we store the result, $\hat{v}_1' \oplus \hat{v}_2'$, in x and extend the path constraint π with π_1 and π_2.

Lazy Static Property Lookup. To evaluate a static property lookup $x := e.p$, we first evaluate e in the current store ρ. Then, if this results in a lazy value v, we create a fresh location l to store a fresh lazy value v' in property p, updating the heap h accordingly. Finally, we store the result v' in x and extend the path constraint π with $v = l$ to mean that l coincides with the lazy value v in the current path.

Lazy Static Property Assignment. To evaluate a static property assignment $e_1.p := e_2$, we first evaluate e_1 and e_2 in the current store ρ. Then, if this evaluation respectively yields a lazy value v and a symbolic value $\hat{v}$, we create a fresh location l to store $\hat{v}$ in property p, updating the heap accordingly. Finally, we extend the path constraint π with $v = l$ to mean that l is storing the lazy value v in the current path.

If-then-else (True). To evaluate an `if` statement $\mathtt{if}(e)\ \{s_1\}\ \mathtt{else}\ \{s_2\}$ using the If-True rule, we first evaluate e in the current store ρ, obtaining the symbolic value $\hat{v}$, which is then converted to the boolean value $\hat{v}'$. Then, if the constraint $\pi \wedge \pi_{\hat{v}'} \wedge \hat{v}'$ is satisfiable, meaning that the expression e may evaluate to `true`, we proceed with evaluating s_1 on the updated path constraint π'.

Lazy Symbolic Function Call To evaluate a function call $x := f(e_1, \ldots, e_n)$, we first evaluate f in the current store ρ. If this results in a lazy value v, we then evaluate the function's arguments $e_1, \ldots, e_n$ and create a fresh lazy value v', representing the return value. Finally, we store v' in x

$$\textsc{Command Injection}\quad \frac{\omega = \texttt{CMI}(\hat{v}, f, \pi) \qquad \pi' \in \texttt{AttackPatts}(\texttt{CMI}, \hat{v}) \qquad \text{sat}(\pi \wedge \pi') \qquad \varepsilon = \text{model}(\pi \wedge \pi')}{\texttt{FindExploit}(\omega) \rightsquigarrow \varepsilon}$$

$$\textsc{Code Injection}\quad \frac{\omega = \texttt{CDI}(\hat{v}, f, \pi) \qquad \pi' \in \texttt{AttackPatts}(\texttt{CDI}, \hat{v}) \qquad \text{sat}(\pi \wedge \pi') \qquad \varepsilon = \text{model}(\pi \wedge \pi')}{\texttt{FindExploit}(\omega) \rightsquigarrow \varepsilon}$$

$$\textsc{Prototype Pollution}\quad \frac{\omega = \texttt{PP}(\pi, l, p, \hat{v}) \qquad \text{sat}(\pi) \qquad \varepsilon = \text{model}(\pi)}{\texttt{FindExploit}(\omega) \rightsquigarrow \varepsilon}$$

Fig. 5. Exploit Symbolic Semantics of Vulnerability Outcomes $\{\texttt{CMI}, \texttt{CDI}, \texttt{PP}\} \in \omega$ (fragment).

and extend the path constraint π with $\nu(\hat{v}_1, \ldots, \hat{v}_n) = \nu'$, indicating that ν' is obtained by applying ν to the arguments $\hat{v}_1, \ldots, \hat{v}_n$.

Command Injection. To evaluate a function call where the callee is a command injection sink, such as `exec(...)`, we first evaluate e in the current store ρ, obtaining a symbolic value $\hat{v}$. Then, if $\hat{v}$ is symbolic, the evaluation produces the command injection outcome $\texttt{CMI}(\pi, \texttt{exec}, \hat{v})$, reporting a command injection vulnerability with path constraint π, sink `exec`, and symbolic payload $\hat{v}$.

Code Injection. The evaluation of a function call where the callee is a code injection sink, such as `eval(...)`, follows the same reasoning as the Command Injection rule, except that it produces a code injection outcome $\texttt{CDI}(\pi, \texttt{eval}, \hat{v})$ if the expression e denotes a symbolic value $\hat{v}$.

Prototype Pollution. When evaluating a property assignment $e_1.p := e_2$, we first evaluate e_1 and e_2 in the current store ρ, obtaining the location l and symbolic value $\hat{v}$, respectively. Then, if p is a modifiable prototype-chain property, and $\hat{v}$ is symbolic, the evaluation produces the prototype pollution outcome $\texttt{PP}(\pi, l, p, \hat{v})$, reporting a prototype pollution vulnerability with path constraint π, prototype object l, property p, and value $\hat{v}$.

5.2 Exploit Solving

At the core of our exploit generation procedure is the function `FindExploit`, whose goal is to compute a concrete exploit from an exploit outcome. Formally, this function takes as input an exploit outcome and generates a set of exploit models, $\varepsilon : \mathcal{X} \rightharpoonup \mathcal{V}$, mapping the symbolic variables of the exploit outcome to concrete values. Fig. 5 shows the definition of `FindExploit` for command and code injection, and prototype pollution outcomes. We write $\texttt{FindExploit}(\omega) \rightsquigarrow \varepsilon$ to mean that the set of exploit models generated by `FindExploit` for the outcome ω contains the model ε.

To concretize a *command injection exploit*, we must first extend the path constraint of the outcome, π, with a constraint that ensures that the payload is a valid shell command that triggers unintended behavior. To this end, we maintain a set of *exploit patterns* for command injections sinks that guarantee that the sink payload is well-formed and contains an attack. In the rule, we write $\pi' \in \texttt{AttackPatts}(\texttt{CMI}, \hat{v})$ to mean that if π' holds, then the symbolic payload $\hat{v}$ fits one of the `CMI` attack patterns. Then, we check if the conjunction of the path constraint and the attack pattern constraint, $\pi \wedge \pi'$, is satisfiable, in which case we output the corresponding model ε. Concretizing a *prototype pollution exploit* outcome is substantially simpler; we simply check if the path constraint of the outcome is satisfiable and output the corresponding satisfying model.

Attack Patterns. When it comes to command and code injection outcomes, we must ensure that the generated payloads satisfy the syntactic rules of their respective sinks and trigger unintended behavior. To this end, we maintain, a set of *exploit patterns* that capture possible attacks and ensure that sink payloads are well-formed. Formally, we model these patterns as a relation `AttackPatterns`

that associates pairs of sink types and symbolic payloads to constraints that guarantee that the given payload yields a working exploit. Table 1 shows the most relevant attack patterns for command injection and code injection sinks. We explain the two classes of attack patterns below.

Table 1. Supported vulnerability patterns by outcome. We use || to separate the attack patterns.

Outcome (ω)	**Patterns ($\mathcal{P}$)**
$\texttt{CMI}(\pi, f, \hat{v})$	$contains(\hat{v}, \texttt{`}Attack\texttt{`}) \quad \| \quad \hat{v} = \hat{x}; Attack\ \# \quad \| \quad \hat{v} = \hat{x}\texttt{"}; Attack\ \# \vee \hat{v} = \hat{x}\texttt{'}; Attack\ \#$
$\texttt{CDI}(\pi, f, \hat{v})$	$\hat{v} = \hat{x}; Attack\ \texttt{//} \quad \| \quad contains(\hat{v}, \texttt{(() => \{ } Attack \texttt{ \})())}$

CMI Attack Patterns. Our `CMI` attack patterns cover three cases:

(1) *Command Substitution*: This pattern captures cases where the given symbolic payload can be concretized to include a specific concrete attack *Attack* within backticks. Recall that backticks are used in the shell to execute the command enclosed within them and can be placed anywhere within a string.
(2) *Sequence Attack*: This pattern captures cases where the symbolic payload is concretized to include a specific attack, appended as a separate command using a shell command separator, followed by a comment. Recall that the "`;`" separator allows chaining commands that execute unconditionally, while the comment truncates any remaining content.
(3) *Escaped Sequence Attack*: This pattern is analogous to the previous one, except that it prepends a quote (either `"` or `'`) before the attack sequence.

CDI Attack Patterns. Out `CDI` attack patterns cover two cases:

(1) *Sequence Attack*: This pattern is analogous to the `CMI` one, except that the comment is written using the JS syntax.
(2) *Anonymous Function Attack*: This pattern captures cases where the given symbolic payload can be concretized to include a specific concrete attack `Attack` within the body of an anonymous function.

6 Evaluation

In this section, we evaluate the effectiveness and performance of Explode.js in generating exploits against *npm* packages, focusing on the following four research questions:

- **RQ1**: How effective is Explode.js in generating exploits and how does it compare to the state-of-the-art tools for exploit generation for Node.js modules?
- **RQ2**: Can Explode.js find zero-day security vulnerabilities in real-world *npm* packages?
- **RQ3**: What is Explode.js' performance and how does it compare to that of state-of-the-art tools for exploit generation for Node.js modules?
- **RQ4**: How does each component of Explode.js contribute to finding vulnerabilities?

Experimental Setup. To address our research questions, we used three datasets. Two of these are complementary vulnerability datasets from prior work, which we use as ground truth: *VulcaN* [10] and *SecBench* [9]. They exhibit different vulnerability distributions over vulnerability types, as summarized in Table 2. The third dataset, which we refer to as *Collected*, consists of popular Node.js packages obtained from the *npm* repository. Next, we describe these three datasets:

(1) *VulcaN* [10] consisting of 957 Node.js packages from the *npm* repository, containing 219 vulnerabilities supported by Explode.js.
(2) *SecBench* [9] consisting of 503 vulnerable Node.js packages reported in the GitHub Advisory Database, Synk, and Huntr.dev, containing 384 vulnerabilities supported by Explode.js.

Table 2. Summary of the reference datasets per vulnerability type: "Raw Total" refers to the total number of vulnerabilities in the dataset, and "Total" refers to the number of vulnerabilities considered in our study.

Vulnerability Type	CWE	*Vulcan*		*SecBench*		Total	Dist. (%)
		Raw Total	Total	Raw Total	Total		
Path Traversal	CWE-22	5	5	170	161	166	27.5%
Command Injection	CWE-78	92	87	101	82	169	28.03%
Code Injection	CWE-94	41	33	40	21	54	8.96%
Prototype Pollution	CWE-1321	98	94	192	120	214	35.49%
Total		236	219	503	384	603	100.00%

(3) *Collected* consisting of 32,137 popular real-world Node.js packages crawled from the *npm* repository in September 2023. We consider a package to be popular if it had $\geq 2,000$ weekly downloads at the time of collection. For the collected dataset, there is no ground truth because we did not manually analyze the source code of the packages to identify exploits.

To compare Explode.js with prior work, we used FAST [36] and NodeMedic [15] on the ground truth datasets SecBench and VulcaN. Our testbed consisted of 6 Ubuntu 22.04.3 servers with 64GB of RAM and two Intel(R) Xeon(R) Gold 5320 2.2GHz CPUs. We set the total analysis timeout to 5 minutes as all tools analyzed over 90% of the datasets within this period.

6.1 RQ1: Effectiveness in Exploit Generation

We assess Explode.js's effectiveness in detecting and confirming vulnerabilities and compare it to FAST and NodeMedic using the ground truth datasets VulcaN and SecBench. Results are summarised in Table 3. Note that NodeMedic has a value of "–" for vulnerability types CWE-22 (Path Traversal) and CWE-1321 (Prototype Pollution), as it does not support these types.

True Positive (TP). A TP vulnerability is a known vulnerability for which the tool is able to find a control path connecting an exported function to the corresponding sensitive sink, regardless of the tool's ability to generate a working exploit. The columns titled "TP" in Table 3 show that Explode.js can find 2.90× and 5.23× more control paths than FAST and NodeMedic, respectively, identifying 351 control paths compared to 121 for FAST and 67 for NodeMedic. In particular, Explode.js finds twice as many command injection control paths (CWE-78) as NodeMedic, three times as many code injection control paths (CWE-94) as FAST, and is the only system capable of finding control paths for prototype pollution, identifying 55.6% of the ground truth dataset. This improvement is

Table 3. Comparison of Explode.js, FAST, and NodeMedic across combined VulcaN and SecBench datasets. Metrics include true positives (TP), false negatives (FN), exploits (E), recall (R) and exploit rate (Er).

CWE	Total	Explode.js					FAST					NodeMedic				
		TP	FN	E	R	Er	TP	FN	E	R	Er	TP	FN	E	R	Er
CWE-22	166	97	69	84	0.584	0.506	6	160	0	0.036	0	–	–	–	–	–
CWE-78	169	112	57	70	0.662	0.414	108	61	66	0.639	0.390	55	114	38	0.325	0.225
CWE-94	54	24	30	13	0.444	0.240	7	47	3	0.130	0.056	12	42	4	0.222	0.074
CWE-1321	214	118	96	111	0.556	0.518	0	214	0	0	0	–	–	–	–	–
Total	603	351	252	278	0.582	0.461	121	482	69	0.200	0.114	67	156	42	0.300	0.188

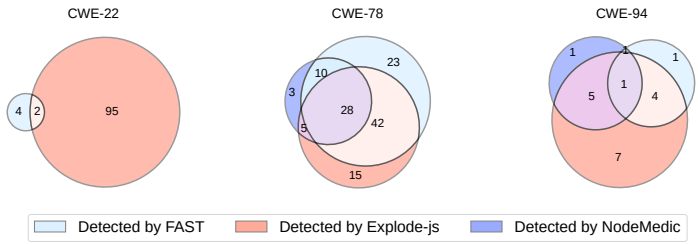


Fig. 6. Overlap of true positives detected by all tools across combined VulcaN and SecBench datasets.

mainly due to our analysis for inferring VISes, which effectively finds the correct sequence of calls and argument types in order to reach the vulnerable sink.

Regarding the specific vulnerabilities each tool can identify, Fig. 6 shows that for CWE-22, Explode.js finds control paths different from those of FAST. This is because FAST can only find path traversal vulnerabilities in web-server modules that register the vulnerable function as a callback for a specific route. In contrast, Explode.js cannot statically identify a VIS for these cases because they all require non-linear VISes. For CWE-78, Explode.js identifies the majority of control paths found by NodeMedic and FAST, with 36 detected exclusively by NodeMedic and FAST. This is because these modules use third-party packages that implement higher-order iterators working with callbacks. Essentially, these iterators take a function as input and invoke it on a collection of elements. Our lazy values do not simulate the execution of callbacks, resulting in these vulnerabilities being undetected. Finally, for CWE-94, the control paths detected by Explode.js largely subsume the ones detected by the other tools.

False Negative (FN) and Recall (R). A FN vulnerability is a known vulnerability for which the tool is unable to find a control path connecting an exported function to the corresponding sensitive sink. Recall is calculated as $TP/(TP + FN)$.

The columns titled "FN" in Table 3 show that Explode.js reported 252 FNs, compared to 482 for FAST and 156 for NodeMedic. The columns titled "R" in the table show the recall obtained by each tool. When comparing only CWE-78 and CWE-94, Explode.js achieved the highest recall of the three tools, obtaining 60.5% which is an increase of 9 percentage points over FAST and 30 percentage points over NodeMedic. When considering the entire dataset, Explode.js achieves a total recall of 58.2% which is 2.90× the recall of FAST (20.0%) and 1.94× the recall of NodeMedic (30.0%).

We manually analyzed the reasons why Explode.js missed 252 vulnerabilities, summarized in Table 4. The largest category, with 39.7%, consists of limitations in the lazy value implementation. Specifically, we currently lack support for array accesses (i.e., $e_1[e_2]$) and for class instantiation using the `new` operator over lazy values. The second-largest category, at 21.8%, includes modules with non-linear VISes. Next, 15.5% of false negatives arise from timeouts during symbolic execution due to the size and complexity of the underlying code, and 12.3% from limitations in our current implementation of the Node.js standard library functions. Finally, 10.7% stem from unsupported JS semantics in our SE engine, such as `async` functions and generators.

Table 4. Reasons for Explode.js's FNs.

Reason	Portion (%)
Lazy Values	39.7%
Unsupported VIS	21.8%
Timeout	15.5%
Stdlib	12.3%
JS Semantics	10.7%

False Positive (FP). While Table 3 does not comment on FPs, note that both Explode.js and NodeMedic have zero FPs, with all generated exploits working as intended, resulting in 100%

Table 5. Summary of exploits identified by Explode.js in the Collected dataset, detailing the total number of control paths and successfully generated exploits, along with the proportion of malicious and 0-day exploits.

Vulnerability Type	Vulnerable Paths		Exploits		Malicious	0-day
	Total	(%)	Total	Er (%)		
Path traversal	88	30.24	70	24.05	23	23
Command Injection	151	51.89	51	17.52	19	18
Code Injection	45	15.46	2	0.69	2	2
Prototype Pollution	7	2.41	2	0.69	2	1
Total	291	100.00	125	42.95	46	44

precision. In contrast, FAST generates 52 FPs, as it uses an approximate model of the JS semantics, causing it to consider symbolic execution paths that have no concrete counterpart. While both Explode.js and NodeMedic may generate models for exploits that do not result in effectful exploits, both tools ensure that such models are not outputted to the user as valid exploits.

Exploited (E) and Exploit Rate (Er). The symbol E refers to vulnerabilities for which the tool generated a working exploit. Hence, $E \subseteq TP$ given that the tool only generates exploits for vulnerabilities correctly identified as such. Exploit rate for each tool is calculated as $E/(TP + FN)$.

The columns titled "E" in Table 3 show that Explode.js can generate 4.02× and 6.61× more exploits than FAST and NodeMedic, respectively. Compared to FAST, Explode.js generates 1.06× more command injection exploits (CWE-78) and 4.33× more code injection exploits (CWE-94). Against NodeMedic, Explode.js generates 1.84× more command injection exploits and 3.25× more code injection exploits. Notably, Explode.js excels in generating prototype pollution exploits, producing 111 exploits compared to 0 for FAST. By combining VISes with precise symbolic execution for JS code, Explode.js uncovers significantly more vulnerable paths than FAST or NodeMedic.

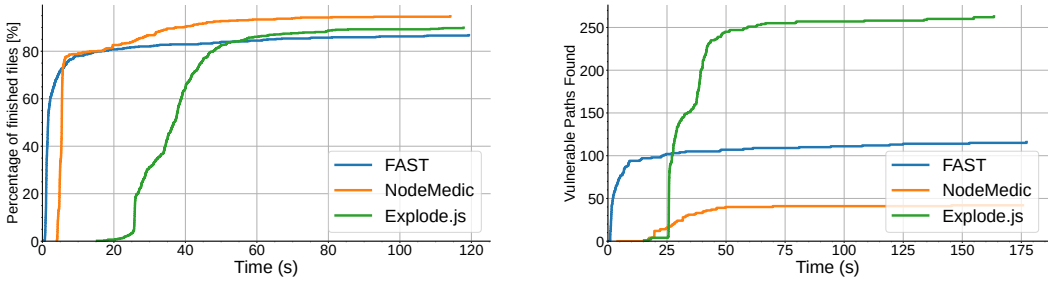
The columns titled "Er" in the table show that, globally, Explode.js achieves an exploit rate of 46.1%, which is an increase of 34 points over FAST's and 27 points over NodeMedic's exploit rate.

Takeaway 1: Explode.js detects 58.2% of existing control paths in the ground truth datasets, outperforming FAST by 2.90× and NodeMedic by 5.23×. It achieves an exploit rate of 46.1% surpassing FAST by 4.02× and NodeMedic by 6.61×.

6.2 RQ2: Exploit Generation in the Wild

We assess Explode.js's capability of finding zero-day vulnerabilities, by applying Explode.js to the Collected dataset, comprising 32k Node.js packages taken from the *npm* repository. In the following, we say that the tool identified a *zero-day exploit* if: (1) Explode.js generates a PoC exploit with an observable side-effect; (2) it is not the intended behavior of the package; and (3) there is no information about the vulnerability in the corresponding package repository.

Table 5 summarizes our results. Initially, Explode.js identified 291 control paths in 95 packages (column "Vulnerable Paths"). From these, Explode.js automatically generated 125 PoC exploits with observable side effects (column "Exploits"). Each exploit was manually verified. We first excluded every exploit targeting packages that merely serve as wrappers for vulnerable sinks (e.g., `exec-sync`, a synchronous wrapper for `exec`), for which the exploit corresponds to the intended behavior of the package. This left us with 46 exploits that can be leveraged by malicious users to trigger non-intended side-effects in the execution platform (column "Malicious"). After further analysis, we excluded two exploits because their corresponding vulnerabilities had already been



(a) CDF of total analysis time for each tool.

(b) CDF of time to find an exploit.

Fig. 7. CDF plots comparing analysis and exploit generation times for Explode.js, FAST, and NodeMedic.

reported, resulting in a total of 44 new zero-day vulnerabilities (see column "0-day").[1] Lastly, we ran NodeMedic and FAST on the 44 zero-days reported by Explode.js. NodeMedic identified 4 control paths but no exploits, while FAST detected 10 control paths and found 4 exploits.

Ethical Disclosure. Disclosure of confirmed vulnerabilities was responsibly performed. For all verified vulnerabilities, we searched for the containing *npm* module's source code repository (e.g. GitHub) to check for disclosure guidelines and package maintainer contacts. For vulnerabilities for which we identified maintainer modes of contact, we sent an email describing the vulnerability, raising awareness and the PoC demonstrating the exploit. For maintainers that did not agree with our assessment, we submitted the vulnerability report to MITRE.

At the time of submission, we contacted maintainers of the *npm* modules of these 44 vulnerabilities, having received twenty replies. Notably: (1) two vulnerabilities were assigned CVEs via MITRE (2024-46503 and 2024-44711); (2) two vulnerabilities were assigned CVEs via GitHub advisories (2024-43370 and 2024-45390); and (3) four additional vulnerabilities were acknowledged by their developers, with three under review by GitHub advisories.

Takeaway 2: Out of 95 packages, Explode.js found 291 vulnerable paths and generated 125 exploits, of which 46 enable malicious actions, leading to 44 zero-days, for which 4 CVEs have already been assigned: CVE-2024-43370, CVE-2024-44711, CVE-2024-45390, CVE-2024-46503.

6.3 RQ3: Performance Evaluation

Execution Time. We measure the time taken by Explode.js, FAST and NodeMedic to detect vulnerabilities in each *npm* package from both our reference datasets, consisting of 219 packages from VulcaN and 384 from SecBench. Fig. 7 plots two cumulative distribution function (CDF) plots for each tool comparing time to finish the analysis and time to find an exploit.

In Fig. 7a, we depict the first 120 seconds, although each package can take as much as 10 minutes to be processed as per our pre-defined analysis timeout. Within just 20 seconds, both FAST and NodeMedic finish analyzing more than 80% of the packages. In contrast, Explode.js took 50 seconds to reach the 80% mark. This is due to a constant startup overhead incurred by stopping and starting a Neo4j instance at the start of the static analysis phase.

[1]Of the 44 new zero-day vulnerabilities, 25 are found in packages that developers claim should not be used in open environments like web servers. However, their respective documentation omits such restrictions potentially leading unsuspecting users to integrate one of these packages into an open system, unwittingly exposing themselves to a vulnerability.

While Explode.js is the tool that least analyzed the dataset by the 40-second mark, Fig. 7b shows that Explode.js detected the most vulnerabilities by that point. Concretely, Explode.js detects 2.0× more real vulnerabilities than FAST and 8× more than NodeMedic.

To further break down performance, we calculated the average package analysis times of Explode.js, FAST, and NodeMedic for packages that did not time out, grouped by vulnerability type. For each tool, we further break down the time into two phases. In particular, for Explode.js we show static analysis time (column "Static") and symbolic execution time (column "Symbolic"); for FAST we show path generation time (column "Path Gen.") and exploit generation time (column "Exploit Gen."); and, for Nodemedic we show fuzzing time (column "Fuzzing") and exploit synthesis time (column "Expl. Synth."). Table 6 summarizes these findings.

Across all vulnerability types, Explode.js is on average 2.9× slower than FAST and 2.0× slower than NodeMedic. However, Explode.js's symbolic execution is more efficient for path traversal (CWE-22) and command injection (CWE-78), with FAST potentially taking up to 2.49 times longer to process a package. Conversely, for prototype pollution (CWE-1321), the situation is reversed: Explode.js takes significantly longer to analyze these vulnerabilities, but it is also the only tool capable of detecting this type of vulnerability. This raises the hypothesis that if FAST and NodeMedic also attempted to identify this type of vulnerability, their performance would be significantly impacted. Notably, unlike code injection and command injection vulnerabilities, which typically have a limited number of sensitive sinks, prototype pollution generally has a large number of potential sinks, one for each dynamic property update.

Takeaway 3: Although Explode.js takes longer to analyze the dataset than FAST and NodeMedic, by the 40-second mark, it finds 2.0× more vulnerability than FAST and 8.0× more than NodeMedic.

6.4 RQ4: Necessity of Explode.js Components

We assess the effectiveness and necessity of Explode.js's components by performing two experiments. In the first, we ran Explode.js without using lazy values; and, in the second we ran Explode.js using lazy values but without using the VISes. Table 7 shows that removing lazy values (columns "No LVes") significantly degrades effectiveness, where the system without lazy values found only 118 TP versus the previous 351, and the number of FN increased substantially. For CWE-78, we can observe that the system with lazy values found 112 TP, while the system without lazy values found 1.96× less, which is only 57 TP. Similarly, for CWE-22, the full system was able to detect 97 TP, whereas when lazy

Table 7. Effectiveness of Explode.js without the VIS and without lazy values.

CWE	Full		No LVes		No VIS	
	TP	E	TP	E	TP	E
CWE-22	97	84	3	1	0	0
CWE-78	112	70	57	46	60	44
CWE-94	24	13	17	10	8	1
CWE-1321	118	111	41	33	18	5
Total	351	278	118	90	86	50

Table 6. Average time taken by each analysis phase for Explode.js, FAST, and NodeMedic across different CWEs, broken down into static analysis, symbolic execution, and fuzzing or exploit generation times.

CWE	Explode.js			FAST			NodeMedic		
	Static	Symbolic	Total	Path Gen.	Exploit Gen.	Total	Fuzzing	Expl. Synth.	Total
CWE-22	29.106	2.462	31.568	1.435	2.836	4.271	–	–	–
CWE-78	33.976	6.545	40.521	4.075	16.331	20.405	13.530	3.246	39.311
CWE-94	42.947	11.926	54.873	4.284	1.655	5.939	2.173	0.760	18.824
CWE-1321	31.162	10.948	42.112	16.980	0	19.130	–	–	–
Total	32.450	7.278	39.728	7.045	5.937	13.674	4.396	0.946	19.730

```
1  module.exports = Git() { ...
2  Git.prototype[command] =
3    function(opts) {
4    // ...
5    var cmd = 'git ' + command;
6    if (opts) cmd += ' ' + opts;
7    // ...
8    this.queue.push((cb) => {
9      exec(cmd, { cwd: base }, (err, out) => {
10         // ...
11     });
12   });
13   return this;
14 }
```

(a) Command construction.

```
1  Git.prototype.run = function(cb){
2    var self = this;
3
4    function next(err, res){
5      if (err) return cb && cb(err);
6
7      var fn = self.queue.shift();
8      if (!fn) return cb && cb(err, res)
9
10     fn.call(self, next);
11   }
12
13   next();
14 }
```

(b) Execution of commands.

Fig. 8. Case Study 1: Command injection in `gity@1.0.5`. (a) The module queues commands based on unsanitized user input. (b) The `run` function executes each queued command, enabling injection.

values were disabled it identified 32.3× less, which is only 3 TP. These results indicate that the lazy values component is a crucial part of Explode.js, since without it, we can only detect trivial exploits that do not include any external dependencies.

Running Explode.js without VISes (columns "No VIS") also has a significant impact on effectiveness, with TP dropping to 86 compared to 351 in the full version. Similarly to the previous experiment, for CWE-78, the results are 1.86× less than those of the full version, corresponding to 60 TP. For CWE-22, the system with no VISes does not find any TP. These results indicate that, without the VIS component, Explode.js is only able to detect trivial exploits in which the vulnerable function is directly exported by the vulnerable module.

Takeaway 4: The core techniques of Explode.js (VISes and lazy values) are key to its effectiveness.

6.5 Case Studies

In this section, we describe two case studies that illustrate the need for the two core components of our architecture: the VISes and the lazy values.

Vulnerable Interaction Scheme. To illustrate the importance of VISes, we analyzed a known vulnerability in the `gity@1.0.5` npm package.[2] This package provides an API for interacting with Git within Node.js, exposing Git commands through a prototype-based function object, `Git`, implemented in a builder-like pattern (see Fig. 8a). Notably, this module is vulnerable to a command injection attack. Specifically, on line 6, a user-controlled argument, `opts`, is concatenated with the `cmd` string, and subsequently passed to the vulnerable sink, `exec`, on line 9, without any proper input sanitization. Additionally, due to the builder-like pattern, these exposed functions do not execute actions immediately. Instead, each function registers a callback in a queue for deferred execution (line 8). As a result, triggering the vulnerability requires chaining a call to the `run` function (see Fig. 8b), which recursively processes the callbacks in the queue (lines 4-13).

Consequently, the VIS for this exploit requires three distinct interactions to reach the vulnerable function. First, we must call the module's constructor (line 1, Fig. 8a) to instantiate the Git command builder; Next, we need to call a Git command on the returned object to register the callback (lines 8-12, Fig. 8a). Finally, we must invoke the `run` function to execute the queued callbacks. A concrete exploit for this vulnerability is:

```
let Git = require("gity"); let git = Git().add('*.js; touch pwned.txt;\#').run();
```

[2] https://security.snyk.io/vuln/SNYK-JS-GITY-1012730

```
1  var exec  = require('child_process').exec;
2  var merge = require('merge');
3  var dargs = require('dargs');
4
5  Compass.prototype.compile = function(options) {
6    var options = options || {};
7    var options = merge(this.defaultOptions, options);
8    var command = generateCommand(options);
9
10   return new Promise(function(resolve, reject) {
11     exec(command, function(error, stdout, stderr) {
12       if (error) // ...
13       else {
14         resolve(compassOptions.cssDir);
15       }
16     });
17   });
18 }
```

(a) Vulnerable function.

```
1  var generateCommand = function(options) {
2    var compassCommand = options.compassCommand;
3    var excludes = ['compassCommand'];
4    var args = dargs(options, { excludes: excludes });
5    var command = [compassCommand, 'compile']
6      .concat(args)
7      .join(' ');
8    return command;
9  };
```

(b) Command construction.

```
1  let Compass = require("compass-compile");
2  let compass = new Compass();
3  let options = { "compassCommand" : "touch pwned" };
4  compass.compile(options);
```

(c) Command injection exploit.

Fig. 9. Case Study 2: Command injection in `compass-compile@0.0.1` package.

Lazy Values. To highlight the need for lazy values, we examine a vulnerability in the npm package `compass-compile@0.0.1`,[3] This package acts as a wrapper for *Compass* [20], a sass-based stylesheet framework that simplifies CSS maintenance. The module (Fig. 9a) exports a `Compass` class with a `compile` function (lines 5-18) that constructs a shell command, based on user-provided options. The `generateCommand` function (Fig. 9b) builds a command string by concatenating `compassCommand` with other options, after processing them with the `dargs` utility and excluding the `compassCommand` itself. The exploit shown in Fig. 9c demonstrates how an attacker can inject arbitrary shell commands by providing a malicious `compassCommand` option, such as `touch pwned`. This input triggers a command injection vulnerability, allowing the attacker to execute malicious commands.

In this example, we use lazy values to model the behavior of the external modules `dargs` and `merge`. Without them, we would need to incorporate the external module's code directly in the analyzed application code, which could significantly impact scalability. In particular, the `dargs` utility converts objects into command-line argument arrays, which heavily rely on string manipulation functions, such as search and replace, making it costly to model directly in symbolic execution. Bundling all external dependencies in the `Compass` module, the code size increases by 6.38×, from 36 lines-of-code (LoC) to 230 LoC, which results in the symbolic execution engine exceeding 8 GiB of RAM usage within the first 300 seconds, at which point the analysis was halted. On the other hand, analyzing this library with lazy objects only takes 0.5 seconds.

7 Limitations

Currently, Explode.js only supports linear call chain generation, producing linear VISes that contain no more than one call to the same function (*limitation 1*). Furthermore, our coverage of the JS semantics is incomplete because ECMA-Symb, the JS symbolic execution engine at the core of Explode.js, is not fully compliant with the JS standard [19]. Specifically, it currently lacks support for `async` and generator functions, Unicode, and the `BigInt` and `SharedArrayBuffer` built-ins (*limitation 2*). Additionally, Explode.js over-approximates the behavior of external functions using lazy objects (*limitation 3*), and it supports only a fixed set of attack patterns (*limitation 4*). For command injection vulnerabilities, the current attack patterns allow for the generation of most

[3] https://security.snyk.io/vuln/SNYK-JS-COMPASSCOMPILE-564429

exploits. For code injection vulnerabilities, Explode.js currently supports only a small subset of the JavaScript syntax, requiring more attack patterns to capture more complex syntactic constructs.

Despite restricting the general applicability of Explode.js, limitations 1 and 2 do not have a significant practical impact on its effectiveness, accounting for only 55 (16.9%) and 27 (8.3%) failed exploit generation cases in the evaluation dataset. In contrast, limitations 3 and 4 have a greater impact, being responsible for 131 (40.3%) and 47 (14.5%) failed exploit generation cases. To mitigate their impact, Explode.js has a modular architecture that streamlines the addition of both: **(i)** new symbolic summaries that precisely model external libraries; and **(ii)** new attack patterns that capture vulnerable sink payloads for which the tool is not currently able to generate effective exploits.

8 Related Work

Exploit Generation Techniques. Most prior work on JavaScript exploit generation targets cross-site scripting vulnerabilities [27, 28, 40, 52], constructing exploits from concrete expressions that reach sensitive sinks [8]. While effective for webpages with simple, primitive input types, this approach is unsuitable for Node.js applications, where inputs are more complex (e.g., multi-nested objects, arrays, and higher-order functions) and often pass through multiple functions before reaching the sensitive sink. Other works have used constraint solvers to generate exploits for client-side JS [63] and languages like PHP [1], employing dynamic symbolic execution to trace paths from attacker-controlled inputs to sensitive sinks. Some approaches combine symbolic execution with fuzzing to address path explosion [31, 64]. However, these methods overlook vulnerabilities requiring multiple function calls in a specific order, making them unsuitable for Node.js.

Vulnerability Detection Tools for Node.js. These tools generally fall into three categories and share an important shortcoming: they report false positives, meaning that developers must manually validate each generated potential-vulnerability report. *Code property graphs (CPGs)-based tools* [22, 23, 36, 38, 44] have been adopted for various programming languages [11, 53] and analytical scopes [22, 60] by performing customizable graph traversal queries on the application's syntax and flow graphs. However, most lack built-in vulnerability confirmation, with FAST [36] being the only exception; even so, it only confirms that vulnerable executions reach sensitive sinks without verifying any observable effect. *Dynamic taint propagation tools* [16, 29, 37] use the Jalangi2 framework [58] to instrument the execution of Node.js packages to detect vulnerable taint flows. NodeMedic [16] is the only tool among them with a built-in vulnerability confirmation mechanism, however, it does not handle vulnerabilities that require chaining multiple function calls to the given package. *Static taint propagation tools* [50, 62] are based on abstract interpretation and primarily focus on scalability. However, these tools do not incorporate vulnerability confirmation mechanisms, typically flagging many false positive vulnerabilities and therefore requiring manual confirmation.

Symbolic Execution. SE tools can be broadly divided into two main groups: *static* and *dynamic*. Static SE tools, such as [2–4], explore the entire symbolic execution tree up to a pre-established bound. Dynamic SE tools, such as [13, 14, 30, 48], pioneered by DART [30], normally work by pairing up concrete execution with SE to fallback to concrete execution whenever SE produces formulas not supported by the underlying SMT solver [54]. For JS there are multiple static and dynamic SE tools, such as [26, 41, 45, 57–59]. One of the main challenges of implementing symbolic execution for Node.js is supporting interaction with third-party code, which we achieved through the use of lazy values. We could have instead used concretization [7] or an incomplete form of concolic execution [30]. However, these techniques under-approximate the set of possible execution paths, potentially introducing false negatives. Most works that use lazy values in symbolic execution target statically typed languages [12, 13, 39], where type information can be leveraged to constrain their behavior. In the context of JS, the only other tool that uses lazy values is JaVerT 2.0 [26], which

employs them solely to lazily initialize object properties. Our use of this technique in Explode.js goes beyond that of JaVerT 2.0, as our lazy values can also transform into values of a specific type when involved in type-specific operations, and can even turn into dummy functions when used in call operations.

Combining Static Analysis with Symbolic Execution. Prior work has demonstrated the effectiveness of combining static analysis with symbolic execution for vulnerability discovery in various domains and analytical scopes, such as identification of memory leaks [43], race conditions [69], and browser bugs [12]. The main difference between these works and Explode.js is that they target entire programs, while Explode.js focuses on libraries. Specifically, the most pressing challenge in these works is that the programs to be tested contain vast amounts of code, making the direct application of symbolic execution to the program's entry point (i.e., the main function) not feasible. To address this, they use static analysis first to identify control-flow paths leading to potential bugs and then instrument symbolic execution to ensure that only those paths are followed.

In contrast, our work focuses on finding vulnerabilities within libraries and confirming them through the generation of effective exploits. Our main challenge lies in constructing specific sequences of calls to a library's (package) top-level functions to activate potentially vulnerable code. In most cases, there is no main function to initiate the library's execution; hence, we must generate the test driver ourselves. Our work is the first to combine static analysis with symbolic execution to create such drivers, as well as exploit payloads for confirming vulnerabilities.

Automated Driver Generation. Automated test generation often relies on fuzzing libraries [42, 46], which require fuzz drivers to exercise specific parts of the code. Our exploit templates function similarly by creating programs that drive symbolic analysis for Node.js packages. There are multiple tools for driver generation across various programming languages, including Fudge [6] for C and C++, FuzzGen [34] for Java, and Crabtree [65] and Syrust [66] for Rust. There are also tools that specialize in generating drivers aimed at specific types of applications, such as RESTler [5] and GraphFuzz [33] for RESTful and C/C++ API testing, and Randoop [51] for Java unit test generation.

When it comes to Node.js, little work has been done on driver generation. However, this problem entails solving language-specific challenges. JS presents unique difficulties, such as higher-order functions, structured objects that can be dynamically extended or shrunk, and implicit runtime type coercions. For instance, obtaining a specific inner function of a given module is not an issue in Randoop, as it was designed for a version of Java that, unlike JavaScript, does not support higher-order functions. Additionally, Explode.js must generate effective exploit payloads, introducing an extra layer of complexity and technical novelty to our methodology.

9 Conclusion

Explode.js demonstrates a significant advancement in automated exploit generation for Node.js packages by synthesizing functional exploits for multi-interaction vulnerabilities. This system leverages a unique blend of static analysis and symbolic execution to create exploit templates that confirm vulnerabilities in complex, multi-step scenarios without producing false positives. Evaluation against state-of-the-art tools shows that Explode.js outperforms FAST and NodeMedic, detecting over twice as many vulnerabilities. Applied to real-world Node.js packages, Explode.js uncovered 44 zero-day vulnerabilities, four of which have already been assigned CVEs, underscoring its effectiveness in addressing security gaps within the Node.js ecosystem.

Acknowledgments

We thank the anonymous reviewers for their comments and insightful feedback. Furthermore, we thank Frederico Ramos and Javi de Muller Santa María for their help in creating the software artifact

for this paper, and Eliana Oliveira for improving the plots and diagrams. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the grants 2021.06134.BD and PRT/BD/154029/2022, as well as the projects UIDB/50021/2020, WebCap (2024.07393.IACDC), SmartRetail (ref. C6632206063-00466847, financed by IAPMEI), and the Future Enterprise Security Initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab).

Data Availability Statement

The Explode.js's artifact is available at Marques et al. [47]. The open-source repository contains the most up-to-date version and is available at https://github.com/formalsec/explode-js.

References

[1] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (SEC '18)*. USENIX Association, Baltimore, MD, 377–392.

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF-SE: a symbolic execution extension to Java PathFinder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 134–138.

[3] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2009. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer* 11, 1 (Feb. 2009), 53–67. doi:10.1007/s10009-008-0090-1

[4] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Santos, and Jean-Christophe Filliâtre. 2024. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *The Art, Science, and Engineering of Programming* 9, 2 (2024). doi:10.22152/programming-journal.org/2025/9/3

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE '19)*. 748–758. doi:10.1109/ICSE.2019.00083

[6] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Association for Computing Machinery, New York, NY, USA, 975–985. doi:10.1145/3338906.3340456

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657

[8] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. 2021. Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis. In *Proceedings of the 14th European Workshop on Systems Security (EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 27–33. doi:10.1145/3447852.3458718

[9] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. 1059–1070. doi:10.1109/ICSE48619.2023.00096

[10] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. 2023. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages. *IEEE Transactions on Reliability* 72, 4 (2023), 1324–1339. doi:10.1109/TR.2023.3286301

[11] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* 118 (2022), 102745. doi:10.1016/j.cose.2022.102745

[12] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, 199–216.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, USA, 209–224. doi:10.5555/1855741.1855756

[14] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*. Association for Computing Machinery, New York, NY, USA, 322–335. doi:10.1145/1180405.1180445

[15] Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. 2025. NODEMEDIC-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities. In *Proceedings of the 2025 Network and Distributed System*

Security Symposium (NDSS '25). doi:10.14722/ndss.2025.241636

[16] Darion Cassel, Wai Tuck Wong, and Limin Jia. 2023. NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs. In *Proceedings of the 2023 European Symposium on Security and Privacy (EuroS&P '23)*. 1101–1127. doi:10.1109/EuroSP57164.2023.00068

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 337–340. doi:10.5555/1792734.1792766

[18] Devon Govett and Parcel contributors. 2021. Parcel – The zero configuration build tool for the web. https://parceljs.org visited on 2024-11-11.

[19] ECMA International. 2025. ECMAScript® 2025 Language Specification. https://tc39.es/ecma262/ visisted on 2025-03-25.

[20] Chris Eppstein, Scott Davis, Miriam Suzanne, Brandon Mathis, and Nico Hagenburger. 2024. Compass Stylesheet Authoring Framework. https://github.com/Compass/compass visited on 2024-11-13.

[21] Evan Wallace. 2020. esbuild - An extremely fast bundler for the web. https://esbuild.github.io/ visited on 2024-11-11.

[22] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2023. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP '23)*. IEEE, San Francisco, CA, USA, 2817–2834. doi:10.1109/SP46215.2023.10179395

[23] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '24, Vol. 8)*. Association for Computing Machinery, New York, NY, USA, Article 164, 25 pages. doi:10.1145/3656394

[24] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 927–942. doi:10.1145/3385412.3386014

[25] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3236950.3236956

[26] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages* 3, POPL '19, Article 66 (Jan. 2019), 31 pages. doi:10.1145/3290379

[27] Yaw Frempong. 2022. *HIJaX: Human Intent to Javascript XSS Generator*. PhD Thesis. The University of North Carolina at Charlotte. http://ninercommons.charlotte.edu/record/2205

[28] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. 2018. Towards Automated Generation of Exploitation Primitives for Web Browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 300–312. doi:10.1145/3274694.3274723

[29] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. 2018. AFFOGATO: runtime detection of injection attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 94–99. doi:10.1145/3236454.3236502

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[31] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. doi:10.1145/2093548.2093564

[32] Google. 2008. V8 JavaScript Engine. https://chromium.googlesource.com/v8/v8.git visited on 2024-11-13.

[33] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE '22)*. 1070–1081. doi:10.1145/3510003.3510228 ISSN: 1558-1225.

[34] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *Proceedings of the 29th USENIX Security Symposium (SEC '20)*. 2271–2287.

[35] Dongseok Jang and Kwang-Moo Choe. 2009. Points-to analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii) *(SAC '09)*. Association for Computing Machinery, New York, NY, USA, 1930–1937. doi:10.1145/1529282.1529711

[36] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP '23)*. 1059–1076. doi:10.1109/SP46215.2023.10179352

[37] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1364–1379. doi:10.1109/TSE.2018.2878020

[38] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *Proceedings of the 30th USENIX Security Symposium (SEC '21)*. USENIX Association, Boston, MA, 2525–2542.

[39] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, Hubert Garavel and John Hatcliff (Eds.). Springer, Berlin, Heidelberg, 553–568. doi:10.1007/3-540-36577-X_40

[40] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1193–1204. doi:10.1145/2508859.2516703

[41] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. Association for Computing Machinery, New York, NY, USA, 449–459. doi:10.1145/2635868.2635913

[42] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (June 2018), 6. doi:10.1186/s42400-018-0002-y

[43] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. 2013. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 112–122. doi:10.1145/2483760.2483778

[44] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *Proceedings of the 31st USENIX Security Symposium (SEC '22)*. USENIX Association, Boston, MA, 143–160.

[45] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN '17)*. Association for Computing Machinery, New York, NY, USA, 196–199. doi:10.1145/3092282.3092295

[46] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (Nov. 2021), 2312–2331. doi:10.1109/TSE.2019.2946563 Conference Name: IEEE Transactions on Software Engineering.

[47] Filipe Marques, Mafalda Ferreira, André Nascimento, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2025. *Automated Exploit Generation for Node.js Packages*. doi:10.5281/zenodo.15009156

[48] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. 2022. Concolic Execution for WebAssembly. In *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP '22)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:29. doi:10.4230/LIPIcs.ECOOP.2022.11

[49] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS '18)*. San Diego, CA, USA. doi:10.14722/ndss.2018.23309

[50] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Association for Computing Machinery, New York, NY, USA, 455–465. doi:10.1145/3338906.3338933

[51] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84. doi:10.1109/ICSE.2007.37

[52] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. DexterJS: robust testing platform for DOM-based XSS vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. Association for Computing Machinery, New York, NY, USA, 946–949. doi:10.1145/2786805.2803191

[53] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1757–1771. doi:10.1145/3133956.3133959

[54] João Madeira Pereira, Filipe Marques, Pedro Adão, Hichem Rami Ait El Hara, Léo Andrès, Arthur Carcano, Pierre Chambart, Nuno Santos, and José Fragoso Santos. 2024. Smt.ml: A Multi-Backend Frontend for SMT Solvers in OCaml. (2024). https://inria.hal.science/hal-04761767

[55] Frederico Ramos, Nuno Sabino, Pedro Adão, David A. Naumann, and José Fragoso Santos. 2023. Toward Tool-Independent Summaries for Symbolic Execution. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:29. doi:10.4230/LIPIcs.ECOOP.2023.24

[56] Ryan Dahl and OpenJS Foundation. 2009. Node.js JavaScript Runtime. https://github.com/nodejs/node visited on 2014-11-13.

[57] Gabriela Sampaio, José Fragoso Santos, Petar Maksimović, and Philippa Gardner. 2020. A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP '20, Vol. 166)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:29. doi:10.4230/LIPIcs.ECOOP.2020.28

[58] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. Association for Computing Machinery, New York, NY, USA, 615–618. doi:10.1145/2491411.2494598 event-place: Saint Petersburg, Russia.

[59] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. Association for Computing Machinery, New York, NY, USA, 842–853. doi:10.1145/2786805.2786830

[60] Faysal Hossain Shezan, Zihao Su, Mingqing Kang, Nicholas Phair, Patrick William Thomas, Michelangelo van Dam, Yinzhi Cao, and Yuan Tian. 2023. CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-language Code Property Graph. In *Proceedings of the 2023 Network and Distributed System Security Symposium (NDSS '23)*. San Diego, CA, USA. doi:10.14722/ndss.2023.2

[61] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *Proceedings of the 27th USENIX Security Symposium (SEC '18)*. USENIX Association, Baltimore, MD, 361–376.

[62] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS '18)*. San Diego, CA, USA. doi:10.14722/ndss.2018.23071

[63] Marius Steffens and Ben Stock. 2020. PMForce: Systematically Analyzing postMessage Handlers at Scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 493–505. doi:10.1145/3372297.3417267

[64] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. Internet Society, San Diego, CA. doi:10.14722/ndss.2016.23368

[65] Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2024. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Proceedings of the ACM on Programming Languages* 8 (2024). doi:10.1145/3689733 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[66] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. Association for Computing Machinery, New York, NY, USA, 899–913. doi:10.1145/3453483.3454084

[67] Tobias Koppers, Sean Larkin, Johannes Ewald, Juho Vepsäläinen, Kees Kluskens, and Webpack contributors. 2014. Webpack Bundler. https://webpack.js.org/ visited on 2024-11-11.

[68] Fish Wang and Yan Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev '17)*. doi:10.1109/SecDev.2017.14

[69] Yu Wang, Fengjuan Gao, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. 2022. Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software. *IEEE Transactions on Software Engineering* 48, 1 (Jan. 2022), 346–363. doi:10.1109/TSE.2020.2989171 Conference Name: IEEE Transactions on Software Engineering.

[70] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE, Berkeley, CA, USA, 590–604. doi:10.1109/SP.2014.44

Received 2024-11-15; accepted 2025-03-06