APPENDIX A
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: NODEMEDIC-FINE can be found here: https://github.com/NodeMedicAnalysis/NodeMedic-FINE. Please use the following commit hash for artifact evaluation: `18bf2e0937ce7c69daffd8b7c8130e0215d25f17`.

2) Hardware dependencies: 5 GB Storage, 4 GB RAM.

3) Software dependencies: Tested operating systems: macOS, Linux. Required software: Docker ($\geq$ version 27).

4) Benchmarks: The evaluations in this paper involved: 1) Large-scale collection of packages from the npm software repository. While that dataset is not part of the artifact, we include two experiments representative of it. 2) Evaluation over the packages in SecBench.js. The dataset can be obtained at https://github.com/cristianstaicu/SecBench.js, but does not need to be downloaded for the artifact experiments.

B. Artifact Installation & Configuration

This section describes the installation steps required to set up NODEMEDIC-FINE. All the steps below are also described in the README.md file present in the repo.

a) Docker installation: Issue the following command in the root of the project (in the same directory as the Dockerfile) to build the Docker container:

```
docker build --platform=linux/amd64 -t nodemedic-fine .
```

For reference, a fresh build takes around 3 minutes on a M1 Mac. After building, the newly created image can be listed:

```
$ docker image ls
REPOSITORY      TAG     IMAGE ID     CREATED        SIZE
nodemedic-fine  latest  5124b389f2b2 8 seconds ago 2.43GB
```

b) Note for ARM-based systems: When running the Docker container you may see the following warning, which can safely be ignored:

```
WARNING: The requested image's platform (linux/amd64) does
    not match the detected host platform (linux/arm64/v8)
    and no specific platform was requested
```

The warning is because the Docker container will be run using cross-architecture emulation.

C. Experiment Workflow

The high-level workflow of using NODEMEDIC-FINE is as follows:

1) A target npm package is selected. Both the name and version of the package must be known.
2) NODEMEDIC-FINE is invoked on the package via a Docker container. The end-to-end NODEMEDIC-FINE infrastructure is executed: the package is automatically downloaded, set up within the Docker container, analyzed, and potentially confirmed to have an exploitable flow.
3) A results file is output by NODEMEDIC-FINE. This file can be processed to measure metrics for that particular run (entry points, provenance graph size, etc.)

To run an evaluation over a set of packages, the above steps are repeated per package, and results are aggregated across packages.

D. Major Claims

This artifact provides two experiments that allow for replication of two underlying claims made by the paper:

- (C1) NODEMEDIC-FINE is able to uncover potential Arbitrary Command Injection (ACI) flows in Node.js packages, and can automatically synthesize exploits that confirm their exploitability. This claim is supported by experiment E1. This corresponds to NODEMEDIC-FINE's ability to find 1788 potential ACI flows in a large-scale dataset and automatically confirm 612 of them, as reported in Section V.B.
- (C2) NODEMEDIC-FINE is also able to find potential Arbitrary Code Execution (ACE) flows in Node.js packages and automatically confirm their exploitability via automatic exploit synthesis. This claim is supported by experiment E2. This corresponds to NODEMEDIC-FINE's ability to find 469 potential ACE flows in a larger-scale dataset and automatically confirm 154 of them, as reported in Section V.B.

The paper also makes claims about the *large-scale* effectiveness of the NODEMEDIC-FINE fuzzer, inference, and enumerator approach to Node.js package exploit discovery and confirmation (Section V.B-E). Given that these claims manifest through analysis of *all* packages in npm with more than 1 weekly download (Section V.A), it is infeasible to replicate those results without a similarly large dataset.

E. Evaluation

1) Experiment (E1): [ACI Flow] [5 human-minutes + 5 compute-minutes]: In this experiment, NODEMEDIC-FINE will analyze a Node.js package, uncover a potential ACI flow, and automatically synthesize an exploit that confirms it is exploitable.

[How to] Use NODEMEDIC-FINE to analyze node-rsync@1.0.3, which has a disclosed ACI vulnerability (https://security.snyk.io/vuln/SNYK-JS-NODERSYNC-568773), and review NODEMEDIC-FINE's output to see the uncovered confirmed-exploitable package API.

[Preparation] As a prerequisite, the previous set of steps (Artifact Sections A-B) must have been followed to the point where a NODEMEDIC-FINE Docker image has been successfully built.

[Execution] Issue the following command to invoke NODEMEDIC-FINE on the package:

```
docker run --rm -it nodemedic-fine --package=node-rsync --
    version=1.0.3 --mode=full
```

The command should take under 5 minutes to complete (52s on an M1 Pro Mac), and should end with the following output:

```
...
info:   Exploit(s) found for functions: execute
...
info: Done with analysis
```

That output is followed by a JSON object:

```
{"rows":[{"id":"node-rsync","index":0,"version":"1.0.3",
...}
```

[Results] NODEMEDIC-FINE emits its key results via the JSON blob output at the end of the package analysis. At the top level, the JSON object is a list of "rows" where each row is an entry about an analyzed package.

In results of the previously run analysis, we see one entry for the target package. In this entry, we can see that an ACI sink was executed (`execSync`), and that an object input (the `exploitString` value) was found that confirms the exploitability of the package API, `runCommand`:

```
"id": "node-rsync",
...
"version": "1.0.3",
...
"sinksHit": ["execSync"],
...
"exploitResults": [{
    "exploitFunction": "execute",
    "exploitString": "{\"flags\":\"BC $(touch\",\"source
        \":\"/tmp/success);#\"}"
}],
```

For completeness, each field is explained below:

- "id": Package name.
- "index": Index in the npm package repo (gathering only).
- "version": Package version.
- "downloadCount": Weekly download count (gathering).
- "packagePath": Path to installed package.
- "hasMain": Whether the package has a main script.
- "browserAPIs": List of browser APIs in the package.
- "sinks": List of NodeMedic-FINE–supported sinks found in the package.
- "sinksHit": List of sinks executed.
- "entryPoints": List of package public APIs.
- "treeMetadata": Metadata about the package's dependency tree (size, depth, etc.).
- "sinkType": Type of sink (ACI, "exec", or ACE, "eval").
- "synthesisResult" Synthesized package exploit input.
- "candidateExploit": Candidate exploit for the package.
- "exploitResults": Results of executing candidate exploit.
- "taskResults": Object with status and runtime for every NODEMEDIC-FINE internal task run.

2) Experiment (E2): [ACE Flow] [5 human-minutes + 5 compute-minutes]: In this experiment, NODEMEDIC-FINE will analyze a Node.js package, uncover a potential ACE flow, and automatically synthesize an exploit that confirms it is exploitable.

[How to] Use NODEMEDIC-FINE to analyze node-rules@3.0.0, which has a disclosed ACE vulnerability (https://security.snyk.io/vuln/SNYK-JS-NODERULES-560426), and review NODEMEDIC-FINE's output to see the uncovered confirmed-exploitable package API.

[Preparation] As with Experiment E1 (A-E1), please ensure the NODEMEDIC-FINE Docker image has been built.

[Execution] Run following command to invoke NODEMEDIC-FINE on the package:

```
docker run --rm -it nodemedic-fine --package=node-rules --
    version=3.0.0 --mode=full
```

The command should take under 5 minutes to complete (51s on an M1 Pro Mac), and should end with the following output, below which a JSON results object will be printed:

```
...
info:   Exploit(s) found for functions: fromJSON
...
info: Done with analysis
{"rows":[{"id":"node-rules"
...}
```

[Results] In the results object, we can see that an ACE sink was executed (`eval`), and that an object input (the `exploitString` value) was found that confirms the exploitability of the package API, `runCommand`:

```
"id": "node-rules",
...
"version": "3.0.0",
...
"sinksHit": ["function", "execSync", "eval"],
...
"exploitResults": [{
    "exploitFunction": "fromJSON",
    "exploitString": "{\"condition\":\"global.CTF())//\"}"
}],
```

F. Customization

In the above experiments, analysis artifacts are stored within the Docker container. This is beneficial for security, but can make it difficult to access all of NODEMEDIC-FINE's outputs. For packages that one has confidence are not malicious/malware, NODEMEDIC-FINE can be run (from the repository root) with a Docker mounted volume to enable direct access to the package under test and the analysis results:

```
docker run -it --rm -v $PWD/packages/:/nodetaint/
    packageData:rw -v $PWD/artifacts/:/nodetaint/
    analysisArtifacts:rw nodemedic-fine --package=node-
    rsync --version=1.0.3 --mode=full
```

Then, `$PWD/packages` directory will contain the package's source code, while the `$PWD/artifacts` directory will now have the analysis results, including coverage files from the fuzzer, the provenance tree, and synthesized exploits. You will find the following files there:

- `results.json`: Overall analysis results.
- `fuzzer_progress.json`: Coverage information from the fuzzer, as a list of pairs (timestamp, coverage).
- `fuzzer_results.json`: General information from the fuzzer.
- `run-<package_name>.js`: Driver that imports the package and the fuzzer and performs fuzzing.
- `run-<package_name>2.js` Second driver which only calls the potentially vulnerable entry point with the fuzzer-generated input, if NODEMEDIC-FINE finds a flow.
- `taint_0.json`: The provenance graph, a `.pdf` visualization of it also exists, if NODEMEDIC-FINE finds a potential flow.
- `poc<argument_number>.js`: Automatically synthesized exploit driver that imports the package and tries to exploit it, if NODEMEDIC-FINE finds a flow.