CHARON: Polyglot Code Analysis for Detecting Vulnerabilities in Scripting Languages Native Extensions

Raoul Scholtes*, Soheil Khodayari*, Cristian-Alexandru Staicu*, and Giancarlo Pellegrino*
*CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
Email: {raoul.scholtes, soheil.khodayari, staicu, pellegrino}@cispa.de

***Abstract*—Scripting languages like Python or JavaScript are extremely popular among developers, in part due to their massive open-source ecosystems that enable smooth code reuse. However, recent work shows that a lot of scripting code runs C/C++ code under the hood, via native extensions. This might introduce subtle security issues that can surprise the users. Prior work in this domain relies on simple, intra-procedural, flow-insensitive data flow analysis to detect such problems, but it is unclear if a more holistic polyglot static analysis could be feasible, and if so, what are its costs and benefits.**

In this work, we propose CHARON, the first inter-procedural, polyglot static analysis for detecting vulnerabilities in scripting languages. Our approach advocates for linking together the code property graphs of the different languages and performing cross-language data flow analysis by switching between code representations, when cross-language function calls are encountered. In this way, CHARON supports data flows that cross several times the language boundary, spanning multiple functions on either side. We evaluated CHARON on 11.8K polyglot packages from npm and PyPI, containing 896M lines of code. CHARON identified 5,813 manually-confirmed, vulnerable data flows in 116 packages. We performed a baseline comparison of CHARON with single-language analysis on native code, showing a ∼6x increase in true positives and ∼4% less false positive alerts. We demonstrated exploitability of the discovered vulnerabilities by creating 63 PoCs across 34 packages, showing, among others, how we can escalate a buffer overflow vulnerability in native extensions to arbitrary code execution, which we believe to be the first of its kind. Overall, our results show that inter-procedural, polyglot analysis is both feasible and effective for detection of native extension vulnerabilities.

1. Introduction

Over the past decades, scripting languages such as Python and JavaScript have become tremendously popular, fostered by extensive repositories of third-party code available via package managers like pip and npm. While most packages are libraries and frameworks written in the same scripting language, e.g., Django for Python or Express.js for JavaScript, more and more packages rely on functionality offered by *native* C/C++ extensions for performance reasons, e.g., `NumPy`, to access hardware devices, e.g., `Cylon.js`, or to reuse system libraries, e.g., `OpenSSL`. Vulnerabilities in native extensions can have devastating effects on the security of scripting programs and on the entire runtime, introducing security risks in the memory management such as buffer overflow vulnerabilities, which might allow attackers to achieve remote code execution.

Detecting vulnerabilities in scripting language packages is a challenging task. The vast majority of code analysis techniques work with one language only, either the scripting language, e.g., JavaScript [1–4] and PHP [5, 6], or the native C/C++ code [7–9]. While these techniques can effectively find vulnerabilities in the targeted language, they operate in isolation and cannot reason on two languages simultaneously, e.g., incorrectly flagging unreachable code. Recently, Staicu et al. [10] survey the security problems caused by native extensions in scripting languages and present a cross-language, static data flow analysis technique. However, their prototype is flow-insensitive, mostly intra-procedural, limited to data-flow analysis, and it only supports the JavaScript language. Moreover, their implementation is also limited to identifying misuses of the API that allow attackers to pass arguments of the wrong type to polyglot packages. Thus, the state of the art in this domain cannot handle flows that span multiple functions in one of the languages or that cross the language boundary multiple times. It is also unclear if bugs in native extensions can be leveraged for serious security attacks like arbitrary code injection. Beyond the security domain, Monat et al. [11] present a more sophisticate multilingual static analysis for Python to find runtime errors in native extensions. However, this approach is relatively slow due to reliance on abstract interpretation, it is Python-specific, and it is unclear how it can be adapted to the security domain to implement flow-based analyses, e.g., taint analysis. As a result, we still lack a technique that can systematically support reusable code analysis across languages.

In this paper, we present CHARON, a holistic static analysis based on fused code property graphs representing both the scripting code and the native extension, which we term polyglot property graph (PPG). To construct this graph, CHARON detects and interprets calls to specific APIs that allow cross-language function invocations. Using PPGs, CHARON performs a backward data flow analysis starting from sinks and traversing cross-language calls as they would be single-language calls. Our prototype includes queries for nine C/C++ security problems, modeled using 16 sinks and 31 mitigation types and marks all the inputs to a given polyglot package as sources. We extensively evaluate CHARON against real-world npm and PyPI polyglot packages, analyzing 11.8K packages

using native extensions, covering three languages, i.e., Python, JavaScript, and C/C++. Our evaluation shows that CHARON can identify 5,813 vulnerable data flows in 116 packages, showing that problematic native extensions are prevalent. For 34 packages, we create proof-of-concept exploits that show the presence of the security problem and we report them to maintainers. For the most serious ones, the community issued security advisories to warn about the high severity of the issues. For one buffer overflow vulnerability we discovered, we performed the difficult task of attempting an arbitrary code injection via overwriting the return address on the stack. We discuss the structure of the payload and the difficulties of deploying it in practice.

At a high level, our results show that a holistic, inter-procedural static analysis is effective at finding vulnerabilities in real polyglot packages. Our baseline comparison of CHARON with single-language analysis of native code suggests that CHARON results in a ∼6x increase in true positives and ∼4% less false positives. For example, we show that CHARON identifies 2,606 flows in which the sink is in the native code, but the vulnerability is unreachable from or mitigated in the scripting language. Such cases would result in false positives in a single-language analysis setup. We also show that the percentage of false positives produced by CHARON are comparable to that of a single-language approach. This may appear surprising, since the tool might flag multiple paths reaching a given flagged code location, thus, amplifying the number of false positives. However, we observed that this potential amplification effect is kept low by the other benefits of a polyglot analysis, such as ignoring unreachable code or detecting cross-language mitigation.

Finally, we believe to be the first to present a code injection payload that exploits a low-level, memory-management vulnerability in native extensions. While this shows the feasibility of such attacks, we also discuss the many hurdles attackers need to tackle when attempting to deliver such payloads at scale, under real-world conditions. In particular, it is extremely difficult to encode the correct jump address into UTF-8 string values that attackers can pass to the scripting code and further to the native code.

To summarize, this paper makes the following contribution:

- We present CHARON, the first polyglot, inter-procedural static analysis powered by code property graphs. At the core of our approach, there are nine novel cross-language taint-style graph queries to identify threats to memory integrity in polyglot packages.
- We evaluate CHARON on 11.8K packages using native extensions, of which 8.2K npm packages and 3.6K PyPI packages. The evaluation uncovers 17,268 potentially vulnerable data flows in 269 packages, of which we manually confirmed 5,813 cases across 116 packages.
- We perform a baseline comparison of CHARON with single-language analysis on the native code, showing ∼6x increase in true positives and ∼4% less false positives.
- We conduct a thorough analysis of the discovered vulnerabilities, showing the impact of language boundary on exploitability and creating 63 PoC exploits in 34 polyglot packages.

2. Background and Motivation

Before presenting our approach, we briefly introduce the APIs of common native extensions (§2.1) and the security threats posed by vulnerabilities in their code (§2.2).

2.1. Native Extensions

Scripting languages offer access to low-level OS primitives via built-in functions and modules, e.g., file or network I/O, that developers can extend via native extensions. A typical native extension has two parts: a wrapper module written in the scripting language and a native module, often in C/C++. Both parts can communicate via the native extension API, which the language interpreter implements, providing, for example, a C/C++ mapping of the scripting language data types, C/C++ primitives to operate with data types, and the mechanism to call functions in the native or scripting side.

2.1.1 Python Extension API The Python extension API [12] requires that C/C++ modules define all exposed functions to the Python interpreter in a specific format. For example, C/C++ functions callable from Python must have the same signature, e.g., `static PyObject* <name>(PyObject* self, PyObject* args)` is a C/C++ function implementing the class method `name` of the object `self`. The Python script can then import the native code as a library, e.g., via the `PyMethodDef` API [13], and then call the method via a library property. Figure 1 shows a Python module `main.py` calling the function `init` of the Python module `_main` (import of this module not shown) which is implemented in C/C++ in the `_main.c` module. Here, we replace the C/C++ signatures with a keyword `PY_METHOD`.

The inverse scenario is also feasible, where developers can call Python functions from the C/C++ module via the `PyObject_CallFunction` or `PyObject_CallMethod` APIs [14]. In this paper, we use the term `PY_CALLMETHOD` to refer to any of these functions for brevity. Figure 1 shows such an example where the C/C++ module `_main.c` calls the function `pack` (implemented in the Python module `_internal.py`) using the `PY_CALLMETHOD` function of the Python extension API.

2.1.2 JavaScript Extension API Similarly to Python, Node.js provides several alternatives for bridging between low-level code and JavaScript. First, the extension API [15] requires that C/C++ programs explicitly mark exposed functions to the JavaScript interpreter. This can be done, via the `NODE_SET_METHOD(exports, "g", f)` API that exposes a C function `f` as `g` to JavaScript [15], or via other APIs like `Nan::SetMethod()` and `Nan::SetPrototypeMethod()`, which are part of the so-called NAN API [16]. Another alternative is that C/C++ programs leverage the N-API library [17],

1. https://pypi.org/project/dulwich

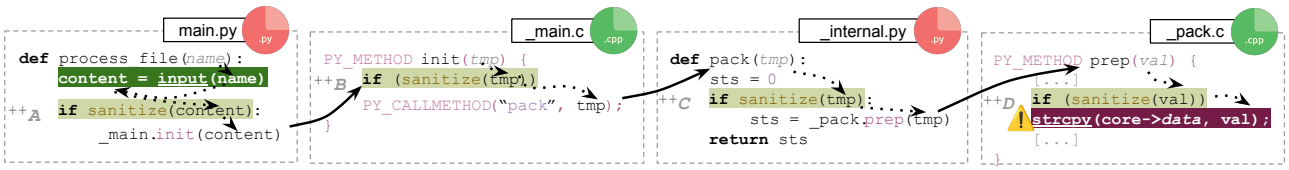


Figure 1: A buffer overflow vulnerability that cannot be discovered using single-language analysis, inspired by complex cross-language interactions that CHARON discovered in the Python package `dulwich`[1]. **Legend:** The lines marked with ++ show where potential mitigations could reside after patching, indicated by the `sanitize()` function. We use `PY_METHOD <name>` to replace the function signature `static PyObject* <name>(PyObject* self, PyObject* args)`; we also use `PY_CALLMETHOD` for `PyObject_CallMethod`.

e.g., using the `napi_call_function` method to call JavaScript functions.

2.2. Threats to Native Extensions

In this section, we illustrate a concrete risk introduced by native extensions, using a simplified, real-world polyglot package. We then discuss how other classes of vulnerabilities might be exposed via native extensions.

2.2.1 Motivating Example. Figure 1 shows a stack-based buffer overflow vulnerability due to complex cross-language interactions, inspired by a real flow in the Python package `dulwich`, which we discovered in our experiments. In this example, the vulnerability is the presence of an unvalidated data flow across language boundaries, where an attacker-controlled input in the Python space ultimately reaches a string copy operation in the C/C++ space. The attacker-controlled value is first copied in the C/C++ native library (pointed by the local variable `tmp`), then sent back to the Python space to the Python function `pack`, implemented in the Python module `_internal`, and, finally, sent to the C/C++ module once more via the `prep` function, which implements in the native space the copy operation. When looking at the four modules in Figure 1, none of them performs input validation, specifically length checking to prevent unbounded copying, leading to a vulnerable data flow pattern. The mitigation for this vulnerability, specifically input validation, can be implemented in any of the four modules, as indicated by the `sanitize()` function in Figure 1.

Accurate detection of the vulnerability in Figure 1 requires: (i) recognizing that the value in the sink is attacker-controlled, and (ii) determining the presence or absence of mitigation checks in the data flow. A single-language analysis in C/C++ space is not sufficient to capture these requirements due to its restricted view of the source code and may lead to both false positives and negatives. In fact, native-only analysis lacks visibility into the JS/Python layer, which may connect multiple C++ functions. Consequently, it cannot infer whether values reaching the sink from the language boundary are attacker-controlled (introducing false negatives), unless it considers all function parameters as such, causing false positives for validated inputs.

Illustrating the problem using Figure 1, a single-language approach can only backtrace incoming flows leading to the dangerous sink up to the method parameter `val` in `_pack.c` forming the language boundary, thus failing to identify that the value `val` originates from attacker-controlled input in `main.py`. Moreover, it only sees the presence or absence of mitigations in the two C/C++ modules of Figure 1, but not in the Python space. Therefore, if a value is properly sanitized before being passed from the scripting language to native code, a native-only analysis may introduce false positives.

As we will show in §4.2.6, cross-language code analysis is critical to accurately detect a significant number of vulnerabilities in polyglot packages.

Vulnerability	Reference
Buffer Overflow	[9, 10, 18–21]
Division by Zero	[9, 22, 23]
Format String	[24–26]
Integer Overflow	[9, 19, 27]
Memory Leak	[9, 28]
Null Pointer Dereference	[9, 28, 29]
Use-After-Free	[9, 10, 30, 31]

TABLE 1: Vulnerabilities affecting native code.

2.2.2 Native Code Vulnerabilities. We conducted a systematic search of the existing literature and MITRE CWE database [32], and compiled a comprehensive list of threats to native extensions, with consequences ranging from arbitrary code execution to violations of memory integrity, memory confidentiality, and program availability [9, 10, 18, 19, 28]. Table 1 summarizes the result of our review. Below, we discuss each threat.

Buffer Overflow. When writing attacker-controlled variables to a memory buffer, the program may overwrite memory regions, e.g., the memory stack, resulting in various security issues, from memory corruption to arbitrary code execution.

Division by Zero. Attacker-controlled variables can also be used in divisions, resulting in divisions by zero, which are undefined and cause a hard crash of the program. Attackers can exploit this vulnerability to conduct persistent denial-of-service attacks by repeatedly triggering the bug after restarting the application.

Format String Attacks. The C `printf` functions family can print messages with dynamic values using special string format identifiers, e.g., `%s` for strings. When attacker-controlled variables reach one of these functions, the attacker can inject such identifiers resulting in memory read operations or, in certain cases, overwriting memory regions.

Integer Overflow. Attacker-controlled variables can be used to create numeric values outside the range of its data type, resulting in unexpectedly small, large, or negative numbers. Attackers can exploit this behavior by tampering with the logic, increasing resource usage, e.g., the number of iterations of a loop, or altering the size of memory allocations.

Memory Leak. Not freeing allocated memory after its use can threaten long-lasting processes like web servers or daemons, where attackers can repeatedly call the affected component, constantly increasing memory consumption. Such an attack eventually exhausts available resources, resulting in a denial of service or a process crash out of memory.

Null Pointer Dereference. Several functions and APIs, including `malloc` and `CPython`, return a null pointer to indicate when an error occurred. If an attacker is able to reliably trigger an error condition and create a null pointer that is dereferenced in a later operation, they can cause a hard crash of the application, leading to a denial of service scenario and thus an availability violation.

Use After Free. After freeing a memory chunk, the corresponding address space is returned to the list of free chunks and can be used for subsequent allocations. Therefore, if an attacker has control over a pointer dereferencing an already freed memory region, they can read or corrupt sensitive information belonging to a later allocation.

3. CHARON

In this section, we describe CHARON, a static analysis framework for security testing of polyglot packages. Given as input the source code of a package, it first performs a series of preprocessing tasks (e.g., code normalization, language identification) and then creates a graph-based representation of the program under test, capturing control and data flows spanning function calls, various files and multiple languages. To achieve this, CHARON starts by constructing distinct graphs for both the high-level and low-level programs, producing a unified code representation for all of them [33]. Then, CHARON links elements of the two graphs together and establishes cross-language interconnections, accounting for function calls and data flows across language boundaries. Finally, CHARON traverses the resulting graph to search for potentially insecure program properties and outputs the results. CHARON provides a comprehensive catalog of built-in queries for vulnerability definitions presented in our threat model (see, i.e., §2.2). For example, it can automatically query the presence of unvalidated data flows from attacker-controlled inputs to security-sensitive instructions, considering common types of sanitization operations applied on data flows. Figure 2 depicts an overview of the CHARON's architecture. The rest of this section details each of the components of our framework.

3.1. Preprocessing

To be able to run CHARON at scale over hundreds of thousands of packages, it can optionally perform a series of package preprocessing steps. Specifically, the preprocessing component performs two tasks. Firstly, it checks the source code to identify the programming languages employed within a package, thereby ascertaining whether the package qualifies as a polyglot package. This initial determination is also essential for triggering the appropriate graph construction module. Secondly, for optimization purposes, it assesses whether the package contains any security-sensitive instructions based on the vulnerability definitions outlined in §2.2. This proactive check aims to circumvent unnecessary processing steps in cases where security-sensitive instructions are absent, aligning with the optimization strategies employed by other graph-based code analysis approaches [5, 9, 18, 34].

3.2. Data Modeling

The data modeling component creates a graph-based model of the preprocessed source code, leveraging a modified engine of Joern [9, 35], as discussed below.

3.2.1 CPG Construction. Code Property Graphs (CPGs) [5, 9] are graph-based representations of code that capture both the structure and behavior of programs, consolidating multiple static code representations, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). These representations collectively model the hierarchical structure of the program's syntactic constructs, the sequence and conditions governing instruction execution, and the data flow and control dependencies among program statements. CHARON uses and extends the concept of CPGs, similarly to other state-of-the-art code analysis tools (e.g., [34, 36]). Starting from the preprocessed source code of a package, CHARON constructs one CPG for the high-level program and another CPG for the low-level code, which are generated independently. CHARON relies on the Joern CPG specification [33, 37] to produce a unified syntax tree for the AST of the constructed CPGs of Python, JavaScript and C/C++ programs, which facilitates the process of creating graph traversal queries.

3.2.2 CPG Linking. After creating CPGs for the script and native code, CHARON analyzes them to discover control and data flow edges connecting the two programs. Specifically, it looks for instructions that call a function of the other part of the program or return information in response to such function calls. As a result, CHARON creates a unified CPG representation, which we call a Polyglot Property Graph (PPG). PPGs can handle bidirectional calls and context switches between script and native code, offering edges that model data dependencies and calls across language boundary. Algorithms 1 and 2 summarize the CPG linking procedure to generate a PPG, starting from the script and native CPGs. At a high level, they operate in two stages, where they model the transfer of control and data propagation: 1) from script program to native code, and 2) from native code to the script program, as detailed next.

Edges from Script to Native Code. Native extensions enable the invocation of their API-exposed functions from script code. This necessitates extensions to explicitly declare or expose accessible functions through API calls, specifying the target function and its corresponding name in the script code (Cf. §2.1). To systematically identify potential cross-language call candidates, Algorithm 2 first searches the AST for such API function expose instructions in the native code (line 2). These instructions should follow the syntax presented in §2.1 in order to make functions in the native code callable by the scripting program. For each API expose instruction, the algorithm extracts the function pointer/identifier and traces the CPG

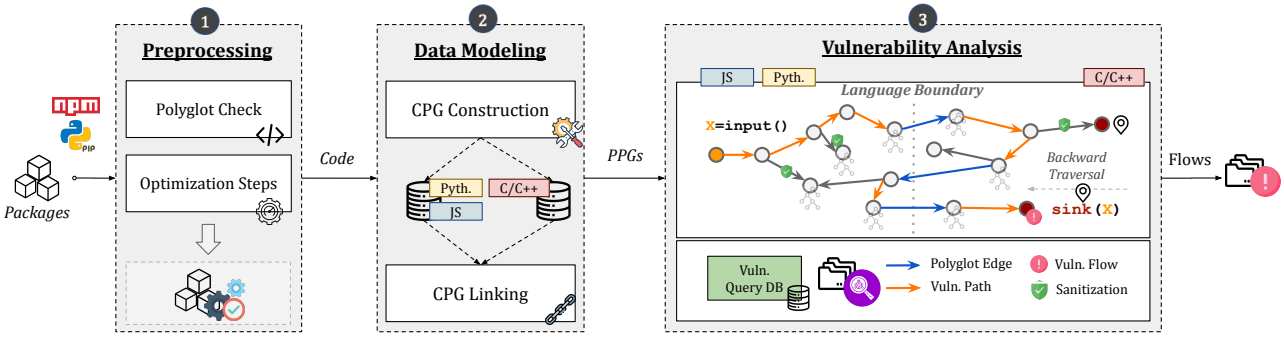


Figure 2: Architecture of CHARON. **Legend:** CPG = Code Property Graph, PPG = Polyglot Property Graph.

following data flow edges to locate the actual function declaration (lines 4-5).

Then, it establishes a map `calleesNative` to store potential callees, keeping track of the function signature (e.g., method name, class, etc) and its declaration (lines 7-9). CHARON supports functions defined in global scope, namespaces, class methods as well as macros. For example, for a method `F` defined in class `C`, the signature would be the string ``C:F''. If the API instruction exposes `F` to the script code with an alias name, say `G`, then the callees map stores the alias name instead. In addition, API expose instructions that make declared functions invokable by the scripting program can happen in an inter-procedural context. CHARON supports these cases by tracking function identifiers specified in these instructions across function boundaries. The complete list of supported expose APIs in C++ that mark a function as invokable in Python and JavaScript are in Table 8.

Finally, to detect and connect cross-language edges from script to native code, Algorithm 2 calls the `AddXLEdges()` routine of Algorithm 1 (line 8). This routine searches the script code CPG for call expressions using any of the function names in the previously-generated callees map (lines 3-5), and create a call edge between the two CFG nodes of the callee-caller and PDG data dependency edges for the invocation variables (line 6-10). Upon identification, these edges allow CHARON to enhance its data flow analysis, successfully traversing the language boundary.

To illustrate this process, the example in Figure 1 exposes two native functions to the script call interface, i.e., `init()` and `prep()`. Both functions are exposed with the `PyMethodDef` API without using any pointers (i.e., the function declaration occurs in the same instruction). Therefore, the function name pointer analysis in line 6 returns the same instruction. Afterwards, Algorithm 2 creates a callees map, associating each function name exposed to the script code to the function definition in the native code, with the signature `'global:prep'` mapped to the `prep()` function, and the signature `'global:init'` mapped to the `init()` function declaration. Note that the 'global' string suggests that both functions are defined in the global scope. In the example in Figure 1, the `strcpy()` call expression in `_pack.c` accepts incoming values from the arguments of `prep()`. Since `prep()` has been identified as a script-exposed function, Algorithm 2 checks for potential call expressions in the script code using the corresponding

Algorithm 1: Add Cross-Language Edges Routine

```
1  Function AddXLEdges(cpg, callees)
2      callEdges, pdgEdges ← []
3      for c in call_expressions(cpg) do
4          k ← get_signature(c.name)
5          if k in callees then
6              callee ← callees[k]
7              callEdges.append([caller, callee])
8              for i ← 1 to c.args.length do
9                  e ← [caller.arg[i], callee.param[i]]
10                 pdgEdges.append(e)
11     return callEdges, pdgEdges
```

name in the callees map. Then, it discovers a matching `prep()` call in `_internal.py`, and connects a call edge between the caller-callee CFG nodes as well as a PDG data dependency edge to link variable `val` in `_pack.c` to `tmp` in `_internal.py`.

Edges from Native to Script Code. Polyglot programs may also perform the inverse scenario, where native code invokes functions situated in the script code. For example, in Figure 1, the `pack()` function in `_internal.py` is invoked from `_main.c` via an API call using the `PY_CALLMETHOD()` function. Algorithm 2 also handles this scenario by adopting a nearly mirrored approach (lines 9-12). First, it searches the script CPG for *all* function definition nodes, creating a callee map of function signature-definition pairs. This is because contrary to the previous scenario, there is no syntactic distinction between the script functions in terms of their exposure to native code, and developers can invoke any function. Therefore, Algorithm 2 retrieves *all* function definitions from the AST (lines 9-10) and stores them in a map `calleesScript` (lines 11).

Then, it searchers the native code for call expressions matching one of the entries in `calleesScript` map by invoking the `AddXLEdges()` routine (line 12). Similarly to the previous case, CHARON connects a call edge between the two nodes and a PDG data dependency edge between the caller passed arguments and callee parameters upon identification.

3.3. Vulnerability Analysis

Given the PPG of a package under test, we now show how to use it to study program properties and detect vulnerabilities automatically.

Algorithm 2: CPG Linking Algorithm

Input : cpgNative, cpgScript
Output: callEdges, pdgEdges

$calleesNative, calleesScript \leftarrow \{\}$
$e \leftarrow get\_function\_expose\_instructions(cpgNative)$
foreach i ***in*** e **do**
  $p \leftarrow extract\_pointer(i)$
  $f \leftarrow trace\_data\_flow(p)$
  $k \leftarrow get\_signature(f)$
  $calleesNative[k] \leftarrow f.declaration$
$callEdges, pdgEdges \leftarrow$`AddXLEdges(`$cpgScript, calleesNative$`)`
foreach f ***in*** $get\_functions(cpgScript)$ **do**
  $k \leftarrow get\_signature(f)$
  $calleesScript[k] \leftarrow f.declaration$
$callEdges, pdgEdges$`+=AddXLEdges(`$cpgNative, calleesScript$`)`

3.3.1 Graph Traversals. Graph traversals provide a way to navigate the complex relationships and dependencies within a software program, enabling analysts to extract meaningful information about the code syntax and semantics, including the detection of security vulnerabilities. CHARON supports traversals in the Joern querying language [38, 39], which is based on the Gremlin language [40] under the hood. CHARON's querying language uses a unified syntax tree specification [33, 37] for different programming languages, which facilitate the creation of graph traversals. The primary design choice of using graph databases is driven by performance; they can efficiently manage large graphs that are challenging to store and process purely in memory or with traditional databases.

3.3.2 Studying Polyglot Program Properties. Given a PPG as described in §3.2, CHARON enables analysts to define and execute arbitrary graph traversals. Specifically, analysts can run queries to identify target graph patterns (i.e., nodes, edges and relationships) matching a specific property or vulnerability. For example, it enables them to conduct data flow analysis, reachability analysis, and points-to analysis, or identify inter-procedural method invocations and perform analysis of control flow transfers, which is necessary for the detection of security vulnerabilities. In the remainder of this section, we show how we instantiated CHARON to detect the vulnerabilities we presented in §2.

3.3.3 Vulnerability Detection. Starting from the constructed models of §3.2, we formulate the vulnerability detection task as a series of PPG traversals. These traversals can be divided in four general categories: (i) traversals that identify attacker-controlled program inputs (i.e., parameters of API functions in script code), hereafter called *sources*; (ii) traversals that identify security-sensitive instructions for each vulnerability in our threat model of §2.2, hereafter *sinks*; (iii) traversals that perform data flow analyses from sources to sinks that may cross the language boundary; and finally (iv) traversals that determine if mitigations are in place for each vulnerability which affect the detected data flows. The rest of this section details each step.

Finding Source Nodes. In our analysis of NPM and PyPI packages, we consider parameters passed to package export functions as our primary input sources . To pinpoint these sources, our CPG queries begin by identifying function export statements at the package level, which detail the function identifiers or pointers intended for export. We then trace these identifiers or pointers to their corresponding function declarations and extract the parameters from function declarations, marking them as our sources in data flow analysis. This method ensures a thorough identification of entry points (untrusted inputs) in the packages under examination.

Finding Sink Nodes. As the next step, CHARON defines and runs a comprehensive catalogue of graph queries to identify sinks in the native code which corresponds to each of the vulnerabilities of our threat model, storing the resulting node information. The complete list of supported sinks in our prototype is in Table 2.

Polyglot Data Flow Analysis. CHARON analyzes the propagation of the data flows in the program by backtracking the PDG and call edges one by one, also across the language boundary, since PPGs contain cross-language edges (Cf. §3.2.2).

To identify the vulnerabilities described in our threat model of §2.2, CHARON conducts one or several data flow analysis queries. Specifically, for vulnerabilities such as buffer overflow, division by zero, format string errors, and integer overflow, CHARON executes a single data flow analysis for each type. This analysis determines whether an input source reaches critical sinks in the code, such as a memory buffer, a division instruction, a format string operation, or an integer assignment.

However, the analysis becomes more intricate for other types of vulnerabilities. For instance, null pointer dereference involves two distinct data flow analyses: first, tracing the path from an input source to an API that returns a null pointer, and second, tracking that null pointer to where it is dereferenced in the code. Use After Free vulnerabilities require three linked data flows: the source reaching a pointer to a memory address, which then reaches a free operation, and finally leads to a dereference of that freed pointer. Memory leaks represent an even more complex challenge. Here, the analysis starts with the input source reaching a memory allocation instruction, and checking if in all following data flow paths, there are no instructions freeing that allocated memory (i.e., N data flow analyses), highlighting the paths that contribute to the memory leak. Even if there are data flows between a source and sink, mitigation instructions along or near the flow may prevent the occurrence of a vulnerability (e.g., input validation checks).

Finding Mitigations and Input Validation Nodes. CHARON defines and executes a set of graph traversals to identify operations in both script and native code containing a sanitization instruction. The goal of this step is to exclude data flows that are properly validated from the output. CHARON can identify various types of input validation operations, including explicit input length checks, equality comparisons against constants, divisor variable not zero checks, mitigations stemming from loop conditions (e.g., controlling divisors or tainted parameters), pointer reference checks such as against the NULL value, casting to unsigned type, dynamic buffer allocations based on input size such as `malloc(strlen(input))`, and finally checks ascertaining the memory allocation size.

Vulnerability \ Sink API	operator:division	operator:modulo	printf()	sprintf()	fprintf()	snprintf()	dprintf()	operator:addition	operator:multiplication	memcpy()	malloc()	calloc()	read()	strcpy()	strcat()	free()	Mitigations JS	Pyth.	C++	Total
Div by zero	●	●															1	1	2	4
Fmt string			●	●	●	●	●												2	2
Sprintf overflow				●													1	1	3	5
Strcpy/strcat overflow														●	●		1	1	2	4
Int. overflow								●	●										1	1
Memcpy overflow										●							1	1	2	4
Mem leak											●	●							2	2
Null pointer dereference											●	●	●				1	1	4	6
UAF																●			3	3
Total																	5	5	21	**31**

TABLE 2: The complete list of sinks supported by CHARON for each vulnerability and the number and context of mitigation queries per vulnerability class. Legend: ● = applicable.

CHARON also implements vulnerability-specific mitigation queries. For example, `sprintf()` calls tainted with attacker data that do not have the string specifier `%s` are considered mitigated against buffer overflows, since they mangle the input value, or shorten it to fit into the buffer (i.e., avoid the attacker's arbitrary length capability). Similarly, memory allocations for which `PyArray_ENABLEFLAGS()` is called with the flag `NPY_ARRAY_OWNDATA` do not lead to a memory leak vulnerability. In total, CHARON can perform PPG queries to detect 31 different types of potential mitigations for vulnerabilities outlined in §2, leveraging the CFG, PDG, AST, and the call graph. Table 2 shows the distribution of queries across different programming language contexts and vulnerability classes. These queries can reduce potential false positives of the vulnerability detection task. For example, consider a scenario where the script code includes a length check on the input before calling a native API from C/C++, which itself lacks any length checks, like in the case of a `memcpy()` instruction. In such instances, CHARON identifies and marks the relevant PPG node associated with the script code. This marking serves to emphasize an input validation operation. Therefore, when checking for a memcpy overflow vulnerability, CHARON has the capability to filter out sanitized data flows from the results. This capability contrasts with a native-only taint analysis approach, which is more likely to report such data flows as potential vulnerabilities.

3.4. Implementation

We implemented our polyglot static analysis approach over Joern [9] in 4,421 LoC (excluding comments and white spaces), where we used Scala and the Joern SAST engine [35] to define and execute CPG queries, such as reachability and data flow analyses. We also use c2cpg [41], jssrc2cpg [42] and pysrc2cpg [43] libraries to build C/C++, JavaScript, and Python CPGs, respectively. CHARON combines the generated CPGs following the algorithm presented in Algorithm 2 to generate a PPG and provide an interactive and API-driven frontend for analysts to run their queries over the PPG. Finally, we implemented the preprocessing steps outlined in §3.1, and the orchestration of the data modeling tasks in Python, which are necessary for scalability reasons.

Enhancements to Static Analysis Engine. Unfortunately, we could not use Joern as-is, and encountered several limitations during our experiments of §4, particularly when Joern generates the inter-procedural data flow and call graphs as a part of the CPG construction. For example, we observed that the c2cpg library fails to create call edges to methods defined in different files. It also generated incorrect PDG edges for unrelated nodes and overlooked edges in certain types of assignments (e.g., `String::Utf8Value address(arr[0])`), reassignments and dereferencing scenarios. Likewise, the jssrc2cpg library faced challenges in creating call graph edges when using the ES6 JavaScript features. Finally, pysrc2cpg lacked inter-procedural call graph and PDG edges, and frequently encountered crashes during the generation of Python-based CPGs.

In total, we identified nine implementation issues across these three CPG generation libraries, which we responsibly disclosed to the Joern creators, Shiftleft [44], all of which were confirmed. As of now, Shiftleft successfully patched six out of the nine reported bugs. For the remaining cases, particularly the ones affecting the PDG generation, Shiftleft responded that they are known issues (see, e.g., [45]), which they plan to fix soon. However, for all remaining issues, we patched or tweaked the implementation of Joern to mitigate them. For example, alongside data flow queries from sources to sinks, CHARON follows dataflow edges until a problematic code pattern [46] is hit. Then, it runs an AST query to determine the correct continuation point, when possible, and continue dataflow analysis from there. Similarly, instead of only relying on call graph edges, CHARON runs an additional AST query to determine call points (i.e., `cpg.call(method.name)` instead of `method.callIn`) to reason about potentially missing call edges.

	npm		PyPI		Total	
	#	*LoC*	#	*LoC*	#	*LoC*
All packages	1.6M	-	318K	-	1.9M	-
With native code	8.7K	-	4K	-	12.8K	-
Contain sinks	8.2K	657M	3.6K	239M	11.8K	896M
PPGs	7.8K	477M	3.3K	196M	**11.2K**	**673M**

TABLE 3: Overview of our dataset and preprocessing steps.

4. Empirical Evaluation

We now assess the efficacy and practicality of CHARON in detecting the vulnerabilities of §2.2 in the ecosystem of two popular programming languages, i.e., npm for JavaScript and PyPI for Python.

Overview. We collected and processed over 1.9M packages from npm and PyPI ecosystems and identified 11.2K packages containing both scripting (i.e., Python or JavaScript) and native (i.e., C/C++) code, as well as at least one security-sensitive instruction. Then, we instantiated CHARON on each of the 11.2K packages, and generated a PPG for for each one, covering about 673M LoC. After analyzing these packages following the approach presented in §3.3, CHARON identified a total of 17,268 potentially insecure data flows across 269 packages in both npm and PyPI. Finally, we demonstrate the exploitability by focusing on a random subset of the data flows to create proof-of-concept exploitations for 34 packages, including popular ones like *barcode4nodejs* and *spinsfast*, resulting in critical consequences like arbitrary code execution, denial of service, memory corruption, and information leakage.

Before presenting our findings, we discuss our methodology and properties of the problem space (§4.1). Then, we report the results of our experiments (§4.2), and finally, conclude with the analysis of CHARON's results.

4.1. Experimental Setup and Methodology

4.1.1 Data Collection and Preprocessing We downloaded all npm and PyPI packages (i.e., 1.9M) in May 2021 and performed the data preprocessing steps described in Figure 2. As a first step, we checked in the input packages for the presence of both native and script code and filtered out the package otherwise. We keep packages that contain C/C++ source code and include one of the following headers: `node.h`, `napi.h`, and `nan.h` for JavaScript, `Python.h` for Python. This resulted in a total of 12,856 packages, of which 8,757 are npm packages, and the remaining 4,099 are PyPI.

Then, we checked if the package source code contained any security-sensitive instructions, i.e., sinks. If the package includes a sink, CHARON analyzes it, searching for vulnerabilities; otherwise, it discards it. In total, 11,859 packages contain at least one sink call, of which 8.2K are npm packages and 3.6K are PyPI. Table 3 summarizes our data collection and preprocessing steps.

4.1.2 Model Construction After identifying packages with native code and sensitive sinks, CHARON generated one PPG for each package. In total, CHARON successfully generated PPGs for 11,270 packages, as shown in Table 3, and failed to generate PPGs for 589 packages. These failures occurred because the underlying static analysis engine that CHARON relies on (i.e., Joern [35]) broke during the construction of CPGs for either the native or script program. We note that CHARON's implementation is influenced by the limitations of the individual CPGs generated by Joern.

Vulnerability	npm		PyPI		Total	
	Flows	*Pkgs.*	*Flows*	*Pkgs.*	*Flows*	*Pkgs.*
Div. by zero	211	22	3,979	16	4,190	38
Fmt string	1	1	66	3	67	4
Int. overflow	715	97	1,855	34	2,570	131
Memcpy overflow	115	13	1,154	7	1,269	20
Mem leak	339	34	2,242	25	2,581	59
Null pointer deref.	4	4	3,486	23	3,490	27
Sprintf overflow	9	2	191	8	200	10
Strcpy/strcat overflow	86	55	2,648	13	2,734	68
UAF	37	2	130	2	167	4
Total	1,517	173	15,751	96	**17,268**	**269**

TABLE 4: Overview of vulnerable data flows detected by CHARON. The table shows the fraction of the data flows crossing the language boundary for different vulnerability types.

4.2. Evaluation Results

After generating PPGs, we run cross-language data flow and control flow reachability analysis queries over them for vulnerability discovery of §3. In this section, we present our results and findings.

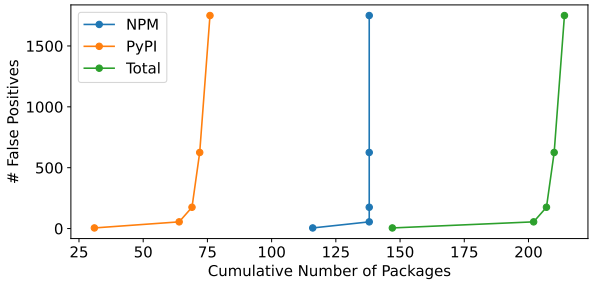
4.2.1 Vulnerable Data Flows In total, CHARON identified 17.2K data flows reaching a sensitive sink in 269 packages, which crossed the language boundary at least once. We observed that the number of data flows in the PyPI ecosystem is $\sim$10 times greater than npm's. However, npm vulnerabilities are more widespread, impacting 173 packages compared to 96 PyPI packages.

When looking at the distribution of data flows over vulnerabilities, the majority of data flows are division by zero, accounting for 4,190 flows across 38 packages in total, which is followed by 3,490 null pointer dereference issues affecting 27 packages. In comparison, the least frequent data flows belong to format string vulnerabilities, accounting for 67 flows in only four packages. Similarly, when looking at the prevalence of issues, the most widespread vulnerability is integer overflow, affecting 131 packages with over 2.5K flows in total, whereas the least widespread issues are format string and use-after-free vulnerabilities, occurring both in only four packages. Table 4 summarizes our findings.

4.2.2 Vulnerability Verification and False Positives Given the substantial volume of data flows detected by our tool, we implemented a semi-automatic method to verify them. First, we used an automated approach to cluster data flows based on their common prefixes, particularly in the sections of native code (backward data flow analysis). This step was crucial because numerous data flows in script code are often linked to a single prefix in native code. If this prefix turns out to be incorrect (a false positive), then all associated data flows are also false positives (i.e., amplification).

Following the automated grouping, we conducted a manual review of the prefix segments to determine their accuracy. If identified as a false positive, we then marked all connected flows accordingly. Conversely, if the prefix was verified as a true positive, we proceeded to manually inspect the connected flows within the scripting program,

Figure 3: Distribution of false positives per cumulative number of packages.



	Path	Flows *PyPI*	Flows *npm*
CASE 1	all(FP) + FP	10,391	895
CASE 2	mix(FP, TP) + TP	152	38
	- FP only	152	17
	- TP only	0	21
CASE 3	all(TP) + TP	5,208	584

TABLE 5: Overview of XL data flow paths and FP amplification.

validating the absence of proper mitigations and the potential impact on security. Our approach identified 2,021 common prefixes in native code, with approximately 66% (1,348 prefixes) ultimately classified as false positives. In total, out of the 17.2K data flows, we marked 5,813 flows across 116 packages as true positive vulnerabilities, observing again a false positive rate of about 66%.

4.2.3 Analysis of False Positives When looking at the underlying reasons for FPs, the side-effects of wrong cross-language edges that were added by CHARON was negligible, accounting for only 0.1% of the FPs. We found that these FP cross-language edges were added for few cases where a native function export was assigned a name identical to an unrelated script function located in a different context, which created naming confusion for CHARON. In stark contrast, the rest of the FPs (i.e., 99.9%) were caused by traditional static analysis issues, such as missing PDG and CFG edges, pointer analysis, array manipulations, and the inability to detect properly mitigated data flows due to path-sensitive propagation of program values and dynamic program features like dynamic buffer allocations, which are similarly applicable for a single-language taint analysis approach. We note that that the capabilities of CHARON's implementation are constrained by the limitations inherent in the underlying (industry-level) static analysis engine we used (i.e., Joern).

Finally, we observed that a significant fraction of FPs are concentrated in only a few packages, where seven packages are causing over 73% of FPs (i.e., 8,451 flows) with four out of these packages having no true positive data flows at all. Figure 3 illustrates this fact, where we can see a sudden spike in FPs in the plot per increase of only few packages. Table 9 shows the distribution of packages across false positive bins.

4.2.4 Analysis of Amplification Effect We observed that the amplifying nature of connecting data flows of two programming languages has a small amplifying impact on the overall number of false positives that CHARON encounters, with only 169 data flows being comprised of both a false positive and true positive part in script and native contexts, respectively, compared to over 10K data flows with false positives in both script and native part. Table 5 presents an overview of different path combinations across npm and PyPI ecosystems.

4.2.5 Accuracy of PPG Cross-Language Edges We also performed comprehensive manual experiments to estimate the accuracy of cross-language edges in the discovered data flows. After reviewing 5,982 cross-language edges included in the PPGs, we found only 17 edges that were a FP, i.e., 0.28%, which is negligible. Since CHARON links cross-language call edges based on the call names defined in the native code access interface, it is susceptible to naming confusion if one or more native methods are overloaded in the script graph. For instance, some Python extensions defined a fallback implementation that is used in case the native code fails to load, causing CHARON to draw additional, incorrect edges to similarly named Python methods.

4.2.6 Comparison with Single-Language Taint Analysis We compared our polyglot code analysis approach, CHARON, with a traditional single-language, native-only taint analysis method based on the Joern engine [35]. Assuming that each value passed through the language boundary is attacker-controlled, the native-only approach generated 3,066 potential vulnerability alerts; however, manual analysis revealed that only **922** of them are true positives, leading to a false positive rate of about 70%. In comparison, CHARON identified ∼6x more true positive vulnerabilities, totaling **5,813**. Furthermore, CHARON achieved a lower false positive rate of 66% (∼4% decrease), which is primarily due to identification of mitigations in high-level code. Overall, we found that polyglot code analysis reduces both false positives and negatives. Table 10 (appendix) summarizes the results.

Although the 5,813 flows reported by CHARON ultimately converge in the same 673 native prefixes, they are distinct attacker-controlled flows, originating from different package exports and following unique paths through the script-level code. Since the native code is not a standalone library, but part of the package itself, sanitization could occur at different points within the package, including the script code. Thus, introducing input validation on one flow does not necessarily mitigate others and each flow needs independent consideration. Consequently, each alert raised by CHARON represents a distinct vulnerability, even if the sink is the same. Moreover, reporting flows individually is important, since dependent applications may use one vulnerable entry point, but not another. Since native-only analysis lacks visibility over the script-level code, it is unable to create a list of vulnerable package entry points. As a result, properly reporting cross-language security issues in native extensions would be unfeasible without substantial manual effort. In contrast, CHARON operates on the entire source code and is thus able to overcome this limitation.

Despite finding more distinct vulnerable flows, the 5,813 alerts issued by CHARON originated from only 673 native prefixes, implying that the native-only analysis discovered 249 additional true positive prefixes. A closer inspection revealed that 123 of these prefixes were

Vulnerability	npm		PyPI		Total	
	Man.	*Exp.*	*Man.*	*Exp.*	*Man.*	*Exp.*
Div by zero	3	1	8	2	11	3
Fmt string	1	-	-	-	1	-
Int. overflow	27	1	2	-	29	1
Memcpy overflow	9	-	5	-	14	-
Mem. leak	17	8	12	1	29	9
Null pointer deref.	3	-	11	-	14	-
Sprintf overflow	-	-	-	-	-	-
Strcpy/strcat overflow	25	20	2	1	27	21
UAF	-	-	-	-	-	-
Total	83	**30**	33	**4**	116	**34**

TABLE 6: Summary of created exploitations. **Legend:** *Man*=Manually confirmed packages, *Exp*= Exploited packages.

Vulnerability	Code Exec	DoS	Inf. Leak	Mem. Corr.
Div. by zero	- ○	3/3 ●	- ○	- ○
Int. overflow	- ◐	1/1 ●	- ○	1/1 ●
Mem. leak	- ○	11/9 ◐	- ○	- ○
Strcpy/strcat overflow	1/1 ◐	23/21 ●	- ○	23/21 ●
Total	1/1	38/34	-	24/22

TABLE 7: Number and impact of exploits created per vulnerability type and package. Each entry shows a value N/M representing N exploits across M packages. Circles represent whether a vulnerability could be exploited for an attack. **Legend:** ●= Applicable; ◐= Partly applicable; ○= Not applicable.

ultimately unreachable to attacker-controlled data in the script domain and thus validly discarded by CHARON. On the other hand, the remaining 126 lost prefixes could receive attacker data and constitute false negatives. Still, CHARON was able to correctly identify and link the cross-language edges for all affected PPGs, indicating that these false negatives are not intrinsic to our cross-language analysis approach.

4.2.7 Contribution of Polyglot Analysis in Script Mitigations One of the key advantages of using a polyglot code analysis approach is its ability to observe how native APIs are called in the scripting program and whether their inputs are properly sanitized, that is, information which a native-only analysis approach does not have access to. This capability allows it to detect mitigations and other input validation measures in script code, significantly reducing the occurrence of false positives in a native-only single-language taint analysis approach. In our experiments, CHARON successfully identified 2,520 explicit mitigations in script code. Additionally, CHARON discovered that 59 alerts were unfounded due to the susceptible function not being used in the script code, and that the attacker lacks control over the input value in 27 cases. In total, CHARON managed to eliminate 2,606 false positives a native-only approach would have reported, emphasizing the potential of polyglot program analysis in preventing false positives.

4.3. Exploitations

Starting from the 116 packages with manually verified data flows, we investigated their exploitability by checking the program behaviour and testing candidate attack payloads. Specifically, for each package, we create a small script calling the function leading to the vulnerable LoC, feeding it with manually-crafted input to trigger the vulnerability. In the majority of the cases, we verified the existence of a vulnerability by causing a hard crash, e.g., due to a division by zero, a null pointer dereference or a memory corruption following a buffer overflow. To detect memory leaks, we repeatedly call the affected sub-routine while monitoring the memory usage of the process. Finally, we assessed the maximal impact of buffer overflow vulnerabilities on native extensions by checking the circumstances under which an attacker can pass shellcode across the language boundary to gain arbitrary code execution.

4.3.1 Results We were able to create 63 proof-of-concept exploitations in 34 packages in total, including popular ones like `bluetooth-serial-port`, `bar-code4nodejs`, and `spinsfast` with far-reaching consequences like code execution, memory corruption, and DoS. For the remaining 82 packages with confirmed data flows, we were unable to create an exploit. However, we note that several packages failed to compile in our test environment due to dependencies on deprecated OS library versions. Furthermore, we did not test exploitability for packages where dependencies were not freely available. In addition, achieving completeness in the manual search for exploits is a challenging task, demanding a profound understanding of each package's semantics and functionality. The fact that we could not find an exploit does not mean that an exploit does not exist. For these cases, we verified that the data flows exists unconditionally, and a determined attacker may still be able to find an exploit. Table 6 summarizes the number of packages successfully exploited for each vulnerability type and Table 7 shows the impact of the generated exploits. In the following section, we present a few case studies of the confirmed attacks.

4.4. Case Studies

In this section, we present a few manually vetted case studies of the confirmed exploitations, which we have responsibly disclosed to the affected vendors.

4.4.1 Node Bluetooth Serial Port Package The *node-bluetooth-serial-port*[2] package is a node module that facilitates communication over Bluetooth serial ports with devices using Node.js. Figure 4 presents an excerpt of a buffer overflow vulnerability we found in this package. In particular, we discovered that an input value coming from the `address` argument of the `findSerialPortChannel()` API, which is defined in `bluetooth-serial-port.js`, is written to a fixed buffer with no length checks as a part of a `strcpy` instruction in C/C++. This led to a buffer overflow vulnerability and we created a PoC exploit causing memory corruption. Furthermore, we were able to escalate this buffer overflow vulnerability to arbitrary code execution (ACC), during which we had to tackle several technical challenges introduced by the language boundary, as discussed next in §4.5.

4.4.2 Leaf-Blade Package *leaf-blade*[3] is an HTML template engine developed in Python. It offers an input sanitizer that escapes the HTML template syntax characters contained in the input through its Parser API.

2. https://www.npmjs.com/package/node-bluetooth-serial-port

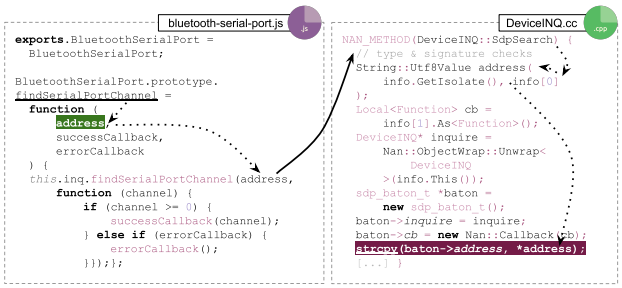


Figure 4: CVE-2023-26109: A buffer overflow vulnerability in `node-bluetooth-serial-port`.

CHARON discovered a memory exhaustion vulnerability in the HTML parser of this package, as illustrated in Figure 5. The root of the matter lies in the `escape_content()` function, which allocates a heap region for file content. However, the package overlooks the crucial step of freeing this region after its operation. The `escape_content()` function in `escape.c` is in turn imported and used in the `Parser` class in Python, leading to a memory exhaustion vulnerability when the `escape` method of the `Parser` class is invoked. Attackers can exploit this vulnerability to obtain a resource-based DoS attack by invoking the affected API many times. The data exchange and communication between the Python and C programs introduces a level of complexity that poses challenges for automatic identification of this vulnerability via traditional taint analysis techniques, requiring polyglot security testing of the package.

4.4.3 MC3 Package The *mc3*[4] package is a Python implementation of the Markov-chain Monte Carlo algorithm. CHARON found a division by zero vulnerability in the implementation logic of *mc3*, where the vulnerable data flow spans across both the Python and C programs. Figure 6 presents an overview of the vulnerability. The function `bin_array()` in Python accepts an integer `binsize` and pass it to the `binarray()` function in C++, which in turn uses this number in two division operations as denominator. However, the program does not validate that `binsize` is not zero, leading to a division by zero vulnerability.

4.5. Impact of Language Boundary on Exploitability

In our work, we explored whether or not the operations at the language boundary hinder exploitation of vulnerabilities (Cf. 7). For example, we assessed the feasibility of an arbitrary code execution attack on a minimalistic clone of the *node-bluetooth-serial-port* package. To ease exploitation in this case, we disabled Address Space Layout Randomization (ASLR) on our VM and compile our sample with an executable stack and no stack protector.

As a result, we successfully created an ACC exploit starting from the buffer overflow vulnerability. However, we observed that the language boundary in polyglot Node.js packages introduces additional challenges for exploitability. This heightened difficulty arises primarily due to the UTF-8 encoding of Node.js strings, which adheres to strict byte value sequences. While benign inputs pose no issues, an attacker's shellcode often faces rejection or distortion because the instruction opcodes tend to create invalid sequences. These sequences may manifest within the opcode of a single instruction or emerge during the linking process when chaining two instructions. In the former scenario, attackers can circumvent the issue by substituting instructions with invalid sequences for semantically equivalent ones [47]. In the latter case, they have the option to inject additional instructions between existing ones, frequently utilizing the NOP instruction, as its opcode is a common valid follow-up byte in sequences [47]. However, the chaining of NOP instructions is constrained by the preceding byte's value, requiring attackers to regularly insert alternative instructions into their NOP sled.

The final hurdle in the attack involves overwriting the return address with a value that not only directs to the shellcode or NOP sled but is also valid UTF-8. This proves to be the most challenging aspect, as multiple constraints come into play. Firstly, certain byte values are nearly entirely restricted in UTF-8. One such restricted value is 0xFF, which is particularly common in addresses. 0xFF may only appear at the beginning of a UTF-8 string and cannot be used when overwriting the return address. Attempting to bypass this limitation, an attacker may position the shellcode with a lengthy NOP sled after the return address position and choose a valid address with the smallest possible offset as the overwrite value. However, depending on the address and architecture, the smallest valid offset might be so extensive that the Node.js interpreter rejects the attack payload for exceeding string length bounds. This mirrors challenges faced in other attack techniques, such as return-oriented programming and return to libc [48, 49], which are subject to the same set of restrictions.

4.6. Vulnerability Notification

We have notified the affected parties following the best vulnerability notification practices [50]. For each affected package, we created a detailed vulnerability report along with one or more PoC exploitations. In the majority of the cases, we used Snyk [51] as the point of contact to help us with the disclosure process. In few cases, we contacted the developers directly when that was not possible through Snyk (e.g., for less popular packages). We also considered reaching out to the package maintainers using alternative channels, particularly Github's confidential vulnerability disclosure system, but found that the feature is disabled in most of the affected repositories.

At the time of writing this paper, 13 out of the 34 packages confirmed our reports, with multiple CVEs already assigned for the patched packages. In contrast, 21 email notifications currently remain unanswered. In addition to issuing reports for packages with PoC exploits, we also notified the authors of the remaining projects with insecure data flows. We are currently awaiting a response. We observed that several of these packages appear to be poorly maintained, as their version and commit histories show no recent activity.

3. https://pypi.org/project/leaf-blade
4. https://pypi.org/project/mc3

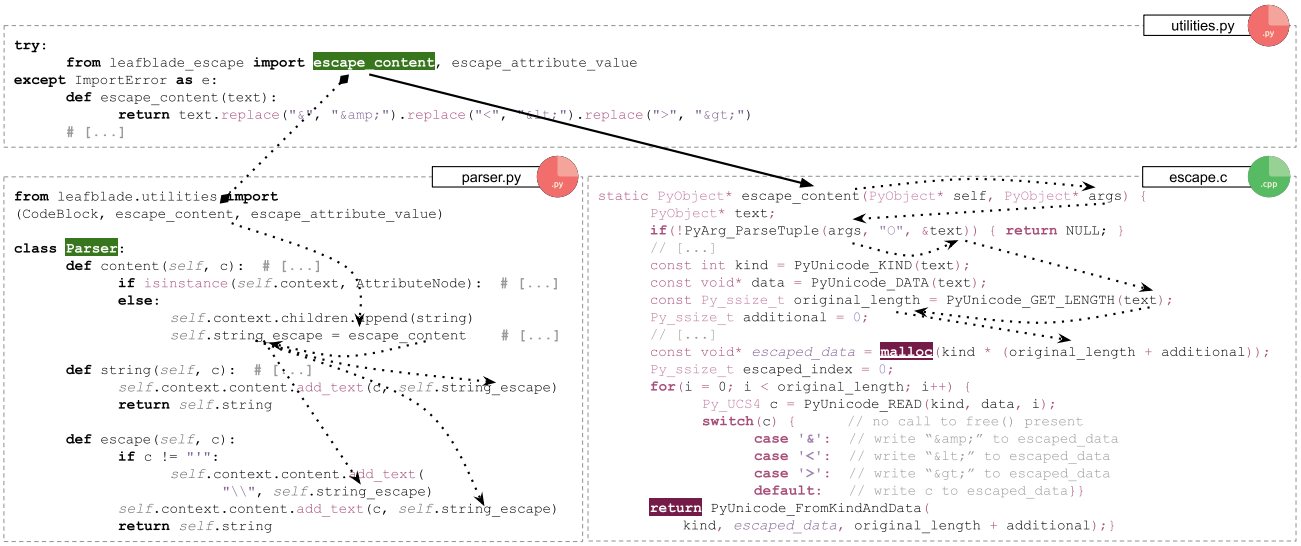


Figure 5: Excerpt of a memory exhaustion vulnerability in the parser of `leaf-blade` HTML template engine. Arrows represent data flows between the variables.

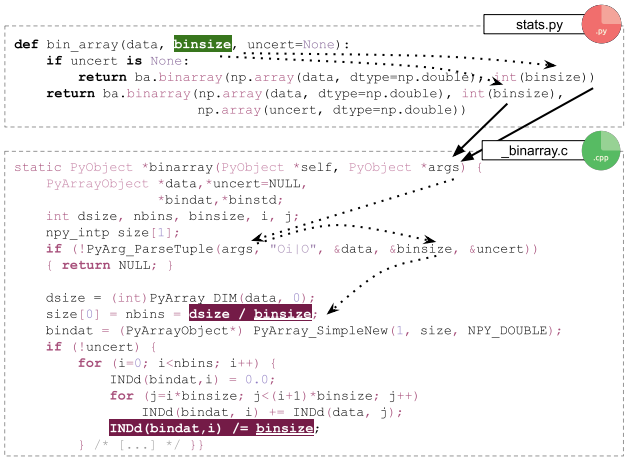


Figure 6: A division by zero in the binarray logic of `mc3`.

5. Discussion

In this section, we put our results into perspective, we discuss limitations of CHARON and future work opportunities.

Support for Stateful Native Extensions. Our work shows that a multi-language static analysis approach is essential for security testing of scripting languages native extensions. First, a polyglot approach enables the detection of complex vulnerabilities that are due to the interplay of cross-language API calls and back-and-forth context switches between native and scripting programs, as exemplified in Figure 1. We observed that such polyglot approach can significantly enhance the identification of vulnerabilities, yielding over six times more true positive detections compared to single-language analyses, which fail to enumerate the vulnerable entry points of the package. Second, our polyglot approach can detect mitigations in high-level code (e.g., input validation) and thereby reduce the false positives by about 4% compared to a single-language taint analysis method (Cf. §4.2.6).

Advanced Program Analysis Features are Essential. We observed that a flow-sensitive and inter-procedural static analysis is necessary for vulnerability detection in scripting languages native extensions (see, e.g., Figure 5), particularly for complex vulnerabilities that require more than one data flow analysis tasks, such as use-after-free, null pointer dereference, and memory leak issues (Cf. §3.3.3). As such, simpler static analyses like the one proposed by Staicu et al. [10] only suffice for bugs directly related to the cross-language conversion, such as type mismatching, and more advanced program analysis features are necessary to detect the vulnerabilities outlined in §2.

Extensibility to Other Programming Languages. In this paper, we evaluated CHARON for C/C++/JavaScript/Python. While our evaluation may be specific to these languages, we designed our methodology with generalizability in mind. In particular, CHARON uses the Joern engine which provides the same syntax tree specification for the CPGs of different programming languages (see, i.e., [37]), making our implementation likely adaptable to different language combinations using build time variability. However, we do not claim generalizability in this work.

Beyond Native Extensions. Cross-language code can appear in many other contexts in scripting languages, beyond native extensions. For example, most runtimes have support for WebAssembly, offering a convenient way to run efficient and portable low-level code. Extending CHARON to WebAssembly would require a Joern frontend for this language, but otherwise would be straightforward to implement, since this type of code is invoked in a similar way to native extensions. On the contrary, runtimes are often written in low-overhead programming languages such as Rust or C++, and glued with scripting code via bindings. While a lot of operations or API calls in scripting code are eventually resolved to these low-level code locations, extending CHARON to support engine code is far from trivial, since the description of the mapping between high-level and low-level is buried deep into the runtime's specifications or documentation, often

expressed in natural language only.

Dynamic analysis to improve Static. Dynamic analysis could help filtering out false positives, but it requires package installation and running automatically. However, we observed that many in-the-wild packages fail to install automatically due to various compilation errors and mismatched Node.js versions, hindering larger-scale analyses.

Limitations. The methodology illustrated by Algorithm 2 ensures a comprehensive approach to recognizing and analyzing cross-language method invocations within native extensions, as it considers any potential relationship between all the call expressions and function definition nodes. However, we note that, similarly to other state-of-the-art static analysis approaches [5, 9, 34, 36, 52], CHARON does not completely handle dynamic function calls and dynamic function definitions in the script code, and is bound by the limitations and soundness properties of the underlying Joern static analysis engine. Finally, the cross-language linking algorithm of CHARON can be confused into creating wrong cross-language edges in specific configurations, although such cases only occurred seldomly in our studied applications.

Open Science. We publicly release CHARON upon publication to benefit future research efforts[5].

6. Related Work

In this section we discuss related work on low-level security bugs in scripting languages and techniques for containing or detecting such problems automatically.

Low-level bugs in scripting languages. For performance reasons, scripting language runtimes are often implemented in C/C++ and thus, may leak low-level bugs into the scripting environment. There are various static [7] or dynamic techniques [53] for identifying such bugs automatically. Gross et al. [54] notice, however, that such approaches must also take into consideration just-in-time compilation, a language feature often used in scripting languages. Of special interest for security analysis is the binding layer that acts as the boundary between scripting and low-level code. Brown et al. [8] were the first to describe such security issues and propose simple static analyses for detecting them. Dinh et al. [55] proposes a fuzzing technique specially designed for testing this type code. More recently, Staicu et al. [10] identify native extensions as another source of low-level bugs in scripting languages. They propose a lightweight analysis technique for JavaScript that analyzes data flows in the two functions closest to the language boundary. To the best of our knowledge, ours is the first systematic work that analyzes low-level bugs in their surrounding context, using inter-procedural analysis that can track values across long dependency chains, possibly crossing the language boundary multiple times.

Cross-language program analysis. Li et al. [56] find that multi-language code is more likely to contain security vulnerabilities. Thus, prior work proposed several cross-language analyses for various pairs of languages: Java-C [57–60], Java-JavaScript[61–64], Python-C [65], Go-C [66], or PHP-JavaScript [67]. There is also work on designing generic cross-language solutions starting from existing analysis frameworks. Arzt [68] proposes extending Soot by building new analyses front-ends, Youn [52] discuss how CodeQL can be enhanced in a similar way, and Kreindl [69] builds dynamic taint analyses in a polyglot virtual machine. To the best of our knowledge, none of this work supports the holistic analysis of cross-language JavaScript or Python code combined with C/++ extensions. Staicu et al. [10] were the first to make a step in this direction for JavaScript, but their analysis does not support multiple crossings of the language boundary, and it is flow-insensitive and limited to data flow analysis. Similarly, Monat et al. [11] were the first to attempt a principled solution for Python, but their heavy formalism makes the approach impractical for most real-world use cases. In comparison, CHARON is flow-sensitive, inter-procedural, supports both JavaScript and Python, and offers various analyses including data-flow analysis, reachability analysis, pointer analysis, and identification of mitigation nodes. Houdaille et al. [70] proposed a method to generate a uniform, language-agnostic AST for code of multiple languages, but they focused on syntactic code analysis rather than tracking complex control and data flows as CHARON does. Li et. al. [19] proposed PolyCruise, a dynamic information flow analysis system for C-Python programs. Similarly to PolyCruise, PolyFuzz [28] is a dynamic fuzzer that tests multi-language programs in C, Python, and Java using cross-language coverage feedback, modeling the relationships between program inputs and conditional predicates. Compared to CHARON, PolyCruise and PolyFuzz use dynamic analysis, which tends to have high FNs and fewer FPs. In addition, these works evaluate a relatively small sample of packages when compared to our work, i.e., PolyFuzz 30 and PolyCruise 12 packages.

Isolating unsafe code in scripting runtimes. An important strategy for reducing the damage caused by the exploitation of low-level vulnerabilities is to contain their effect. Abbadini et al. advocate for sandboxing native extensions in Node.js [71] or Deno [72], using existing Linux security modules like Landlock and eBPF. Moreover, Wyss et al. [73] and Wang et al. [74] propose system call filtering for preventing unprivileged operations in scripting language. Narayan et al. [75] design a sophisticated solution at C++ level that integrates static information flow analysis with lightweight dynamic checks to isolate buggy parts of the runtime. Vasilakis et al. advocate both for process-level isolation [76] and lighter language-based techniques [77] for compartmentalizing scripting language code. However, AlHamdan and Staicu [78] show that language-based enforcement at scripting level is often easy to bypass. Scripting languages often allow integration with WebAssembly, which might also execute vulnerable low-level code. While WebAssembly runs in a highly-isolated environment, Lehmann et al. [79] note that low-level bugs in WebAssembly might still lead to security consequences. Thus, Michael [80] proposed a novel memory-safety property that inform compiler-level enforcement mechanisms that in turn prevent runtime exploitation. Johnson et al. [81] highlight the importance of verifying the interactions of such code with the outside world. While in our work we focus on detecting low-level

5. https://github.com/VainlyStrain/charon

bugs instead of preventing their exploitation, we note the importance of deploying multiple layers of defense against buggy low-level code in scripting languages.

Program analysis for security in scripting languages. In recent years, there were many sophisticated program analyses proposed for scripting languages. Backes et al. [5] were the first to show that code property graphs can be used for statically detecting vulnerabilities in PHP. Subsequently, Pellegrino et al. [6] propose extracting such property graphs from dynamic traces and expanding this data structure across multiple application's tiers to incorporate causality and state transitions. Similarly, Khodayari and Pellegrino [3] propose hybrid property graphs, a program representation that aggregates information from both static and dynamic program analysis. Li et al. [1, 2] propose object dependence graph, a data structure that allows static analyses to reason about dynamically added properties and prototype inheritance in JavaScript. There is also great interest in the industry to build static analyses for scripting languages, with prominent examples being Facebook's Pysa and GitHub's CodeQL. Prior academic work proposes enhancing such tools with dynamic taint summaries [82] or with machine learning models that leverage semantic information contained in the code [83]. CodeQL is also a prominent example of analysis that can handle code written in different languages, albeit one language at a time. Similarly, while Yamaguchi et al. [9] first proposed code property graph for static analysis of C/C++, their prototype Joern grew into a mature industrial product that also supports languages like JavaScript or Python. To the best of our knowledge, none of these work performs cross-language static analysis and we are the first to propose a polyglot approach that leverages code property graphs.

7. Conclusion

In this paper, we presented CHARON, a polyglot code analysis framework to detect vulnerabilities in scripting languages native extensions. Our approach can track the propagation of program properties like control and data flows across different programming languages. We instantiated CHARON over the npm and PyPI ecosystems, processing over 673M lines of code and creating PPGs for 11.2K polyglot packages. CHARON detected 5,813 confirmed data flows across 116 packages, out of which we created 68 PoC exploitations across 34 packages, including popular ones like `bluetooth-serial-port` and `barcode4nodejs`, enabling attackers to achieve arbitrary code execution, memory corruption, denial of service attacks or sensitive information leakage. Our empirical evaluation suggests that CHARON is practical and can be used for security testing of polyglot packages at scale.

References

[1] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[2] ——, "Mining Node.js vulnerabilities via object dependence graph and query," in *USENIX Security Symposium*, 2022.

[3] S. Khodayari and G. Pellegrino, "JAW: studying client-side CSRF with hybrid property graphs and declarative traversals," in *USENIX Security Symposium*, 2021.

[4] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Network and Distributed System Security Symposium, (NDSS)*, 2021.

[5] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities," in *IEEE European Symposium on Security and Privacy*, 2017.

[6] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[7] F. Brown, D. Stefan, and D. R. Engler, "Sys: A static/symbolic tool for finding good bugs in good (browser) code," in *USENIX Security Symposium*, 2020.

[8] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in JavaScript bindings," in *Symposium on Security and Privacy (S&P)*, 2017.

[9] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *IEEE S&P Symposium*, 2014.

[10] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes, "Bilingual problems: Studying the security risks incurred by native extensions in scripting languages," in *USENIX Security Symposium*, 2023.

[11] R. Monat, A. Ouadjaout, and A. Miné, "A multilanguage static analysis of Python programs with native C extensions," in *International Symposium on Static Analysis (SAS)*, 2021.

[12] Extending python with C or C++. [Online]. Available: https://docs.python.org/3/extending/extending.html

[13] PyMethodDef API. [Online]. Available: https://docs.python.org/3/c-api/structures.html#c.PyMethodDef

[14] Calling Python Functions from C. [Online]. Available: https://docs.python.org/3/extendinextending.html#calling-python-functions-from-c

[15] C++ addons for node.js. [Online]. Available: https://nodejs.org/api/addons.html

[16] JavaScript-accessible methods. [Online]. Available: https://github.com/nodejs/nan/blob/main/doc/methods.md

[17] C/C++ addons - N-API. [Online]. Available: https://nodejs.org/dist/latest-v11.x/docs/apn-api.html#n_api_napi_call_function

[18] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the Annual Computer Security Applications Conference*, 2012.

[19] W. Li, J. Ming, X. Luo, and H. Cai, "{PolyCruise}: A {Cross-Language} dynamic information flow analysis," in *31st USENIX Security Symposium*, 2022.

[20] H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan, "A study of c/c++ code weaknesses on stack overflow," *IEEE Transactions on Software Engineering*, pp. 2359–2375, 2022.

[21] MITRE CWE-121: Stack-based Buffer Overflow. [Online]. Available: https://cwe.mitre.org/data/definitions/121.html

[22] MITRE CWE-369: Divide By Zero. [Online]. Available: https://cwe.mitre.org/data/definitions/369.html

[23] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, 2022.

[24] M. F. Ringenburg and D. Grossman, "Preventing format-string attacks via automatic and efficient dynamic checking," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, p. 354–363.

[25] Format string vulnerabilities. [Online]. Available: https://book.hacktricks.xyz/binary-exploitation/format-strings

[26] F. Kilic, T. Kittel, and C. Eckert, "Blind format string attacks," in *International Conference on Security and Privacy in Communication Networks*, 2015, pp. 301–314".

[27] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 760–770.

[28] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "{PolyFuzz}: Holistic greybox fuzzing of {Multi-Language} systems," in *32nd USENIX Security Symposium*, 2023.

[29] S. Ma, M. Jiao, S. Zhang, W. Zhao, and D. W. Wang, "Practical null pointer dereference detection via value-dependence analysis," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2015.

[30] Cwe-416: Use after free. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html

[31] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, p. 999–1010.

[32] MITRE Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/

[33] Code Property Graph: specification, query language, and utilities. [Online]. Available: https://github.com/ShiftLeftSecurity/codepropertygraph

[34] S. Khodayari and G. Pellegrino, "JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals," in *USENIX Security Symposium*, 2021.

[35] Joern engine. [Online]. Available: https://github.com/joernio/joern

[36] S. Khodayari and G. Pellegrino, "It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses," in *IEEE S&P Symposium*, 2023.

[37] Code property graph syntax tree specification for Joern. [Online]. Available: https://cpg.joern.io/

[38] Querying the Joern Database. [Online]. Available: https://joern.readthedocs.io/en/latest/querying.html

[39] The Anatomy of a Joern Query. [Online]. Available: https://docs.joern.io/traversal-basics/

[40] Gremlin Graph Traversal Language. [Online]. Available: https://github.com/tinkerpop/gremlin/wiki

[41] Joern c2cpg library. [Online]. Available: https://github.com/joernio/joern/tree/c2132ede7c199853c33764de567059511bff6353/joern-cli/frontends/c2cpg

[42] Joern jssrc2cpg library. [Online]. Available: https://github.com/joernio/joern/tree/c2132ede7c199853c33764de567059511bff6353/joern-cli/frontends/jssrc2cpg

[43] Joern pysrc2cpg library. [Online]. Available: https://github.com/joernio/joern/tree/c2132ede7c199853c33764de567059511bff6353/joern-cli/frontends/pysrc2cpg

[44] ShiftLeft Security. [Online]. Available: https://github.com/ShiftLeftSecurity

[45] Joern fixes for the OSS data flow engine. [Online]. Available: https://github.com/joernio/joern/pull/1376

[46] F. Al Kassar, G. Clerici, L. Compagna, F. Yamaguchi, and D. Balzarotti, "Testability tarpits: the impact of code patterns on the security testing of web applications," in *NDSS*, 2022.

[47] Writing UTF-8 Compatible Shellcode. [Online]. Available: http://phrack.org/issues/62/9.html

[48] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM conference on Computer and communications security*, 2007.

[49] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Annual Computer Security Applications Conference*. IEEE, 2009.

[50] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow, "Didn't you hear me?-towards more successful web vulnerability notifications." in *NDSS*, 2018.

[51] Snyk. [Online]. Available: https://snyk.io/

[52] D. Youn, S. Lee, and S. Ryu, "Declarative static analysis for multilingual programs using codeql," *Softw. Pract. Exp.*, vol. 53, no. 7, pp. 1472–1495, 2023.

[53] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines," in *NDSS*, 2019.

[54] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "FUZZILLI: fuzzing for javascript JIT compiler vulnerabilities," in *NDSS*. The Internet Society, 2023.

[55] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases," in *NDSS*, 2021.

[56] W. Li, L. Li, and H. Cai, "On the vulnerability proneness of multilingual code," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (ESEC/FSE)*, 2022.

[57] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis," in *International Conference on Automated Software Engineering, (ASE)*, 2020.

[58] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *USENIX Security Symposium*, 2008.

[59] G. Tan and G. Morrisett, "Ilea: Inter-language analysis across Java and C," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[60] S. Almanee, A. Unal, and M. Payer, "Too quiet in the library: An empirical study of security updates in Android apps' native code," 2021.

[61] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps - A report on the state of Apache Cordova Nation," in *Engineering Secure Software and Systems (ESSoS)*, 2016.

[62] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, "Bridgetaint: A bi-directional dynamic taint tracking method for JavaScript bridges in Android hybrid applications," *IEEE Trans. Inf. Forensics Secur.*, 2019.

[63] S. Bae, S. Lee, and S. Ryu, "Towards understanding and reasoning about Android interoperations," in *International Conference on Software Engineering (ICSE)*, 2019.

[64] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: Static analysis framework for Android hybrid applications," in *International Conference on Automated Software Engineering (ASE)*, 2016.

[65] M. Hu and Y. Zhang, "The Python/C API: evolution, usage statistics, and bug patterns," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.

[66] A. Sorniotti, M. Weissbacher, and A. Kurmus, "Go or no go: Differential fuzzing of native and C libraries," in *IEEE Security and Privacy Workshops - Workshop On Offensive Technologies (WOOT)*, 2023.

[67] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[68] S. Arzt, T. Kussmaul, and E. Bodden, "Towards cross-platform cross-language analysis with soot," in *International Workshop on State Of the Art in Program Analysis, (SOAP@PLDI)*, 2016.

[69] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseder, and H. Mössenböck, "Multi-language dynamic taint analysis in a polyglot virtual machine," in *International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2020.

[70] P. Houdaille, D. E. Khelladi, R. Briend, R. Jongeling, and B. Combemale, "Polyglot ast: Towards enabling polyglot code analysis," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2023.

[71] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "Natisand: Native code sandboxing for javascript runtimes," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023.

[72] ——, "Cage4deno: A fine-grained sandbox for deno subprocesses," in *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2023.

[73] E. Wyss, A. Wittman, D. Davidson, and L. D. Carli, "Wolf at the door: Preventing install-time attacks in npm with latch," in *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2022.

[74] W. Wang, X. Lin, J. Wang, W. Gao, D. Gu, W. Lv, and J. Wang, "HODOR: shrinking attack surface on node.js via system call limitation," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[75] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *USENIX Security Symposium*, S. Capkun and F. Roesner, Eds., 2020.

[76] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *NDSS*, 2018.

[77] N. Vasilakis, C. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on node.js via rwx-based privilege reduction," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[78] A. AlHamdan and C. Staicu, "Sanddriller: A fully-automated approach for testing language-based javascript sandboxes," in *USENIX Security Symposium*, 2023.

[79] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *USENIX Security Symposium*, 2020.

[80] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan, "Mswasm: Soundly enforcing memory-safe execution of unsafe code," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 425–454, 2023.

[81] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "Wave: a verifiably secure webassembly sandboxing runtime," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.
[82] C. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *International Conference on Software Engineering (ICSE)*, 2020.
[83] Y. W. Chow, M. Schäfer, and M. Pradel, "Beware of the unexpected: Bimodal taint analysis," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

Appendix

1. Additional Evaluation Details

#	API	Node	Pyth.
1	NODE_SET_METHOD()	●	
2	Nan::SetMethod()	●	
3	Nan::SetPrototypeMethod()	●	
4	<exports>.Set(<Napi patterns>)	●	
5	<exports>.Set(<V8 patterns>)	●	
6	PyObject_Call()		●
7	PyObject_CallObject()		●
8	PyObject_CallFunction()		●
9	PyObject_CallMethod()		●
10	PyMethodDef()		●
11	PyObject<name>()		●

TABLE 8: Overview of function expose APIs supported by CHARON.

Bin	NPM	PyPI	Total
$1 \leq$ FP $<$ 10	116	31	147
$10 \leq$ FP $<$ 100	22	33	55
$100 \leq$ FP $<$ 250	0	5	5
$250 \leq$ FP $<$ 1000	0	3	3
$1000 \leq$ FP $<$ 2500	0	4	4
FP $\geq$ 2500	0	0	0

TABLE 9: Distribution of packages across false positive bins.

	Cross-Language									Single-Language		
	PyPi	*N*	*S*	*L*	**npm**	*N*	*S*	*L*	**Total**	*PyPi*	*npm*	*Total*
TPs	5,208	-	-	-	605	-	-	-	5,813	271	651	922
FPs	10,543	10,391	151	1	912	895	1	16	11,455	1,039	1,105	2,144
Total	15,751				1,517				17,268			3,066

TABLE 10: Comparison of CHARON with single-language taint analysis. Legend (FP introduction point): *N*=native code, *S*=script code, *L*=language boundary.

2. Artifact Availability

We will publicly release CHARON upon publication under an open-source license. The source code is accessible here: `https://github.com/VainlyStrain/charon`.

For ethical reasons, we will not publicly release our dataset of vulnerable NPM and PyPI packages. Since not all affected parties may patch the vulnerabilities, sharing these details could pose risks. Our paper does not currently include any specific information of the vulnerable packages, except when they have been already patched, and reports only aggregated results. The anonymity set is large enough to minimize the risk of someone trying to find the vulnerable packages using the results in our paper. Furthermore, CHARON identified a large number of sensitive data flows (i.e., 5,813), for a significant fraction of which we were unable to create exploits manually. However, this does not mean that an exploit does not exist, and determined attackers may still find a way to exploit these behaviours (Cf. §4.3). Therefore, instead of releasing the dataset and in the spirit of open science, we will set up an online form where interested researchers can apply and describe their need. We will vet the provided information and grant or deny access to our dataset.