Identifying Parser Functions in Binary Executables with WebParserHunter

Marco Scapin[0009−0009−4189−1181]✉, Fabio Pinelli[0000−0003−1058−6917], and Letterio Galletta[0000−0003−0351−9169]

IMT School for Advanced Studies Lucca, Lucca, Italy
{marco.scapin,fabio.pinelli,letterio.galletta}@imtlucca.it

Abstract. We present WebParserHunter, an interactive web application for automatically identifying parsing and validation functions in x86-32 and x86-64 binary executables without source code. This demo is the companion of the ParserHunter methodology [4] that relies on the SAFE embedding model and Graph Sage GNNs for code classification: analysts can upload any binary executable to the application and receive a ranked list with predicted probabilities of it being a parser, along with CFG visualizations and inline assembly. They can then launch an `angr`-based symbolic execution or a fuzzing campaign for deeper auditing within the same browser session.

Keywords: Binary analysis · Parser identification · Graph Neural Networks · Security auditing · Symbolic execution

1 Introduction

Security analysts frequently audit binaries without access to their source code. Among the many functions in a compiled executable, *parsing and validation functions* are prime targets of security analyses. Indeed, these functions read unstructured byte streams, implement complex state-machine logic, and build data structures from raw input, making them hotspots for memory corruption and integer errors. Due to their complexity, parsing and validation functions are highly susceptible to bugs: weaknesses in input validation are one of the leading causes of software security vulnerabilities, according to Mitre's Top 25 Most Dangerous Software Weaknesses list [1]. Yet locating parser functions manually in a binary with hundreds of functions is tedious and time-consuming.

To address this gap, we recently introduced the ParserHunter methodology [4], which relies on Graph Neural Networks and embedding models to identify parsing functions. More precisely, we analyze each binary by identifying its functions, extracting their Control Flow Graphs (CFGs), and enriching these graphs with features derived from the SAFE embedding model [3] that captures both structural and the semantic aspects of their behavior. These annotated CFGs serve as input to a Graph SAGE GNN [2] trained to carry out the classification task.

In this paper, we present WebParserHunter, an interactive frontend of the ParserHunter methodology. We describe how a security analyst can upload a

binary to the application, initiate the analysis, and obtain—within a relatively short time, depending on the binary's size—a ranked table of parser candidates with associated probabilities. The results are accompanied by LLM-based explanations of the classification, as well as CFG visualizations and assembly code views. In addition, the analyst can select a function from the same interface and generate a Python script to perform symbolic execution using the `angr` tool [5] or run a fuzzing campaign on the selected function.

Below, we describe the architecture of WebParserHunter and present a walkthrough of its usage. Code, models, datasets, and a demo video are online [7].

2 The WebParserHunter System

WebParserHunter implements the ParserHunter methodology of [4], providing analysts with access to a pre-trained model for parser identification. Fig. 1 shows the identification pipeline implemented by the tool. Below, we recap our methodology and then walk through the demo.

2.1 ParserHunter Methodology

The methodology defines a function as a parser when it meets six formal criteria, called **IIDDOC**: *Input Handling*, *Internal State*, *Decision-Making*, *Data Structure Creation*, *Outcome*, and *Compositionality*. These criteria guide dataset labelling and make predictions cross-checkable against the function's CFG and assembly. The pre-trained model that is exposed to analysts is obtained by an offline pipeline that relies on Radare2 [6] to enumerate functions inside a binary; `angr` to reconstruct CFGs; SAFE to provide node embeddings, producing Attribute CFGs (ACFGs); GraphSAGE for the classification. Our training dataset is derived from ten open-source C libraries (XML, JSON, HTTP, PCAP, ELF) compiled with GCC and Clang at six optimisation levels for x86-32/64 architectures, yielding 262 binaries and 7,898 functions (1,977 parsers). The GraphSAGE classifier was trained via leave-one-program-out cross-validation, achieving a mean recall of **0.92** (std. 0.09), stable at 0.90–0.96 across all compiler configurations.

2.2 Web Application and Demo Walkthrough

The back-end of WebParserHunter is written in Python/Flask and orchestrates the full analysis pipeline accessible via the browser. An async task queue keeps the interface responsive while a progress bar tracks each phase in real time. Below, we show how an analyst can analyze the binary of `Jsmn`[1], a library with several functions to parse JSON files.

Step 1 – Upload and Analysis. The analyst uploads a binary onto the upload page. The tool supports two CPU architectures (x86-32 and x86-64), two compilers

[1] `https://github.com/zserge/jsmn/tree/master`

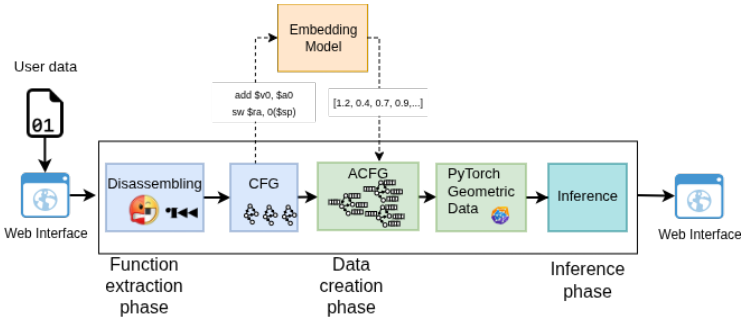

Fig. 1. The WebParserHunter system architecture and pipeline

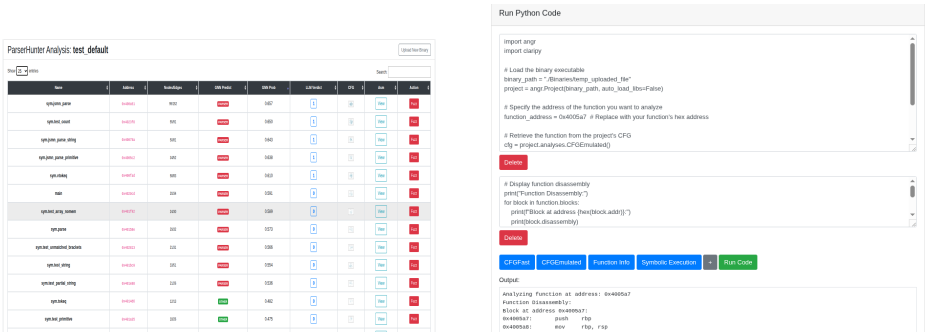

Fig. 2. *Left:* Results table for `Jsmn`: per-function parser probabilities, CFG statistics, and expandable inline assembly. *Right:* Symbolic execution panel with a pre-filled `angr` script.

(GCC and Clang), and multiple optimization levels. Behind the scenes, the back-end launches the multi-stage pipeline Fig. 1: *(i) Function discovery*—Radare2 scans the binary and enumerates all functions with their entry-point addresses; *(ii) CFG reconstruction*—`angr` builds the CFG of each function, identifying basic blocks and edges; *(iii) Semantic embedding*—SAFE encodes each basic block's assembly code into a 100-dimensional numeric vector, producing an Attribute CFG (ACFG) per function that captures both structural and semantic content; *(iv) Inference*—the pre-trained GraphSAGE model classifies each ACFG and outputs a parser probability.

Step 2 – Results and Inspection. Fig. 2 (left) shows the results page: a sortable table where each row lists the function name, address, CFG statistics (nodes and edges), predicted label (using a standard threshold of 0.5 on the predicted probability), parser probability, and an LLM-based prediction that follows the IIDDOC criteria. Also, clicking on the corresponding column of any row expands it to reveal three interactive components: (i) a visualizable CFG rendering, (ii) the full disassembled assembly code of the function, and (iii) a *Fuzzing* button to configure and launch the fuzzing procedure. The analyst can

sort the table by probability to focus on the top parser candidates. Additionally, clicking the LLM button opens a pop-up with a detailed textual explanation justifying the classification.

Step 3 – Function Analysis. For deeper auditing, the analyst copies the address of a candidate function and launches a symbolic execution or fuzzing campaign directly from the browser. As shown in Fig. 2 (right), the system generates a Python script pre-filled with the standard `angr` setup for symbolic execution. Clicking "Run Code" executes the script in a sandboxed environment and streams the symbolic execution output back to the browser, allowing the analyst to move from classification to program-level analysis without leaving the interface. Alternatively, the analyst can click the "Fuzz" button in Fig. 2 (left) to configure the values of the standard input registers representing the function's inputs, and launch the fuzzing procedure with those specified values.

A walkthrough video is available at [7].

3 Conclusions

WebParserHunter enables accessible parser identification in binaries via a browser interface that combines GNN classification, CFG visualization, and basic fuzzing and symbolic execution, helping software analysis and audit. Future work includes extending support to ARM and RISC-V architectures, handling stripped and obfuscated binaries, and incorporating semi-supervised learning for label propagation.

References

1. Corporation, M.: Cwe top 25 most dangerous software weaknesses (2024), `https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html`
2. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects (2019)
3. Massarelli, L., Luna, G., Petroni, F., Querzoni, L.: Safe: Self-attentive function embeddings for binary similarity (11 2018)
4. Scapin, M., Pinelli, F., Galletta, L.: ParserHunter: Identify Parsing Functions in Binary Code. Journal of Systems and Software (2026)
5. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
6. Team, R.: Radare2 github repository. `https://github.com/radare/radare2` (2017)
7. ParserHunter: Identify Parsing Functions in Binary Code (Supplementary Material) (2025), `https://sites.google.com/view/parserhunter-paper/`