BINARY DIFF SUMMARIZATION USING LARGE LANGUAGE MODELS

Anonymous authorsPaper under double-blind review

000

001

002003004

006

008

010 011

012

013

014

015

016

017

018

019

021

025

026

027

028

029

031

032033034

035

037

040

041

042

043

044

046

047

048

051

052

ABSTRACT

Security of software supply chains is necessary to ensure that software updates do not contain maliciously injected code or introduce vulnerabilities that may compromise the integrity of critical infrastructure. Verifying the integrity of software updates involves binary differential analysis (binary diffing) to highlight the changes between two binary versions by incorporating binary analysis and reverse engineering. Large language models (LLMs) have been applied to binary analysis to augment traditional tools by producing natural language summaries that cybersecurity experts can grasp for further analysis. Combining LLM-based binary code summarization with binary diffing can improve the LLM's focus on critical changes and enable complex tasks such as automated malware detection. To address this, we propose a novel framework for binary diff summarization using LLMs. We introduce a novel functional sensitivity score (FSS) that helps with automated triage of sensitive binary functions for downstream detection tasks. We create a software supply chain security benchmark by injecting 3 different malware into 6 open-source projects which generates 104 binary versions, 392 binary diffs, and 46,023 functions. On this, our framework achieves a precision of 0.98 and recall of 0.64 for malware detection, displaying high accuracy with low false positives. Across malicious and benign functions, we achieve FSS separation of 3.0 points, confirming that FSS categorization can classify sensitive functions. We conduct a case study on the real-world XZ utils supply chain attack; our framework correctly detects the injected backdoor functions with high FSS.

1 Introduction

Binary analysis is fundamental to cybersecurity, enabling critical tasks like vulnerability discovery, malware analysis, and software supply chain integrity. Reverse engineering low-level binaries to extract high-level functionalities is essential for understanding their behavior without access to source code (Cifuentes, 1994; Schulte et al., 2018; Shoshitaishvili et al., 2016). Binary differential analysis (binary diffing) extends binary analysis by comparing two versions of a binary to understand what has changed, allowing analysts to focus on the modifications to find newly introduced bugs or security flaws. This approach is especially relevant for software supply chain security that involves verifying the integrity of software updates (Reichert & Obelheiro, 2024). Binary diffing supports critical security tasks like identifying patched vulnerabilities (Brumley et al., 2006), clustering malware variants (Royal et al., 2006), detecting vulnerabilities in binary distributions (Zhao et al., 2022). Malicious actors may inject hidden code into a program binary or into the source code of dependent open-source libraries, leading to devastating downstream impact as exemplified by recent incidents such as 3CX (FortiGuard Labs, 2023), SolarWinds, log4j, and XZ utils (Williams et al., 2025). These concerns are magnified in the context of the embedded systems supply chain. Unlike enterprise software, embedded devices are deployed in remote or inaccessible locations. Software updates require significant effort. Due to this, corrupted or compromised updates may persist for extended periods. Embedded firmware is distributed as monolithic binaries which incorporate several projects into one blob, making verification difficult (Shirani et al., 2017). This necessitates initial verification of software integrity before deployment via binary analysis and reverse engineering.

Reverse engineering is an inexact process, hence binary analysis tools often recover source code in an obscure format, requiring significant effort and domain expertise to understand (Cao et al., 2024). Machine learning (ML) and large language models (LLMs) have been used to improve the reverse

055

056

057

060

061

062

063

064

065

066

067

069

071

072

073 074 075

076 077

079

081

083

084

085

087

880

089

090

091

092

094

095

096

098

100

101

102

103

104

105

106

107

engineering output quality, such as by predicting variable names and types (Lacomis et al., 2019; Nitin et al., 2021), decompiling with translation ML models (Armengol-Estapé et al., 2024; Udeshi et al., 2025), and binary code summarization with LLMs (Jin et al., 2023; Tan et al., 2024).

Binary diffing tools are built on top of binary analysis methods and hence face similar issues of obscurity, hard-to-understand outputs. Current tools reliably identify modified binary components by employing binary code similarity metrics; however, cybersecurity experts require significant effort to understand the code changes to identify vulnerabilities or detect malicious injected code. We propose *binary diff summarization* to augment binary diffs with natural language summaries produced by an LLM. Additionally, we introduce the *functional sensitivity score* (FSS), a novel categorization method to triage binary functions such that sensitive behaviors that reveal vulnerabilities or malware are marked with a high score. We evaluate the binary diff summarization and functional sensitivity score for the software supply chain security task of detecting malware injected into opensource programs. For this, we construct a benchmark by injecting malware into multiple versions of open-source programs to construct compromised software updates across clean/injected versions.

The contributions of this paper are threefold: (i) A novel framework for *binary diff summarization* that augments outputs from binary diffing tools with LLM-generated natural language summaries for improved code understanding; (ii) The *functional sensitivity score*, a novel method to triage sensitive function behaviors that highlight vulnerabilities and malicious code; (iii) A *software supply chain security benchmark* of Linux open-source programs injected with 3 different malware, comprising of 6 projects, 104 versions, 392 binary diffs, and 46,023 functions.

2 BACKGROUND AND RELATED WORK

Binary differential analysis: Binary differential analysis (binary diffing) is the process of identifying changes between compiled binaries at different granularities, such as instructions, basic blocks, or complete binary formats (Haq & Caballero, 2021). Unlike source-level diffing, which benefits from static code analysis, binary diffing is considerably more challenging due to compiler optimizations, instruction set variations, and obfuscation techniques (Linn & Debray, 2003). Early efforts such as BinDiff (Flake, 2004), DarunGrim (Oh, 2008), and BMAT (Wang et al., 2000) relied on syntactic and graph-based similarity across control-flow graphs (CFGs), with later improvements addressing register allocation and instruction reordering (Dullien & Rolles, 2005). Subsequent works have expanded these ideas. Diaphora leverages SQL-based heuristics for CFG matching (Koret, 2015-2025), while Asm2Vec (Ding et al., 2019) introduced function embeddings resilient to compiler optimizations. More recently, deep learning methods such as jTrans (Wang et al., 2022) incorporated transformers with jump and control flow awareness. QBinDiff (Cohen et al., 2024) reframed diffing as a graph-alignment problem, achieving robustness against obfuscation as shown in the evaluation of Cohen et al. (Cohen et al., 2025). Other approaches, including Binhunt (Gao et al., 2008), and BinSlayer (Bourquin et al., 2013) employed symbolic execution, graph isomorphism and unsupervised learning to capture semantic differences more precisely.

LLMs for Binary Analysis: Jin et al. (2023) introduced BinSum, a benchmark of over 557K binary functions across multiple architectures and optimization levels, along with novel prompt optimization strategies and semantic evaluation metrics for binary summarization. Dil et al. (2025) applied LLM-guided prompting to filter noisy vulnerability patch data in the BigVul dataset (Fan et al., 2020), improving the accuracy of downstream vulnerability prediction models, while Yu (2025) proposed DeepDiff, which embeds decompiled functions for similarity search and combines control and data flow analysis to detect logic-altering changes in binaries. Shang et al. (2024) constructed a benchmark for reverse engineering tasks such as function name recovery and summarization, systematically evaluating LLM capabilities. Hussain et al. (2025) developed Vul-BinLLM, which augments decompilation with contextual vulnerability annotations and employs in-context learning, chain-of-thought prompting, and memory management to improve detection accuracy. Lin & Mohaisen (2025) systematically evaluated LLMs for vulnerability detection in Java and C/C++ programs, highlighting cross-language performance, prompting strategies, and configuration best practices. Wong et al. (2023) explored "recompilable decompilation," proposing a hybrid two-stage approach where LLMs correct syntax errors in decompiled outputs and resolve runtime memory errors, enabling regenerated executables that preserve original functionality. Chen et al. (2025) introduced ReCopilot, an expert LLM for binary analysis that integrates variable data-flow and call-

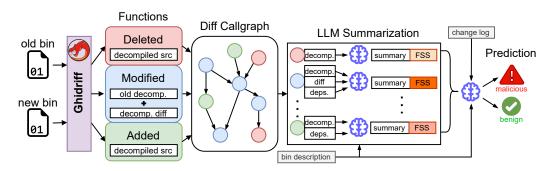


Figure 1: Overview of the binary diff summarization framework. Ghidriff provides *added*, *modified*, and *deleted* functions that are merged into a diff callgraph. Each function undergoes LLM summarization and FSS classification. Finally, prediction happens to label the diff as malicious or benign.

graph information with test-time scaling, and through continued pretraining, supervised fine-tuning, and direct preference optimization, achieved up to a 13% improvement over state-of-the-art models in function name recovery and variable type inference.

3 Method

108

109

110

111 112

113

114

115

116

117

118 119

120

121

122 123

124

125

126 127

128 129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

Figure 1 is an overview of the binary diff summarization pipeline. In the context of software supply chain security, the pipeline concludes by producing a malicious/benign prediction, however the summaries and FSS scores can be used for tasks such as vulnerability detection or patch identification.

Ghidriff: The pipeline begins by taking two binaries namely old and new. We use Ghidriff (McIntosh, 2023) as the binary diff tool. Ghidriff uses the Ghidra decompilation engine to perform the initial analysis of both binaries, then computes correlations across functions from old and new. The correlation reveals whether a pair of functions match exactly, match approximately, or do not match. Ghidriff outputs three lists of functions: deleted contains functions present in old that do not match with any function in new, added contains functions present in new that do not match with any function in old, and modified contains functions that match approximately. In this manner, functions that match exactly are removed from the diff, so only the binary changes remain. As the symbols in both binaries are stripped, the *modified* functions will show up with different names in the decompiled code depending on their hexadecimal address. We rename the *modified* functions to a consistent name incorporating both the old and new address, and update all referenced locations, so that this name difference does not show up unnecessarily.

Diff Callgraph: For the functions in the three lists, we extract the decompiled source code from Ghidra analysis. For *modified* functions, we additionally compute a textual diff of the decompiled code using the Python difflib¹ module to provide a succinct representation of the changes. The LLM summarization happens function by function and some infor-

You are a skilled cybersecurity professional. You are an expert at binary analysis, binary diffing, and patch diffing. Your goal is to process a binary diff ... flag any sensitive code changes and sensitive behaviors ... The binaries you are analyzing are two consecutive versions from the microhttpd library.... a lightweight C versions from the micronitipo iiprary... a hydracig... HTTP server library. The function you are analyzing lcurrently is FUN 00104ac0 in the old binary and oa0 in the new binary. It was modified. Here is the decompiled code. Here is the diff .. pthread_attr_destroy(param_1) socket(param_1,param_2,param_3); Describe the code functionality and ...changes. func_summary: The original function ... effectively cleaning up a thread attribute object. ...

diff_summary: The behavior transitions from destroying ... to attempting to create a socket ...Semantic shift implies the function is no longer performing resource cleanup.. Based on the functionality summary ...rate the function based on ...; sensitive behaviors, sensitive resources accessed, confidentiality impact, integrity impact, availibility impact. Each should be classified as none, low, medium, or high. Below are a few examples ... behavior: medium resource: low confidentiality: none score = 3.5 integrity: none availability: low **(‡)**

Figure 2: Example of LLM summarization and FSS for a modified function from microhttpd.

mation about the function dependencies (in terms of other functions it calls) need to be provided for the LLM to understand the functionality correctly. This dependency information is captured in the diff callgraph. The diff callgraph is essentially the merged callgraph of the *old* and *new* binaries,

https://docs.python.org/3/library/difflib.html

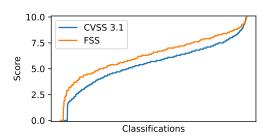
where only the *added*, *deleted*, and *modified* functions are preserved. Instead of merging and trimming down the full callgraphs of the binaries, we construct the diff callgraph by directly analyzing referenced dependencies in the three diff function lists.

LLM Summarization: The functions are processed in a reverse breadth-first traversal starting from leaf nodes of the diff callgraph to ensure that a function's dependencies are processed before it. For each function, the decompiled source code along with summaries of the dependencies are passed to the LLM. For *modified* functions, the textual diff of decompiled code between *old* and *new* is also passed. The summary and FSS are generated via two separate prompts. The first prompt asks for a functionality summary and an optional diff summary. The second prompt continues the conversation (the LLM sees its previous output) and asks for the FSS. Figure 2 shows an example conversation for a modified function from microhttpd, where the LLM first correctly identifies the changed functionality and then proceeds to mark the FSS categories appropriately.

Functional Sensitivity Score: The FSS is designed similar to the common vulnerability scoring system (CVSS) (Howland, 2022) such that functions of interest can be marked during binary analysis using consistent categories. CVSS helps score the severity of vulnerabilities after they are identified with distinct categories and classification options, for example attack complexity (low, high) and privileges required (none, user, administrator). This allows for better vulnerability classification by cybersecurity professionals than picking abstract numerical values. CVSS aggregates the category classifications into a severity score from 0 to 10. CVSS does not directly apply for vulnerability or malware detection. Thus we design FSS with similar goals to provide meaningful categories and classifications for scoring functional sensitivity. We pick five categories with examples:

- Sensitive behaviors (B): reading system info, opening sockets, forking processes
- Sensitive resources (R): network, system files, hardware devices
- Confidentiality impact (C): sending files over network, reading passwords or keys
- Integrity impact (I): modifying system configuration, overwriting files, encrypting data
- Availability impact (A): disabling system services, consuming unnecessary resources

Each category is classified as none, low, medium, or high. We provide examples to the LLM of each category and each classification to ground its outputs. These examples can be adapted to different scenarios and environments to better guide the LLM.



Score = 5.4							
Behavior	none	low	medium	high			
Resource	none	low	medium	high			
Confidentiality	none	low	medium	high			
Integrity	none	low	medium	high			
Availability	none	low	medium	high			

Figure 3: CVSS 3.1 and FSS scores across all classifications in increasing order.

Figure 4: Example of an FSS classification with the aggregate score.

Similar to CVSS 3.1, the final score is aggregated using the formula:

$$S = 1 - (1 - B)(1 - R)$$

$$M = 1 - (1 - C)(1 - I)(1 - A)$$

$$FSS = \begin{cases} \text{roundup}(5.3S + 6.1M) & M > 0\\ 0 & \text{otherwise} \end{cases}$$

where B,R,C,I,A are as defined above, S is sensitivity aggregate, M is impact aggregate, and roundup rounds up values to one decimal place. Weights for B and A are $\{\text{none}=0,\text{low}=0.1,\text{medium}=0.35,\text{high}=0.6\}$, while weights for C, I, and A are $\{\text{none}=0,\text{low}=0.22,\text{medium}=0.39,\text{high}=0.56\}$. Equations for S and M are structured to produce a high score when any one of the components are marked higher, similar to CVSS 3.1. The weights and coefficients were tuned such that FSS captures the scores from 0 to 10. Figure 3 shows the scores

of all classifications in increasing order for CVSS 3.1 and FSS, demonstrating that FSS behaves similarly to the industry-standard CVSS 3.1. Figure 4 shows an example classification and its score.

Prediction: The last step of the pipeline is the prediction that outputs whether the summarized diff contents resemble malicious injection or a benign software update. We implement this step by passing the top k functions with highest FSS to the LLM and prompt it to output either MALICIOUS or BENIGN by reasoning about whether the changes match the project description.

224

4 **EVALUATION**

225 226

SUPPLY CHAIN SECURITY BENCHMARK

227 228 229

230

231

232

We construct a benchmark for software supply chain security by picking six popular opensource projects spanning command line utilities and libraries gzip, openssl, tar, sqlite, microhttpd, and paho-mqtt. Additional details are provided in Appendix A.2. Table 1 shows the project description, number of versions selected, number of diffs from taking consecutive version pairs, and total number of added, deleted, and modified functions across all diffs. Each project is compiled in a Ubuntu 20.04 docker container with the default compiler GCC 9.4.0. Pairs of binaries of consecutive versions are treated as software updates and we use them for binary diffing. We collected 104 versions across the 6 projects, generating 98 software update pairs.

Project	Description	Versions	Diffs	Functions
gzip	File compression utility	5	16	1209
openssl	Cryptography library and utility	29	112	9722
tar	File and directory archival utility	10	36	8936
sqlite	Single-file SQL database library	28	108	16682
microhttpd	Lightweight HTTP server library	23	88	6648
paho-mqtt	MQTT lightweight messaging library	9	32	2826
Total		104	392	46023

242 243 244

Table 1: Details of software supply chain security benchmark.

245 246 247

248 249

250

251

253

254

256

257

Additionally, we implement three malwares to inject into the source code of each project. Details of each malware are provided in Appendix A.3.

- rware: a ransomware (Li, 2021) that encrypts user files using AES and ECDH encryption
- rat: a remote access trojan (Kara & Aydos, 2019) that initiates a reverse shell with remote server

• botnet: a bot network (Antonakakis et al., 2017) for denial-of-service attacks on servers

Each malware is implemented as C code contained in one source file that is copied into the project source directory and added to the build system. The entry point of each malware is a C function that takes no arguments and returns no values. For each project, we determine a trigger point that is not reachable in normal operation of the project but can be triggered by the attacker with specific malformed configurations, for example, passing an attacker-defined command line option. We obtain 4 binary diffs per version pair by considering a software update from a clean binary of the former version to the clean and injected binaries of the latter version. In total, this makes 392 diffs.

258 259 260

266

267

268

269

4.2 METRICS

261 262 263 264 265

Malware Detection: Prediction output is evaluated against ground truth labels for each diff. Diffs with binaries containing the injected malware are labeled MALICIOUS and diffs with clean binaries are labeled BENIGN. Treating the MALICIOUS label as positive, we compute precision and recall to evaluate accuracy of malware detection. False positives would be clean diffs labeled as MALICIOUS. False negatives would be diffs with injected malware labeled as BENIGN.

FSS Separation: It is difficult to evaluate the quality of FSS scores assigned by an LLM without human-labeled scores for functions in the diff. Even in clean diffs, functions may show different behaviors and thus different FSS. In our benchmark, we mark functions from the original code as benign and injected functions as malicious. FSS scores are averaged as $FSS_{\rm ben}$ and $FSS_{\rm mal}$ across

Model /	<i>k</i> =	= 5	k =	= 10	k = 5	w/ change	k = 10	w/ change
Program	P	R	P	R	P	R	P	R
GPT5 mini	0.96	0.58	0.96	0.54	0.98	0.64	0.98	0.60
gzip	1.00	0.75	1.00	0.50	1.00	0.75	1.00	0.67
openssl	0.83	0.33	0.83	0.33	0.90	0.60	0.82	0.30
tar	0.83	0.37	0.80	0.30	0.79	0.41	0.77	0.37
sqlite	1.00	0.70	0.98	0.81	1.00	0.66	0.98	0.78
microhttpd	0.98	0.62	0.97	0.58	1.00	0.64	1.00	0.65
paho-mqtt	1.00	0.71	1.00	0.71	1.00	0.62	1.00	0.79
GPT5 nano	0.83	0.35	0.87	0.36	0.85	0.29	0.87	0.34
gzip	0.86	0.50	0.80	0.33	0.67	0.17	0.60	0.25
openssl	0.81	0.40	0.78	0.30	0.81	0.31	0.88	0.33
tar	0.78	0.67	0.76	0.59	0.72	0.48	0.74	0.52
sqlite	0.82	0.31	0.95	0.53	0.92	0.31	0.92	0.46
microhttpd	0.93	0.20	0.94	0.24	0.93	0.21	0.94	0.24
paho-mqtt	1.00	0.25	1.00	0.17	1.00	0.17	1.00	0.17
GPT OSS 20B	0.89	0.41	0.93	0.42	0.89	0.40	0.92	0.41
gzip	0.80	0.33	0.86	0.50	0.83	0.42	0.83	0.42
openssl	0.85	0.33	0.88	0.33	0.83	0.29	0.88	0.35
tar	0.86	0.29	0.80	0.19	0.80	0.19	0.75	0.14
sqlite	0.92	0.72	0.95	0.72	0.95	0.75	0.97	0.72
microhttpd	0.95	0.32	1.00	0.26	0.81	0.26	0.90	0.27
paho-mqtt	0.67	0.08	1.00	0.29	1.00	0.12	1.00	0.17

Table 2: Malware detection precision (P) and recall (R) for each model and each program. k refers to how many functions with highest score are provided for prediction step. "w/ change" refers to program changelog being provided for the prediction step.

a binary. Their distributions are checked to see if malicious functions score higher than benign ones. Higher separation of the distributions of $FSS_{\rm ben}$ and $FSS_{\rm mal}$ will indicate better FSS quality.

5 RESULTS

Experimental Setup: We evaluate with two commercial LLMs, *GPT5 mini* and *GPT5 nano* (OpenAI, 2025a), and three open-source LLMs, *GPT OSS 20B* (OpenAI, 2025b), *Qwen3 30B*, and *Qwen3 8B* (Qwen, 2025). All four LLMs are run in thinking/reasoning mode. Reasoning effort is set to "low" for *GPT5 mini*, *nano*, and *GPT OSS 20B* models. *Qwen3 30B*, *8B*, and *GPT OSS 20B* models are run via Ollama on a server with two NVidia L40 GPUs. The models are run with default hyperparameter settings as follows: temperature of 1.0 and top-p of 1.0 for *GPT5 mini*, *GPT5 nano*, and *GPT OSS 20B*; temperature of 0.6 and top-p of 0.95 for *Qwen3 8B* and *Qwen3 30B*. We do not evaluate the highest capability *GPT5* on the full benchmark due to high API costs and because the smaller models suffice as seen in the results.

Table 2 shows the accuracy of malware detection as measured by precision (P) and recall (R) described in Section 4.2. The same model is used for final prediction as for summarization. We run the prediction step with four configurations by modifying k, the number of functions with highest FSS provided to the LLM, and whether or not the program changelog was provided. The changelog is extracted from the program source code or online repository for each version and appended to the initial prompt when needed. The gray highlighted rows show the overall P and R for each model, while the rows beneath them show per-program results. We do not display the results of Qwen3 30B and Qwen3 8B as both models stumble in the final prediction step; in many cases they do not produce either MALICIOUS or BENIGN as instructed and a prediction is not obtained from their response. To overcome this, we perform the final prediction step with GPT5 mini for these and other models, with results described later in Table 3.

Precision: Across all models, P ranges from 0.83 to 0.98, indicating that the framework obtains low false positives. GPT5 mini achieves highest P of 0.98 for both k with changelog, with only a slight improvement over without changelog. GPT OSS 20B achieves lower P from 0.89 to 0.93.

Its performance improves with k=10 over k=5, and degrades with changelog for both k. This indicates that giving a larger context of function summaries is better than providing changelog. GPT5 nano ranks third with P from 0.83 to 0.87, showing similar configuration trends as GPT5 mini. Performance improves slightly with higher k, but is not affected by changelog. Overall scores show that providing a changelog does not significantly impact P, but larger k helps in some cases.

Recall: R sees interesting trends across all models, ranging from 0.35 to 0.64, indicating wide variation in the false negatives and accuracy of detecting malware. This is expected as it is harder to classify newly introduced code as malicious or benign than analyzing legitimate changes. Nonetheless, this performance indicates that the framework detects malicious injected code in upto 64% cases. For GPT5 mini, maximum R is achieved for k=5 with changelog. R improves by 0.06 with changelog than without, indicating that the changelog grounds the model in terms of what differences to expect thus improving malware detection accuracy. R degrades by 0.04 for k=10 than k=5, indicating that providing more functions in the context may confuse the model. For GPT OSS 20B, R ranges from 0.40 to 0.42, showing consistent results across configurations. For this model, providing changelog degrades R slightly by 0.01, while larger k improves it by 0.01. GPT5 nano obtains R from 0.29 to 0.36, ranking lower than GPT OSS 20B. R drops significantly with changelog, but increases with larger k. GPT5 nano reiterates the behavior of GPT OSS 20B, suggesting that lower capability models perform better with a larger function context instead of providing changelogs, whereas higher capability models utilize the changelog better.

Programs: The per program results of P and R show wide variation. $GPT5\ mini$ obtains near perfect P for all programs across configurations, except openssl and tar where false positives are high likely because of crypto operations and directory access in them. k=5 with changelog improves P for openssl, but other configurations degrade. For $GPT5\ nano$ and $GPT\ OSS\ 20B$, P is in the range 0.60 to 1.00, with microhttpd and paho-mqtt achieving high P. Variations are consistent with P of the model overall. On R, $GPT5\ mini$ consistently outperforms $GPT5\ nano$ and $GPT\ OSS\ 20B$ for configuration k=5 with changelog, except for tar where $GPT5\ nano$ is higher. The impact trends of the configuration are not consistent across programs, highlighting variable dependence on the amount of functions and whether changelog is relevant to malware detection.

Model	k = 5		k = 10 k		k = 5	k = 5 w/ change		k = 10 w/ change	
	P	R	P	R	P	R	P	R	
GPT5 nano	0.92	0.47	0.88	0.39	0.92	0.45	0.92	0.45	
GPT OSS 20B	0.96	0.45	0.97	0.47	0.95	0.45	0.95	0.47	
Qwen3 30B	0.97	0.51	0.97	0.57	0.97	0.58	0.97	0.52	
Qwen3 8B	0.97	0.42	0.93	0.38	0.97	0.47	0.97	0.46	

Table 3: Malware detection precision (P) and recall (R) when using GPT5 mini as the predictor on summaries generated by each model.

Combination of LLMs: As described above, the *Qwen3 30B* and *Qwen3 8B* models face difficulty in the final prediction step. To overcome that, we explore how the framework performs when using a lower capability model for the summarization and using a higher capability model for the prediction. Table 3 shows P and R with GPT5 mini as the predictor model combined with the other models for summarization. Comparatively, P and R are consistently higher with GPT5 mini as the predictor than GPT5 nano and GPT OSS 20B. P ranges from 0.88 to 0.92 for GPT5 nano, lower than for other models where it ranges from 0.93 to 0.97. This reiterates the trend of Table 2, indicating that GPT5 nano summaries lead to higher false positive malware detection. Qwen3 30B performs best in terms of R from 0.51 to 0.58 which implies higher quality summaries. For GPT OSS 20B, R stays consistent for both R and with or without changelog. Qwen3 30B gets a boost in R with either higher R or with changelog, whereas R improves only with changelog.

FSS Separation: Figure 5 presents the distribution of FSS scores for benign $(FSS_{\rm ben})$ and malicious $(FSS_{\rm mal})$ functions as described in Section 4.2. Figure 5(a) shows distribution by model. All models demonstrate a clear separation between $FSS_{\rm ben}$ and $FSS_{\rm mal}$. Except for GPT5 mini, other models have a wider spread in the $FSS_{\rm mal}$ distribution that overlaps with $FSS_{\rm ben}$, yet the boxes and medians remain clearly separated. GPT5 mini shows a tighter distribution of $FSS_{\rm mal}$ than the rest, demonstrating greater scoring consistency. Overall, the median scores show a difference of 1.5

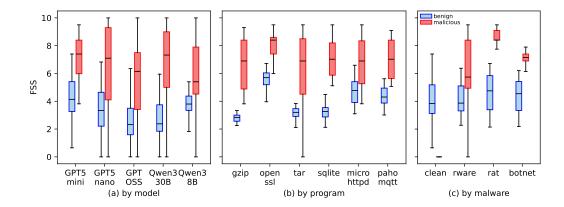


Figure 5: Distribution of $FSS_{\rm ben}$ and $FSS_{\rm mal}$ (a) by LLM, (b) by program for *GPT5 mini*, and (c) by malware for *GPT5 mini*. The boxes show first to third quartile, the middle line shows median, and the whiskers show $1.5 \times$ inter-quartile range.

to 5.0 points, with *GPT5 mini* having a separation of 3.0. The summarization framework reliably marks malicious injected functions with higher FSS than benign functions. The benign functions are consistently scored with median FSS of 4.0 or lower; LLM understands function sensitivity correctly, illustrating efficacy of FSS categorization.

To further investigate the performance of the best-performing model $GPT5\ mini$, Figures 5(b) and 5(c) provide a granular breakdown of $GPT5\ mini$'s scores. The analysis by program demonstrates that $GPT5\ mini$'s discriminative power is robust across the set of programs. Interestingly, the $FSS_{\rm ben}$ distributions across programs are narrow, showing that $GPT5\ mini$ consistently marks the functions similarly. Additionally, openssl, microhttpd, and paho-mqtt get higher $FSS_{\rm ben}$ as expected because the benign functions have cryptographic and network functionalities. Similarly, Figure 5(c) illustrates the model's effectiveness against different malwares. Distribution of $FSS_{\rm mal}$ for rware is largest, while for rat and botnet is very narrow, indicating that it is easier to identify sensitive behaviors with the network access in the later two. Nonetheless, there is a clear separation across all malwares which makes it easy to configure thresholds for detection.

6 CASE STUDY: XZ BACKDOOR

We analyze the XZ Utils supply chain attack detected in 2024 (Przymus & Durieux, 2025), where the open-source XZ repository was compromised to inject a backdoor into the liblzma.so library. This library is ubiquitous on Linux systems ranging from servers to embedded controllers, so the attack would have devastating consequences, however it was caught before the backdoor was distributed as part of up-We compile the XZ utils source code for the compromised version v5.6.0 and a previous version v5.4.7. We evaluate our binary diff summarization framework on the

Summ.	Predictor GPT5 GPT5 mini		Top-5 functions		
· ·		01 10 111111			
		malicious	FUN_00104794(6.5)		
GPT5	malicious		FUN_00104720(6.5)		
			_get_cpuid(5.3)		
			lzma2_decode(4.3)		
			$FUN_0011e4a0(4.2)$		
			x86_code(4.2)		
GPT5 mini	malicious	benign	FUN_00104794(4.2)		
			crc64_set_fun(3.4)		
			_get_cpuid(3.4)		
			FUN_00104720(3.4)		

Table 4: XZ backdoor detection by *GPT5* and *GPT5 mini* along with sensitive functions identified by both.

generated liblzma.so libraries. We run GPT5 mini and GPT5 for both the summarization and prediction step with k=5 and no changelog.

Table 4 shows the output of malware detection by GPT5 and GPT5 mini when run on each other's summarizations. GPT5 correctly marks the diff summaries as mali-

cious for both the summaries generated by itself and by *GPT5 mini*. On the other hand, *GPT5 mini* marks its own summaries as benign, however it correctly marks GPT5 summaries as malicious. This indicates that both models highlight the injected malicious behavior sufficiently, while it takes the more capable GPT5 for a correct prediction.

The top-5 highest scored functions are shown along with their scores for both models. The red highlighted functions were those injected with malicious behavior. Out of 79 functions in the diff, both models score the relevant malicious functions higher so they appear among the top 5. GPT5 scores the malicious functions highest, whereas *GPT5 mini* scores them generally lower. This demonstrates that the LLMs correctly identify sensitive behaviors using the FSS categorization.

Figure 6 shows the diff summaries generated by GPT5 for the two highlighted functions. Highlighted in red, we see the model describe how the functionalities are "atypical for liblzma" and differ from "liblzma's expected functionality". This case study illustrates that LLMs utilize the binary diff summarization framework and FSS categorization to produce meaningful summaries that highlight malicious behavior when analyzing software updates.

FUN_00104720 diff summary



Newly added function. It introduces a call-count–gated side effect (only when the global counter equals 1 on entry) that invokes NEWF_00104794 with a crafted local buffer containing param_2. That callee performs an unchecked transient memory write to an address derived from pointer arithmetic and issues a CPUID call before restoring memory. This is atypical for libizma and poses potential integrity/availability risks (fault or probe) and increases hardware fingerprinting/anti-analysis behaviors...

FUN_00104794 diff summary



New function added...introduces opaque initialization and a transient write to an address computed from (vtable_pointer - this + 0x2e8), followed by a CPUID call and restoration... Unchecked computed-pointer dereference/write (may crash ... integrity/availability risk)... Stores pointer/offset deltas into object fields, suggesting obfuscation or anti-reverse-engineering behavior rather than typical libizma state setup. Given libizma's expected functionality, this routine is anomalous and warrants deeper review for malicious or vulnerable behavior...

Figure 6: GPT5 summaries for the XZ backdoor functions.

7 CONCLUSION

In this work, we presented a novel framework for binary diff summarization using LLMs, with a specific focus on enhancing software supply chain security. We introduce the functional sensitivity score (FSS), a metric designed for automated triage of sensitive functions within binary diffs. To evaluate our approach, we created a new benchmark for software supply chain security, comprising 104 versions of 6 open-source projects, into which we injected 3 different types of malware. Our framework achieved a high precision of 0.98 and a recall of 0.64 for malware detection. Furthermore, the FSS demonstrated a clear separation of 3.0 points between malicious and benign functions, highlighting its effectiveness. On the real-world XZ backdoor case study, our framework correctly captured the injected malicious functions with high FSS and correctly marked the software update as malicious, exemplifying the applications to real-world scenarios. These findings illustrate the significant potential of leveraging LLMs for automation of software supply chain security. Future work could explore the application of this framework to other security-critical domains, such as vulnerability detection and patch analysis. The FSS could be adapted and refined for other security applications, and the framework could be extended to support a wider range of architectures.

Ethics: This work explores the use of large language models (LLMs) for binary diff summarization, which identifies changes between binary versions to help analysts detect bugs, vulnerabilities, and supply chain threats. Although the technique strengthens patch management and software integrity verification, it also has dual-use implications. Malicious actors could potentially exploit the same methods for reverse engineering, intellectual property theft, or scalable attacks on software supply chains. Our study is conducted purely for defensive and research purposes, aiming to advance the ability of the security community to manage patches and identify vulnerabilities. We acknowledge the risks of misuse and emphasize the importance of safeguards, rigorous evaluation, and governance mechanisms in guiding responsible adoption of LLM-based tools. By contextualizing and transparently reporting our findings, we seek to raise awareness of emerging attack vectors while supporting the development of effective countermeasures.

Reproducibility: We provide the evaluation artifacts containing the ghidriff outputs, LLM-generated summaries, and predictions as part of the supplementary materials. For safety reasons, the malware source code and compiled binaries have been omitted. We will release source code and build scripts at the time of publication.

REFERENCES

- Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In 26th USENIX security symposium (USENIX Security 17), pp. 1093–1110, 2017.
- Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F. P. O'Boyle. Slade: A portable small language model decompiler for optimized assembly. CGO '24, pp. 67–80. IEEE Press, 2024. ISBN 9798350395099. doi: 10.1109/CGO57630.2024.10444788. URL https://doi.org/10.1109/CGO57630.2024.10444788.
- Martial Bourquin, Andy King, and Edward Robbins. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, pp. 4:1–4:10, 2013. doi: 10.1145/2430553.2430557.
- David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 2–16, 2006. doi: 10.1109/SP.2006.41.
- Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 491–502, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3652144. URL https://doi.org/10.1145/3650212.3652144.
- Guoqiang Chen, Huiqi Sun, Daguang Liu, Zhiqi Wang, Qiang Wang, Bin Yin, Lu Liu, and Lingyun Ying. Recopilot: Reverse engineering copilot in binary analysis. *arXiv preprint* arXiv:2505.16366, 2025.
- Cristina Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, 1994.
- Roxane Cohen, Robin David, Riccardo Mori, Florian Yger, and Fabrice Rossi. Improving Binary Diffing Through Similarity and Matching Intricacies. In SSTIC Proceedings (SSTIC 2024), pp. 1-8, 2024. URL https://www.sstic.org/media/SSTIC2024/SSTIC-actes/qbindiff_a_modular_differ/SSTIC2024-Article-qbindiff_a_modular_differ-rossi_yger_mori_david_cohen.pdf.
- Roxane Cohen, Robin David, Riccardo Mori, Florian Yger, and Fabrice Rossi. Experimental Study of Binary Diffing Resilience on Obfuscated Programs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 223–243. Springer, 2025.
- Charlie Dil, Hui Chen, and Kostadin Damevski. Towards higher quality software vulnerability data using llm-based patch filtering. *Journal of Systems and Software*, pp. 112581, 2025.
- Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search Against Code Obfuscation and Compiler Optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 472–489, 2019. doi: 10.1109/SP.2019.00003. URL https://dmas.lab.mcgill.ca/fung/pub/DFC19sp.pdf.
- Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Network and Distributed System Security Symposium* (NDSS), 2020. doi: 10.14722/ndss.2020.24311.
- Thomas Dullien and Rolf Rolles. Graph-Based Comparison of Executable Objects. In Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC), 2005. URL https://actes.sstic.org/SSTIC05/Analyse_differentielle_de_binaires/SSTIC05-article-Flake-Graph_based_comparison_of_Executable_Objects.pdf. English version (PDF). See also BinDiff historical references and implementations.

- Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pp. 508–512, 2020.
 - Halvar Flake. Structural Comparison of Executable Objects. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pp. 161–173, 2004.
 - FortiGuard Labs. 3CX Desktop App Compromised (CVE-2023-29059), 2023. URL https://www.fortinet.com/blog/threat-research/3cx-desktop-app-compromised.
 - Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the International Conference on Information and Communications Security (ICICS)*, pp. 238–255, 2008. doi: 10.1007/978-3-540-88625-9_16.
 - Irfan Ul Haq and Juan Caballero. A Survey of Binary Code Similarity. *ACM Comput. Surv.*, 54(3), April 2021. ISSN 0360-0300. doi: 10.1145/3446371. URL https://doi.org/10.1145/3446371.
 - Henry Howland. Cvss: Ubiquitous and broken. *Digital Threats*, 4(1), February 2022. doi: 10.1145/3491263. URL https://doi.org/10.1145/3491263.
 - Nasir Hussain, Haohan Chen, Chanh Tran, Philip Huang, Zhuohao Li, Pravir Chugh, William Chen, Ashish Kundu, and Yuan Tian. Vulbinllm: Llm-powered vulnerability detection for stripped binaries. *arXiv preprint arXiv:2505.22010*, 2025.
 - Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601*, 2023.
 - İlker Kara and Murat Aydos. The ghost in the system: technical analysis of remote access trojan. *International Journal on Information Technologies & Security*, 11(1):73–84, 2019.
 - Joxean Koret. Diaphora Program Diffing Plugin for IDA Pro (GitHub). https://github.com/joxeankoret/diaphora, 2015-2025. URL https://github.com/joxeankoret/diaphora.
 - Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltos Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 628–639. IEEE, 2019.
 - Adrian Shuai Li. An analysis of the recent ransomware families. *Project Report. Purdue University*, 2021.
 - Jie Lin and David Mohaisen. From large to mammoth: A comparative evaluation of large language models in vulnerability detection. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*, 2025.
 - Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 290–299, 2003.
 - John McIntosh. Ghidriff: Python Command-Line Ghidra Binary Diffing Engine, 2023. URL https://github.com/clearbluejar/ghidriff.
 - Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. Direct: a transformer-based model for decompiled identifier renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pp. 48–57, 2021.
 - Jeong Wook Oh. DarunGrim: A Patch Analysis and Binary Diffing Tool. https://sgros-students.blogspot.com/2014/06/binary-diffing-using-darungrim.html, 2008.

```
OpenAI. GPT-5 System Card, 2025a. URL https://openai.com/index/gpt-5-system-card/.
```

- OpenAI. GPT-OSS System Card, 2025b. URL https://openai.com/index/gpt-oss-model-card/.
- Piotr Przymus and Thomas Durieux. Wolves in the repository: A software engineering analysis of the xz utils supply chain attack. In 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), pp. 91–102. IEEE, 2025.
- Qwen. Qwen3 System Card, 2025. URL https://huggingface.co/Qwen/Qwen3-8B.
- Beatriz M. Reichert and Rafael R. Obelheiro. Software supply chain security: a systematic literature review. *International Journal of Computers and Applications*, 46(10):853–867, 2024. doi: 10. 1080/1206212X.2024.2390978. URL https://doi.org/10.1080/1206212X.2024.2390978.
- Paul Royal, Matthew Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pp. 289–300, 2006. doi: 10.1109/ACSAC.2006.20.
- Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop On Binary Analysis Research (BAR)*, 2018. doi: 10.14722/ndss. 2018.23002.
- Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in binary code understanding using large language models. In 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 1–12. IEEE, 2024.
- Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and Robust Binary Library Function Identification Using Function Shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 301–324. Springer, 2017.
- Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, USA, 2016. IEEE.
- Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.203. URL http://dx.doi.org/10.18653/v1/2024.emnlp-main.203.
- Meet Udeshi, Prashanth Krishnamurthy, Ramesh Karri, and Farshad Khorrami. Remend: Neural decompilation for reverse engineering math equations from binary executables. *ACM Trans. Intell. Syst. Technol.*, July 2025. ISSN 2157-6904. doi: 10.1145/3749988. URL https://doi.org/10.1145/3749988. Just Accepted.
- Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jTrans: Jump-Aware Transformer for Binary Code Similarity. *arXiv Preprint*, 2022. URL https://arxiv.org/abs/2205.12713.
- Zheng Wang, Ken Pierce, and Scott McFarling. BMAT A Binary Matching Tool for Stale Profile Propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.
- Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, Michel Cukier, Christian Kästner, Alexandros Kapravelos, Dominik Wermke, and William Enck. Research directions in software supply chain security. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025. ISSN 1049-331X. doi: 10.1145/3714464. URL https://doi.org/10.1145/3714464.

Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*, 2023.

Sheng Yu. DeepDiff: Next-Generation Binary Diffing for Precise Vulnerability and Patch Detection, 2025. URL https://www.deepbits.com/blog/DeepDiff.

Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, et al. A Large-Scale Empirical Analysis of the Vulnerabilities Introduced by Third-Party Components in IoT Firmware. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 442–454, 2022.

A APPENDIX

A.1 RELATED WORK COMPARISON

Tool / Paper	Category / Approach
BMAT (Wang et al., 2000)	Symbol / Name-based / Fuzzy
BinDiff (Flake, 2004)	Graph-based
BinDiff extended (Dullien & Rolles, 2005)	Graph-based
BinHunt (Gao et al., 2008)	Graph-based + Symbolic Execution
DarunGrim (Oh, 2008)	Graph-based
BinSlayer (Bourquin et al., 2013)	Graph-based + Bipartite matching
Diaphora (Koret, 2015–2025)	Graph-based
Asm2Vec (Ding et al., 2019)	ML-based embedding
DeepBinDiff (Duan et al., 2020)	ML-based embedding
jTrans (Wang et al., 2022)	Deep Learning
QBinDiff (Cohen et al., 2024)	Network alignment/Belief Propagation

A.2 BENCHMARK DETAILS

Program	URL
gzip openssl tar sqlite microhttpd paho-mqtt	https://ftp.gnu.org/gnu/gzip https://github.com/openssl/openssl/releases/download http://mirror.rit.edu/gnu/tar https://sqlite.org https://ftp.gnu.org/gnu/libmicrohttpd https://github.com/eclipse-paho/paho.mqtt.c/archive/ refs/tags

Table 5: URLs for each project in the benchmark.

A.3 MALWARE IMPLEMENTATIONS DETAILS

Ransomware: The ransomware is implemented as a C program that utilizes self-contained versions of tiny-AES² and tiny-ECDH³ for its cryptographic operations. The malware recursively scans for files and encrypts each one with a unique, randomly generated AES-128 key in Counter (CTR) mode, appending a .CRYPT extension to the filename. To protect these individual file keys, it employs an Elliptic Curve Diffie-Hellman (ECDH) key exchange using the NIST B-163 curve; it generates a shared secret by combining a new local private key with a hardcoded attacker's public

²https://github.com/kokke/tiny-AES-c

³https://github.com/kokke/tiny-ECDH-c

key. This shared secret is then used as a master key to encrypt all the individual file keys and their paths into an info.bin file, after which the ransomware drops a note containing the victim's public key needed for decryption.

Remote access trojan: The RAT implements a stealthy reverse shell that connects a target machine back to an attacker. It begins by reading the attacker's IP address and port from an environment variable, a technique used to avoid hardcoding sensitive information. The program then uses fork() to create a child process, allowing the parent to exit immediately while the malicious code continues to run in the background, detached from the original application. This child process establishes a network connection to the attacker's machine. The core of its functionality lies in using the dup2() system call to redirect the standard input, output, and error streams to the network socket. Finally, it calls execve() to replace its own process with /bin/sh, which is cleverly obfuscated in the code as a series of integer multiplications. Because the I/O streams are already redirected, this new shell process is fully interactive for the remote attacker, granting them command-line control over the compromised system.

Botnet: The botnet client is implemented based on the leaked source code of the Mirai botnet (Antonakakis et al., 2017). The program is designed to connect to a Command and Control (C2) server, which is hardcoded as "localhost" on port 5034. Once connected, the bot enters a loop where it sends a periodic keep-alive message to the C2 server and listens for attack commands. When a command is received, it is parsed to extract a target IP address, port, payload size, and the number of packets to send. Unlike the original Mirai, which featured multiple attack vectors, this simplified version only implements a basic UDP flood attack. This attack function bombards the specified target with a high volume of UDP packets containing randomized data, generated by a Xorshift pseudo-random number generator identical to the one used in Mirai, with the goal of overwhelming the target's network resources.

A.4 TOKEN CONSUMPTION

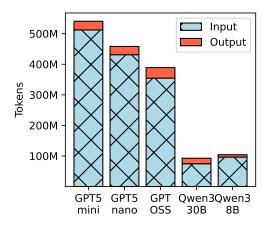


Figure 7: Token consumption.

Figure 7 shows the total input and output token consumption per model on the entire benchmark. The tokens range from 100M for Qwen3 models to 500M for *GPT5 mini*. The wide difference in token consumption may be due to different tokenizers for each model and because *GPT5 nano* and *GPT5 mini* may produce larger and more detailed function summaries that are sent back in the followup prompt. Output tokens are around 5% to 25% of input tokens. Considering 46K functions in the benchmark, the average per-function token consumption is around 2K to 12K.

A.5 LLM USAGE

Apart from the LLMs that we evaluate as part of our research, we have used the LLM tools ChatGPT and Gemini for refining and polishing content written by us.