USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

From Constraints to Cracks: Constraint Semantic Inconsistencies as Vulnerability Beacons for Embedded Systems

Jiaxu Zhao, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology;* Yuekang Li, *University of New South Wales;* Yanyan Zou, Yang Xiao, Naijia Jiang, Yeting Li, Nanyu Zhong, Bingwei Peng, Kunpeng Jian, and Wei Huo, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology*

https://www.usenix.org/conference/usenixsecurity25/presentation/zhao

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

From Constraints to Cracks: Constraint Semantic Inconsistencies as Vulnerability Beacons for Embedded Systems

Jiaxu Zhao[1,2,3,4], Yuekang Li[5], Yanyan Zou[1,2,3,4✉], Yang Xiao[1,2,3,4], Naijia Jiang[1,2,3,4]
Yeting Li[1,2,3,4], Nanyu Zhong[1,2,3,4], Bingwei Peng[1,2,3,4], Kunpeng Jian[1,2,3,4], Wei Huo[1,2,3,4✉]
[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, China
[4]Beijing Key Laboratory of Network Security and Protection Technology, China
[5]University of New South Wales, Australia

Abstract

Embedded systems have a profound impact on our daily lives and work by powering IoT devices and network devices. Ensuring their security is therefore critical. To enhance security and robustness, embedded systems often utilize constraints to validate user inputs. Through an empirical study, we identified that these constraints can be categorized into distinct types and may exhibit semantic inconsistencies across different components. Notably, over 86% of embedded system vulnerabilities originate from such inconsistencies. However, existing static analysis techniques struggle to systematically and accurately identify these inconsistencies, resulting in high false positive rates and an inability to detect certain vulnerabilities effectively.

This paper introduces NÜWA, a novel static analysis technique that leverages constraint semantic inconsistencies to detect vulnerabilities in embedded systems. NÜWA achieves scalable and precise vulnerability discovery by addressing the challenges of identifying constraint semantics across diverse implementations and accurately extracting them. We implemented NÜWA and evaluated it using known vulnerability datasets, including 31 vulnerabilities from 13 vendors, and compared its performance to five state-of-the-art (SOTA) tools. NÜWA identified 18, 22, 6, 17, and 19 more vulnerabilities than the respective SOTA tools. Further analysis demonstrates that NÜWA effectively extracts constraints with minimal false positives. To date, NÜWA has uncovered 152 previously unknown vulnerabilities which are all confirmed by the developers, and 88 were assigned with CVE IDs.

1 Introduction

Embedded systems serve as the foundational operating systems for the majority of Internet of Things (IoT) and network devices, enhancing daily life and work with greater convenience and intelligence. These devices are network-connected and primarily configured and managed through web services comprising front-end and back-end components. However, the growing complexity of web service code, driven by increasing device functionalities and services, has introduced more vulnerabilities. These vulnerabilities not only risk exposing sensitive information but also enable attackers to gain control of devices [1, 9, 16]. Consequently, detecting vulnerabilities in embedded system web services is of critical importance.

Existing approaches for vulnerability detection in embedded system web services focus on identifying execution paths between user-controllable inputs (sources) and sensitive operations (sinks), such as system calls or memory accesses [5, 11, 12, 31, 40]. Some methods employ taint analysis to trace these paths [10, 11, 21]. However, their effectiveness is constrained by the limited ability to identify sources and sinks. While recent advancements in taint-based techniques have improved source identification, they continue to struggle with accurately detecting sinks [4, 49]. Additionally, these approaches often yield high false positive rates due to their failure to consider path constraints. Alternatively, static symbolic execution has been applied to explore source-to-sink paths [6, 30, 45], resulting in fewer false positives. However, these techniques face significant challenges in handling complex constraints, limiting their effectiveness.

While existing approaches focus on detecting suspicious operations leading to vulnerabilities—i.e., sensitive operations potentially controllable by user inputs—we take a different perspective: What are the root causes of these vulnerabilities during development, and can this information be leveraged for detection?

Currently, most vulnerability defense mechanisms rely on input constraints to ensure proper validation of user inputs before processing. However, as front-end and back-end components in web services are often developed by separate teams, we hypothesize that inconsistencies may exist between front-end and back-end constraints, with some constraints potentially missing in the back-end.

✉Corresponding Authors.

To test this hypothesis, we conducted a comprehensive analysis of web services from 18 widely used embedded vendors. Constraints in these web services were classified into three types: (1) *front-end and back-end explicit constraints*, implemented as conditional statements in the code; (2) *desired constraints*, which represent essential validation rules needed to prevent vulnerabilities but are not properly implemented [2].

Our analysis of 324 recent vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database revealed that over 86% were caused by missing constraints in the back-end, either compared to the front-end explicit constraints or to the desired constraints. These inconsistencies allow attackers to bypass front-end checks and deliver malicious inputs to vulnerable code. In summary, we conclude that ***constraint inconsistencies can serve as a reliable indicator of potential vulnerabilities***.

Inspired by these findings, we propose NÜWA [3], a novel static analysis technique that leverages constraint semantic inconsistencies to detect vulnerabilities in embedded systems. The first key component of NÜWA is the accurate identification of constraint semantics across different code implementations. NÜWA performs targeted analysis of both front-end HTML and JavaScript source code, as well as back-end binary programs, representing constraints in six standardized semantic formats. It extracts explicit constraints from both the front-end and back-end code and infers desired constraints from the back-end code. The second component focuses on extracting and comparing constraint semantics for each source-sink path to identify inconsistencies and reduce false positives. NÜWA constructs an inter-procedural control flow graph (ICFG) centered on input-related elements and their contextual interactions. Using summary-based analysis and code slicing, it associates explicit and desired back-end constraints with relevant ICFG nodes and edges. Finally, NÜWA compares the semantic differences between back-end explicit constraints and the other two types (front-end explicit and back-end desired), flagging any inconsistencies. A manual review is then conducted to confirm whether the missing back-end constraints result in vulnerabilities.

We implemented NÜWA as a firmware static analysis framework and evaluated its performance against five state-of-the-art (SOTA) tools using a dataset of 31 known vulnerabilities from 13 vendors. The results demonstrate that NÜWA outperforms the SOTA tools, detecting 18, 22, 6, 17, and 19 more vulnerabilities than SATC, EMTAINT, LARA, MANGO, and OCTOPUSTAINT, respectively, while achieving the highest precision. For the 28 detected vulnerabilities, NÜWA achieved precision rates of 95%, 86%, and 85% in extracting front-end explicit, back-end explicit, and back-end desired constraints, respectively. The inconsistencies identified among these constraint semantics facilitated the detection of 12 additional vulnerabilities. Furthermore, NÜWA was applied to analyze 38 devices, uncovering 152 previously unknown vulnerabilities. All findings were confirmed by the respective vendors, with 88 vulnerabilities assigned CVE IDs.

In summary, we make the following contributions:

- **Novel Design:** We propose NÜWA, a novel static analysis technique that, for the first time, identifies constraint semantic inconsistencies in embedded systems and leverages them to discover vulnerabilities.
- **Superior Performance:** We implemented NÜWA and evaluated it on a dataset of 31 known vulnerabilities. NÜWA outperforms five state-of-the-art tools in both vulnerability detection and precision.
- **Real-world Impact:** We applied NÜWA to analyze 38 embedded devices, uncovering 152 previously unknown vulnerabilities. All vulnerabilities were confirmed by vendors, with 88 assigned CVE IDs.

2 Motivating Example

2.1 Threat Model

This paper focuses on attacks targeting web services in embedded systems. Such attacks require the presence of a vulnerability in the web service of an IoT device, which is known to the attacker. We assume that the attacker can directly communicate with the back-end of the web service over a local area network (LAN) or a wide area network (WAN). By sending crafted malicious inputs, the attacker can exploit the vulnerability, potentially leading to severe consequences such as Denial of Service (DoS) or Remote Code Execution (RCE). This study aims to provide white-hat professionals with a static analysis technique to detect vulnerabilities in firmware web services caused by missing constraints or inadequate input validation.

2.2 Example Vulnerabilities

Figure 1 illustrates an example of front-end and back-end code in a D-Link router, focusing on configuring a parameter named `pskname`. Typically, users modify this parameter via the front-end interface. The process begins with the user providing an input value through the front-end table. The front-end performs validation to ensure the input does not exceed 200 characters and consists only of alphanumeric characters. If these constraints are met, the input is encoded as a base64 string and transmitted to the back-end function `sub_42AA54`. Upon reaching the back-end, the encoded input is decoded using the base64 decoding function (from `sub_42E5D0` to `decode`). The decoded input is then subjected

[2] *Explicit constraints* and *desired constraints* are terms introduced for clarity in our methodology. Explicit constraints refer to those explicitly defined in the code, while desired constraints denote necessary validations analogous to sanitization checks.

[3] NÜWA is named after a goddess in ancient Chinese mythology known for mending holes in the sky. The holes symbolize the missing constraints in back-end systems, inspiring the name of our technique.

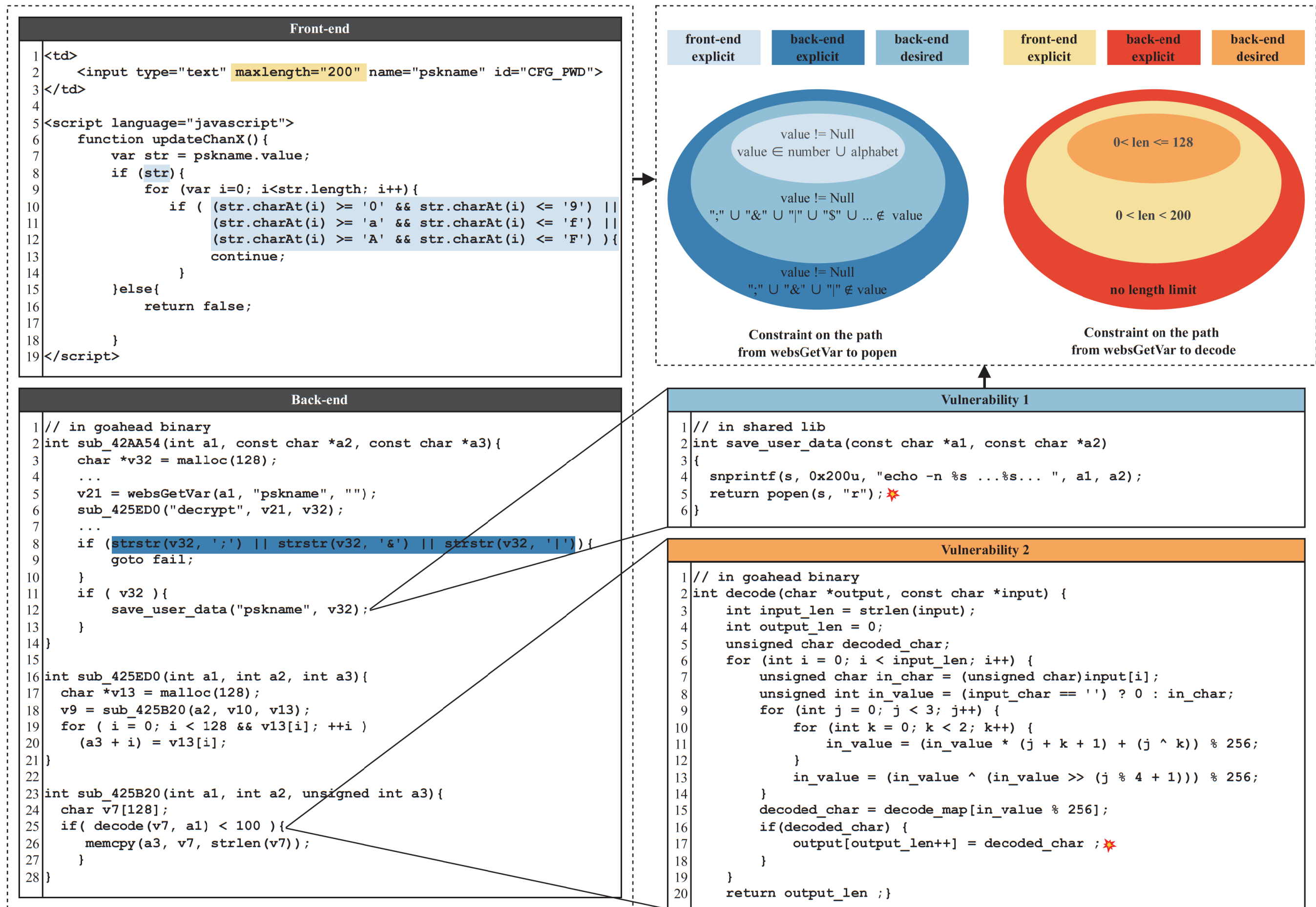

Figure 1: Motivating Example

to an additional validation check (line 8 in the back-end code). If this validation succeeds, the input is saved to the router using the save_user_data function.

The back-end code of this web service contains two critical vulnerabilities. First, the save_user_data function employs popen to execute the echo command for storing user input. This approach introduces a significant risk of arbitrary command execution because the input is not properly sanitized. Second, the decode function stores the decoded user input in a 128-byte buffer (v7 at line 24 of the back-end code). If the input exceeds 128 bytes, it can cause a buffer overflow, potentially leading to severe security breaches.

Both vulnerabilities arise from missing or incomplete input validation in the back-end. For the first vulnerability, the back-end developers implemented a rudimentary check for special characters such as ;, &, and |, to filter some malicious inputs. However, this check is inconsistent with the front-end validation, which strictly limits user input to alphanumeric characters. An attacker bypassing front-end validation by directly sending HTTP requests to the back-end API can exploit this inconsistency. For instance, a crafted input containing the $ character (e.g., test$(touch /tmp/test)) could trigger arbitrary command execution via the popen function. For the second vulnerability, the back-end implicitly assumes that user input will not exceed 128 bytes, as it is stored in a 128-byte buffer. However, this assumption is not enforced through explicit validation. Additionally, the front-end length check is insufficiently strict, allowing an attacker to input data exceeding the buffer size, resulting in a buffer overflow vulnerability.

2.3 Limitations of Existing Techniques

Existing static analysis techniques struggle to handle the example in Figure 1, often resulting in false negatives or false positives depending on the strictness of the predefined analysis rules.

For taint analysis-based techniques [11,30,49], defining appropriate taint propagation rules is challenging. For instance, in vulnerability 2, the buffer index output_len depends on both input_len and the value of decoded_char, which itself is influenced by complex interactions. Accurately modeling how decoded_char is determined is difficult. Loosely defined prop-

agation rules may fail to track whether the write position of the array changes across loop iterations, leading to numerous false positives, such as writes to the array decode_char. Conversely, overly strict rules may prevent the detection of vulnerabilities, as indirect propagation through global variables like decode_map might incorrectly suggest that decoded_char is untainted. Additionally, the modeling of branch and loop iterations further complicates the analysis [25].

Symbolic execution-based techniques [4–6] face different challenges, such as path explosion in the presence of long call chains and nested for loops. Moreover, for vulnerability 1, detecting the issue would require symbolic execution to rely on predefined *sanitizers* to identify potential command injections when execution reaches the popen function. Hence the vulnerability detection performance is also dependent on the quality of the sanitizers.

In summary, existing techniques rely on precise analysis to detect vulnerabilities while minimizing false positives. However, achieving this balance between preciseness and scalability remains challenging and is often unattainable in real-world scenarios.

2.4 Observations

From these two vulnerabilities, it can be observed that a set of *back-end desired constraints* could effectively prevent such issues by guarding dangerous operations (e.g., invoking popen or writing to memory pointers). These constraints need not enforce feature-based restrictions; instead, they focus exclusively on securing critical operations to mitigate potential vulnerabilities. Based on this observation, the constraints in these web services can be categorized as follows:

- **Explicit Constraints** are rules and conditions explicitly implemented for user inputs. These rules are defined through HTML attributes, such as the maxlength attribute in the <input> element, and conditions enforced in the code using constructs like if statements.
- **Desired Constraints** are implicit requirements that are not explicitly implemented in the code but are essential for preventing vulnerabilities. These constraints focus on securing operations that pose security risks. For instance, when using the popen function, it is crucial to ensure that inputs do not include special characters that could lead to command injection.

User inputs may satisfy different types of constraints. Let S denote the set of all possible input strings. We define S_{fc} as the set of inputs that satisfy the front-end explicit constraints, S_{ec} as the set that satisfies the back-end explicit constraints, and S_{dc} as the set that meets the back-end desired constraints. Then, for $\forall s \subseteq S$:

- If $s \notin S_{ec}$, the input is considered invalid. As shown in Figure1, when the input pskname contains the special character &, it is not processed further.
- If $s \in S_{ec} \cap S_{dc}$, the input is considered safe. That is, when the input pskname contains none of the special characters ;, &, or |, and its length does not exceed 100, it is handled normally without introducing potential vulnerabilities.
- If $s \in (S_{ec} \setminus S_{dc}) \vee (S_{ec} \setminus S_{fc})$, the input is potentially risky. For instance, if the input pskname contains the special character $ or exceeds 200 characters in length, it may trigger command injection or buffer overflow vulnerabilities.

Inconsistencies between back-end explicit and desired constraints, as well as between front-end and back-end explicit constraints, can both indicate potential vulnerabilities (e.g., vulnerabilities 1 and 2 in Figure1). However, they differ in detection precision and coverage. Inconsistencies between back-end explicit and desired constraints are typically more precise, as each sensitive operation in the back-end generally corresponds to a specific desired constraint. However, this precision hinges on accurately identifying sensitive operations and may not generalize across diverse vulnerability types. For instance, due to vendor-specific implementations, operations related to SQL injection or stored XSS are often difficult to systematically identify. In contrast, inconsistencies between front- and back-end explicit constraints offer broader coverage, particularly for cases where front-end validations are not enforced back-end. Yet, these inconsistencies are less precise and may require additional analysis to confirm, as in the case of vulnerability 1 in Figure1.

Focusing on detecting constraint inconsistencies in execution paths can reduce the dependence on precise path-level analysis for vulnerability detection. Leveraging information about constraint inconsistencies helps mitigate the limitations of existing analysis techniques. However, extracting explicit constraints from web services and inferring the corresponding desired constraints necessitates further investigation and analysis. Applying constraint inconsistency checking to embedded devices requires addressing two primary challenges:

- **Challenge 1. Accurate Identification of Constraint Semantics Across Diverse Implementations.** Constraints implemented in diverse programming languages or forms may represent the same underlying semantics. A critical challenge is consistently identifying and unifying semantically equivalent constraints for each source, even when they are expressed in varied forms.

- **Challenge 2. Accurate Extraction of Constraint Semantics for Source-Sink Paths.** In back-end code, a single source may propagate to multiple sinks, each governed by distinct constraints. Extracting constraint semantics for all potential source-sink paths prematurely may result in numerous false positives. The challenge lies in accurately identifying and extracting constraint semantics specific to each source-sink path, ensuring precise vulnerability detection while avoiding irrelevant or redundant information.

Table 1: Vulnerabilities Dataset and Constraint Inconsistency Analysis Results.

CWE-Type	CWE-119/125/787	CWE-77	CWE-134	CWE-79	CWE-476	CWE-415/416	CWE-22	CWE-190	CWE-259	Total
#Collected	2014	1654	20	108	44	38	26	33	11	3,948
#Sampled	156	112	6	18	6	9	5	9	3	324
#Inconsistency-Related	138	112	6	12	6	0	3	4	0	281 (86.73%)

Table 2: Missing Constraints for CWE Vulnerabilities.

CWE	Type	Missing Constraints	Semantic Type
CWE-22	Path Traversal	Particular Characters	Content-releated
CWE-77	Command Injection	Particular Characters	Content-releated
CWE-79	Cross-site Scripting	Particular Characters	Content-releated
CWE-119	Buffer Overflow	Length Range	Length-releated
CWE-125	Out-of-bounds Read	Read Length Range	Content/Length-releated
CWE-134	Controlled Format String	Particular Characters	Content-releated
CWE-190	Integer Overflow	Integer Value Range	Content-releated
CWE-476	NULL Pointer Dereference	Null Value	Content-releated
CWE-787	Out-of-bounds Write	Write Length Range	Content/Length-releated

Table 3: Representations for Constraint Semantics.

Semantic Type	Iuput Type	Representation	Description
Content-related	string	<not_null, True/False>	Whether the input is null
	string	<fix, value>	The fixed value
	string	<include, character>	The characters contained
	string	<excluded, character>	The characters not included
	number	<num, [min, max]>	The value range
Length-related	string	<len, [min, max]>	The length range

3 An Empirical Study of Constraints

To systematically examine the characteristics of constraints in the web services of embedded systems and understand how constraint inconsistencies contribute to vulnerabilities, we conducted an empirical study.

Dataset Analysis. We collected 3,948 vulnerabilities from the CVE records of 18 IoT device vendors over the past three years [33]. Given the impracticality of manually reviewing the entire dataset, we applied Cochran's formula [17] to determine an appropriate sample size for each CWE (Common Weakness Enumeration) category. A total of 324 vulnerabilities were selected for manual analysis, as detailed in TABLE 1. Our analysis revealed that 281 (86.73%) of these vulnerabilities were associated with constraint inconsistencies, underscoring the potential of leveraging these inconsistencies to detect a substantial proportion of vulnerabilities.

Constraint Semantics. As illustrated in Figure 1, constraints can take various forms, such as HTML attributes or conditional statements. Despite these syntactic differences, constraints may exhibit the same behavior, referred to as constraint semantics. For example, `maxlength=128` and `strlen(input )<=128` differ in syntax but share identical semantics. When discussing constraint inconsistencies, we specifically refer to inconsistencies in constraint semantics, not their forms. Therefore, it is crucial to unify semantically equivalent constraints into a consistent representation to facilitate accurate analysis.

Constraint Semantics Types. To develop a proper representation for constraint semantics, it is essential to first understand their types. We categorized the CWEs from our dataset, focusing on those related to inconsistent constraints. As shown in TABLE 1, CWEs such as CWE-259, CWE-415, and CWE-416, which are unrelated to constraint inconsistencies, were excluded from the analysis. We then identified and labeled the missing constraints contributing to these CWEs, acknowledging that some vulnerabilities may arise from factors beyond constraint inconsistencies. As detailed in TABLE 2, the missing constraints were categorized into two types: Content-related constraints specify which elements should or should not be included in the input, preventing vulnerabilities like command injection. On the other hand, length-related constraints focus on input length, mitigating vulnerabilities such as buffer overflows.

Constraint Semantics Representation. Based on the types of constraint semantics and input data, we found that constraint semantics can be represented as `<name, value>`, as shown in TABLE 3. When extracting constraint semantics, it is critical to consider the operations guarded by the constraints. Constraints can be categorized as either *positive constraints* or *negative constraints*. A positive constraint allows input processing when the condition is met, whereas a negative constraint triggers error handling under the same circumstances. For example, a constraint limiting input length to a threshold t can appear as `if (len < t){ process input; }`, a positive constraint, or `if (len >= t){ goto error; } else { process input; }`, a negative constraint. To maintain consistency, we negate all negative constraints during processing, aligning them with a unified positive logic representation.

4 Methodology

Figure 2 depicts the overview of NÜWA, which takes a firmware image as input and outputs constraint semantic inscnnsistency alerts. NÜWA works in four steps: ❶ *Preprocessing.* The preprocessor extracts front-end files (HTML, JavaScript), back-end files (HTTP handling programs, shared libraries) and identifies user input information, recorded as a URI-key mapping [49]. ❷ *Front-end Constraint Semantic Extraction.* NÜWA locates URIs and input keys in the front-end using a method similar to SATC, and extracts explicit constraint semantics associated with user inputs from HTML attributes and JavaScript conditions. ❸ *Back-end Constraint Semantic Extraction.* Based on URI-keys mappings, NÜWA constructs the call graph of each URI handling function and analyzes summaries of directly or indirectly invoked functions, where each summary depends on its callees. It then performs forward slicing to eliminate irrelevant code and backward

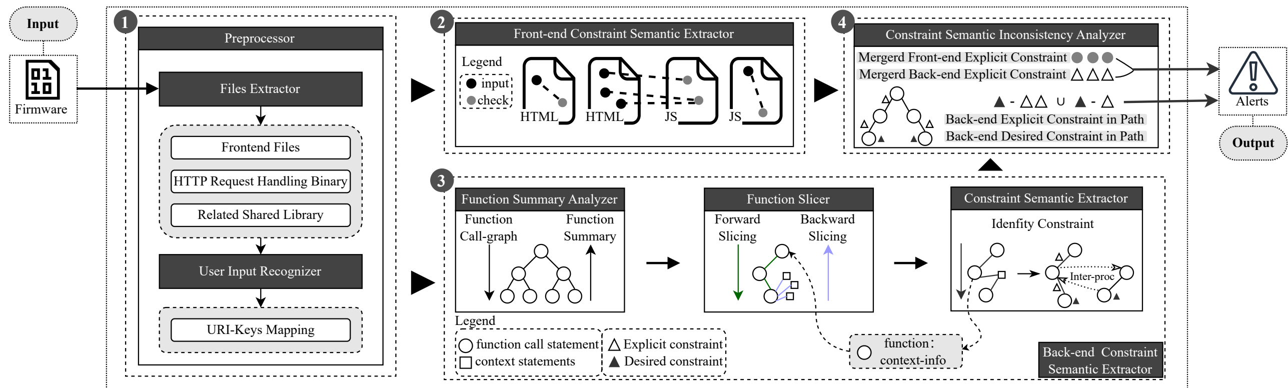

Figure 2: The overview of NÜWA

slicing to extract context from the control flow graph (CFG). With the sliced CFG, NÜWA extracts explicit and desired constraint semantics and identifies the functions required inter-procedural analysis. These functions are iteratively sliced and analyzed. All input-related back-end constraint semantics are recorded within the inter-procedural control flow graph (ICFG). ❹ *Static Analysis on Constraint Semantic Inconsistency.* NÜWA identifies two categories of constraints semantic inconsistencies and reports them as alerts.

NÜWA effectively addresses the two aforementioned challenges. To address challenge 1, NÜWA designs targeted rules to identify constraints and express them in unified constraint semantics for different constraint implementations in the front-end and back-end. To address challenge 2, NÜWA designs a novel binary analysis framework that extracts explicit and desired constraint semantics within the ICFG, enabling path-sensitive merging and comparison.

4.1 Preprocessing

Firmware Unpack and Files Extraction. NÜWA use unpacking tools [28,32] to extract the file system from firmware. The preprocessor traverses the file system and matches file types to extract these front-end files, including HTML files, XML files, and JavaScript files. For back-end programs handling HTTP requests, NÜWA employs a method similar to SATC [4] to extract relevant programs and considers programs in the same folder as the front-end files as handling programs and gathers related shared libraries from the ELF headers [29].

User Input Recognition. NÜWA leverages the source extractor of LARA [49] to extract user input from the firmware, classifying inputs as URI–key pairs. User input is classified as hidden or non-hidden depending on whether it is detectable in the front-end. For hidden sources, NÜWA extracts constraint semantics from both front-end and back-end; For non-hidden sources, only from the back-end.

4.2 Front-end Constraint Semantic Extraction

Front-end explicit constraints are implemented via HTML attributes and JavaScript conditional statements, typically triggered directly or through HTML elements attributes. NÜWA identifies relevant JavaScript handling code by locating shared keywords [4] (e.g., pskname in Figure 1) and extracts constraints from associated conditional statements.

Explicit Constraints in HTML. HTML constraints are implemented in three primary forms. First, validation-related attributes within the <input> element, encompassing eight standard classes [26], are translated into corresponding constraint semantics. Such as maxlength in Figure 1 was transformed into <len,[0,200]>. Second, the <select> element restricts inputs to predefined options, enforcing fixed-value constraints. Third, certain non-user-controlled fields (e.g., input indices) are included in the payload and transmitted to back-end programs. For example, if the input value is 'wan' or 'lan', the corresponding constraint semantics are <fix, ['wan', 'lan']> and <len, [3, 3]>, respectively.

Explicit Constraints in Conditional Statements. In addition to HTML-based constraints, most explicit constraints are implemented via conditional statements [24]. These constraints consist of one or more conditions combined using logical operators. NÜWA analyzes each condition individually and merges them as necessary. A condition may contain only operands (e.g., variables, function calls), as in if(v32) or if(strstr(v32, ';')) , or both operators (e.g., numeric and comparison operations) and operands, as in if(decode(v7, a1)<100). NÜWA analyzes the semantics of the operands first, then evaluates expressions involving arithmetic operations, and finally processes comparison and logical operations. ❶ *Variables and function calls.* For standard library function calls (e.g., isalnum), NÜWA maps them to predefined semantics. For custom functions, it performs inter-procedural analysis to extract constraint semantics. When analyzing variables, NÜWA determines whether they are assigned from conditional expressions (e.g., a = x > y) or function calls (e.g., a = strlen(x)), and extracts semantics

accordingly. Otherwise, it assesses whether the variable represents input content or length. ❷ *Arithmetic operations.* If the operand is a string or number, it is returned without modification. If it is a variable or function call, a corresponding analysis is conducted. Subsequently, the constraint semantics are determined according to the operator and the relationship between the variable and the input. ❸ *Comparison operations.* Comparison operators (e.g., ==) directly encode constraints. Similar to arithmetic operations, NÜWA determines the semantics of these operators based on the operand types and the operators. ❹ *Logical operations.* Logical operators (e.g., &&, ||) combine multiple constraints. NÜWA extracts constraints from each sub-condition and then merges them based on the logical context.

Furthermore, determining whether explicit constraints are positive is also crucial. In JavaScript, the return value typically indicates the outcome of a condition. Therefore, when return True or does not impact the construction of the HTTP request packet, the corresponding constraint semantics are considered positive. Otherwise, they are considered negative.

Algorithm 1: Extract Function Summary

Input: $func$ for the function to be analyzed
S for the obtained function summaries.
Output: $info$ for the summary of function to be analyzed.

1 $AST, info \leftarrow$ Decompiler($func$);
2 **while** *visit(AST)* **do**
3   **if** *is_assign(expr)* $\lor$ *is_call(expr)* **then**
4   **if** *expr.src.is_par* **then**
5       $info[expr.src] \xleftarrow{+} expr.dest$;
6   **if** *expr.dest.is_var* **then**
7         $info[expr.dest] \xleftarrow{+} expr.src$;
8   **if** *expr.src.is_var* **then**
9   **if** *expr.dest.is_par* **then**
10         $info[expr.dest] \xleftarrow{+} expr.src$;
11   **if** *expr.dest.is_var* **then**
12   // Which parameter the variable is derived from ;
13         $from_par \leftarrow expr.src$.from() ;
14         $info[expr.dest] \xleftarrow{+} from_par$;
15         $info[from_par] \xleftarrow{+} expr.dest$;
16   **if** *is_loop(expr)* $\land$ *is_len(expr.count_var)* **then**
17     $info[expr.condition] \xleftarrow{+} expr.count_var$;
18   **if** *is_return(expr)* **then**
19     $info[expr.src] \xleftarrow{+} 'ret'$;

4.3 Back-end Constraint Semantic Extraction

Explicit and desired constraints reside in the back-end programs and shared libraries, with most explicit constraints spanning multiple functions or binaries. Moreover, extracting constraint semantics requires tracking the values of relevant variables and expressions, and not all constraints are input-related. To address this, NÜWA employs code slicing to extract input-related constraint semantics along with their defining values. However, iteratively analyzing callee functions during slicing can be time-consuming. To improve efficiency, NÜWA generates summarizes of relevant callee functions, enabling intra-procedural slicing without extensive inter-procedural analysis. This subsection first presents the methods for function summary analysis and code slicing, followed by the procedure for extracting constraint semantics.

4.3.1 Function Summary Analyzer

Most existing techniques [6, 30, 31] adopt cloning-based inter-procedural analysis, which suffers from two major limitations: analysis timeouts that can cause the entire process to fail, and repeated analysis of the same process increases overhead. To address these issues, NÜWA adopts summary-based inter-procedural analysis [13], where functions are represented by dependencies among parameters. For example, the summary <0:[], 1:[0]> for strcpy indicates that the first parameter (dest) is fully determined by the second (src), as src's content overwrites that of dest. Similarly, strcat(dest, src) is summarized as <0:[0], 1:[0]>, meaning dest is influenced by both its original value and src, which is appended to it.

Analyzing the summary of a URI handling function involves two main steps. NÜWA constructs a top-down function call graph rooted at the specified URI handling function, encompassing all directly and indirectly invoked functions. The leaf nodes correspond to either standard library functions, which have predefined summaries, or shared library functions, which are recursively expanded until all leaves are resolved to standard libraries. Subsequently, a bottom-up analysis is conducted to derive function summaries based on data and control flow. As detailed in Algorithm 1, NÜWA distinguishes between parameters and variables to track data flow in assignment and function call statements (lines 3–15). When the source is a parameter, it is recorded as the origin of the destination, and any variables assigned to it are tracked. Conversely, when the source is a variable, the affected parameters are noted, and variable-to-variable relationships are updated accordingly. For control flow, if a variable reflects the length of another (e.g., in loops), their association is recorded (line 17). Additionally, parameters or variables used as return values are captured (line 19). Following these rules, the summary of the function decode in Figure 1 is <0:[], 1:[0]>, indicating that the second parameter influences the first.

4.3.2 Function Slicer

The back-end code of embedded systems often handles complex functionality and multiple user inputs, with input-handling logic typically implemented independently. Consequently, direct analysis of the back-end code complicates extracting constraint semantics and associated context. To mitigate this, NÜWA applies forward and backward slicing to simplifythe code to be analyzed.

Forward slicing removes constraints unrelated to the in-

put. Based on input-handling logic, NÜWA identifies tainted variables and marks variables requiring backward slicing. These include variables involved in input-based comparisons, computations, concatenations, assignments, or conditional branches. Backward slicing then traces relevant context, such as assignments, buffer allocations, and related operations, enriching the semantic extraction process.

Some statements retained in the sliced code are not directly influenced by the input but still contribute to semantic understanding. To differentiate these, NÜWA classifies variables as direct or indirect. Direct variables are input-related, and constraint extraction is performed only for them. Indirect variables, though input-unrelated, may carry meaningful constant values or strings. Certain variables may be marked as direct or indirect depending on context. For instance, v32 in Figure 1 is marked as indirect before the call to sub_425ED0 and as direct afterward. Consequently, statements at lines 6, 8, 11, and 12 were included through forward slicing, while the statement at line 3 was recovered through backward slicing.

4.3.3 Constraint Semantic Extractor

NÜWA extracts constraint semantics as content- or length-related based on context, and records their semantics using the representations in TABLE 3. In the back-end programs, explicit constraints are expressed in source handling code and conditional statements, while desired constraints are inferred through sink operations in the code.

Explicit Constraints in Source Handling Code. User input is typically received and handled by specific functions in back-end programs [49]. These functions often include constraints, such as length limitations and default value checks [10]. For example, in the function GetValue shown in Listing 1, the corresponding explicit constraint semantic is <len,[0,length]>, indicating a length constraint. Similarly, the function websGetVar provides a default value when the input is parsed, the corresponding explicit constraint semantic is <not_null, True>, ensuring that the input is not a null value. Function summary can help identify explicit constraints in the source handling code. When the output of a parameter matches the parameter input, it often signifies a constraint, such as a length constraint for non-zero integers or a default value for strings.

Explicit Constraints in Conditional Statements. NÜWA employs a method similar to that used in JavaScript to extract explicit constraint semantics. However, it differs by using various contextual information to determine whether a constraint is positive or negative based on the subsequent code behavior. Statements that alter the control flow, such as goto, break and abort functions, indicate negative constraints. Similarly, error-logging functions that record failure messages also signify negative constraints. Additionally, variable reassignment within a condition, such as replacing the character & with an empty string to correct invalid input, is treated as a negative constraint. In contrast, conditions not followed by

```
1  int GetValue(char *in, char *key, char *out, int len){
2      int output_len = 0;
3      current = StrLookup(in, key);
4       while (*current && output_len < len - 1){
5          *out++ = *current++;
6          output_len++;
7       }
8      return 0;
9  }
10 char *websGetVar(Webs *wp, char *in, char *def){
11   char *sp = hashLookup(wp->vars, in);
12   if ( sp && sp->content.value.integer )
13     return sp->content.value.string;
14   else
15     return def;
16 }
```

Listing 1: Examples of the source handling code.

Algorithm 2: Extract Desired Constraint Semantics in Memory-related Sink Operations

Input: *sink* for the Memory-related sink operations to be analyzed.
Output: *DC* for the set of desired constraint semantics.

1 **if** *is_call(sink)* **then**
2   *src*, *dest* $\leftarrow$ identify_par(*sink*);
3   **if** *is_direct(src)* **then**
4         $DC \xleftarrow{+}$ <len, [0, len($dest$)]>;
5   **if** *is_direct(dest)* **then**
6         $DC \xleftarrow{+}$ <len, [len(src), MAX]>;
7 **if** *is_integer_arithmetic(sink)* **then**
8     $default_range \leftarrow$ Decompiler();
9   // Calculate the range that source should satisfy at each computation;
10     $DC \xleftarrow{+}$ <num, calc($default_range$, $operators$)>;
11 **if** *is_array/global_access(sink)* **then**
12   *array*, *index* $\leftarrow$ identify_par(*sink*);
13   **if** *is_direct(index)* **then**
14         $DC \xleftarrow{+}$ <len, [0, len($array-1$)]>;
15   **if** *is_direct(array)* **then**
16         $DC \xleftarrow{+}$ <len, [len($index+1$), MAX]>;
17 **return** *DC*;

such corrective actions are classified as positive constraints.

Desired constraints are intended to mitigate the vulnerabilities associated with sink operations, which are categorized into memory-related and non-memory-related.

Desired Constraints in Memory-related Sink Operations. Memory-related sink operations are categorized into function-based and non-function-based types. Relevant boundary is obtained from both slicing context and the decompilation output. ❶ For function-based operations, constraint semantic extraction consists of two steps. NÜWA first identifies the source (src) and destination (dest) parameters (line 2). Standard library functions like strcpy are handled using predefined rules, while custom functions (e.g., decode in Figure 1) are analyzed using function summaries. Then, constraint semantics are inferred based on input-parameter correlations. As illustrated

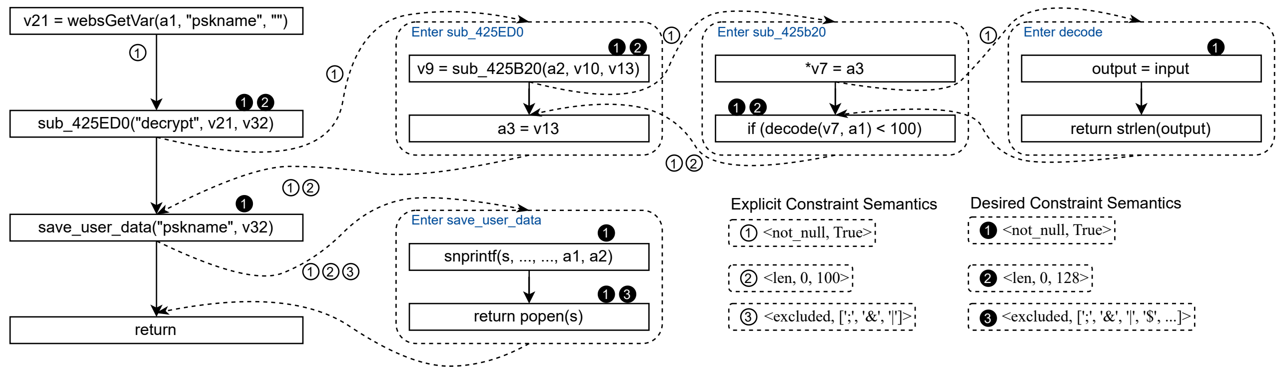

Figure 3: The ICFG of Motivating Example

in Algorithm 2, if src is direct, its content length must not exceed the size of dest (line 4); if dest is direct, its size must exceed the length of src (line 6). ❷ Non-function-based operations include arithmetic, array access, and global variable access. The corresponding constraint semantics are inferred from array and integer boundary information. For arithmetic, the default integer range and operator precedence are applied to prevent overflow at each computation step(lines 8–10). For array or global variable access, if the array variable variable is direct, its length must exceed the value of the index variable (line 14). If the index is direct, its value must remain within the bounds of the array (line 16).

Desired Constraints in Non-memory-related Sink Operations. Non-memory operations typically involve access to external environments or system resources, such as command execution functions (e.g., system, popen) and file operations (e.g., open, fopen). The corresponding constraint semntics are extracted based on prior knowledge summarized from manual experience and vulnerability exploitation analysis.

Constraint semantics are extracted via inter-procedural analysis through the following steps: ❶ *Intra-procedural Analysis.* After deriving function summaries for all nodes in the call graph, NÜWA begins from the root function. Guided by these summaries, it performs forward slicing to eliminate statements unrelated to source handling, followed by backward slicing to preserve those associated with constraint semantics. Both explicit and desired constraint semantics are then extracted from the sliced code. NÜWA also identifies function-specific features (e.g., strlen) to distinguish content-related and length-related variables, enhancing semantic extraction precision. ❷ *Capturing Functions and Execution Contexts for Subsequent Analysis* During inter-procedural analysis, NÜWA records custom functions involved in source handling along with context such as parameter mappings and explicit constraints at call sites. This facilitates more accurate extraction of constraint semantics. ❸ *Recursive Analysis.* NÜWA performs intra-procedural analysis on pending functions using the recorded context and identifies additional functions requiring analysis. This process repeats until all source-handling functions have been analyzed.

In the back-end, different paths from a source to various sinks may involve distinct explicit constraints. For instance, in line 26 of Figure 1, the function decode returns the length of v7. While a length check exists, it is applied after the risky operation, leaving the explicit constraint unenforced along the path to the sink decode, resulting in a buffer overflow alert. In contrast, the length check is enforced along the paths to sinks sub_425B20 and sub_425ED0, reducing false positives. Moreover, some explicit constraints are split across branches following conditionals and require separate handling. To model back-end constraint semantics, NÜWA employs an inter-procedural control flow graph (ICFG), as shown in Figure 3, which depicts back-end constraint semantics stored within the ICFG for the motivating example.

The ICFG serves to delineate the scope of constraint enforcement along source-to-sink paths. Its nodes represent statements that impose explicit or desired constraints, or involve functions requiring inter-function analysis. Each node is annotated with the explicit and desired constraints applicable at that point. Explicit constraints are propagated along the ICFG to model their impact on data flow, while desired constraints are recorded only at the corresponding sink nodes without propagation.

Further details about parsing a source twice along a single path are in Appendix §A.

4.4 Semantic Inconsistency Analysis

NÜWA identified semantic inconsistencies separately between back-end explicit constraints and desired constraints, as well as between back-end and front-end explicit constraints.

To identify semantic inconsistencies between back-end explicit and desired constraints, NÜWA performs path-sensitive analysis over all source-to-sink paths in the ICFG. Let *EC* and *DC* denote the set of back-end explicit and desired constraint semantics. For a given path *P* in the ICFG, we define

Table 4: Operational Rules for Constraint Semantics Sets.

Representation	Constraint Semantic Set Operations			
	Initial Value	$Merge()$	$Diff(EC, DC)$	$Diff(EC, FC)$
not_null	False	True if either value is true	Log discrepancy if only the back-end explicit constraint is false	
fix	$\emptyset$	$\cup$	$DC \setminus EC$	$FC \setminus EC$
include	$\emptyset$	$\cup$	$DC \setminus EC$	$FC \setminus EC$
excluded	$\emptyset$	$\cup$	$DC \setminus EC$	$FC \setminus EC$
num	$[MIN_INT, MAX_INT]$	$\cap$	$EC \setminus DC$	$EC \setminus FC$
len	$[0, MAX]$	$\cap$	$EC \setminus DC$	$EC \setminus FC$

$EC(P) \subseteq EC$ and $DC(P) \subseteq DC$ as the explicit and desired constraint semantics along path P. The inconsistency set $I(P)$ is defined as:

$$I(P) = Diff(Merge(EC(P)), DC(P)) \quad (1)$$

Equation 1 defines NÜWA identifies constraint semantic inconsistencies along a program path P by comparing the merged set of back-end explicit constraints $EC(P)$ with the desired constraints $DC(P)$. A non-empty $I(P)$ indicates that some desired constraints are not enforced by any explicit check, potentially leaving the embedded system vulnerable.

Semantic inconsistencies between front-end and back-end explicit constraints often span different components. To detect them, NÜWA employs a different strategy. The set I is defined to quantify these inconsistencies:

$$I = Diff(Merge(EC), Merge(FC)) \quad (2)$$

Equation 2 defines NÜWA compares the merged front-end explicit constraints (FC) with the merged back-end explicit constraints (EC). A non-empty I indicates missing checks in the back-end programs, suggesting under-validation and inconsistent sanitization that may increase exploitation risk.

In Equation 1 and 2, the operator $Merge()$ denotes a merge over heterogeneous constraint semantics, while the function $Diff()$ captures semantic inconsistencies between constraint sets. As shown in TABLE 3, constraint semantics are expressed in six distinct forms. Accordingly, set operations are applied separately to each type. The processing is guided by two principles: ❶ *Per-representation constraint operations.* Each representation is initialized with a default value (see TABLE 4) indicating the absence of constraints and is updated upon extraction. Supported operations include union ($\cup$), intersection ($\cap$), and difference ($\setminus$). The semantics of $Merge()$ and $Diff()$ are defined in TABLE 4. ❷ *Cross-representation consistency.* The *fix* representation is semantically linked to *include*, *excluded*, *num*, and *len*. Fixing an input to a specific value implies constancy in these related forms. During set operations, implied values from *fix* are propagated to maintain consistency across representations.

4.5 Implementation

We implemented the NÜWA prototype with over 9,800 non-comment lines of Python code. NÜWA employs unblob [28] to extract file systems from firmware images across various vendors. Front-end analysis uses regular expressions to parse HTML and handles JavaScript in JSON format. Back-end static analysis, including function summary analyzer, function slicer, and the constraint semantic extractor, is built on IDA Pro [14] 9.0.20241217(SP1) and the IDALib [15] plugin, supporting multi-architecture analysis.

Vulnerability Validation and Reporting. To assess whether constraint semantic inconsistencies reported by NÜWA may lead to exploitable vulnerabilities, we follow a three-step validation process: ❶ *Validation Environment.* We establish a testing setup using either physical devices or emulation frameworks [2, 18]. ❷ *PoC Construction.* Based on identified semantic inconsistencies, we manually craft packets that satisfy the conditions to trigger the vulnerability. ❸ *Vulnerability Reporting.* If successful, we report the vulnerability with technical details and the PoC. For issues that cannot be validated due to unavailable environments, we generate detailed analysis reports and notify vendors. If the vendor is unresponsive within 90 days, we pursue further validation and disclosure through the CVE program [7].

5 Evaluation

Research Questions. We conducted an evaluation of NÜWA on real-world firmwares with embedded systems to address the following research questions (RQs):

- **RQ1.** How is the detection capability of NÜWA compare with baselines on known vulnerabilities? (§5.2).
- **RQ2.** How effective is NÜWA in extracting constraint semantics and identifying inconsistencies? (§5.3)
- **RQ3.** How effectively is NÜWA in discovering unknown vulnerabilities in real-world embedded systems? (§5.4)

5.1 Evaluation Setup

Evaluation Design. To ensure clarity, fairness and effectiveness, we designed an evaluation approach based on known vulnerabilities, differing from previous methods. Given that the goal of static analysis in embedded systems is to discover vulnerabilities effectively, the number of known vul-

nerabilities detected serves as an intuitive and practical metric. These vulnerabilities span diverse firmware series and vendors, demonstrating the ability to analyze heterogeneous firmware. Each case requires manual verification of input sources and trigger paths to assess accuracy of the results. To minimize the influence of taint source identification, we provide fixed sources for both NÜWA and the baselines. As vendors often reuse firmware across series and share similar code across functionalities, resulting in recurring vulnerabilities [41], we curated a dataset with clear triggering conditions and low code similarity. This focused design enables more meaningful and fair comparisons between tools.
Dataset. We constructed two datasets to evaluate NÜWA: a known vulnerability dataset and a recent firmware dataset.

- **Known Vulnerability Dataset.** The vulnerabilities in this dataset were manually collected from CVE records [7] and exploit-db [34], following these criteria: ❶ Vulnerabilities were selected from embedded device vendors referenced in SOTA research [4, 6, 11, 18, 30, 49, 52]. ❷ Only buffer overflow vulnerabilities (CWE-119) and command injection vulnerabilities (CWE-77) were included, due to limitations in baseline tools. ❸ Vulnerabilities with clear trigger information were prioritized, ensuring practical relevance. ❹ Redundant vulnerabilities, such as highly similar ones (e.g. CVE-2023-34933 and CVE-2022-37095) and those requiring interactions between multiple processes [31], as NÜWA does not focus on these. In total, 31 known vulnerabilities from 13 vendors were included in this dataset.

- **Firmware Dataset.** This dataset consists of the latest firmware from several popular vendors, covering five device types and 38 firmware samples.

Baseline and Configurations. We compares NÜWA with five SOTA tools including SATC [4], EMTAINT [6], LARA [49], MANGO [11], OCTOPUSTAINT [30]. All of them are published in top-tier conferences. We excluded KARONTE [31], as it focuses on vulnerabilities related to multi-process interactions, which are beyond the scope of our focus.

- SATC, which employs shared-keyword-aware source identification and symbolic execution to efficiently track user input and identify back-end vulnerabilities.

- EMTAINT, which incorporates a structured symbolic expression (SSE)-based on-demand alias analysis technique to enhance indirect call analysis.

- LARA, which utilizes the semantic relationship in code and data to identify more sources and sinks to improve the accuracy of taint analysis.

- MANGO, which introduces two novel optimizations, named sink-to-source analysis and Assumed Nonimpact, facilitating precise and large-scale taint analysis.

Table 5: Overall known Vulnerability Detection Results.

	NÜWA	SATC	EMTAINT	LARA	MANGO	OCTOPUSTAINT
Alert	67	31	8	44	33	15
TP	51	16	6	29	15	11
Prec.	76%	52%	75%	66%	45%	73%
#Vuln	28, 12	10, 6	6, 0	22, 7	11, 4	9, 2

- OCTOPUSTAINT, which is based on data dependency graphs to mitigate the state explosion problem and analyzes sanitization measures to reduce the false positive rate.

All tools were executed on a host machine with an 28-core Intel Xeon Processor and 256GB of RAM running the Ubuntu 22.04 operating system.

5.2 RQ1: Known Vulnerability Detection

We evaluated NÜWA, SATC, EMTAINT, LARA, MANGO, and OCTOPUSTAINT using the known vulnerability dataset to assess their vulnerability detection capabilities. To ensure fair and valid comparisons, we applied the following adjustments to NÜWA and all baseline tools: ❶ We manually specified the source or source handling code for each vulnerability to ensure accurate source modeling; ❷ We unified the sink definitions by configuring all tools to use standard library function calls as sinks, enabling consistent evaluation of their inter-procedural and inter-binary analysis capabilities.

NÜWA demonstrated superior performance in vulnerability detection, as summarized in TABLE 5. It identified 67 potential vulnerabilities, of which 51 were confirmed as true positives, achieving a precision of 76%. These true positives revealed 28 known vulnerabilities from the existing dataset, along with 12 additional ones.

In comparison, SATC, EMTAINT, LARA, MANGO, and OCTOPUSTAINT reported 31, 8, 44, 33, and 15 vulnerabilities, respectively, with precisions of 52%, 75%, 66%, 45%, and 73%. Although several tools identified additional vulnerabilities, with LARA detecting the most 7 and OCTOPUSTAINT the fewest 2, these numbers were significantly lower than the 12 additional vulnerabilities detected by NÜWA. Notably, all vulnerabilities identified by the baseline tools were also covered by NÜWA. In summary, NÜWA surpassed all five baseline tools in both detection capability and precision.

As detailed in TABLE 10, the 31 known vulnerabilities from dataset are categorized into intra-function, inter-function, and inter-binary based on their triggering paths. Intra-function vulnerabilities occur within a single function, inter-function vulnerabilities span multiple functions within the same binary, and inter-binary vulnerabilities involve shared libraries, distinct from the multi-process cases addressed by KARONTE [31]. NÜWA successfully detected all 13 intra-function vulnerabilities, whereas SATC, LARA, EMTAINT, MANGO, and OCTOPUSTAINT detected 9, 13, 4, 9, and 8, respectively. Among 10 inter-function vulnerabilities, NÜWA detected 9, while

Table 6: Overall Extraction Results of Constraint Semantics.

Constraints	Semantic Extraction				Constraint Semantic Representations						Halstead Effort of TP		
	#Alert	#TP	#Unique	Prec.	#Not_null	#Fix	#Include	#Exclude	#Num	#Len	Min	Max	Avg
Front-end Explicit Constraint	58	55	41	95%	26	9	8	4	3	5	1	2606	357
Back-end Explicit Constraint	258	222	77	86%	115	8	59	23	6	11	1	2747	66
Back-end Desired Constraint	257	218	77	85%	105	0	0	12	16	85	1	1793	60

Table 7: Constraint Classification.

Constraint	Front-end Explicit Constraint			Back-end Explicit Constraint				Back-end Desired Constraint		
	HTML	HTML+JS	JS	Src	Cont.	Len.	Path	F-Memory	NF-Memory	N-Memory
#	14	15	26	13	112	9	88	76	46	96
% of TP	26%	27%	47%	6%	50%	4%	40%	35%	21%	44%

Src = Source Handling, Cont. = Content-related, Len. = Length-related, Path = Path Condition, F-Memory = Function-based Memory sinks, NF-Memory = Non-Function-based Memory Sinks, N-Memory = Non-memory Sinks.

SATC, EMTAINT, LARA, MANGO, and OCTOPUSTAINT detected 1, 2, 6, 2, and 1, respectively. For 8 inter-binary vulnerabilities, NÜWA and LARA detected 6 and 3 vulnerabilities, respectively, while the others failed entirely. These results further demonstrate that NÜWA outperforms existing baseline tools in both inter-procedural and intra-procedural analysis.

FN Analysis for NÜWA. We analyzed 3 cases missed by NÜWA and summarized 2 main reasons. ❶ Disassembly Engine Limitations. In CVE-2024-2713, incorrect identification of function parameters prevented accurate analysis. In EDB-42344, improper variable decompilation prevented valid data flow tracking, leading to missed constraint extraction. ❷ Complex Input Structure. In CVE-2023-27997, the input comprises two components: size and data. The size determines buffer allocation, and the data is written into the buffer. A buffer overflow occurs if the constraint between size and data is violated. However, NÜWA is unable to infer relationships between multiple input components, thus failing to detect the constraint inconsistency required to identify the vulnerability.

FN Analysis for Baselines. Our manual analysis of false negatives in 5 baselines revealed 3 main reasons. ❶ Loop-Based Assignment Vulnerabilities. The dataset includes 7 such cases, yet EMTAINT detecting only 1 and LARA 3. As discussed in §2.4, existing tools often struggle with these vulnerabilities due to challenges like path explosion. ❷ Limited Inter-Binary Analysis. LARA identifies wrapper functions in shared libraries as sinks, whereas others rely on manually specified sinks. This limitation results in missed sinks, thereby causing missed vulnerabilities. ❸ Complex Code Structures. Some tools fail to complete analysis due to timeouts triggered by complex code, further underscoring the effectiveness of constraint semantic inconsistency analysis in such scenarios.

5.3 RQ2: Comprehensive Analysis of Output

The output of NÜWA includes explicit and desired constraint semantics extracted from font-end files and back-end programs, along with any identified semantic inconsistencies between them. To evaluate the accuracy of the extracted constraints, we applied NÜWA to 28 known vulnerabilities it can detect. TABLE 6 summarizes the overall constraint semantics extraction results. NÜWA extracted 58 front-end explicit constraints semantics, 258 back-end explicit constraint semantics and 257 back-end desired constraint semantics, whth precisions of 95%, 86%, and 85%, respectively. Certain constraint semantics (e.g., <not_null, True>) can be extracted from multiple locations in the code, leading to redundancy. After manual deduplication, we identified 41, 77 and 77 unique constraint semantics in the front-end explicit, back-end explicit, and back-end desired categories.

TABLE 7 details the classification of constraint semantics extracted by NÜWA. NÜWA derives front-end explicit constraint semantics from both HTML and JavaScript sources. Of the 55 correctly extracted constraints, 14 originate from HTML, 26 from JavaScript invoked within HTML, and 15 from standalone JavaScript code. Among the 222 back-end explicit constraint semantics, 13 come from source handling code, while the rest are from conditional statements. These conditionals are categorized into length checks, content checks, and path conditions. Although path conditions also involve input length and content, they primarily serve program logic rather than mitigate security risks. Manual analysis identified 112 content-related, 9 length-related, and 88 path-condition constraints. For the 218 back-end desired constraints semantics inferred by NÜWA, 76 are associated with function-based memory-related sink operations, 46 with non-function-based memory-related sink operations, and 96 with non-memory-related sink operations.

Constraints Complexity. To further assess the effectiveness of constraint extraction, we analyze three key dimensions: Halstead effort of relevant statements, the number of statements per constraint, and the call depth from source to the constraint location. ❶ Halstead effort. Halstead effort [39] measures the complexity of analyzing constraint-related statements. For instance, a multi-character check `if (strchr(s, '<')|| strchr(s, '>')|| strchr(s, '&')|| strchr(s, '#'))` yields an effort of 100. As shown in

TABLE 6, front-end explicit constraints exhibit higher average effort than back-end constraints, as decompilation tends to decompose complex statements into multiple simpler ones. Notably, both front-end and back-end cases contain highly complex statements. We observe 13 constraint-related statements with an effort exceeding 1000, suggesting that while such complexity exists, it remains manageable. These results also demonstrate that NÜWA is capable of analyzing even highly complex constraint expressions. ❷ Statement Count. This metric reflects the number of statements required to extract each constraint semantic. Among the correctly extracted results, 33 back-end explicit and 82 back-end desired constraint semantics span multiple statements, with some requiring up to 7. These findings underscore the effectiveness of code slicing in supporting precise semantic extraction. ❸ Call Depth. Call depth captures the number of inter-procedural depth between the source and the constraint location. We observe that 179 back-end explicit and 130 back-end desired constraint reside in functions different from where the source is handled, with 16 spanning across 5–6 functions. This underscores the strength of NÜWA in supporting deep constraint semantic extraction.

False Positives in Constraint Semantic Extraction. As shown in TABLE 6, NÜWA produces several inaccurate constraint semantics, including 55 constraint false positives and 23 semantic false positives. The majority of constraint false positives arise from the loss of data and context semantics, as well as variable over-tainting. Due to stripping in embedded system binaries, structure member accesses often appear as pointer-offset expressions, which may be misclassified as computation-related sensitive operations. Moreover, the lack of context information can result in certain operations being mistakenly identified as a sensitive operation. For example, the operation `v3[12]` is associated with the desired constraint `len(v3)> 12` to prevent an out-of-bounds read, even though `v3` may already contain data of sufficient length. Similar misclassifications occur in formatted parameter handling. A small number of false positives also stem from disassembly errors and inaccurate function summaries, leading to over-tainting. Semantic false positives primarily involve misinference of variable data types or content, and incorrect classification of constraint semantics. A single variable may represent the entire input or only a subset, causing confusion between constraint representations such as *include* and *fix*. Additionally, some logging functions record both normal and abnormal events, leading to positive constraints being misidentified.

TABLE 8 presents the detailed results of constraint semantic inconsistency identification conducted by NÜWA, based on source form 28 known vulnerabilities. For semantic inconsistencies between back-end explicit constraints and desired constraints, NÜWA reported 147 alerts corresponding to 53 unique inconsistencies, of which 39 were validated as true positives. Each TP represents a distinct vulnerability, revealing 28 vulnerabilities of the dataset and 11 additional ones.

Table 8: Results of Semantic Inconsistency Analysis.

Source From	*Diff* (EC, DC)			*Diff* (EC, FC)			
	#Alert	#Unique	#TP	#Alert	#Unique	#TP	#Vuln
CVE-2022-43000	2	2	1	3	3	2	$1 \rightarrow 1$
CVE-2022-25106	4	2	2	1	1	1	$1 \rightarrow 2$
CVE-2022-46566	7	2	1	0	0	0	0
CVE-2023-29665	16	3	2	0	0	0	0
CVE-2023-38933	2	1	1	0	0	0	0
CVE-2023-50983	2	2	1	1	1	1	$1 \rightarrow 1$
CVE-2024-0538	3	2	2	0	0	0	0
CVE-2023-51099	2	2	2	3	3	3	$3 \rightarrow 2$
CVE-2022-37805	1	1	1	0	0	0	0
CVE-2023-26806	1	1	1	1	1	1	$1 \rightarrow 1$
CVE-2022-25130	2	2	2	0	0	0	0
CVE-2023-26978	1	1	1	0	0	0	0
CVE-2024-0579	2	2	2	0	0	0	0
CVE-2024-0296	2	2	1	0	0	0	0
CVE-2023-46977	3	2	1	1	1	1	$1 \rightarrow 1$
CVE-2023-39550	1	1	1	0	0	0	0
CVE-2021-45756	3	2	1	0	0	0	0
CVE-2023-50361	1	1	1	0	0	0	0
CVE-2024-27129	1	1	1	0	0	0	0
CVE-2024-53703	1	1	1	0	0	0	0
CVE-2023-27806	1	1	1	0	0	0	0
CVE-2023-34933	30	2	1	0	0	0	0
CVE-2023-33538	1	1	1	1	1	1	$1 \rightarrow 1$
CVE-2023-31701	3	3	2	0	0	0	0
CVE-2020-10825	41	5	2	0	0	0	0
CVE-2023-24229	7	3	2	0	0	0	0
CVE-2023-20117	5	3	2	2	2	1	$1 \rightarrow 1$
CVE-2023-31741	2	2	2	1	1	1	$1 \rightarrow 1$
Total	147	53	39	14	14	12	$11 \rightarrow 11$

For semantic inconsistencies between front-end and back-end explicit constraints, NÜWA reported 14 unique alerts, with 12 validated as TPs. In contrast to the previous category, these TPs do not always map one-to-one with specific vulnerabilities. The final column of TABLE 8 summarizes the mappings between TPs and vulnerabilities, highlighting three key observations: ❶ One semantic inconsistency may reveal multiple vulnerabilities. For example, in CVE-2022-25106, the front-end enforces a fixed input value, so inputs with different contents or lengths may lead to different vulnerabilities. ❷ Multiple semantic inconsistencies may correspond to a single vulnerability. In CVE-2023-51099, the input is a ping command followed by an IP address, and front-end constraints over the command string and IP format both indicate a potential command injection. ❸ Some semantic inconsistencies only affect the normal system functionality. In CVE-2022-43000, specific inputs may prevent Wi-Fi connectivity but do not lead to further exploitation. Overall, semantic inconsistencies between front-end and back-end explicit constraints revealed 10 vulnerabilities of the dataset and one XSS vulnerability that could only be uncovered through this approach.

False Positives in Constraint Semantic Inconsistencies. False positives of constraint semantic inconsistencies stem from incorrect constraint inference and path condition constraints. All 14 false positives in semantic inconsistencies between back-end explicit and desired constraints resulted from misidentified back-end constraints. For semantic inconsistencies between front-end and back-end explicit constraints, one false positive caused by incorrectly extracted front-end constraint semantics, and the other is due to path conditions related to input encoding, which poses no security risk.

Table 9: Unknown Vulnerability Discovery Results.

Vendor	#Series	#Unknown Vuln	SOTA Tools				
			SATC	EMTAINT	LARA	MANGO	OCTOPUSTAINT
DLink	6	17	0	0	6	2	2
Tenda	6	9	1	3	4	4	2
Zyxel	8	1	0	0	1	1	1
QNAP	4	68	0	2	25	8	0
Draytek	4	39	0	1	16	1	0
TOTOLink	2	7	1	1	4	1	3
Linksys	3	9	0	0	5	2	2
Trendnet	5	2	0	0	1	0	1
ToTal	38	152	2	7	62	19	11

5.4 RQ3: Unknown Vulnerability Discovery

We also applied NÜWA to detect unknown vulnerabilities in the firmware dataset in the wild. Each vulnerability was verified in the latest version of the firmware sample and confirmed by the vendor. As illustrated in TABLE 9, NÜWA discovered a total of 152 previously unknown vulnerabilities. 88 of them have been assigned CVE IDs following responsible disclosure. These vulnerabilities involve all other types of vulnerability in the TABLE 2, excluding integer overflow. Among these vulnerabilities, 7 are related to loop-based assignments, including AES decoding, Base64 decoding, and others; 3 are path traversal vulnerabilities; 5 are null pointer dereference vulnerabilities; and 1 is a cross-site scripting vulnerability. Furthermore, semantic inconsistencies between front-end and back-end explicit constraint allowed NÜWA to detect 57 vulnerabilities, of which 3 were exclusively identifiable through this inconsistency-based approach.

Meanwhile, SOTA tools were applied to analyze these firmware samples. Due to its source identification method, LARA is able to detect 62 vulnerabilities, while other tools can only identify fewer vulnerabilities. Furthermore, all vulnerabilities found by SOTA tools can be found by NÜWA.

6 Discussion and Limitation

Web Service Application Scope. Front-end and back-end of embedded systems are typically integrated, enabling comprehensive analysis. In contrast, web services of open-source projects are often decoupled, which increases the difficulty of analysis. While NÜWA is designed for embedded web services (commonly with C/C++ back-ends), our methodology generalizes to non-embedded web services and diverse programming languages.

Firmware-unpack Impact. Binwalk [32] and Unblob [28] are the most widely used tools for firmware unpacking, both relying on underlying libraries to support diverse firmware formats. Compared to Binwalk, Unblob provides more convenient usage and integrates automated analysis for certain encrypted firmware, reducing manual effort. Leveraging Unblob, NÜWA can automatically analyze vulnerabilities such as CVE-2024-27129 and CVE-2024-27130 with minimal human intervention.

Approach vs Implementation. Binary analysis relies on the capabilities of reverse engineering tools. Among common tools, IDA Pro [14] provides superior decompilation support. While NÜWA and baselines adopt different underlying engines (e.g. MANGO with angr and LARA with IDA Pro), the superior performance of NÜWA stems primarily from its methodology rather than implementation.

Threat to Validation. First, NÜWA only supports back-end programs developed in C and C++. Second, the precision of the disassembly engine may affect the analysis results. Third, there may be potential errors in the ground truth that has been manually confirmed.

7 Related Work

Static Analysis of Embedded Systems. Due to limited fuzz testing and emulation [20, 52], various static analysis techniques [12, 43, 44] have been developed for vulnerability detection in embedded systems. Source identification is key, with SaTC [4] using shared front- and back-end keywords, and Lara [49] refining this by categorizing inputs and leveraging LLMs. FITS [21] infers source-handling functions from static and dynamic properties but neglects code constraints, which HermeScan [10] partially addresses with length constraints. DTaint [5] and EmTaint [6] resolve indirect calls via pointer aliasing, while HermeScan [10] and Mango [11] improve data flow analysis through reaching definition analysis, value analysis, and data dependency analysis.

Missing-Check Bugs Detection. Missing security checks, common in OS kernels [24], are a significant class of bugs. Early research [23] assumed security checks could be applied to similar code sections, particularly after indirect jumps. Later works [24, 42] combined semantic-aware and context-sensitive analysis to detect bugs in the Linux kernel, while others [19, 37, 46] focused on missing rechecks and permission checks. These methods detect missing constraints but only represent part of the necessary conditions. AMCheX [38] addresses challenges in identifying required security checks but relies on semantic information, limiting its applicability in binary analysis. Additionally, some research [22, 35] focuses on bugs in kernel APIs and cloud services, mainly inferring variables needing checks through source identification.

Dynamic Analysis of Embedded Systems. Fuzz, a key vulnerability detection technique, testing focuses on packet mutation and execution feedback through black-box and gray-box approaches. Black-box fuzzing tools like IoTFuzzer [3] and SRFuzzer [47] target web interfaces of embedded devices, while SNIPUZZ [8] and LABRADOR [20] utilize response packets and logs to infer execution states, enhancing mutation guidance. In gray-box fuzzing, Firm-AFL [51] leverages QEMU for high-throughput coverage-guided fuzzing, balancing user-mode efficiency with system-mode completeness. Tools like FirmFuzz [36], FirmAE [18], and NDFuzz [48] improve system emulation and coverage collection, with FIRMAE successfully emulating 1,124 devices.

8 Conclusion

This paper presents NÜWA, a novel static analysis technique that identifies potential vulnerabilities in embedded systems by identifying constraint semantic inconsistencies. NÜWA accurately extracts explicit constraints from both front-end and back-end components, infers desired constraints of the back-end, and identifies semantic inconsistencies indicative of vulnerabilities. Compared to five state-of-the-art tools, NÜWA significantly outperforms them in detecting known vulnerabilities, with precision rates of 95%, 86%, and 85% for front-end explicit constraints, back-end explicit constraints, and back-end desired constraints, respectively. Furthermore, NÜWA identified 152 unknown vulnerabilities, all of which have been confirmed by vendors, with 88 CVE IDs assigned.

9 Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103900, Strategic Priority Research Program of the CAS under Grant #XDCO2030200 and Chinese National Natural Science Foundation (Grants #62032010, #62202462, #62302500)

10 Ethics Considerations

The primary ethical consideration in our work is ensuring that the vulnerabilities identified by NÜWA are responsibly disclosed. We follow the vulnerability disclosure process outlined in §4.5 to notify vendors and relevant organizations [7], aiming to reduce the risk of exploitation by attackers. All vulnerabilities are confirmed in a controlled local environment, ensuring the legitimacy of the investigation. Additionally, our work does not involve human subjects, personal data, or other activities with significant ethical concerns.

11 Open Science

In alignment with USENIX Security's open science policy, we commit to making our research results publicly available [50]. The source code of NÜWA and the datasets will be released under an opensource license, which will allow other researchers to replicate our findings, build upon our work, and further the advancement of embedded system security. However, there may be limitations regarding the release of certain details, especially concerning firmwares with unknown vulnerabilities. We may delay the release of specific firmware dataset information until it is deemed secure. Overall, our commitment to open science aims to balance the need for transparency and reproducibility with the ethical responsibility to protect software systems and users from potential harm.

References

[1] BISHOPFOX. Its 2024 and over 178,000 sonicwall firewalls are publicly exploitable. https://bishopfox.com/blog/its-2024-and-over-178-000-sonicwall-firewalls-are-publicly-exploitable, 2023.

[2] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, San Diego, California, USA, 2016. The Internet Society.

[3] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[4] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 303–319, USA, 2021. USENIX Association.

[5] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[6] Kai Cheng, Yaowen Zheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, Kejiang Ye, and Limin Sun. Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 360–372, Seattle, WA, USA, 2023. ACM.

[7] CVE. Common vulnerabilities and exposures. https://cve.mitre.org/, 1999.

[8] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 337–350, 2021.

[9] Fortiguard. Heap buffer underflow in administrative interface. https://www.fortiguard.com/psirt/FG-IR-23-001, 2023.

[10] Zicong Gao, Chao Zhang, Hangtian Liu, Wenhou Sun, Zhizhuo Tang, Liehui Jiang, Jianjun Chen, and Yong Xie. Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*, San Diego, California, USA, 2024. The Internet Society.

[11] Wil Gibbs, Arvind S. Raj, Jayakrishna Menon Vadayath, Hui Jun Tay, Justin Miller, Akshay Ajayan, Zion Leonahenahe Basque, Audrey Dutcher, Fangzhou Dong, Xavier J. Maso, Giovanni Vigna, Christopher Kruegel, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Operation mango: Scalable discovery of taint-style vulnerabilities in binary firmware services. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[12] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.

[13] Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 253–267, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[14] Hex-Rays. The interactive disassembler pro is a computer software disassembler which generates assembly language code from machine-executable code. https://hex-rays.com/ida-home/, 2005.

[15] Hex-Rays. Idalib allows you to use the c++ and ida python apis outside ida as standalone applications. https://docs.hex-rays.com/user-guide/idalib, 2024.

[16] IVANTI. Cve-2023-46805 & cve-2024-21887 for ivanti connect secure and ivanti policy secure gateways. https://forums.ivanti.com/s/article/CVE-2023-46805-Authentication-Bypass-CVE-2024-21887-Command-Injection-for-Ivanti-Connect-Secure-and-Ivanti-Policy-Secure-Gateways?language=en_US, 2023.

[17] Bartlett J.E., Kotrlik J.W., and Higgins C.C. Organizational research: Determining appropriate sample size in survey research. *Information technology, learning, and performance journal*, 19:43, 2001.

[18] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 733–745, TX, USA, 2020. ACM.

[19] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting missed security operations through differential checking of object-based similar paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, page 1627–1644, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Hangtian Liu, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Xiangzhi Wang, and Guangming Gao. Labrador: Response Guided Directed Fuzzing for Black-box IoT Devices . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1920–1938, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[21] Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Chuan Qin, Dongliang Fang, Mingdong Liu, and Limin Sun. FITS: inferring intermediate taint sources for effective vulnerability analysis of iot device firmware. In Tor M. Aamodt, Michael M. Swift, and Natalie D. Enright Jerger, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 138–152. ACM, 2023.

[22] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. Detecting missing-permission-check vulnerabilities in distributed cloud systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2145–2158, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Automatically identifying security checks for detecting kernel semantic bugs. In *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II*, page 3–25, Berlin, Heidelberg, 2019. Springer-Verlag.

[24] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check bugs via semantic- and Context-Aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, August 2019. USENIX Association.

[25] Peng Luo, Deqing Zou, Yajuan Du, Hai Jin, Changming Liu, and Jinan Shen. Static detection of real-world buffer overflow induced by loop. *Computers & Security*, 89:101616, 2020.

[26] Mozilla. Constraint validation in html. https://developer.mozilla.org/docs/Web/HTML/Constraint_validation, 2024.

[27] Nvwa. Nvwa-site. https://sites.google.com/view/nvwa-site, 2025.

[28] Onekey-Sec. An accurate, fast, and easy-to-use extraction suite. https://github.com/onekey-sec/unblob, 2023.

[29] Oracle. Elf header. https://docs.oracle.com/cd/E19620-01/805-4694/6j4enatct/index.html, 2010.

[30] Abdullah Qasem, Mourad Debbabi, and Andrei Soeanu. Octopustaint: Advanced data flow analysis for detecting taint-based vulnerabilities in iot/iiot firmware. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 2355–2369. ACM, 2024.

[31] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy(SP)*. IEEE, 2020.

[32] ReFirm-Labs. The tool for analyzing, reverse engineering, and extracting firmware images. https://github.com/ReFirmLabs/binwalk, 2014.

[33] Route_fileter. Tool for collecting statistics on router cve. https://github.com/Joe1sn/route_fileter, 2022.

[34] Offensive Security. exploit database:a public and open source vulnerability database. https://www.exploit-db.com/, 2016.

[35] Qintao Shen, Hongyu Sun, Guozhu Meng, Kai Chen, and Yuqing Zhang. Detecting api missing-check bugs through complete cross checking of erroneous returns. In *Information Security and Cryptology*, page 391–407, Berlin, Heidelberg, 2023. Springer-Verlag.

[36] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.

[37] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1899–1913, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Ying-Jie Wang, Liangze Yin, and Wei Dong. Amchex: Accurate analysis of missing-check bugs for linux kernel. *J. Comput. Sci. Technol.*, 36(6):1325–1341, 2021.

[39] WIKIPEDIA. Halstead complexity measures. https://en.wikipedia.org/wiki/Halstead_complexity_measures, 2024.

[40] Jiahui Xiang, Lirong Fu, Tong Ye, Peiyu Liu, Huan Le, Liming Zhu, and Wenhai Wang. Luataint: A static analysis system for web configuration interface vulnerability of internet of things devices, 2024.

[41] Haoyu Xiao, Yuan Zhang, Minghang Shen, Chaoyang Lin, Can Zhang, Shengli Liu, and Min Yang. Accurate and efficient recurring vulnerability detection for iot firmware. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, page 3317–3331, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, page 499–510, New York, NY, USA, 2013. Association for Computing Machinery.

[43] Yao Yao, Wei Zhou, Yan Jia, Lipeng Zhu, Peng Liu, and Yuqing Zhang. Identifying privilege separation vulnerabilities in iot firmware with symbolic execution. In *24th European Symposium on Research in Computer Security(ESORICS)*, 2019.

[44] Xiaokang Yin, Ruijie Cai, Yizheng Zhang, Lukai Li, Qichao Yang, and Shengli Liu. Accelerating command injection vulnerability discovery in embedded firmware with static backtracking analysis. In *12th International Conference on the Internet of Things(IoT)*, 2022.

[45] Xiaokang Yin, Ruijie Cai, Xiaoya Zhu, Qichao Yang, Enzhou Song, and Shengli Liu. Precise discovery of more taint-style vulnerabilities in embedded firmware. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2024.

[46] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, Santa Clara, CA, 2019.

[47] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. Srfuzzer: an automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. In *Proceedings of the 35th annual computer security applications conference*, pages 544–556, 2019.

[48] Yu Zhang, Nanyu Zhong, Wei You, Yanyan Zou, Kunpeng Jian, Jiahuan Xu, Jian Sun, Baoxu Liu, and Wei Huo. Ndfuzz: a non-intrusive coverage-guided fuzzing framework for virtualized network devices. *Cybersecurity*, 5(1):21, 2022.

[49] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, Xinyi Wang, Wei Wang, and Wei Huo. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7067–7084, Philadelphia, PA, August 2024. USENIX Association.

[50] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Yang Xiao, Naijia Jiang, Yeting Li, Nanyu Zhong, Bingwei Peng, Kunpeng Jian, and Wei Huo. Usenix security 25'-NÜWA-artifact-evaluation. https://doi.org/10.5281/zenodo.15605329, June 2025.

[51] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl:high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[52] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 417–428, New York, NY, USA, 2022. Association for Computing Machinery.

Table 10: Known Vulnerability Detection Results of NÜWA with Baselines.

CVE-ID	Vendor	Device	Binary	Type	Trigger Path	NÜWA	SATC	EMTAINT	LARA	MANGO	OCTOPUSTAINT
CVE-2021-45756	ASUS	RT-AC68U	httpd	CWE-119	Intra-Func	✔	✔	✔	✔	✔	✗
CVE-2022-37805	Tenda	AC12	httpd	CWE-119	Intra-Func	✔	✗	✗	✔	✗	✗
CVE-2023-20117	Cisco	RV32x	ssi.cgi	CWE-77	Intra-Func	✔	✔	✗	✔	✔	✔
CVE-2023-24229	Draytek	Vigor2960	mainfunction.cgi	CWE-77	Intra-Func	✔	✔	✗	✔	✔	✔
CVE-2023-26806	Tenda	W20E	httpd	CWE-119	Intra-Func	✔	✔	✔	✔	✗	✔
CVE-2023-31741	Linksys	E2000	httpd	CWE-77	Intra-Func	✔	✔	✗	✔	✔	✔
CVE-2023-33538	Tplink	TL-WR841Nv10	httpd	CWE-119	Intra-Func	✔	✔	✔	✔	✔	✔
CVE-2023-34933	H3C	B1 ST	webs	CWE-119	Intra-Func	✔	✗	✗	✔	✔	✗
CVE-2023-39550	NetGear	JWNR2000v2	uhttpd	CWE-119	Intra-Func	✔	✔	✔	✔	✔	✔
CVE-2023-50361	QNAP	QNAP	userConfig.cgi	CWE-119	Intra-Func	✔	✔	✗	✔	✗	✔
CVE-2024-0538	Tenda	W9	httpd	CWE-119	Intra-Func	✔	✔	✗	✔	✔	✔
CVE-2024-0579	TOTOLink	X2000R	boa	CWE-77	Intra-Func	✔	✗	✗	✔	✔	✗
CVE-2024-27129	QNAP	QNAP	utilRequest.cgi	CWE-119	Intra-Func	✔	✗	✗	✔	✗	✗
CVE-2020-10825	Draytek	Vigor2960	activate.cgi	CWE-119	Inter-Func	✔	✗	✗	✗	✗	✗
CVE-2022-25106	Dlink	DIR859	cgibin	CWE-119	Inter-Func	✔	✗	✔	✔	✗	✔
CVE-2022-43000	Dlink	DIR816	goahead	CWE-119	Inter-Func	✔	✗	✗	✔	✗	✗
CVE-2022-46566	DLink	DIR882	prog.cgi	CWE-119	Inter-Func	✔	✗	✗	✔	✗	✗
CVE-2023-27806	H3C	R100	webs	CWE-119	Inter-Func	✔	✗	✗	✔	✗	✗
CVE-2023-27997	Fortigate	Fortigate	init	CWE-119	Inter-Func	✗	✗	✗	✗	✗	✗
CVE-2023-29665	DLink	DIR823G	goahead	CWE-119	Inter-Func	✔	✗	✗	✗	✗	✗
CVE-2023-31701	Tplink	TL-WPA4530	httpd	CWE-77	Inter-Func	✔	✗	✗	✔	✔	✗
CVE-2023-38933	Tenda	FH1203	httpd	CWE-119	Inter-Func	✔	✔	✗	✔	✔	✗
CVE-2023-46977	TOTOLink	N600R	cstecgi.cgi	CWE-119	Inter-Func	✔	✗	✔	✗	✗	✗
CVE-2022-25130	TOTOLink	T10	wireless.so	CWE-77	Inter-Bin	✔	✗	✗	✔	✗	✗
CVE-2023-26978	TOTOLink	A7100RU	cstecgi.cgi	CWE-77	Inter-Bin	✔	✗	✗	✗	✗	✗
CVE-2023-50983	Tenda	i29	httpd	CWE-77	Inter-Bin	✔	✗	✗	✗	✗	✗
CVE-2023-51099	Tenda	W9	httpd	CWE-77	Inter-Bin	✔	✗	✗	✔	✗	✗
CVE-2024-0296	TOTOLink	N200RE	cstecgi.cgi	CWE-77	Inter-Bin	✔	✗	✗	✔	✗	✗
CVE-2024-27130	QNAP	QNAP	utilRequest.cgi	CWE-119	Inter-Bin	✗	✗	✗	✗	✗	✗
CVE-2024-53703	SonicWALL	SMA100	mod_httprp	CWE-119	Inter-Bin	✔	✗	✗	✗	✗	✗
EDB-42344	SonicWALL	SMA1000	sitecustomization	CWE-77	Inter-Bin	✗	✗	✗	✗	✗	✗
Total	—	—	—	—	—	**28**	**10**	**6**	**22**	**11**	**9**

A Multiple Invocations of the Input.

An input might be invoked at multiple different locations, and sometimes these different invocations can be grouped into a single handling path. Listing 2 demonstrates how an input is parsed across two functions. During the first parsing, the input id_flag must be equal to 2 to proceed to the second function. In the second parsing, the input id_flag is passed directly to strcpy without any length check. Due to ignoring the connection between the two invocations, the use of strcpy may be incorrectly identified as a vulnerability. Therefore, when different invocations of the same input are called consecutively within a single path, the corresponding constraints need to be considered sequentially and holistically. NÜWA will merge ICFGs from different invocations of the same input. Constraints from different invocations but within the same path will be considered together.

```
1  int func1(...){
2      v3 = get_cgi(a1, "id_flag");
3      if (v3 == 2){
4          func2(...);
5      }
6  }
7  int func2(...){
8      char *v8[10];
9      v2 = get_cgi(a1, "id_flag");
10     ...
11     strcpy(v8, v2);
12 }
```

Listing 2: Multiple Invocations of the Input

B Detail Vulnerability Detection Result

TABLE 10 presents the detailed detection results of NÜWA and baselines on the known vulnerability dataset.

C False Negatives of NÜWA

To further assess NÜWA's capabilities, we analyzed false negatives in constraint semantic analysis and vulnerability detection. A primary cause is the difficulty in analyzing key semantic information, such as variable names and contents, which are crucial for inferring data types and extracting constraints. In addition, some complex code paths involve user input flowing through structures unsupported by NÜWA. Finally, the use of custom JavaScript frameworks on the front end can lead to missed explicit constraints.

D The Distribution of Complexities

Figure 4 presents the Halstead effort distribution of all statements associated with the correctly extracted front-end explicit constraints, back-end explicit constraints, and back-end

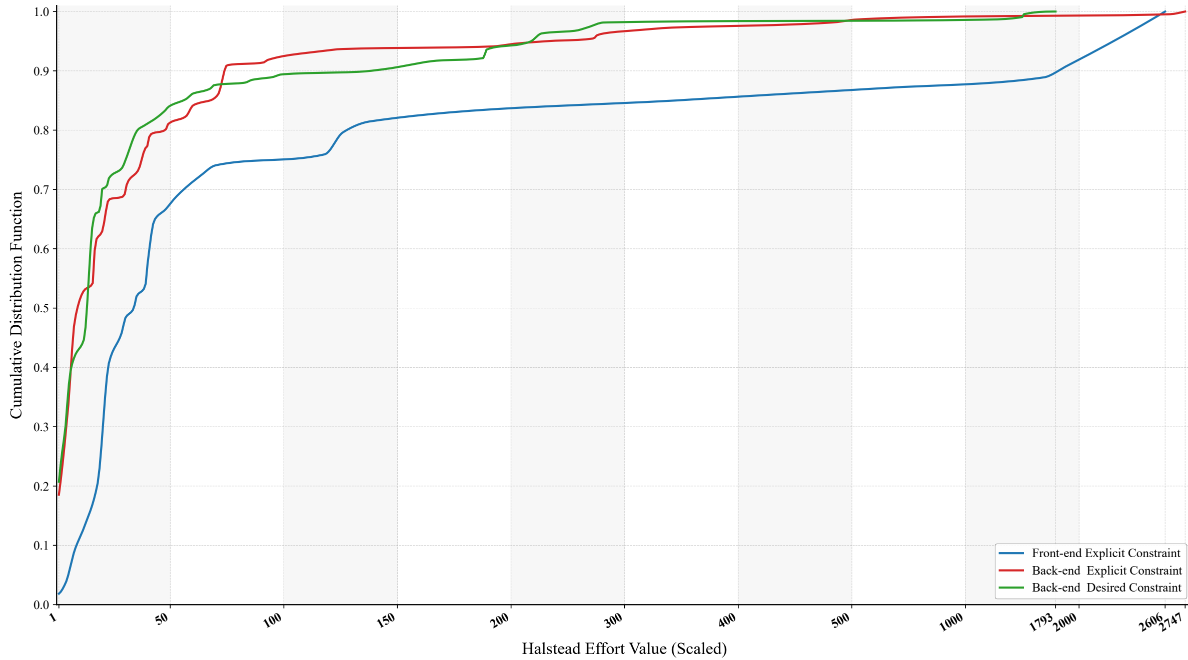

Figure 4: The Distribution of the Halstead Effort

desired constraints.

E CVE ID and Case Study

NÜWA-Site [27] provides the complete list of CVE identifiers corresponding to the previously undisclosed vulnerabilities discovered by NÜWA, along with analyses of three distinct types of vulnerabilities detected by the tool.