中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

UNSW
SYDNEY

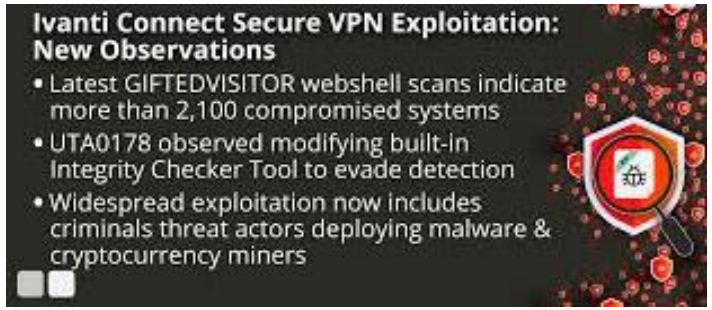
From Constraints to Cracks: Constraint Semantic Inconsistencies as Vulnerability Beacons for Embedded Systems

Jiaxu Zhao, Yuekang Li, Yanyan Zou*, Yang Xiao, Naijia Jiang, Yeting Li
Nanyu Zhong, Bingwei Peng, Kunpeng Jian, Wei Huo*

Background

- **These vulnerabilities not only risk exposing sensitive information but also enable attackers to gain control of devices. Consequently, detecting vulnerabilities in embedded system web services is of critical importance.**

- **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?**



The Ivanti VPN vulnerabilities CVE-2023-46805 and CVE-2024-21887 can be chained to achieve unauthorized arbitrary code execution.

CVE-2023-25610, disclosed in 2023, affects over 50 models of Fortinet network devices.

An improper authentication control vulnerability exists in AiCloud of ASUS router, leading to unauthorized execution of functions.

Motivating Example

- **The lack or incompleteness of input constraints is a key cause of vulnerability.**
  - **Incomplete validation of special characters in user input can lead to command injection: the user-provided *pskname* is checked for special characters (*;* , *&*, |) and is passed to the command execution function *popen*.**

```
1. int sub_42AA54(int a1, const char *a2, const char *a3){
2.   char *v32 = malloc(128);
3.   v21 = websGetVar(a1, "pskname", "");
4.   sub_425ED0("decrypt", v21, v32);
5.   ...
6.   if (strstr(v32, ';') || strstr(v32, '&') || strstr(v32, '|')){
7.     goto fail;
8.   }
9.   if ( v32 ){
10.    snprintf(s, 0x200u, "echo -n %s ...%s... ", a1, a2);
11.    popen(s, "r");
12.  }
13.}
```

The constraint that has been enforced.

Input must not contain ;, &, or |.

The constraint that should be enforced.

Input must not contain ;, &, |, $, \r, \n, etc.

Construct a PoC based on missing input constraints.

```
POST …cgi HTTP
HTTP Header
…&pskname="aa\r\ntelnetd -l /bin/sh"&…
```

The constraint restricts the range of allowed input characters.

Motivating Example

- **Semantic inconsistencies between back-end explicit and desired constraints can reveal potential vulnerabilities.**
  - Explicit Constraint are rules and conditions explicitly implemented for user inputs.
  - Desired Constraint are implicit requirements that are not explicitly implemented in the code but are essential for preventing vulnerabilities.

```
1. int sub_42AA54(int a1, const char *a2, const char *a3){
2.   char *v32 = malloc(128);
3.   v21 = websGetVar(a1, "pskname", "");
4.   sub_425ED0("decrypt", v21, v32);
5.   ...
6.   if (strstr(v32, ';') || strstr(v32, '&') || strstr(v32, '|')){
7.     goto fail;
8.   }
9.   if ( v32 ){
10.    snprintf(s, 0x200u, "echo -n %s ...%s... ", a1, a2);
11.    popen(s, "r");
12.  }
13.}
```

Explicit Constraints

Input must not contain ;, &, or |.

Desired Constraints

Input must not contain ;, &, |, $, \r, \n, etc.

Construct a PoC based constraints inconsistenci.

```
POST …cgi HTTP
HTTP Header
…&pskname="aa\r\ntelnetd -l /bin/sh"&…
```

The constraint restricts the range of allowed input characters.

Motivating Example

- **Semantic inconsistencies between front-end and back-end explicit constraints also can reveal potential vulnerabilities.**
  - **The diverse implementations of back-end sinks, such as those in XSS and SQL injection vulnerabilities, hinder accurate extraction of corresponding desired constraints.**

```
1. <td>
2.     <input type="text" pattern="^[^<>/()]+$" name="pskname" id="PSK">
3. </td>

4. <script language="javascript">
5.     function updateChanX(){
6.         var str = pskname.value;
7.         if (str){
8.             for (var i=0; i<str.length; i++){
9.                 if ( (str.charAt(i) >= '0' && str.charAt(i) <= '9') ||
10.                     (str.charAt(i) >= 'a' && str.charAt(i) <= 'f') ||
11.                     (str.charAt(i) >= 'A' && str.charAt(i) <= 'F') ){
12.                     continue;
13.                 }
14.         }else{
15.             return false;
16.         }
17.</script>
```

Front-end explicit constraints in HTML

Input must not contain <, >, /, (, or).

Front-end explicit constraints in Javascript

Input must consist of letters or digits only (no special characters allowed).

XSS Exploit：<script>…</script>

Motivating Example

- **Formal definition of vulnerability detection based on constraint semantic inconsistencies**
  - S: the set of all possible input strings
  - S_{fc}: the set of inputs that satisfy the front-end explicit constraints
  - S_{ec}: the set of inputs that satisfies the back-end explicit constraints
  - S_{dc}: the set of inputs that satisfies the back-end desired constraints

  For $\forall s \in S$:
  - If $s \notin S_{\mathrm{ec}}$, the input is considered invalid.
  - If $s \in S_{\mathrm{ec}} \cap S_{dc}$, the input is considered safe.
  - If $s \in (S_{\mathrm{ec}} \setminus S_{dc}) \vee (S_{\mathrm{ec}} \setminus S_{fc})$, the input is potentially risky.

The representations for constraint semantics

- **Represent constraint semantics consistently and equivalently.**
  - **Constraint semantics are represented in the form of a <name, value> pair.**

Different implementations of constraints

```
<input type="text" maxlength="200" …>
```

```
If (strlen(v1)< 200)
```

```
V2 = strlen(v1);
If ( v2 < 200 )…
```

```
char v7[128];      strcpy(v7, v1);
```

Consistent constraint semantics:
input string length < 200.

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

The representations for constraint semantics

Represent constraint semantics consistently and equivalently.

- Constraint semantics are represented in the form of a <name, value> pair.
- **Positive vs. negative** constraints: distinguished by whether the program proceeds with input processing or triggers error handling upon satisfying the explicit constraint.

Constraint semantics are context-dependent.

```
if (strstr(v32, ';')){
        goto fail;
}
```

```
if (strstr(v32, ';')){
        v32.replace(';', '-');
}
```

Equivalent constraint semantics: the input string must not contain `;`.

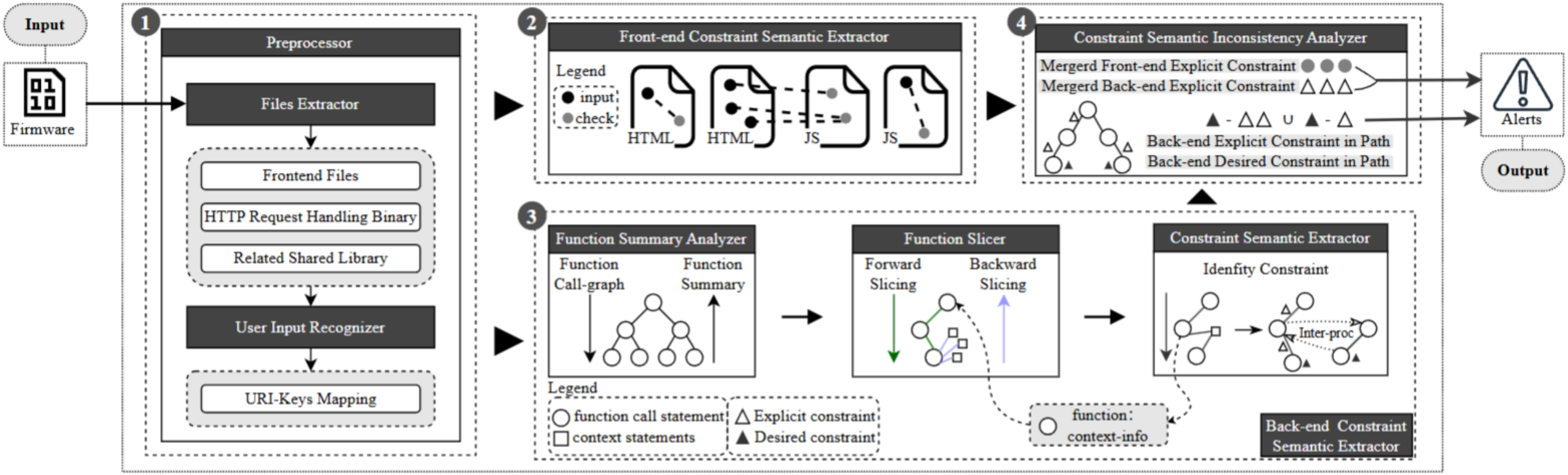
The decision for the error-handling branch.

Return value: return false
Abort function: exit()
Control flow alter: break、goto
Error logging: log('error ....')
Character replacement: substitute certain special characters

Negative constraint semantics are recorded after being negated.

Nüwa | Overview

- Preprocessing
- Front-end Constraint Semantic Extraction
- Back-end Constraint Semantic Extraction
- Static Analysis on Constraint Semantic Inconsistency



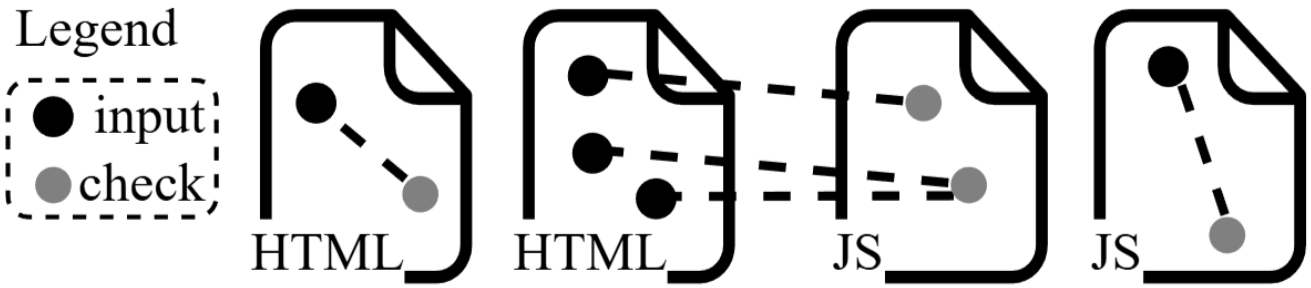
Nüwa | Front-end Constraint Semantic Extraction

Front-end explicit constraints in HTML attributes.

- the validation-related attributes within the <input> element, encompassing eight standard classes, such as maxlength to <len, …>
- the <select> element enforces fixed-value constraints
- the fields automatically filled in enforces fixed-value constraints

Front-end explicit constraints in JavaScript conditional statements.

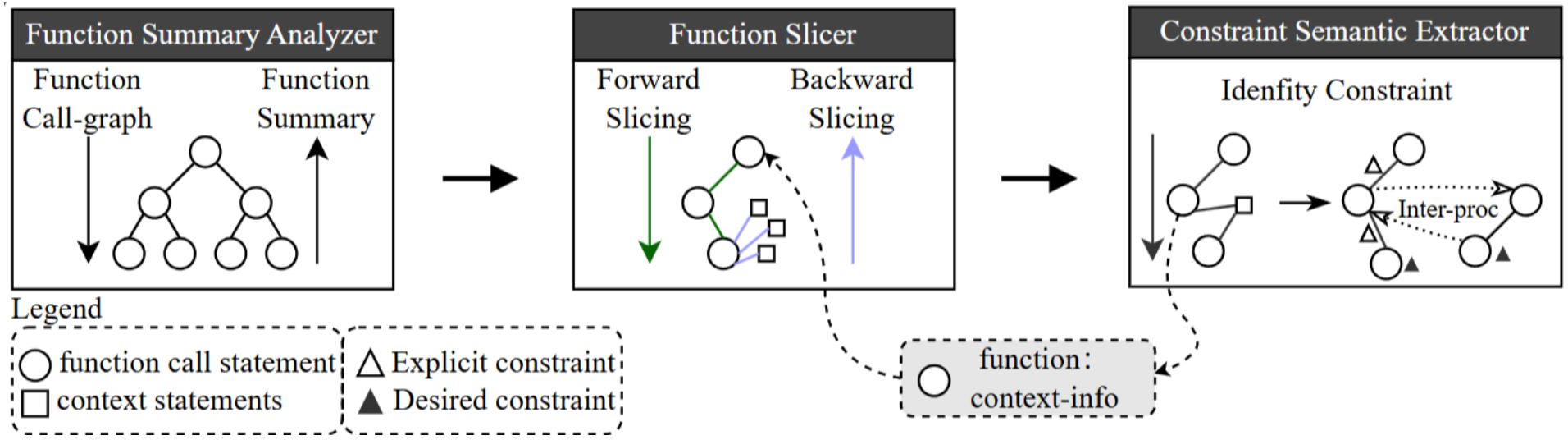
- Variables and function calls: e.g., if(v32) or if(strstr(v32, ';'))
- Arithmetic operations: e.g., +, -
- Comparison operations: e.g., ==, >=
- Logical operations: e.g., &&, ||



Nüwa | Back-end Constraint Semantic Extraction

Inter-procedural and cross-binary binary analysis

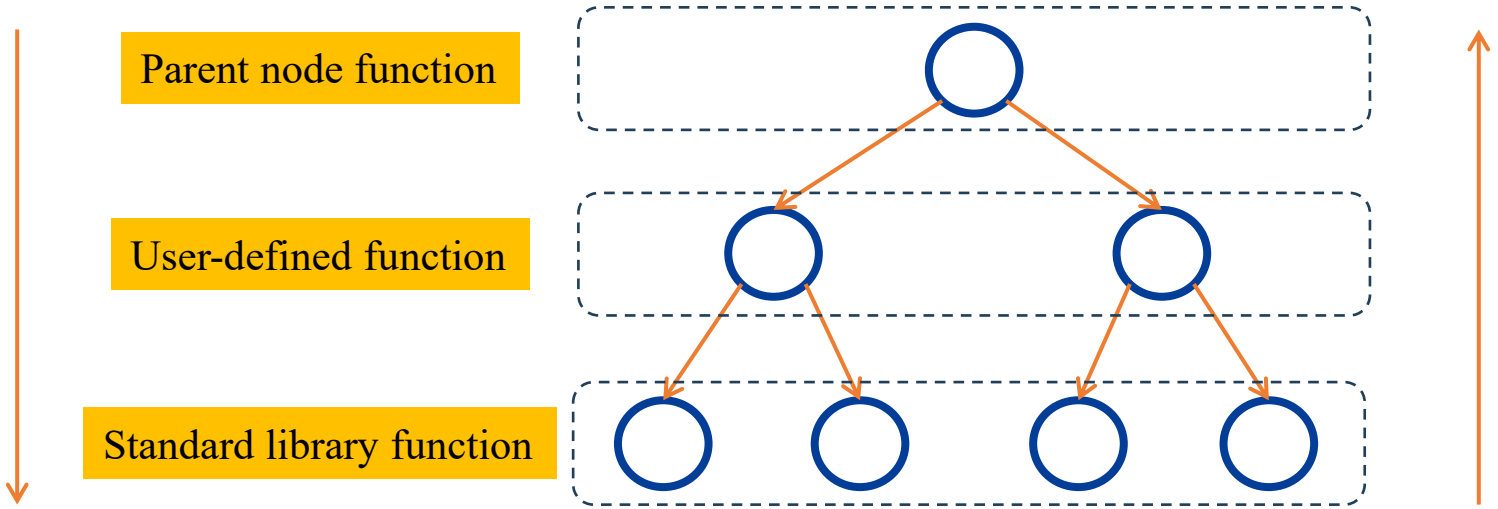
- function summary analyzer —> inter-procedural analysis
- forward and backward slicer —> simplify analysis code and collect context
- constraint semantic extractor —> record back-end constraint semantics



Nüwa | Back-end Constraint Semantic Extraction

Function Summary Analyzer

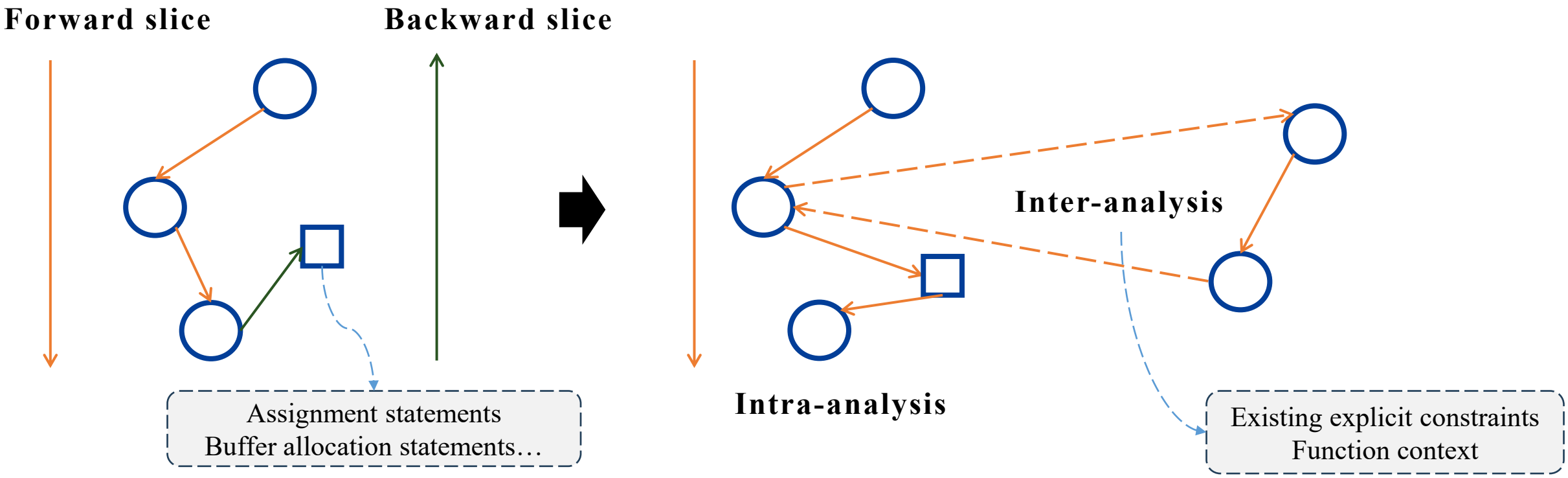
- A top-down function call graph
- A bottom-up function summary analysis
- The summary of Standard library function is predefined



Nüwa | Back-end Constraint Semantic Extraction

Constraint semantics extraction based on function slicing

- Forward slicing removes input-irrelevant statements, while backward slicing gathers context for constraint semantics.
- Intra-proce analysis extracts explicit and desired constraints and records relevant functions.
- Inter-proce analysis repeatedly applies slicing and intra-proce analysis with context.



Nüwa | Back-end Constraint Semantic Extraction

Constraint semantics extraction based on function slicing

- Explicit constraints in source handling code and conditional statements

```
1.v21 = websGetVar(a1, "pskname", "");

2.char *websGetVar(Webs *wp, char *in, char *def){
3.   char *sp = hashLookup(wp->vars, in);
4.   if ( sp && sp->content.value.integer )
5.     return sp->content.value.string;
6.   else
7.     return def;
8.}
```

a default value is assigned if the input is absent.

Parse constraints via function summaries

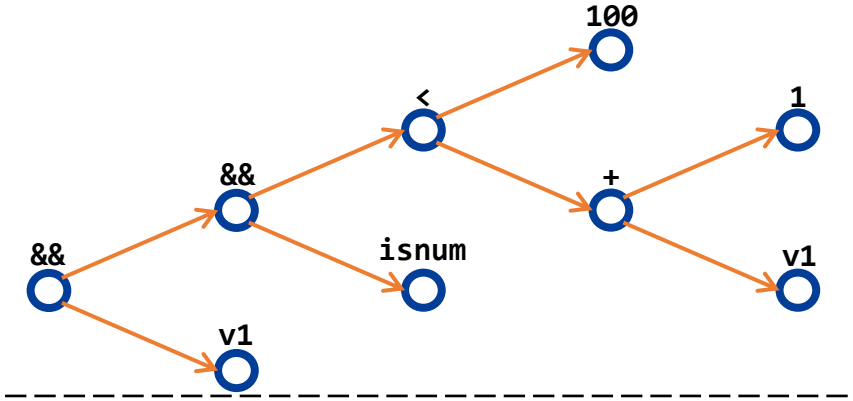
Summary of `websGetVar`: `<0: ['ret'], 1: [], 2: ['ret']>` Parameter 1 represents data from the HTTP request, and parameter 3 serves as the default value if it is a string or a number less than 10.

conditional statements

1. Variables and function calls
2. Arithmetic operations
3. Comparison operations
4. Logical operations

Parse constraints from AST

if (v1 && isnum(v1) && v1+1 < 100)



Nüwa | Back-end Constraint Semantic Extraction

Constraint semantics extraction based on function slicing

- Explicit constraints in source handling code and conditional statements
- Desired constraints in sink operations
  - Memory-related sink operations: relevant memory boundary is obtained from both slicing context and the function summary.
  - Non-memory-related sink operations: based on prior knowledge summarized from manual experience and vulnerability exploitation analysis.

```
1.// array copy or read/write operations
2.char *v7 = malloc(128);
3.strcpy(v7, input); // Heap overflow constraint: <len, [0, 128]>
4.…=*(v7+(input++));// over-write constraint: <num, [0, 127]>

5.// User-defined function
6.char *v7 = malloc(128);
7.Decode(v7, input); // Heap overflow constraint: <len, [0, 128]>
```

Based on context

Obtain the size of the memory pointed to by pointer `v7` through backward slicing.

Based on function summary

The function summary indicates that the `Decode` function passes `input` to `v7`.

Nüwa | Back-end Constraint Semantic Extraction

Constraint semantics extraction based on function slicing

- Explicit constraints in source handling code and conditional statements
- Desired constraints in sink operations
- **Record backend explicit and desired constraints with in the ICFG** to differentiate constraint semantics across distinct execution paths.

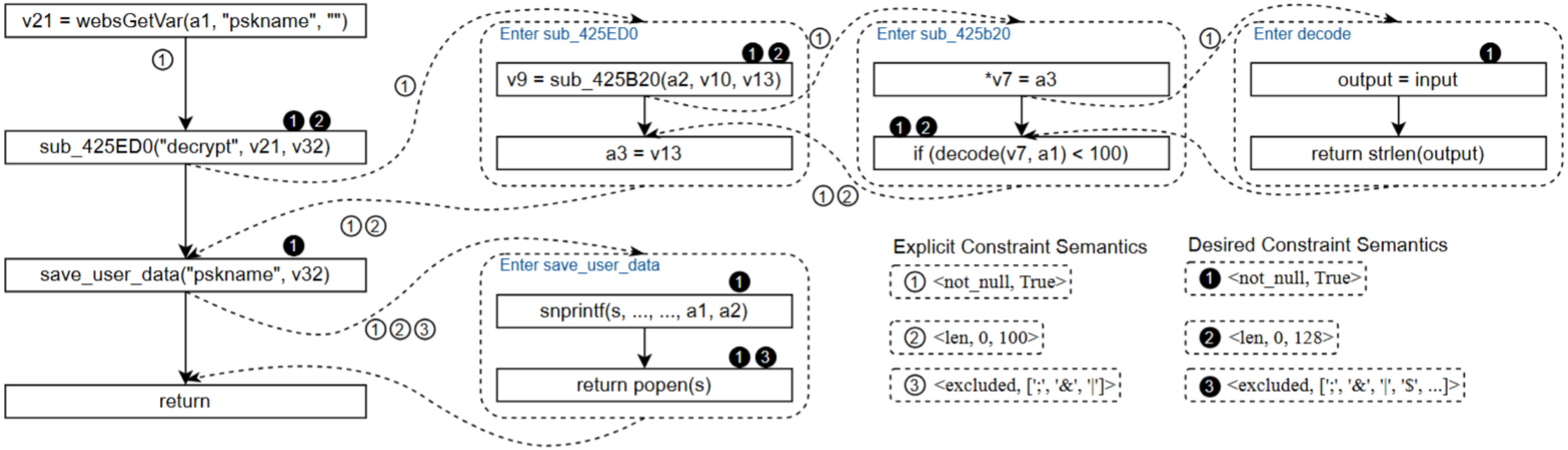


Figure 3: The ICFG of Motivating Example

Nüwa | Constraint Semantic Inconsistency Analysis

- ***EC*: back-end explicit constraint semantics**
- ***DC*: back-end desired constraint semantics**
- ***FC*: front-end explicit constraint semantics**

- **For a given path *P* in the ICFG, The inconsistency set *I(P)* is defined as**

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

- **And the inconsistency set *I* between front-end and back-end explicit constraints as**

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

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	$[0, MAX]$	$\cap$	$EC \setminus DC$	$EC \setminus FC$

Evaluation

- **RQ1**: How is the detection capability of Nüwa compare with baselines on known vulnerabilities?
- **RQ2**: How effective is Nüwa in extracting constraint semantics and identifying inconsistencies?
- **RQ3**: How effectively is Nüwa in discovering unknown vulnerabilities in real-world embedded systems?

Evaluation | Dataset and Baselines

- A known vulnerability dataset
  - 31 known vulnerabilities from 13 vendors
  - Vulnerabilities were selected from embedded device vendors referenced in SOTA research
  - Only buffer overflow vulnerabilities and command injection vulnerabilities were included
  - Vulnerabilities with clear trigger information were prioritized, ensuring practical relevance
  - Redundant vulnerabilities, such as highly similar ones (e.g.CVE-2023-34933 and CVE-2022-37095), only select one
  - And those requiring interactions between multiple processes are not concerned

- Five SOTA tools
  - SaTC (USENIX Sec 2021)
  - EmTaint (ISSTA 2023)
  - Lara (USENIX Sec 2024)
  - Mango (USENIX Sec 2024)
  - OCTOPUSTaint (ACM CCS 2024)

RQ1: Comparison with the SOTA tools

- **Nüwa 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%.**
- **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.**

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

RQ2: Comprehensive Analysis of Output

- **The precision and distribution of constraint semantic extraction.**
- **Constraints Complexity: Halstead effort， Statement Count， Call Depth.**

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.

RQ2: Comprehensive Analysis of Output

For *Diff(EC, DC)*

- 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.

For *Diff(EC, FC)*

- 14 unique alerts, with 12 validated as TPs.
- These TPs do not always map one-to-one with specific vulnerabilities.
- revealing 10 vulnerabilities of the dataset and one XSS vulnerability that could only be uncovered through this approach.

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 → 1
CVE-2022-25106	4	2	2	1	1	1	1 → 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 → 1
CVE-2024-0538	3	2	2	0	0	0	0
CVE-2023-51099	2	2	2	3	3	3	3 → 2
CVE-2022-37805	1	1	1	0	0	0	0
CVE-2023-26806	1	1	1	1	1	1	1 → 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 → 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 → 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 → 1
CVE-2023-31741	2	2	2	1	1	1	1 → 1
Total	147	53	39	14	14	12	11 → 11

RQ3: Real-world Vulnerability Discovery

- **Nüwa uncovered 152 previously unknown vulnerabilities, and 88 of them have been assigned CVE IDs following responsible disclosure.**
- **These vulnerabilities involve 8 types of vulnerability. 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**

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

Summary

- **NÜWA, a novel static analysis technique that identifies potential vulnerabilities in embedded systems by identifying constraint semantic inconsistencies.**

- **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.**

- **NÜWA identified 152 unknown vulnerabilities, all of which have been confirmed by vendors, with 88 CVE IDs assigned.**

https://doi.org/10.5281/zenodo.15605329