CPscan: Detecting Bugs Caused by Code Pruning in IoT Kernels

Lirong Fu   Shouling Ji   Kangjie Lu   Peiyu Liu   Xuhong Zhang
YuXuan Duan   Zihui Zhang   Wenzhi Chen   Yanjun Wu

2021

Code pruning in real-world IoT kernels

Linux Kernel

IoT Kernels

Table 1: The third-party customization on Linux kernel.

ID	IoT Vendor	IoT Kernel	Customized Files	Customized Funcs
1	DD-WRT	universal-3.5.7	2,254	13,779
2	DD-WRT	universal-3.10.108	661	3,306
3	DD-WRT	universal-3.18.140	764	3,871
4	DD-WRT	universal-4.4.198	642	3,615
5	DD-WRT	universal-4.14.151	700	3,123
6	ASUSWRT	Asuswrt-rt-6.x.4708	742	808
7	ASUSWRT	Asuswrt-rt-7.14.114.x	740	802
8	ASUSWRT	Asuswrt-rt-7.x.main	740	807
9	TUYA	tuya-3.10	352	2,648
10	TUYA	tuya-4.9	177	556
11	TUYA	hisi3518e_v300	177	766
12	TUYA	tuya-4.1.0	309	356
13	NETGEAR	A90-620025	706	1,508
14	NETGEAR	VER_01.00.24	211	327
15	NETGEAR	C6300BD_LxG1.0.10	118	634
16	NETGEAR	R7450_AC2600	825	3,151
17	NETGEAR	R6700v2_R6800	828	3,232
18	TPLink	Archer-AX20	490	3,649
19	TPLink	Archer-AX6000	425	2,391
20	TPLink	Archer-AX11000	425	2,408
21	TPLink	KC200	413	1,218
22	DLink	DCS-T2132	1,017	1,018
23	DLink	DAP-X2850	1,094	4,247
24	QNAP	Qhora	1,264	5,213
25	QNAP	Turbo	2,947	4,726
26	Arris	DCX4220	315	1,574
27	Level One	WAC-2003	289	611
28	Linksys	E8450	1,540	4,198
Total			21,165	74,542

Security bugs caused by code pruning

```
/*drivers/char/n_gsm.c*/
static void gsm_control_reply(struct gsm_mux *gsm, ...){
    struct gsm_msg *msg;
    msg = gsm_data_alloc(gsm, 0, dlen + 2, gsm->ftype);
    // The deleted NULL pointer check
-   if (msg == NULL)
-       return;
    msg->data[0] = (cmd & 0xFE) << 1 | EA;
    msg->data[1] = (dlen << 1) | EA;
}
```

Figure 1: A deleted security check in an IoT kernel found by CPScan. The missed NULL pointer check against security-critical variable *msg* leads to a NULL pointer dereference.

Challenges

- Challenge 1: a significant structural change makes precisely locating the deleted security operations (DSO) difficult.

```
1 /*net/ipv6/ip6_output.c*/
2 void ip6_flush_pending_frames(struct sock *sk){
3     while (...) {
4 +         if (skb->dst)
5               IP6_INC_STATS(...);
6           kfree_skb(skb);
7     }
8 }
```

Figure 12: *kfree_skb* (line 6) is mistakenly reported as a deleted resource-release operation, because the new added conditional statement (line 4) changes the function CFG.

Challenges

- Challenge 2: inferring the security impact of a DSO is not trivial since it requires complex semantic understanding, including the developing logic and context of the corresponding IoT kernel.

  - where the security-critical variable associated with an DSO comes from;
  - how it is checked;
  - what it is used for;
  - how and where it is used;
  - what the potential reliability and security impact is.

Approach

- Cpscan uses graph matching to perform precise code pruning identification because graph comparison can capture not only structural information but also semantic information.

- CPscan employs inconsistency analysis to infer the security impact of a DSO by comparing the bounded uses of the security-critical variable associated with it.

Design of CPscan

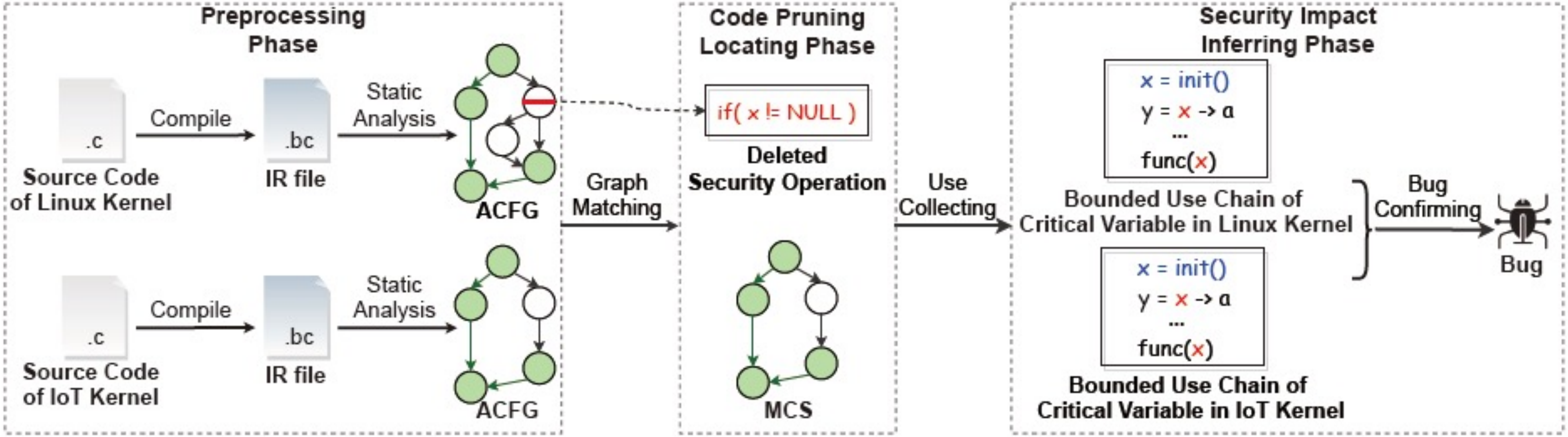


Figure 2: Workflow of CPscan. ACFG = Attributed control flow graph, MCS = Maximum common subgraph.

Design of CPscan

➢ Preprocessing phase

Attributed control flow graphs (ACFG) generation: basic block attributions extraction

Table 3: Basic-block attributes used in CPscan.

Type	Feature Name
Statistical Attribute	Constant Value Instruction Sequence Function Call Sequence Security Operation Instruction Distribution
Structural Attribute	Neighbor Nodes in MCS

Design of CPscan

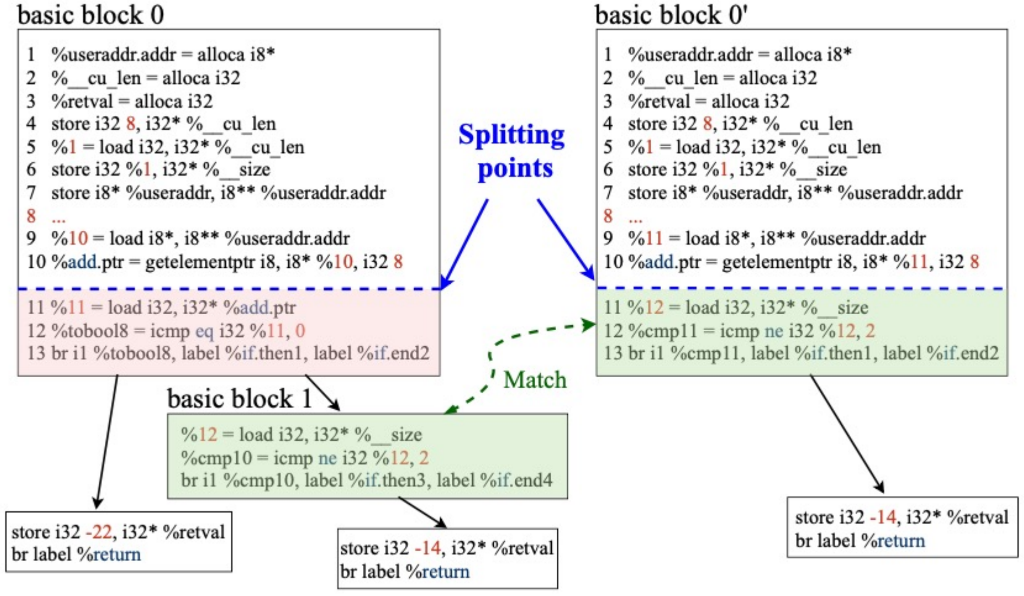
➢ Graph matching phase

Graph matching main idea

1. Use the distinguishable basic blocks containing security operations to guide an initial fast basic block matching.

2. Utilize maximum common subgraph to guide the match of the neighbor nodes of the already matched basic block pairs.

3. Perform a one-to-many match for the remaining basic blocks.



(a) Linux kernel (b) IoT kernel

Design of CPscan

➢ Security Impact inferring phase

Security-critical variable determination

1. A security-critical variable is closely associated with a DSO and is usually the parameter or the return value of this DSO
2. The security-critical variable should also have subsequent uses in the function, which can be utilized to determine the security impact of the corresponding DSO.

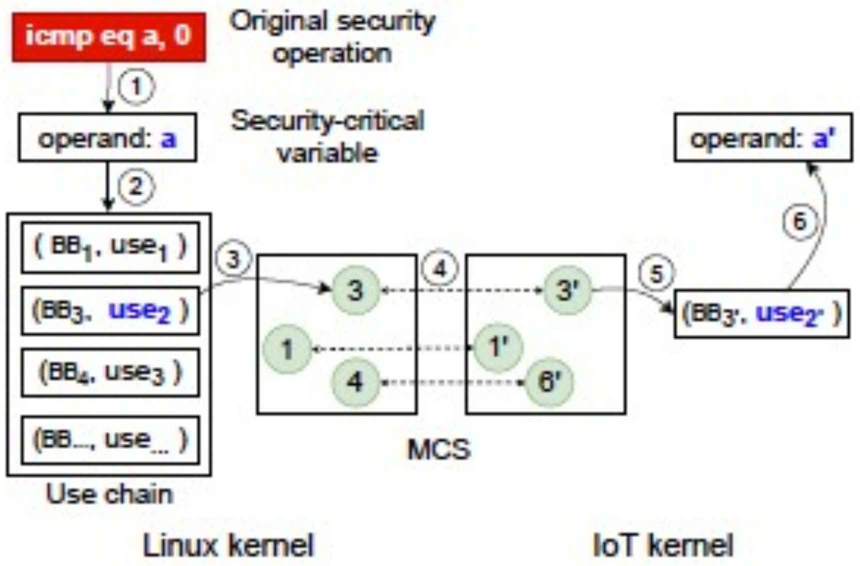


Figure 4: (1) Identify the security-critical variable a. (2) Obtain the use chain of a. (3) check which use is contained in the maximum common subgraph; (4) Find the corresponding paired basic block $3'$ in IoT kernel. (5) Locate the identical use use_2'. (6) Identify the corresponding security-critical variable a' in IoT kernel.

Design of CPscan

- **Security Impact inferring phase**

Bounded use chain generation and comparison

1. Each security operation has its own influence scope, e.g., a security check protects a checked variable from being used under erroneous states within its successor branches, and

2. Only the uses in the influenced code segments are security-critical.

Evaluation

Experiment Settings

Environment

- Ubuntu 16.04 LTS
- LLVM version 10.0.0
- 64 GB RAM
- An Intel CPU (Xeon R CPU E5-2680 with 20 cores)

Evaluation metrics

- The accuracy of CPscan
- The efficiency of CPscan

Dataset

- 28 IoT kernels from 10 popular IoT vendors

Evaluation

➢ **Performance of Locating DSOs**

Table 4: Performance of locating *DSOs* on the real dataset.

ID	GumTree [20]		LLVM-Diff [6]		LLVM-Diff-N		CPscan		
	TP	Re.	TP	Re.	TP	Re.	TP	Pre.	Re.
2	70	54%	43	33%	42	32%	**127**	**83%**	**97%**
3	151	49%	62	20%	61	20%	**290**	**86%**	**94%**
6	46	64%	3	4%	5	7%	**68**	**89%**	**94%**
7	48	65%	5	7%	4	5%	**70**	**90%**	**94%**
10	56	73%	1	1%	1	1%	**75**	**89%**	**97%**
11	56	73%	1	1%	1	1%	**74**	**89%**	**96%**
12	24	83%	3	10%	3	10%	**27**	**75%**	**93%**
22	31	69%	5	11%	5	11%	**42**	**81%**	**92%**
Average	61	66.0%	15	11.0%	15	11.0%	**97**	**85.4%**	**94.9%**

- ✓ The accuracy of DSO identification is good
- ✓ The recall of CPscan is 44% - 763% higher than the baselines

Evaluation

- **Identification efficiency of Cpscan and baselines on the real-world dataset**

  - ✓ The average analyzing time is ~4.05 s

Table 5: The average analyzing time (per file) of CPscan and baseline tools.

Tool	GumTree [20]	LLVM-Diff [6]	LLVM-Diff-N	CPscan
Time (s)	3.06	**0.93**	0.99	4.05

Evaluation

➢ **Distribution of the DSOs in the real-world dataset**

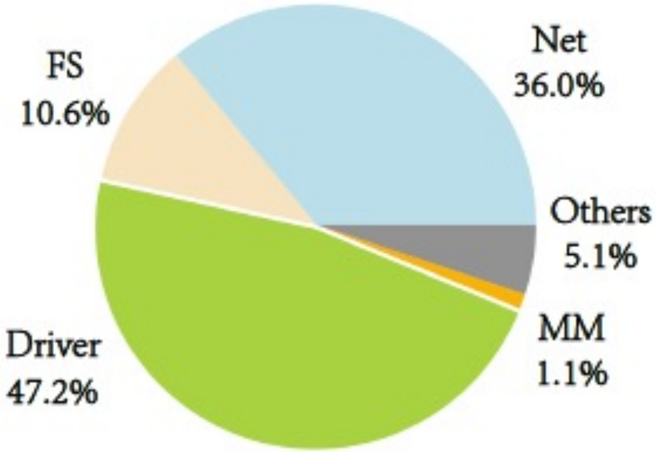


Figure 7: The distribution of the *DSOs*.

✓ About 90% of DSOs and the detected bugs exist in the driver and net modules

Evaluation

- **Identified DSOs and bugs in the real-world dataset**

Table 7: The detected security bugs caused by code pruning.

ID	# of DSC	# of DI	# of DRR	# of DSO	# of reported bugs	# of confirmed bugs
1	267	148	55	470	15	2
2	69	17	1	87	13	4
3	119	21	15	155	22	8
4	78	11	1	90	10	4
5	90	33	3	126	16	5
6	25	10	3	38	5	2
7	26	10	3	39	5	2
8	27	8	3	38	5	2
9	175	78	20	273	17	3
10	41	15	11	67	9	3
11	41	15	11	67	9	3
12	23	9	11	43	7	1
13	51	7	13	71	16	10
14	21	4	6	31	0	0
15	22	6	1	29	1	0
16	40	13	3	56	15	5
17	41	12	3	56	15	5
18	37	33	0	70	10	2
19	26	24	0	50	7	2
20	26	25	0	51	6	2
21	32	16	1	49	8	2
22	33	19	1	53	8	5
23	120	27	7	154	22	5
24	140	43	7	190	21	5
25	152	51	19	222	24	4
26	121	13	15	149	29	16
27	19	9	0	28	4	1
28	210	190	41	441	40	11
Total	**2,072**	**867**	**254**	**3,193**	**359**	**114**

- ✓ The number of the reported DSOs is 3193
- ✓ The number of the manual confirmed bugs is 114

Evaluation

➢ **The comparison of the performance of detecting missing security-check bugs.**

Table 8: The comparison of the performance of detecting missing security-check bugs.

ID	Crix [33]			PeX [58]			CPscan		
	TP	Pre.	Re.	TP	Pre.	Re.	TP	Pre.	Re.
1*	3	100%	N/A	0	0%	N/A	34	64%	59%
3*	3	20%	N/A	0	0%	N/A	34	42%	46%
6	21	35%	N/A	35	43%	N/A	2	40%	66%
10	19	37%	N/A	4	40%	N/A	1	13%	100%
22	0	0%	N/A	8	89%	N/A	3	50%	60%
Average	9	38.5%	N/A	10	34.4%	N/A	13	41.7%	66.2%

✓ CPscan detects 74 missing security-check bugs caused by code pruning

Evaluation

- **False Positives**

- 245 out of 359 bugs are FPs.

  Code re-implementation (36%).
  Inaccurate graph matching (51%).

```
1 /*net/ieee80211/ieee80211_tx.c*/
2 int ieee80211_xmit(struct sk_buff *skb, ...){
3     /* Ensure zero initialized */
4 -   struct ieee80211_hdr_3addrqos header = {
5 -   .duration_id = 0,
6 -   .seq_ctl = 0,
7 -   .qos_ctl = 0
8 -   };
9 +   memset(&header, 0, sizeof (struct ieee80211_hdr_3addrqos));
10 }
```

Figure 11: ***header*** **initialization (lines 4 - 8) is deleted. However, the same semantics is implemented as** ***memset*** **(line 9).**

Evaluation

➢ **False Negatives**

- Cpscan missed 276 bugs (the recall is 60%).

Incorrect graph matching (39%).
Different bounded use chains (40%).

```
/* drivers/net/ethernet/stmicro/stmmac/stmmac_platform.c*/
static int stmmac_probe_config_dt( ...){
    struct device_node *np = pdev->dev.of_node;
-   if (!np)
-       return -ENODEV;
-   *mac = of_get_mac_address(np);
    plat->interface = of_get_phy_mode(np);
+   of_property_read_u32(np, ...);
}
```

Figure 8: An example of a changed bounded use chain in an IoT kernel.

Conclusion

- Deep understanding of the bugs caused by code pruning in IoT kernels -perform the first comprehensive study on code pruning with a large corpus of real-world IoT kernels.

- New techniques - propose a new deterministic graph matching algorithm to precisely identify the DSOs in IoT kernels and solve the problem of security impact inference by comparing the bounded use chains of the security -critical variable associated with a DSO before and after the pruning.

- Comprehensive evaluation - find 114 new bugs in 28 IoT kernels from 10 popular IoT vendors, which affect billions of devices. These bugs can lead to critical security issues such as NULL pointer deference, memory leakage, and denial of service.

THANKS