CPscan: Detecting Bugs Caused by Code Pruning in IoT Kernels

Lirong Fu
Zhejiang University
fulirong007@zju.edu.cn

Shouling Ji*
Zhejiang University &
Binjiang Institute of Zhejiang University
sji@zju.edu.cn

Kangjie Lu
University of Minnesota
kjlu@umn.edu

Peiyu Liu
Zhejiang University
liupeiyu@zju.edu.cn

Xuhong Zhang
Zhejiang University &
Binjiang Institute of Zhejiang University
xuhongnever@gmail.com

Yuxuan Duan
Zhejiang University
22021304@zju.edu.cn

Zihui Zhang
Zhejiang University
zhangzihui@zju.edu.cn

Wenzhi Chen*
Zhejiang University
chenwz@zju.edu.cn

Yanjun Wu
Institute of Software, Chinese Academy of Sciences
yanjun@iscas.ac.cn

ABSTRACT

To reduce the development costs, IoT vendors tend to construct IoT kernels by customizing the Linux kernel. Code pruning is common in this customization process. However, due to the intrinsic complexity of the Linux kernel and the lack of long-term effective maintenance, IoT vendors may mistakenly delete necessary security operations in the pruning process, which leads to various bugs such as memory leakage and NULL pointer dereference. Yet detecting bugs caused by code pruning in IoT kernels is difficult. Specifically, (1) a significant structural change makes precisely locating the deleted security operations (*DSO*) difficult, and (2) inferring the security impact of a *DSO* is not trivial since it requires complex semantic understanding, including the developing logic and the context of the corresponding IoT kernel.

In this paper, we present CPscan, a system for automatically detecting bugs caused by code pruning in IoT kernels. First, using a new graph-based approach that iteratively conducts a structure-aware basic block matching, CPscan can precisely and efficiently identify the *DSO*s in IoT kernels. Then, CPscan infers the security impact of a *DSO* by comparing the bounded use chains (where and how a variable is used within potentially influenced code segments) of the security-critical variable associated with it. Specifically, CPscan reports the deletion of a security operation as vulnerable if the bounded use chain of the associated security-critical variable remains the same before and after the deletion. This is because the unchanged uses of a security-critical variable likely need the security operation, and removing it may have security impacts. The experimental results on 28 IoT kernels from 10 popular IoT vendors show that CPscan is able to identify 3,193 *DSO*s and detect 114 new bugs with a reasonably low false-positive rate. Many such bugs tend to have a long latent period (up to 9 years and 5 months). We believe CPscan paves a way for eliminating the bugs introduced by code pruning in IoT kernels. We will open-source CPscan to facilitate further research.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**.

KEYWORDS

Bug detection; Inconsistency analysis; Missing security operation; Static analysis

ACM Reference Format:
Lirong Fu, Shouling Ji, Kangjie Lu, Peiyu Liu, Xuhong Zhang, Yuxuan Duan, Zihui Zhang, Wenzhi Chen, and Yanjun Wu. 2021. CPscan: Detecting Bugs Caused by Code Pruning in IoT Kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3484738

1 INTRODUCTION

As the core of IoT devices, IoT kernels play an important role in ensuring the security, reliability, and stability of IoT devices. However, for tremendous device vendors, the development of a secure, reliable, stable, and efficient kernel requires high expertise costs and long development cycles. Therefore, massive downstream IoT vendors, e.g., ASUSWRT [1] and NETGEAR [8], tend to adopt the Linux kernel in billions of their commodity devices.

During the practical adoptions, nearly all IoT vendors customize the Linux kernel for their own needs by removing/adding code segments or using non-standard building configurations [26]. In the

*Shouling Ji and Wenzhi Chen are co-corresponding authors.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484738

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

customization process, considering the limited hardware and system resources of IoT devices, IoT vendors tend to delete "irrelevant" code segments from the Linux kernel to save computation power. Specifically, as shown in Table 1, thousands of source files are modified by directly deleting code lines. In general, most code pruning (code line deletions) performed by IoT vendors is reasonable, which satisfies the various needs of different downstream IoT kernels. However, due to the intrinsic complexity of the Linux kernel and the lack of long-term effective maintenance, some code pruning becomes unreliable and insecure. Specifically, many necessary security operations, including security checks, variable initializations, and resource-release operations, are mistakenly deleted, introducing various security bugs (memory leakage, denial of service, and NULL pointer dereference) that can affect the security and reliability of the whole IoT kernel. For instance, Figure 1 shows a concrete example of mistakenly deleting a security check (lines 6 - 7) in the kernel of an IoT router. This NULL pointer check ensures that *msg* cannot be NULL when used in the subsequent code segment (lines 8 - 9). Without this check, *msg* can reach an error state (being a NULL pointer) and cause a NULL pointer dereference. The deletion of security operations can make IoT kernels unreliable and insecure. Moreover, the resulting bugs will affect a large number of widespread IoT devices. Therefore, there is an urgent need for a practical approach to detect the bugs caused by code pruning in downstream IoT kernels.

To address this problem, it is intuitive to use state-of-the-art detectors to discover bugs in IoT kernels. Indeed, researchers have proposed many effective approaches to detect bugs in the Linux

```
1 /* drivers/char/n_gsm.c */
2 static void gsm_control_reply(struct gsm_mux *gsm, ...){
3     struct gsm_msg *msg;
4     msg = gsm_data_alloc(gsm, 0, dlen + 2, gsm->ftype);
5     // The deleted NULL pointer check
6 -   if (msg == NULL)
7 -       return;
8     msg->data[0] = (cmd & 0xFE) << 1 | EA;
9     msg->data[1] = (dlen << 1) | EA;
10 }
```

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.

kernel. Specifically, to detect 0-day bugs, UniSan [34], MemorySanitizer [46], Crix [33], PeX [58], Hector [42], and HFL [28] are proposed to report security bugs caused by missing security checks, missing variable initializations, and missing resource-release operations. However, these approaches are insufficient for our problem because they can only find bugs under certain conditions [33, 34, 42, 56, 58] or lead to heavy performance overhead when applied to complex IoT kernels [28, 46]. For instance, Crix [33] and PeX [58] are insufficient to detect missing security-check bugs caused by code pruning because they employ cross-checking, which has an inherent limitation: they must find enough similar cases for cross-checking. Hector [42] only targets the detection of error-handling code. To pinpoint N-day bugs in IoT kernels, Gemini [54], Genius [22], and DeepBinDiff [19] perform code similarity detection by referencing the known bugs. However, the bugs caused by code pruning are mostly new bugs. In summary, detecting security bugs caused by code pruning in IoT kernels remains an open problem.

By intuition, a proper way of solving this problem can be dividing it into two concrete sub-problems—❶ precisely locating which security operations are deleted in the downstream IoT kernels and ❷ analyzing whether each deleted security operation (*DSO*) introduces security risks. For instance, as shown in Figure 1, we need to precisely and efficiently locate the *DSO* (line 6) and then analyze the security impact of it. For solving the first problem, state-of-the-art approaches [6, 11, 19, 20, 22, 43, 54, 61] are inadequate. First, existing approaches [19, 22, 43, 54, 61] are non-deterministic [47], which only report a similarity score, but fail to report the exact code deletions. Second, the deterministic methods [6, 11, 20] also fail to precisely locate code pruning in IoT kernels because the structure of code segments may have been significantly changed during the customization process [26]. For instance, Unix-Diff [11] mistakenly reports simple code segment movements as code additions and deletions (as shown in §A.1). However, code segment movements are quite common in the customization process. The tree-based approach GumTree [20] uses abstract syntax trees (AST) to detect the added and deleted lines. However, for complicated code pruning, finding a correct AST match is difficult, which results in poor precision and recall. When comparing LLVM IR files, LLVM Diff [6] is very sensitive to CFG changes. Therefore, a significant CFG change in IoT kernels can cause massive false positives and false negatives. At present, *precisely locating the DSOs in an IoT kernel with significant structural changes is still a challenging problem.*

For the second problem, the removal of a security operation may not be a bug if it is indeed unnecessary. To determine if a *DSO* is still needed in an IoT kernel, we need to figure out whether the relevant code that a security operation protects is still reachable without the corresponding security operation. If so, the security-critical variable associated with the *DSO* may enter an erroneous state and hence cause bugs. In summary, we need to answer the following questions—in the Linux kernel, which security-critical variable is validated, freed, or initialized by the corresponding security operation? does this security-critical variable still exist in the corresponding IoT kernel? how and where is this security-critical variable used in the subsequent code segments? which potential error state can this security-critical variable reach without the corresponding security operations? However, it is hard to automatically answer these questions because security operations, the associated security-critical variable, and its uses are highly diverse in form. It is difficult to analyze them without field expertise, not to mention using universal rules/patterns to summarize them. Therefore, *automatically determining the security impact of a DSO requires complex semantic understanding, which is not trivial.*

To address the above problems, in this paper, we present CPscan, a system that overcomes the aforementioned two challenges with multiple novel techniques. For problem ❶, we use the idea of graph matching to perform precise code pruning identification because graph comparison can capture not only structural information but also semantic information [39]. However, the closely related graph matching approaches such as McGregor [36] and Koch [29] are unsuitable for our problem. Their limitations are twofold. First, they can only map the exact same basic blocks. Second, they suffer from a high computational complexity [39] Therefore, we need an efficient and precise code differential analysis to locate the *DSO*s in IoT kernels. To achieve this goal, we manually identify the *DSO*s in two randomly selected IoT kernels to obtain heuristics. From our empirical analysis, we observe that the security operations contained in each basic block are distinguishable and hence can be used to form a high-quality seed pool of the matched basic blocks. In addition, two basic blocks are highly possible similar if they point to or from the matched basic blocks in the seed pool. According to these observations, we propose a new deterministic graph matching approach. First, we represent the paired functions in the Linux kernel and an IoT kernel, as attributed control flow graphs (ACFG) [54]. Then CPscan performs an initial fast basic block matching to form a seed pool of the matched basic blocks by computing the similarity score between two basic blocks according to the highly distinguishable attributes (security operations) within each basic block. Finally, CPscan iteratively performs a structure-aware basic block matching on top of the seed pool, during which CPscan prioritizes the match of the neighbor nodes of the matched basic blocks in the seed pool. In this way, CPscan achieves both high precision and efficiency.

To solve problem ❷, CPscan employs inconsistency analysis to infer the security impact of a *DSO*. Specifically, we take a conservative approach by comparing the bounded use chains (where and how a variable is used within the potentially influenced code segments) of the security-critical variable associated with it. The intuition is that in the potentially influenced code segments of a security operation (e.g., the successor branches of a security check), suppose the security-critical variable protected, freed, or initialized by the security operation has the same uses as that in the original Linux kernel. In that case, we expect that the security operation is still necessary, and removing it will likely have a security impact. For example, in Figure 1, the bounded use chain of *msg* (lines 8 - 9) remains the same in the IoT kernel. Therefore, the security check (lines 6 - 7) is necessary, which prevents *msg* from being used as a NULL pointer and causing NULL pointer deference. With the above techniques, we can automatically infer if a security operation's deletion brings in security risks.

We implement CPscan on top of LLVM [7] as multiple static-analysis passes and evaluate it on 28 IoT kernels from 10 popular vendors in about 9 hours. CPscan features high accuracy and efficiency in locating *DSO*s in IoT kernel. Particularly, CPscan's precision of locating *DSO*s is about 85%. Besides, CPscan's efficiency in locating *DSO*s is 400 times faster than state-of-the-art graph matching approaches. In total, CPscan locates 3,193 *DSO*s. After automatically inferring their security impact, CPscan reports 359 potentially vulnerable deletions of security operations. By further manual analysis, we confirm 114 new bugs, 10 of which have been confirmed by the corresponding IoT developers. These bugs, which have been in the IoT kernels for a long time (up to 9 years and 5 months), can lead to critical security risks, including NULL pointer deference, memory leakage, and denial of service. The experimental results show that CPscan is accurate and effective in finding bugs caused by code pruning in IoT kernels. Overall, we make the following contributions:

- **Deep understanding of the bugs caused by code pruning in IoT kernels**. We perform the first comprehensive study on code pruning with a large corpus of real-world IoT kernels. We find that various security operations, including security checks, variable initializations, and resource-release operations, are mistakenly deleted during the code customization process by IoT vendors.
- **New techniques**. We propose a new deterministic graph matching algorithm to precisely identify the *DSO*s in IoT kernels by iteratively performing a structure-aware basic block matching. Furthermore, we 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**. We develop CPscan[1], the first system to perform highly reliable bug detection introduced by code pruning in IoT kernels. With CPscan, we 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.

2 PROBLEM UNDERSTANDING

2.1 Code Pruning in IoT kernels

Downstream IoT vendors widely customize the Linux kernel, hence code pruning is prevalent in IoT kernels to achieve better performance and a lower resource consumption. To better reveal this fact,

[1] https://github.com/zjuArclab/CPscan.

Table 2: The security impact of missing security operations.

Missing Security Operation	Security Impact
Security Check	Denial of service, NULL pointer dereference, Out-of-bound access, System crashes
Variable Initialization	Unintialized use
Resource Release	Resource leakage, Denial of service

we conduct an evaluation of code customization on 28 Linux-based IoT kernels from 10 popular IoT vendors. Specifically, we collect the source code of each IoT kernel and the corresponding Linux kernel. We then pair the functions in an IoT kernel and Linux kernel according to their function name and filter out the function pairs with no code changes by comparing their hash values. Finally, we count the number of the source files and functions that IoT vendors customized. As shown in Table 1, code customization is common in the investigated IoT kernels. All the studied IoT vendors customize the Linux kernel. Among them, DD-WRT [3] has performed the most customization. In total, there exists code customization in over 70,000 functions, which belong to the key subsystems of an IoT kernel, such as file systems, network software, and device drivers. Considering that obtaining precise code deletions is known as a complex task [20], we randomly select 500 customized functions and manually check them. Our analysis shows that 429 (85.8%) of them contain code pruning, which indicates that code pruning is prevalent in IoT kernels. Next, we elaborate on the security impact of the deleted code in these functions.

2.2 Security Impact of Code Pruning

The *DSO*s have a significant security impact on IoT kernels. In this work, CPscan mainly investigates three kinds of security operations that are particularly common (including security checks, variable initializations, and resource-release operations, as summarized in Table 2) and missing them can cause great security risks [48, 51]. Specifically, missing security checks can lead to insecure and unreliable running states of an OS, which further results in denial of service, NULL pointer deference, out-of-bound access, and so on. Uninitialized memory can store arbitrary values that are previously stored in it. Thus, missing initialization results in uninitialized use. If an uninitialized variable is used to control the execution flow of a program, e.g., the use of an uninitialized function pointer, the control flow of this program can potentially be hijacked. Resource-release operations in error handling code are usually used to recover a system from an error state. Deleting a needed resource-release operation may cause resource leakage, denial of service, and so on. More details about the security impact of the bugs found by CPscan is described in Table 13 in Appendix A.

2.3 Problem Scope

Our evaluation demonstrates the prevalence of code pruning in downstream IoT kernels. Considering its severe potential security risks, in this work, we focus on detecting the security bugs caused by code pruning in IoT kernels. Particularly, CPscan is designed only to identify the *DSO*s rather than all the deleted lines of code in IoT kernels. In this research, we have the following assumptions. First, the downstream IoT vendors are not intentionally malicious but may mistakenly remove some security operations in IoT kernels. Second, the customers of CPscan include vendors, the providers of security services, or the users of downstream IoT kernels. Third, the source code of an IoT kernel is available for CPscan, which is reasonable, as vendors should release their source code according to the license of Linux.

3 DESIGN OF CPSCAN

3.1 Workflow of CPscan

To detect bugs caused by code pruning in IoT kernels, CPscan first locates the *DSO*s via a structure-aware graph matching. Then, it performs security impact inference by comparing the bounded use chains of the security-critical variable associated with a *DSO*. We develop CPscan as an easy-to-use system, whose workflow is shown in Figure 2. CPscan consists of three phases: (1) *preprocessing* phase, which prepares the required graph input of our graph matching method; (2) *code pruning locating* phase, which utilizes the graphs generated in phase one to perform a structure-aware graph matching, obtains the maximum common subgraph (MCS), and locates the *DSO*s; and (3) *security impact inferring* phase, which performs inconsistency analysis to infer the security impact of the *DSO*s obtained in phase two. In the following, we elaborate on the design of each phase.

3.2 Preprocessing

Given the source code of kernels, CPscan aims to generate suitable graphs for the following structure-aware graph matching. Inspired by Gemini [54] and Genius [22], CPscan utilizes an attributed control flow graph (ACFG), a widely used representation in program analysis, to perform graph matching. In a traditional ACFG, the graph structure is the same as the control flow graph (CFG), and the vertexes are the basic blocks within a function. Each vertex is labeled with a set of attributes. However, the traditional ACFG has two limitations when used to precisely locate the *DSO*s. On the one hand, the attributes used in a traditional ACFG are coarse-grained. For instance, the attributes employed by Gemini [54] include the number of instructions and the number of function calls. It is difficult to match basic blocks with these unrepresentative attributes. On the other hand, a traditional ACFG directly uses a CFG as the graph structure, in which a function is split into basic blocks in a coarse-grained manner, and hence may cause a high false-positive and false-negative rate when locating the *DSO*s. For instance, Figure 3(a) and Figure 3(b) show part of two CFGs of a function, in which basic block 0 is matched with basic block 0′ with high similarity since they contain many identical attributes. Therefore, the security check in Figure 3(a) (basic block 1), which should be matched with that in Figure 3(b) (lines 11 - 13 in basic block 0′), will be mistakenly reported as a *DSO* (a false positive), because there is no basic block can be matched with it. To address this problem, according to our empirical analysis on two randomly chosen IoT kernels, CPscan employs more fine-grained and representative attributes within a basic block. Additionally, CPscan further splits a raw CFG in a fine-grained manner. Next, we introduce the attributes used in CPscan and how CPscan splits a raw CFG.

Attributes extraction. Basic-block attributes are used to evaluate the similarity of two basic blocks. Considering the problem we solve and prior insightful knowledge in this field [22, 54], the

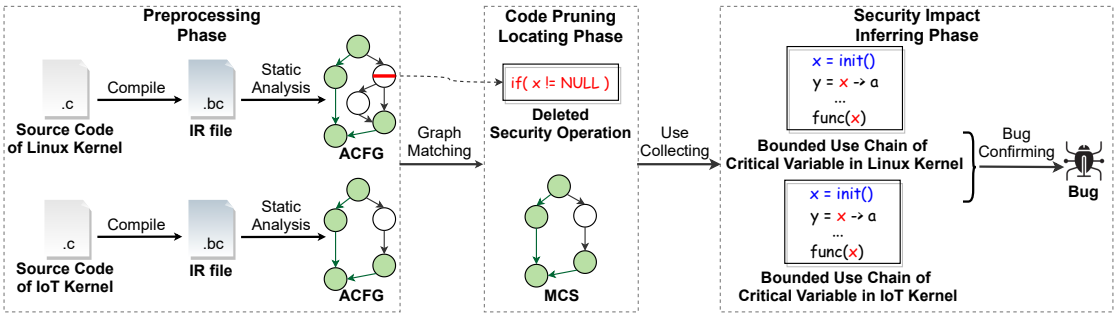

Figure 2: Workflow of CPscan. ACFG = Attributed control flow graph, MCS = Maximum common subgraph.

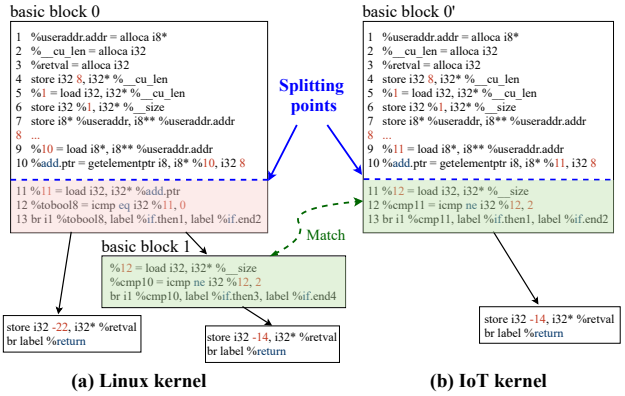

Figure 3: (a) and (b) are part of *ethtool_set_features* definition in Linux kernel and IoT kernel, respectively. CPscan divides one basic block into two at the splitting points.

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

basic-block attributes used in CPscan are obtained according to our empirical program analysis together with the metrics used in existing approaches [22, 54]. In this work, CPscan uses two types of attributes to characterize a basic block: (1) *statistical attributes* which describe the local representative characteristics within a basic block, and (2) *structural attributes* which show the position characteristics of a basic block in the whole ACFG. In total, we extract five types of statistical attributes and one structural attribute as listed in Table 3. For a basic block, *constant value* refers to constant strings, numbers, or other data structures. *Instruction sequence* and *function call sequence* record the order of instructions and function calls within a basic block, respectively. CPscan identifies three kinds of *security operations*: security checks, variable initializations, and resource-release operations. When identifying *security operations*, security checks are identified by locating conditional statements that sanitize erroneous states. Variable initializations are recognized by locating operations that zeroing/rewriting the allocated memory. Resource-release operations are identified by tracing the empirically determined function calls that free computing resources. *Instruction distribution* records the number of different types of instructions. *Neighbor nodes in MCS* stores the neighbor nodes that are contained in the *MCS*. It is worth noting that *neighbor nodes in MCS* are dynamically updated during the matching process. By contrast, other basic-block attributes can be obtained in the preprocessing phase.

CFG split. Based on the aforementioned analysis on the inaccurate basic block matching caused by the coarse-grained CFG splitting, we observe the root cause is that in a CFG, the basic block that contains a conditional statement may also contain many other instructions that are not related to this conditional statement. However, these instructions will influence the match of this conditional statement. Thus, we propose to split them apart with a fine-grained splitting method. Namely, when generating the CFG of a function in IR, except for the original branch jumps in a control flow, CPscan adds a new unconditional jump at the nearest splitting point (where the variables in an *if* condition are assigned) to an *if* condition. Specifically, the process of CFG split is as follows. (1) CPscan locates the conditional statement in a basic block. (2) CPscan identifies the "splitting point". (3) CPscan splits a basic block into two basic blocks at the "splitting point". For instance, we add new jumps between lines 10 and 11 in both Figure 3(a) and Figure 3(b). Such fine-grained splitting allows the security check in basic block 1 in Figure 3(a) to have the opportunity to be matched with lines 11 - 13 in Figure 3(b). Thus, CPscan can perform a precise basic block matching. It is worth noting that theoretically, CFG split can be used in any CFG-based deterministic code differential analysis regardless of the specific code representation.

3.3 Code Pruning Identification

After obtaining the ACFGs of a function pair, CPscan next performs a precise graph matching, obtains the *MCS* of the paired function, and finally identifies the *DSOs* in IoT kernels. It is worth noting that code customization in IoT kernels often results in a significant CFG change, which may divide an original basic block into many basic blocks. Only performing the one-to-one match (the match of one basic block to one basic block) may miss many basic block pairs, causing a high false-positive and false-negative rate. Therefore, to ensure a precise graph matching, we consider the one-to-one match and the one-to-many match (the match of one basic block to many

basic blocks). Particularly, the one-to-many match is bidirectional in an IoT kernel and Linux kernel.

Inspired by the way developers manually match ACFGs, we design an efficient and precise graph matching algorithm. For the one-to-one match, the main idea is that when *MCS* is empty, we use the basic blocks that contain security operations as seed basic blocks to guide an initial fast basic block matching. The reason is that the basic blocks that contain the same security operations are more likely to be matched. After updating *MCS* with the matched basic block pairs, we utilize *MCS* to guide the match of the neighbor nodes of the already matched basic block pairs. The intuition is that two basic blocks are highly possible similar if they point to or from the same matched basic blocks. In this way, most one-to-one matches in a paired function can be precisely and efficiently identified. In a one-to-many match, for a basic block that is similar to N basic blocks, we first carefully construct a new basic block by linking the N similar basic blocks according to their CFG order and then perform a one-to-one match for the newly constructed basic block to confirm the correctness of this one-to-many match. If the similarity of the one-to-one match is lower than the threshold we define, CPscan reorders the N similar basic blocks and links them to a new basic block again to repeat the one-to-one match. If the one-to-one match fails for all the permutations of the N basic blocks, this match is discarded.

Algorithm 1: The algorithm of locating *DSO*s

Input: Two attributed control flow graphs $ACFG_L$ and $ACFG_I$; Two sets of basic blocks SBB_L and SBB_I that contain security operations in $ACFG_L$ and $ACFG_I$, respectively; Two sets of basic blocks CBB_L, CBB_I that do not contain any security operations in $ACFG_L$ and $ACFG_I$, respectively

Output: The *MCS* of $ACFG_L$ and $ACFG_I$; The deleted security operations *DSO* in $ACFG_I$

1 **CalDSO:**
while *(!MCS.Isempty())* **do**
2 $(bb_L, bb_I) \leftarrow MCS.pop()$;
3 $Neighbors_L \leftarrow GetNeighbors(bb_L)$;
4 $Neighbors_I \leftarrow GetNeighbors(bb_I)$;
5 $NodePairs = Match(Neighbors_L, Neighbors_I)$;
6 $MCS \leftarrow MCS \cup NodePairs$;
7 **foreach** sbb_L **in** SBB_L **do**
8 **foreach** sbb_I **in** SBB_I **do**
9 **if** $GetSimilarity(sbb_L, sbb_I) > threshold$ **then**
10 $MCS \leftarrow MCS \cup (sbb_L, sbb_I)$;
11 $DSO \leftarrow DSO \cup GetDSO(sbb_L, sbb_I)$;
12 **goto CalDSO;**
13 **foreach** cbb_I **in** CBB_I **do**
14 **if** $GetSimilarity(sbb_L, cbb_I) > threshold$ **then**
15 $MCS \leftarrow MCS \cup (sbb_L, cbb_I)$;
16 $DSO \leftarrow DSO \cup GetDSO(sbb_L, cbb_I)$;
17 **goto CalDSO;**
18 $RBB_L \leftarrow ACFG_L - MCS$;
19 $RBB_I \leftarrow ACFG_I - MCS$;
20 $(L_1I_N, L_NI_1) \leftarrow OneToMany(RBB_L, RBB_I)$;
21 $(bbL, Nbb_I) \leftarrow ConvertToOneToOne(L_1I_N)$;
22 $(Nbb_L, bb_I) \leftarrow ConvertToOneToOne(L_NI_1)$;
23 $MCS \leftarrow MCS \cup (bbL, Nbb_I) \cup (Nbb_L, bb_I)$;
24 $DSO \leftarrow DSO \cup GetDSO(bbL, Nbb_I) \cup GetDSO(Nbb_L, bb_I)$;

Algorithm 1 shows how we perform the graph matching and then obtain the *MCS* and *DSO* of a function pair. In the beginning, CPscan divides the basic blocks within an ACFG into two groups: (1) the sensitive basic blocks (*SBB*), which contain the basic blocks that have security operations and serve as the seed set of basic blocks to be matched in the initial round of match, and (2) the common basic blocks (*CBB*) which include the remaining basic blocks in an ACFG. CPscan first prepares SBB_L and SBB_I for a paired function in the Linux kernel and IoT kennel, respectively. Then, CPscan transverses each basic block pair (sbb_L, sbb_I) in SBB_L and SBB_I to perform a fast basic block matching by comparing their similarity with a threshold θ_0, which is set to a high value to ensure the quality of this initial basic block matching. In particular, CPscan obtains the similarity of two basic blocks by averaging all the similarity scores of the attributes within the basic block pair. If the similarity between sbb_L and sbb_I is larger than θ_0, CPscan inserts this basic block pair into *MCS* (line 10), compares the instruction sequences in sbb_L and sbb_I to obtain the *DSO*s by *GetDSO* (line 11), and jumps to label *CalDSO* to further match the neighbor basic blocks of a matched basic block pair (lines 2 - 6). However, if sbb_L cannot be matched with any node in SBB_I, CPscan continues to match it with the basic blocks in CBB_I (lines 13 - 17).

For the remaining basic blocks that are not matched in the one-to-one match, CPscan performs a one-to-many match (lines 18 - 22). Specifically, for each node rbb_L in RBB_L (the remaining unmatched basic blocks in $ACFG_L$), the function *OneToMany* (line 20) collects the basic blocks, whose similarity with rbb_L is higher than θ_1, in RBB_I. θ_1 is defined to be relatively lower than θ_0, indicating that when the similarity of a basic block pair is higher than θ_1 and lower than θ_0, these two basic blocks are relatively similar, but CPscan cannot match them in a one-to-one manner. After bidirectly conducting the above collections for each node in RBB_L and RBB_I, the function *OneToMany* (line 20) returns L_1I_N (the basic block pairs that match one basic block in the **L**inux kernel with N basic blocks in the **I**oT kernel) and L_NI_1 (the basic block pairs that match N basic blocks in the **L**inux kernel with one basic block in the **I**oT kernel). Then, by the function *ConvertToOneToOne* (line 21), CPscan constructs a new basic block by linking the N basic blocks according to their CFG order and performs the one-to-one match for the newly generated basic block. If the similarity of the one-to-one match is higher than θ_0, function *ConvertToOneToOne* (line 21) returns a newly matched pair. Consequently, CPscan can obtain the resulting *MCS* and *DSO* (lines 23 - 24).

3.4 Security Impact Inference

After obtaining the *MCS* and *DSO*s of a function pair in an IoT kernel and Linux kernel, CPscan should perform security impact inference for all the *DSO*s. However, the removal of a security operation may not introduce security bugs because the security

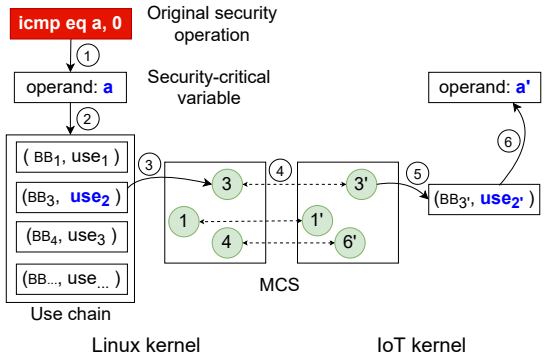

Figure 4: (1) Identify the security-critical variable a. (2) Obtain the use chain of a. (3) check which use is contained in the maximum common subgraph; (4) Find the corresponding paired basic block $3'$ in IoT kernel. (5) Locate the identical use use'_2. (6) Identify the corresponding security-critical variable a' in IoT kernel.

operation may no longer be needed or have been changed in the IoT kernel. Therefore, to improve the accuracy of CPscan and decrease potential false positives, it is unnecessary for CPscan to further consider those reasonable DSOs. Based on our observation, those reasonable DSOs are mainly caused by three factors: *legitimate program patch*, *code encapsulation* (syntactically similar code that is moved to a new function), and *code re-implementation* (syntactically dissimilar code segments which implement the same functionality). Therefore, CPscan first filters out the *DSO*s caused by the first two factors mentioned above to decrease false positives. For the third factor, we point out that code re-implementation is out of the scope of this work, which is known as a difficult task that needs further investigation [43]. Therefore, CPscan currently cannot filter out the *DSO*s caused by code re-implementation. Then, CPscan needs to infer the security impact of the remaining *DSO*s. In this step, if the security-critical variable has the same uses as that in the original Linux kernel, we expect that the security operation is still necessary, and removing it will likely have a security impact. Therefore, we conservatively determine the impact of a *DSO* by comparing the bounded use chains of the security-critical variable associated with it. Next, we elaborate on how we recognize program patch and code encapsulation, find the security-critical variable, and compare the bounded use chains of the security-critical variable.

Program patch and code encapsulation recognition. As discussed before, the *DSO*s caused by legitimate program patch and code encapsulation are reasonable and do not need further security analysis. Therefore, to filter out reasonable *DSO*s caused by program patch, CPscan first obtains the patch information of the corresponding function from the git commit information of the Linux kernel [4]. Then, CPscan confirms whether a *DSO* is contained in a patch through a text-based comparison method. Specifically, CPscan searches each code line in a patch to check if the *DSO* is included in this patch. If so, CPscan ceases to analyze this *DSO*. Then, CPscan recognizes code encapsulation by comparing the deleted code sequence to the code sequence implemented in the corresponding new function. If the edit distance [41] of these two code sequences is small (no larger than θ_{ed}, which indicates

```
1 /* drivers/mtd/spi-nor/hisi-sfc.c*/
2 static int hisi_spi_nor_probe(struct platform_device *pdev){
3 +    struct hisi_fmc *fmc = dev_get_drvdata(dev->parent);
4 +    host->regbase = fmc->regbase;
5 -    host->regbase = devm_ioremap_resource(dev, res);
6 -    if (IS_ERR(host->regbase))
7 -        return PTR_ERR(host->regbase);
8 }
```

Figure 5: A deleted security check in an IoT kernel, in which the source of security-critical variable *host->regbase* has been changed from the return value of a function call to a member of *fmc*.

these two code sequences are syntactically similar) and the original *DSO* remains in the new function, this *DSO* also does not need further analysis.

Security-critical variable determination. This step aims to find the paired security-critical variables associated with a *DSO* in an IoT kernel and Linux kernel. However, the security-critical variable, which is initialized, freed, or validated by a security operation, appears in diverse forms in IR. For instance, a security-critical variable can be the parameter or return value of a function call, a global variable, or a Macro. Therefore, we first manually identify several paired security-critical variables for the *DSO*s to summarize a suitable pattern to determine them. From our empirical analysis on two randomly selected IoT kernels, CPscan can identify a security-critical variable according to the following two insights: (1) a security-critical variable is closely associated with a *DSO* and is usually the parameter or the return value of this *DSO* or the propagation of them; and (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*.

CPscan first collects the security-critical variable for a *DSO* in the Linux kernel. Next, CPscan needs to find the corresponding security-critical variable in the IoT kernel. Unfortunately, the security operation has been deleted in the IoT kernel. We do not know which variable in the IoT kernel should be matched with that in the Linux kernel. To address this problem, CPscan utilizes the previously obtained *MCS* to infer the corresponding security-critical variable in the IoT kernel. Specifically, as shown in Figure 4, *icmp eq a, 0* is a *DSO* in IoT kernel. a is the security-critical variable in the Linux kernel. CPscan first obtains the use chain of a. Then CPscan transverses the *MCS* and finds that basic block 3 in the Linux kernel is matched with basic block $3'$ in the IoT kernel. Through basic block $3'$, CPscan gets the identical use of a in basic block $3'$ and finally obtains the corresponding security-critical variable a' in the IoT kernel. After obtaining the matched security-critical variables, we need to confirm that their sources (where a critical variable propagates from) are identical. Otherwise, the IoT kernel may not need the *DSO*. For instance, as shown in Figure 5, a NULL pointer check (lines 6 - 7) is deleted in *hisi_spi_nor_probe*, which is reasonable and does not need further analysis since the source of the security-critical variable *host->regbase* has been changed from the return value of a function call to a member of another variable. Further manual analysis shows that the newly added function call *dev_get_drvdata* has performed this NULL pointer check. Now, for each *DSO*, CPscan maintains a mapping, which stores the paired security-critical variables in an IoT kernel and Linux kernel.

Bounded use chain generation and comparison. After obtaining the paired security-critical variables of a *DSO*, a straightforward method of security impact inference is performing inconsistency analysis on all the uses of a security-critical variable. However, significant code customization in an IoT kernel is highly likely to change the use chain of a variable. Therefore, comparing all the uses of a security-critical variable might result in a high false-negative rate for bug detection because many irrelevant use changes are also compared. To address this problem, we also conduct an empirical study to infer the security impact of a *DSO* by manually analyzing the uses of a security-critical variable. Our empirical research indicates that (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. Therefore, CPscan infers the security impact of a *DSO* by comparing the bounded use chains (where and how a variable is used in the potentially influenced code segments). Specifically, CPscan first obtains all the uses of a security-critical variable. Then, it determines the use boundary for a *DSO*. For a security check, CPscan considers all the uses in the successors of the check condition; for variable initializations and resource-release operations, all the subsequent uses of the security-critical variable are considered. Only if all of these bounded use chains remain the same, can CPscan report that this security operation deletion brings in a new security risk.

4 IMPLEMENTATION

We implement CPscan on top of LLVM (of version 10.0.0) with multiple passes [7], which are based on LLVM data-flow and control-flow analysis. In total, CPscan contains about 6.5K lines of C++ code.

4.1 Preprocessing

The preprocessing phase provides the required graph inputs for the following analysis, consisting of a source-code filter, IR generation and normalization, and ACFG generation.

Source-code filter. The source-code filter reports all the paired functions that IoT vendors customize. First, CPscan pairs functions in an IoT kernel and the corresponding Linux kernel by their function name. Then, CPscan recognizes function customization by comparing their hash values, which is obtained by hashing the string concatenating all the normalized instructions in a function. Specifically, we generate the hash value for a function by the following two steps: (1) obtaining a new string by concatenating all the normalized instruction sequences in the function, and (2) generating the hash value for this string by the MD5 message-digest algorithm [10] and using this hash value to represent this function. If their hash values are different, CPscan performs further analysis as described in the following sections.

IR generation and normalization. CPscan invokes LLVM to compile the source files into LLVM IR files. Specifically, we compile an IoT kernel under the architecture on which an IoT vendor has performed the most code customization. Moreover, to obtain as many IR files as possible, similar to existing static analysis [32], we use *allyesconfig* in the kernels during the compiling process. Then, we compile the corresponding Linux kernel by using as many of the same building configurations as possible in the IoT kernel to exclude the IR differences caused by different building configurations. Additionally, we use a compiler option "-g" to generate debugging information, which is helpful for manual analysis. Generally, compiling kernels into LLVM IR often has compatibility issues. For source files that cannot be compiled successfully, we choose to discard them. After obtaining the IR files, CPscan performs IR normalization to exclude unnecessary IR changes. First, CPscan removes minor debug information, including metadata and prefetch instructions. Second, due to the static single assignment form [9] of LLVM, memory behavior in IR (loading or storing a variable) changes variable names, which causes massive false positives in basic block matching. Therefore, CPscan uniforms the name of a numeric operand as a constant string "Var" concatenated with the position of this operand as the suffix. For instance, in Figure 3, CPscan normalizes *%1* in *store i32 %1, i32* %__size* (line 6) to $Var0$.

ACFG generation. In this step, CPscan generates ACFGs for a precise graph matching. First, when constructing a CFG, to avoid the path explosion problem, we unroll the loops by treating for and *while* statements as if statements according to [52, 53]. Then, based on the standard control-flow analysis, CPscan identifies the "splitting point" in IR, i.e., the positions where the variables in an if condition are assigned. In brief, CPscan first identifies the if condition in IR by LLVM *getCondition* API. Then CPscan obtains the variables in this condition. For each variable, CPscan iteratively performs backward analysis to get its source until meeting a local variable or global variable. CPscan returns the nearest variable source position to the if condition as a "splitting point" to exclude extra code segments in the newly generated basic block. In this way, CPscan obtains a fine-grained CFG. Then, CPscan transverses each basic block to obtain the attributes (as described in Table 3). Specifically, in LLVM, a constant value can be identified by its instruction type. Then, CPscan identifies the three kinds of security operations described in Table 2. For security checks, CPscan obtains a list of conditional statements with at least one normal branch and one error handling branch through existing work [32]. For variable initialization, CPscan searches the store instructions that assign zero to a variable and the call instructions that are employed to initialize a variable, including *memset* and *memcpy*. For resource-release operations, CPscan traces the empirically identified function calls that free resources such as dev_kfree_skb, usb_free_urb, and $kfree$.

4.2 Code Pruning Identification

After obtaining the ACFGs of a function pair in an IoT kernel and Linux kernel, CPscan performs a precise graph matching to identify the *DSO*s in the IoT kernel. During the matching process, CPscan calculates the similarity between two basic blocks by averaging all the similarity scores of the attributes described in Table 3. Particularly, CPscan calculates the similarity score of each type of attribute using the known classic algorithms [2, 5, 41]. For *constant value* and *neighbor nodes in MCS*, CPscan calculates their similarity score by Jaccard index [5]. For *instruction sequence* and *function call sequence*, CPscan calculates their similarity score by the levenshtein metric [41]. The similarity score of a pair of *instruction distribution* (a vector, in which each element is the count of a type of instruction) is obtained by calculating their cosine distance [2].

Moreover, we perform a set of principled parameter evaluations on two randomly selected IoT kernels (whose IDs are 2 and 3 in Table 1) to obtain a set of suitable thresholds. We set θ_0 and θ_1 to be 0.95 and 0.6 respectively, to achieve the highest graph matching precision. The graph matching results showed in Table 4 demonstrate that CPscan can also achieve high accuracy when performing *DSO* identification on unseen IoT kernels, which indicates the adopted thresholds are suitable and generic.

4.3 Security Impact Inference

This section elaborates on the implementation details in security impact inference. First, we propose to find a security-critical variable according to the two rules described in §3.4. Particularly, when CPscan fails to find the subsequent uses for an associated variable, it iteratively performs backward analysis to check a new associated variable, i.e., the parameters or return values of the previously checked variable until CPscan finds a variable that satisfies the previously described two rules. CPscan chooses not to use the alias analysis [24] through the "AliasAnalysis" class in LLVM to trace a use chain due to its poor accuracy. Instead, CPscan obtains the use chain of a security-critical variable by checking the use-def chain defined in LLVM. Then CPscan extends the use chain with the propagated use chains of the security-critical variable's propagation. CPscan can achieve a relatively precise data flow analysis in identifying the security-critical variable's propagation. Specifically, when performing pointer analysis, CPscan traces and analyzes a set of LLVM instructions that can propagate a variable to other variables, such as *GetElementPtrInst*, *BitCastInst*, and so on.

5 EVALUATION

In this section, we first describe the experimental setup (§5.1). Then, we evaluate the accuracy and efficiency of CPscan in code pruning identification (§5.2) and security impact inference (§5.3). Finally, we present the results of bug detection (§5.4) and elaborate on the causes of false positives and false negatives (§5.5).

5.1 Experimental Setup

All the experiments are conducted on a machine with 64GB RAM and an Intel CPU (Xeon R CPU E5-2680, 20 core), running Ubuntu 16.04.10 LST with LLVM version 10.0.0 installed.

Datasets. We evaluate CPscan on two datasets. (1) A *real dataset* consists of 28 IoT kernels from 10 popular IoT vendors, which is used for evaluating the security state of the real-world IoT kernels (as shown in Table 1). (2) A *synthetic dataset* consists of 4 manually modified kernels obtained by deleting some code segments in the randomly chosen Linux kernels. This dataset contains as many various and complex *DSO*s as possible and is used for a straightforward comparison of *DSO* identification and false-negative evaluation (as shown in Table 9 in Appendix A). While the real dataset is pragmatic, the synthetic is comprehensive, covering representative cases. Particularly, the synthetic dataset is constructed by following the deletions performed by real-world IoT developers, namely containing as various *DSO*s as possible. Compared to the original Linux kernels, 700 security operations are deleted in the synthetic dataset, including 261 security checks, 214 variable initializations, and 225 resource-release operations.

5.2 Code Pruning Identification

As described in §3.3, we propose a structure-aware graph matching approach to locate the *DSO*s, which is an important technique in CPscan. In this section, we evaluate the accuracy and efficiency of this approach by comparing it with the state-of-the-art approaches.

To the best of our knowledge, there is no dedicated tool that can be used to identify the *DSO*s in IoT kernels. Thus, we compare CPscan with the most related state-of-the-art code differential analysis tools that can report deterministic code pruning, including GumTree [20], LLVM-Diff [6], and LLVM-Diff-N (an enhanced LLVM-Diff implemented by us via normalizing LLVM IR before adopting LLVM-Diff to reduce the false positives caused by memory behavior and debug information in IR). However, as discussed in §2.3, the design goals of CPscan and the baseline approaches are different. Specifically, CPscan aims to locate the *DSO*s, while the baselines aim to find all the deleted code lines. For these baselines, directly evaluating their precision on the task of locating *DSO*s might be unfair for them since the reported *DSO*s are a small part of all the reported code deletions, which results in very low precision for them. Therefore, for a fair comparison, we first evaluate the precision and recall of CPscan and the baselines on their own task with the synthetic dataset. Furthermore, the deleted code lines reported by the baselines may also contain *DSO*s. Thus, we also compare the recall (how many manually labeled *DSO*s can be located) of CPscan and the baselines for locating the *DSO*s with the real dataset. In this way, we can fairly compare the accuracy of CPscan and the baseline approaches.

Accuracy on the synthetic dataset. We evaluate the precision and recall of CPscan and state-of-the-art approaches for locating the *DSO*s and the deleted code lines, respectively. The results are shown in Table 10 and Table 11 (deferred to Appendix A). From the experimental results, the precision and recall of the baseline approaches are low. For instance, the precision and recall of LLVM-Diff are 46.0% and 25.5%, respectively, which indicates that this tool mistakenly reports wrong deletions, and at the same time misses many real deletions because LLVM-Diff is very sensitive to control flow changes. Besides, the precision of LLVM-Diff-N is about one time higher than that of LLVM-Diff, demonstrating that LLVM memory behavior indeed causes massive false positives. Thus, IR normalization in code differential analysis is necessary. For CPscan, its average identification precision of locating the deleted security checks, variable initializations, and resource releases is 96.0%, 98.0%, and 99.0%, respectively, which shows that the accuracy of CPscan is high in locating different kinds of *DSO*s, which benefits from our structure-aware graph matching approach.

Accuracy on the real dataset. As discussed before, we further compare the recall of CPscan and the baselines for locating the *DSO*s in the real-world IoT kernels. Specifically, we manually labeled all the 811 *DSO*s in 8 randomly chosen IoT kernels. Then, we use CPscan and the baselines to locate them. The experimental results are shown in Table 4. The average recall of GumTree, LLVM-Diff, and LLVM-Diff-N is 66.0%, 11.0%, and 11.0%, respectively. The experimental results show that the structural change in code customization is significant in the real-world IoT kernels, which leads to poor code differential analysis performance for the baselines. Particularly, LLVM-Diff and LLVM-Diff-N are barely usable with

Table 4: Performance of locating *DSO*s 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%**

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

such a low recall. By contrast, by utilizing a precise graph matching, the recall of CPscan is 94.9%, which is 44% - 763% higher than the baselines. Additionally, CPscan is specifically designed for locating the *DSO*s. Therefore, we also evaluate its precision (85.4%). Though the identification precision of CPscan decreases a little on the real dataset, compared to the state-of-the-art approaches, CPscan still can precisely locate most *DSO*s. The experimental results demonstrate that our structure-aware graph matching approach enables CPscan to effectively identify the *DSO*s in the real-world IoT kernels.

Efficiency on the real dataset. Efficiency is also important for locating code pruning because there are abundant customized functions in IoT kernels. We evaluate GumTree, LLVM-Diff, LLVM-Diff-N, and CPscan on over a half of the IoT kernels in the real dataset (whose IDs are from 1 to 17 in Table 1) to compare their efficiency. As shown in Table 5, the baseline approaches have high efficiency, which takes few seconds to analyze one file on average. Among them, LLVM-Diff performs the best (0.93 seconds per file). Compared to the baseline tools, CPscan takes more time to locate the *DSO*s (4.05 seconds per file, which is acceptable in practical settings). This result is reasonable because CPscan utilizes a graph-based approach, which is inherently more accurate but more time-consuming than the text-based or AST-based approaches used by the baselines. Furthermore, we compare the efficiency of CPscan with McGregor [36], the most closely related MCS algorithm. On average, the efficiency of the graph matching in CPscan (4.05 seconds per file) is about 400 times faster than McGregor.

5.3 Security Impact inference

As described in §3.4, after obtaining the *DSO*s in an IoT kernel, CPscan performs security impact inference to detect potential bugs caused by code pruning. Specifically, (1) CPscan filters out the reasonable *DSO*s caused by *legitimate program patch* and *code encapsulation*. (2) For the remaining *DSO*s, CPscan automatically identifies the security-critical variables associated with them. (3) CPscan compares the bounded use chains of the security-critical variables to confirm whether the remaining *DSO*s are vulnerable.

Table 6: Performance of security impact inference.

ID	*DSO* Filter			Security-critical Variable Identification			Bug Detection		
	TP	Pre.	Re.	TP	Pre.	Re.	TP	Pre.	Re.
1*	N/A	N/A	N/A	127	77%	76%	96	45%	61%
3*	N/A	N/A	N/A	143	81%	79%	122	48%	68%
6	47	97%	77%	24	85%	85%	2	40%	67%
10	33	94%	82%	39	79%	79%	3	33%	100%
12	13	86%	81%	17	80%	77%	1	15%	50%
22	20	83%	80%	23	82%	74%	5	50%	63%
Average	28.3	90.5%	80.2%	62.2	81.2%	78.7%	38.2	38.0%	67.9%

```
1 /*arch/mips/mm/init.c*/
2 void copy_from_user_page(struct vm_area_struct *vma, ...){
3 -   if (cpu_has_dc_aliases)
4 -       SetPageDcacheDirty(page);
5     ...
6 }
```

Figure 6: The variable compared in the conditional statement is not the real security-critical variable for the *DSO*.

According to the above three procedures, as shown in Table 6, we evaluate the precision and recall of each step on 4 randomly chosen real-world IoT kernels (whose IDs are 6, 10, 12, and 22 in Table 1) and 2 randomly selected synthetic kernels (whose IDs are 1* and 3* in Table 9 in Appendix A). Particularly, the synthetic dataset is constructed by manually deleting the security operations in the Linux kernel. Thus, there is no *DSO* that needs to be filtered out.

For step one, we manually identify the reasonable *DSO*s that need to be filtered out in the evaluated real-world IoT kernels. Then, we evaluate how many reasonable *DSO*s CPscan can identify. From Table 6, the average precision and recall of *DSO* filter are 90.5% and 80.2%, respectively, which demonstrates that CPscan is effective when filtering out the reasonable *DSO*s. The false positives and false negatives are mainly caused by inaccurate graph matching, which will be further discussed in §5.5. For step two, we confirm whether the reported security-critical variable is the one protected, initialized, or freed by the corresponding security operation. As shown in Table 6, the precision and recall of security-critical variable identification are 81.2% and 78.7%, respectively. This experiment demonstrates that the rules used to identify the security-critical variable in CPscan can correctly identify most security-critical variables. The false positives and false negatives mainly happen when identifying the security-critical variables associated with the deletion of permission checks. For instance, as shown in Figure 6, the checked variable is *cpu_has_dc_aliases* (line 3), which is a Macro used to determine whether the OS allows the alias of CPU cache, while the actually protected variable is *page*.

For the final step, the precision and recall of bug confirmation are 38.0% and 67.9%, respectively. The accuracy is consistent with the state-of-the-art approach [33]. Most false positives and false negatives are caused by the code re-implementation in the *DSO* filter and inaccurate graph matching, which can be easily identified with limited manual efforts and will be further discussed in §5.5.

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**

5.4 Bug Detection

By running CPscan over all the 28 IoT kernels in the real dataset, CPscan analyzes 74,542 customized functions in total. CPscan finishes this analysis in about 9 hours and reports 359 potential bugs caused by code pruning. This section elaborates on the results of bug detection and the comparison to other bug detectors.

New bugs. As shown in Table 7, CPscan identifies 3,193 *DSO*s in the real dataset, including 2,072 deleted security checks, 867 deleted variable initializations, and 254 deleted resource-release operations. The experimental results demonstrate that the deletion of security operations is pervasive in IoT kernels. Moreover, CPscan automatically reports 359 potential bugs caused by code pruning. To manually confirm all these reported bugs, three researchers spend about 38 person-hours on analyzing their developing logic and root causes. The manual efforts are mainly spent on confirming the security impact of the *DSO*s, including (1) the use analysis of the security-critical variable that can potentially bring in security risks and (2) the reachability analysis of a bug by tracing the call chain between the attacker-controllable functions to the function that contains the *DSO*s. Finally, we confirm 114 new bugs, consisting of 76 missing security-check bugs, 22 missing variable initialization bugs, and 16 missing resource-release bugs. We have submitted these bugs to the corresponding IoT developers, 10 of which have been accepted, and the remaining of them are under review. We also evaluate the distribution of the discovered *DSO*s, which is deferred to §A.3 (as shown in Figure 10).

The existence time of bugs. We find that compared to the Linux kernel, IoT kernels are updated slowly because of compatibility issues. In this paper, the newest kernel version of the tested IoT devices is 4.9.198. Overall, the existence time of the bugs is long. Specifically, we compute the latent period of the reported bugs and find that the longest time between the first release of the IoT kernel that contains bugs and our detection is 3,435 days (approximately nine years and five months).

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%

```
1 /* net/bridge/br_multicast.c*/
2 static int br_ip4_multicast_igmp3_report(struct sk_buff *skb, ...)
3 {
4     struct igmpv3_report *ih;
5 -   if (!pskb_may_pull(skb, sizeof(*ih)))
6 -       return -EINVAL;
7     br_vlan_get_tag(skb, &vid);
8 }
```

Figure 7: A bug caused by code pruning in an IoT kernel.

Case study. Figure 7 shows a concrete example of mistakenly deleting a security check (lines 5 - 6) in the kernel of an IoT router. *pskb_may_pull* ensures that the length of the data stored in the memory space pointed by *skb->data* is at least as long as the IP header since each IP packet must include a complete IP header. Now that the length of the data will not be checked, the following uses of IP packets may cause failures in parsing an IP packet, denial of service, and so on. Particularly, Crix [33], which aims to detect missing security-check bugs, fails to report this bug due to the lack of sufficient reference slices to enable cross-checking. More details about the confirmed bugs are listed in Table 13, in Appendix A.

Comparison to the related bug detection methods. We compare our system to Crix [33] and PeX [58] (two state-of-the-art detectors used to discover bugs in the Linux kernel) to further understand the usefulness of CPscan. Specifically, we use these tools to detect the missing security-check bugs in three randomly chosen real-world IoT kernels (whose IDs are 6, 10, and 22 in Table 1) and 2 randomly selected synthetic kernels (whose IDs are 1* and 3* in Table 9). Particularly, Crix and CPscan can finish the analysis of each kernel in Table 8 within an hour. By contrast, Pex cannot finish the same detection within four days. Thus, for PeX, we only analyze the bugs reported within four days.

As shown in Table 8, Crix's false-positive rate is 61.5%, which is similar to that (65%) reported in the original paper [33]. However, Crix detects 6 and 0 missing security-check bugs discovered by CPscan in the synthetic dataset and the real dataset, respectively. The average recall of Crix on the synthetic dataset is only 4.7% because the synthetic dataset mainly contains the bugs caused by code pruning rather than the missing security-check bugs that may have sufficient reference paths. Thus, we report the recall of Crix to be N/A. The average precision of PeX is 34.4%. However, it can find

none out of the 74 missing security-check bugs found by CPScan in the evaluated kernels. Similar to the evaluation of Crix, we also report the recall of PeX to be N/A because it mainly focuses on the detection of missing permission checks. The evaluation of baselines demonstrates that the state-of-the-art approaches are insufficient to detect bugs caused by code pruning. By contrast, CPScan detects 74 missing security-check bugs caused by code pruning. The false-positive rate of CPScan is 58.3%, which is similar to that of the state-of-the-art static method Crix. Moreover, the false positives can be easily identified with limited manual efforts (less than 1 minute per FP), which will be further discussed in §5.5.

From the experimental results, we observe that the distribution of the bugs targeted by these three detectors is very different, which indicates that these detectors actually complement each other. PeX and Crix aim to detect bugs such as missing permission checks and API misuses. Thus, they are insufficient to detect bugs found by CPScan. By contrast, CPScan is a complementary system that completes this missing task. In summary, existing state-of-the-art approaches are insufficient to solve the problem studied in this paper—detecting bugs caused by code pruning.

5.5 Accuracy of Bug Detection

False positives. As shown in Table 7, CPScan analyzes 3,193 *DSO*s. Though 245 out of 359 bugs (68%) reported by CPScan are false positives, CPScan can correctly analyze the remaining 2,834 (89%) *DSO*s. Next, we investigate the detailed causes of the false positives on the real dataset. (1) *Code re-implementation* (36%). Currently, CPScan can successfully identify 3,193 *DSO*s, in which only 89 (2.7%) *DSO*s are false positives caused by code re-implementation. These 89 false positives happen in *DSO* identification. For instance, in Figure 11 (deferred to §A.4), CPScan reports the initialization of *header* is deleted, which is a false positive because *header* is initialized by *memset*. Fortunately, these false positives can be identified with limited manual efforts. One can realize that the deleted variable initialization in Figure 11 within a few seconds. Thus, code re-implementation can hardly influence the effectiveness of CPScan.(2) *Inaccurate graph matching* (51%). Precisely locating code additions and deletions is known as a complex task [20]. Though CPScan has achieved high accuracy on graph matching (as shown in §5.2, both the identification precision and recall of CPScan on identifying the DSOs in the synthetic dataset are over 96%. The precision and recall of CPScan on the real dataset are about 85% and 95%), there still exists incorrect basic block matching as illustrated in §A.4. We believe if a more accurate code differential analysis is proposed, these false positives can be consequently removed. The discussion of other false positives (13%) is deferred to §A.4.

False negatives. We evaluate the recall (67%) of CPScan on the 8 randomly selected IoT kernels in the real dataset (as shown in Table 4). However, the bugs contained in these IoT kernels are limited. Thus, we leverage the synthetic dataset, which contains diverse bugs caused by *DSO*s to evaluate the false negatives of CPScan. Recall that we insert 700 bugs into the synthetic dataset by deleting various security operations. CPScan reports 424 bugs while missing the remaining 276 bugs (the recall is about 60%). We investigate these false negatives and summarize their causes as below.

```
1 /* drivers/net/ethernet/stmicro/stmmac/stmmac_platform.c */
2 static int stmmac_probe_config_dt( ...){
3     struct device_node *np = pdev->dev.of_node;
4 -   if (!np)
5 -       return -ENODEV;
6 -   *mac = of_get_mac_address(np);
7     plat->interface = of_get_phy_mode(np);
8 +   of_property_read_u32(np, ...);
9 }
```

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

(1) *Unrecognized security checks* (7%). CPScan utilizes cheQ [32] to locate security checks in both Linux kernels and IoT kernels and then perform graph matching to identify *DSO*s. However, cheQ [32] misses some security checks when no error code is returned in the corresponding error handling branch [32]. (2) *Inaccurate graph matching* (39%). As discussed before, currently, there are some cases that CPScan fails to accurately perform graph matching. Therefore, part of the *DSO*s are wrongly matched and will not be further analyzed by CPScan. (3) *Missing security-critical variables* (14%). Part of security-critical variables associated with the *DSO*s are Macros or global variables, which do not have any uses in the subsequent code segments. Consequently, CPScan cannot infer the security impact of these *DSO*s. (4) *The change of the bounded use chains* (40%). For some bugs, the bounded use chains are unnecessary to be the same. For instance, as shown in Figure 8, the deletion of the NULL pointer check (lines 4 -5) results in a NULL pointer dereference (line 7). Thus, such deletion of NULL pointer check is vulnerable, even though the use chain of the security-critical variable (*np*) has changed in the IoT kernel (line 6).

6 DISCUSSION AND LIMITATIONS

6.1 Limitations

Source code coverage. Currently, we use the *allyesconfig* configuration that allows us to obtain most customized files, through which the current dataset covers about 45% of the customized source files and 60% of the customized functions, including various *DSO*s. We cannot obtain all the customized files because (1) we need to specify the architecture during the compiling process. For example, if we compile IR files under MIPS, we will ignore the modified files under ARM. In our experiments, we choose the compilation architecture with the most customized files; (2) config conflicting issues and files that are not compilable under LLVM will further decrease the generated IR files. Considering the proposed system is used to identify and analyze *DSO*s, with this dataset (74,542 customized functions), to some extent, we can perform a high-qualified and fair evaluation of CPScan. Moreover, code coverage can be increased by using different compilation configurations. For instance, we compile the IoT kernel whose ID is 1 in Table 1 under a different architecture. The source file coverage increases from 39.7% to 47.5%.

Graph matching. Graph matching is an NP hard problem. The current implementation of the graph matching in CPScan is based on basic block matching according to the similarity score of basic-block attributes. If the attributes of two basic blocks are similar and the similarity score is higher than a threshold, CPScan will match them. Even though CPScan has achieved high precision of graph

matching, as shown in §5.2, part of code customization significantly changes function CFGs and brings in several inaccurate graph matching scenarios.

Security impact inference. In this step, if CPscan cannot find the paired security-critical variables or the bounded use chains in Linux kernels and IoT kernels, respectively, the security impact of this *DSO* is unclear, which inevitably causes a false negative. Besides, to reduce false positives, CPscan reports a *DSO* to be vulnerable iff the paired security-critical variables' bounded use chains are the same. Therefore, CPscan filters out all the *DSO*s that have different bounded use chains but are vulnerable as listed in Figure 8. Finally, to find a security-critical variable and its bounded use chain in an IoT kernel (as described in §4.3), CPscan performs pointer analysis to find the propagation of this security-critical variable. Inaccurate alias analysis results in an inaccurate bounded use chain of this critical variable. To solve this problem, Andersen-style pointer analysis [24] and Steensgaard alias analysis [45] would be helpful and could be a potential future work.

6.2 Discussion

Extending CPscan. Currently, the implementation of CPscan supports identifying the detection of missing security checks, missing variable initializations, and missing resource-release operations. In theory, the techniques developed in CPscan can support identifying other bug classes such as missing lock/unlock and reference count by specifying suitable patterns. In the future, we would like to extend CPscan to support more types of vulnerabilities. Moreover, it is worth noting that code customization is not specific to IoT kernels. The security bugs caused by code pruning also exist in the widely-used mobile OS kernels and other projects. We can naturally extend CPscan to detect missing security operations in other software. Finally, code additions or the deletions of non-security operations may also potentially change the data or control flow dependence of sensitive operations, hence causing security bugs. We will investigate this interesting problem in the future.

Exploitation. We conduct manual analysis on the bugs reported by CPscan carefully to investigate their security impact and exploitability. To boost the efficiency of confirming the security impact of the bugs reported by CPscan, we can generate inputs to trigger a missing security operation bug by using symbolic execution [40] and a theorem prover [18]. Besides, fuzzing can also dynamically trigger a bug by inputting mutated seeds [30, 35, 55]. In fact, how to automatically exploit a class of security bugs remains a challenging research problem requiring dedicated research. In this paper, the goal of CPscan is not to automatically exploit bugs but automatically identify security bugs caused by code pruning. Automatically generating proof-of-concept exploits by leveraging kernel fuzzing or symbolic execution is an interesting future work.

7 RELATED WORK

IoT device bug detection. Many detection systems are proposed to discover bugs in IoT devices and have greatly improved the security of IoT ecosystems. For one thing, the state-of-the-art approaches [15, 16, 22, 44, 54, 59] perform static analysis on IoT firmware. However, these static approaches aim to discover N-day bugs by code clone detection. By contrast, CPscan can find 0-day bugs caused by code pruning. For another thing, existing approaches [12–14, 17, 21, 27, 31, 37, 49, 57, 60] support dynamic analysis on IoT firmware. However, these methods currently aim to fuzz IoT apps or IoT libraries rather than IoT kernels. Meanwhile, the efficiency of these dynamic methods is low due to their slow throughput. Furthermore, dynamic analysis is inherently limited by the code coverage problem [23].

Inconsistency-based bug detection. Inconsistency-based detection is widely used in the closely related approaches [33, 56, 58] that find bugs in the Linux kernel. It is intuitive to apply these bug detectors to discover bugs in IoT kernels. However, these methods are insufficient for our problem due to the following reasons. (1) Crix [33], PeX [58], and APISan [56] are based on cross-checking. However, cross-checking has two main limitations. First, it is difficult to find similar paths for cross-checking, which needs semantic- or context-aware analysis. Second, cross-checking requires sufficient reference paths to detect buggy paths against the majority of usage patterns. Unfortunately, the evaluation in Table 8 indicates that in most cases, there are no sufficient similar paths to enable cross-checking for a *DSO*. (2) Most bugs discovered by Crix, PeX and APISan are API misuses or missing permission checks. For the deletion of other security operations such as variable initialization, these methods are not suitable. Unlike the above approaches that check the inconsistency of different paths, CPscan determines the inconsistency between different kernels. In the scenario of code pruning where a reference (the compared Linux kernel with the same edition) exists, we believe that we should make full use of the references for detecting bugs in IoT kernels in a more effective way to avoid the limitation with cross-checking.

8 CONCLUSION

Code pruning is prevalent in IoT kernels. In this paper, we present CPscan, an effective detection system to automatically identify various bugs caused by the *DSO*s in IoT kernels. First, to precisely locate the *DSO*s, we design and implement a new structure-aware graph matching algorithm by iteratively performing basic block matching. Then, to automatically infer the security impact of a *DSO*, CPscan employs inconsistency analysis by comparing the bounded use chains of the security-critical variable associated with a *DSO*. Overall, the identification accuracy and efficiency of CPscan are much higher than state-of-the-art tools. To the best of our knowledge, CPscan is the first system that aims to detect bugs caused by code pruning in IoT kernels. Extensive experiments with 28 IoT kernels from 10 vendors show that CPscan effectively discover 114 new bugs with outstanding performance.

9 ACKNOWLEDGMENT

This work was partly supported by NSFC under No. U1936215 and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), and Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies. Kangjie Lu was supported in part by the NSF awards CNS-1815621 and CNS-1931208.

REFERENCES

[1] 2021. *ASUSWRT Open Source Code for Programmers.* https://www.asuswrt-merlin.net/.
[2] 2021. *Cosine-similarity-usage.* https://en.wikipedia.org/wiki/Cosine_similarity.
[3] 2021. *DD-WRT Open Source Code.* https://github.com/mirror/dd-wrt.
[4] 2021. *index : kernel/git/torvalds/linux.git.* https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/.
[5] 2021. *Jaccard-index-usage.* https://en.wikipedia.org/wiki/Jaccard_index.
[6] 2021. *LLVM-diff.* https://llvm.org/docs/CommandGuide/llvm-diff.html.
[7] 2021. *LLVM-usage.* https://llvm.org/.
[8] 2021. *NETGEAR Open Source Code for Programmers (GPL) form.* https://kb.netgear.com/2649/NETGEAR-Open-Source-Code-for-Programmers-GPL.
[9] 2021. *Static single assignment form.* https://en.wikipedia.org/wiki/Static_single_assignment_form.
[10] 2021. *the MD5 message-digest algorithm.* https://en.wikipedia.org/wiki/MD5.
[11] 2021. *Unix-diff-usage.* https://man7.org/linux/man-pages/man1/diff.1.html.
[12] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.. In *NDSS*, Vol. 16. 1–16.
[13] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.
[14] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1–18.
[15] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. 2015. PIE: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 251–260.
[16] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 95–110.
[17] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 437–448.
[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[19] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. *DeepBinDiff: Learning program-wide code representations for binary diffing.* eScholarship, University of California.
[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.
[21] Bo Feng, Alejandro Mera, and Long Lu. 2020. P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.
[22] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.
[23] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.
[24] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 290–299.
[25] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. 2017. Pallas: Semantic-aware checking for finding deep bugs in fast path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 709–722.
[26] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1149–1163.
[27] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 329–340.
[28] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.
[29] Ina Koch. 2001. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science* 250, 1-2 (2001), 1–30.
[30] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. {UNIFUZZ}: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.
[31] Peiyu Liu, Shouling Ji, Xuhong Zhang, Qinming Dai, Kangjie Lu, Lirong Fu, Wenzhi Chen, Peng Cheng, Wenhai Wang, and Raheem Beyah. 2021. iFIZZ: Deep-State and Efficient Fault-ScenarioGeneration to Test IoT Firmware. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[32] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Automatically identifying security checks for detecting kernel semantic bugs. In *European Symposium on Research in Computer Security*. Springer, 3–25.
[33] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. 1769–1786.
[34] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 920–932.
[35] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.
[36] James J McGregor. 1982. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience* 12, 1 (1982), 23–34.
[37] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.. In *NDSS*.
[38] Eugene W Myers. 1986. AnO (ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.
[39] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2017. A Comparison of Code Similarity Analysers. *RN* 17, 04 (2017), 04.
[40] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 49–64.
[41] Shama Rani and Jaiteg Singh. 2017. Enhancing Levenshtein's edit distance algorithm for evaluating document similarity. In *International Conference on Computing, Analytics and Networks*. Springer, 72–80.
[42] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. 2013. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–12.
[43] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168.
[44] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.. In *NDSS*.
[45] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.
[46] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 46–55.
[47] Jeffrey Svajlenko, Iman Keivanloo, and Chanchal K Roy. 2013. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *2013 7th International Workshop on Software Clones (IWSC)*. IEEE, 16–22.
[48] Jacob P Tyo. 2016. Empirical Analysis and Automated Classification of Security Bug Reports. (2016).
[49] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X. Liu, and Reheem Beyah. 2021. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.
[50] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.
[51] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Network and Distributed System Security Symposium (NDSS)*.
[52] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 359–368.
[53] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 661–678.
[54] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity

detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.
[55] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2139–2154.
[56] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. Apisan: Sanitizing {API} usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*. 363–378.
[57] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *NDSS*, Vol. 14. 1–16.
[58] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1205–1220.
[59] Binbin Zhao, Shouling Ji, Wei-Han Lee, Changting Lin, Haiqin Weng, Jingzheng Wu, Pan Zhou, Liming Fang, and Raheem Beyah. 2020. A Large-scale Empirical Study on theVulnerability of Deployed IoT Devices. *IEEE Transactions on Dependable and Secure Computing* (2020).
[60] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1099–1114.
[61] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).

A APPENDIX

A.1 Detecting code movement by Unix-Diff

In Figure 9, Unix-Diff [11] compares two versions of the source code by performing the Myers algorithm [38] at the text line granularity. In fact, there are no code line additions or deletions in this example. However, Unix-Diff [11] reports Part B in Figure 9(a) as deleted code lines. This is because Unix-Diff [11] is limited by its coarse-grained comparison, which is not aligned with the source code structure. Therefore, it can only recognize additions and deletions instead of simple code segment movements. However, code movements are common in the customization process of IoT kernels. Therefore, Unix-Diff [11] is not suitable for identifying code pruning in IoT kernels.

A.2 Efficiency comparison of graph matching

The graph-based approach used in CPscan is much more efficient than the traditional graph matching approaches. Therefore, we further compare the efficiency of CPscan with McGregor [36], the most closely related MCS algorithm. Although McGregor is not suitable in locating code pruning since it cannot match the modified nodes, comparing CPscan with it can help us understand the difference between the efficiency of CPscan and the traditional graph matching methods. We intend to evaluate the efficiency of McGregor and CPscan on all the functions in the real dataset. However, limited by its design, the matching process of McGregor is too long. McGregor can only perform graph matching for the functions whose number of basic blocks is less than 25 within a certain time (300 hours in this experiment). Therefore, we can only compare the matching efficiency of 2,671 functions. For McGregor, the matching time significantly increases when the number of basic blocks becomes larger. In contrast, the matching time of CPscan remains at a very low value for the functions with large sizes. On average, the efficiency of the graph matching algorithm used in CPscan (4.05 seconds per file) is about 400 times faster than McGregor. The reason is that CPscan is structure-aware and fully utilizes the characteristics of a basic block in the matching process. Besides, there are about 30% functions whose number of basic block is larger than 25 (as shown in Table 12 in Appendix A), which further shows that CPscan can achieve high efficiency in the graph matching for locating the *DSO*s.

A.3 The distribution of the DSOs

We evaluate the distribution of the *DSO*s on over half of the IoT kernels in the real dataset (whose IDs are from 1 to 17 in Table 1). Similar to prior research [25, 50, 53], a majority of bugs caused by code pruning exist in the driver or network subsystem of an IoT kernel. The distribution of the *DSO*s is shown in Figure 10. Based on our communication with the developers of IoT kernels, IoT vendors usually need to customize the driver and network code to adapt to the new hardware or functions in IoT devices. In total, about 90% of the detected bugs exist in the driver and net modules. Other subsystems such as file system are also subject to missing security operation bugs. Moreover, we discover that a class of bugs is highly likely to appear in other versions of the same IoT device.

A.4 Other false positives

As shown in Figure 12, the newly added conditional statement in line 4 splits an original basic block (lines 5 and 6 in Linux kernel) into two basic blocks, which causes the CFG change of $ip6\_flush\_pending\_frames$. Therefore, CPscan mistakenly reports $kfree$ (line 6) as a deleted resource-release operation because there is no basic block that can be matched with it.

From our manual analysis, the other false positives (13%) consist of the following aspects. 1) In IoT kernels, a security check is deleted while its error handling branch remains in the function. Thus, the corresponding critical variables will not reach error states and the *DSO* does not cause any security impact. Therefore, such reported *DSO* is a false positive. 2) In the compiling process, even though we try to make sure that the building configurations of

```
1 int spectrum_cs_config(...) {
2     Part A;
3     Part B;
4     ret = pcmcia_request_irq(link, ...);
5     if (ret)
6         goto failed;
7 }
```

(a) Function definition in Linux kernel

```
1 int spectrum_cs_config(...) {
2     Part A;
3     ret = pcmcia_request_irq(link, ...);
4     if (ret)
5         goto failed;
6     Part B;
7 }
```

(b) Function definition in IoT kernel

Figure 9: An example of simple code movement in an IoT kernel, where code segment Part B is moved from line 3 to line 6.

Table 9: The manually constructed synthetic dataset. DSC = deleted security check, DVI = deleted variable initizlization, DRR = deleted resource release, and NCL = normal code line.

ID	Kernel	# of DSC	# of DVI	# of DRR	# of NCL
1*	linux-2.6.30	57	53	54	54
2*	linux-2.6.36	70	55	58	188
3*	linux-4.9.37	73	55	53	96
4*	linux-3.18	61	51	60	108
Total		261	214	225	446

Table 10: The identification precision (Pre.) and recall (Re.) of baselines on the deleted code lines in the synthetic dataset generated from the Linux kernels.

ID	GrumTree			LLVM-Diff			LLVM-Diff-N		
	TP	Pre.	Re.	TP	Pre.	Re.	TP	Pre.	Re.
1*	193	68%	**89%**	48	53%	22%	48	**55%**	22%
2*	341	36%	**92%**	82	50%	22%	89	**90%**	24%
3*	233	59%	**84%**	60	53%	22%	59	**88%**	21%
4*	252	54%	**91%**	100	28%	36%	92	**94%**	33%
Average	255	54.2%	**89.0%**	72	46.0%	25.5%	72	**81.7%**	25.0%

Table 11: The identification precision and recall of CPScan on the DSOs in the synthetic dataset. DSC = deleted security check, DVI = deleted variable initialization, and DRR = deleted resource release.

ID	# of DSC			# of DVI			# of DRR		
	TP	Pre.	Re.	TP	Pre.	Re.	TP	Pre.	Re.
1*	55	93%	96%	52	**100%**	98%	54	**100%**	**100%**
2*	68	97%	97%	54	98%	98%	58	**100%**	**100%**
3*	72	97%	98%	55	98%	**100%**	51	96%	96%
4*	58	97%	95%	51	96%	**100%**	60	**100%**	**100%**
Average	63	96.0%	96.5%	53	98.0%	99.0%	56	**99.0%**	**99.0%**

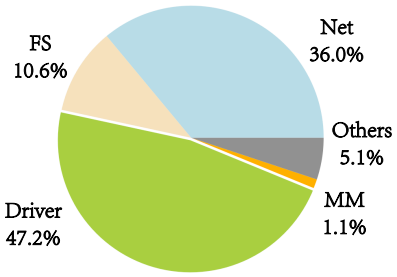

Figure 10: The distribution of the *DSO*s.

an IoT kernel and Linux kernel are the same, there are some extra building configurations only in IoT kernels. The differences between building configurations cause the deletion of unnecessary security operations, which brings in new false positives.

```
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).

Table 12: The distribution of functions in the Linux kernel with respect to the number of basic blocks in a function.

# of Basic Blocks in a Function	[1,10)	[10,20)	[20,30)	[30, +∞)
Ratio of Functions	40.0%	22.8%	17.4%	19.8%

Table 13: List of some new bugs detected with CPscan in IoT kernels. DSC = deleted security check, DVI = deleted variable initialization, and DRR = deleted resource release. The S and C in the Status field represent patch status - Submitted and Confirmed, respectively.

Subsystem	File	Function	Type	Security Operation	Critical Variable	Consequence	Status
Driver	core.c	regulator_unregister	DRR	kfree	rdev->constraints	Memory leakage	S
Driver	hub.c	hub_port_init	DSC	device check	hub->tt.hub	Denial of service	S
Driver	sierra.c	sierra_outdat_callback	DRR	kfree	urb->transfer_buffer	Memory leakage	S
Driver	serial_core.c	uart_poll_init	DVI	device initialization	tport	Denial of service	S
Driver	cdc-acm.c	acm_probe	DSC	running state check	control_interface	system crash	S
Driver	dvb_demux.c	dvbdmx_release_section_feed	DSC	running state check	&demux->mutex	dead lock	S
Driver	hci_ldisc.c	hci_uart_tty_open	DSC	NULL pointer check	tty_ops_write	NULL pointer dereference	S
Driver	n_gsm.c	gsm_control_reply	DSC	NULL pointer check	msg	NULL pointer dereference	S
Driver	synclink_cs.c	hdlcdev_ioctl	DVI	memset	&newline	Memory leakage	S
Driver	iowarrior.c	iowarrior_ioctl	DVI	memset	&info	Memory leakage	S
Driver	m25p80.c	m25p_probe	DRR	kfree	flash	Memory leakage	S
Driver	serial_core.c	uart_poll_init	DRR	kfree	tport	Memory leakage	S
Driver	xhci.c	xhci_resume	DVI	device initialization	xhci	Denial of service	S
Driver	f_uac1.c	f_audio_free_inst	DRR	kfree	opts->fn_play	Memory leakage	C
Driver	f_uac1.c	f_audio_free_inst	DRR	kfree	opts->fn_cap	Memory leakage	C
Driver	f_uac1.c	f_audio_free_inst	DRR	kfree	opts->fn_cntl	Memory leakage	C
Driver	ion.c	ion_client_create	DVI	device initialization	&client->idr	Denial of service	C
Driver	nand_base.c	check_offs_len	DSC	bound check	mtd->size	Out-of-bound access	S
Driver	nandsim.c	do_read_error	DVI	variable initialization	ns->buf.byte	Denial of service	S
Driver	onenand_base.c	onenand_panic_write	DSC	bound check	mtd->size	Out-of-bound access	S
Driver	m25p80.c	m25p80_erase	DSC	bound check	instr->addr	Out-of-bound access	S
Driver	pegasus.c	alloc_urbs	DRR	resource free	pegasus->ctrl_urb	Denial of service	S
Driver	configfs.c	usb_string_copy	DRR	kfree	copy	Memory leakage	S
Driver	of-thermal.c	thermal_zone_of_sensor_register	DSC	running state check	child	Denial of service	S
Driver	mt29f_spinand.c	spinand_probe	DSC	NULL pointer check	mtd	NULL pointer dereference	S
Driver	dm-thin.c	check_for_space	DSC	running state check	pool	Denial of service	S
Driver	ch.c	ch_probe	DVI	variable initialization	&ch->ref	Memory leakage	S
Driver	of-thermal.c	of_thermal_set_emul_temp	DSC	running state check	data->ops	Denial of service	S
Net	br_multicast.c	br_multicast_set_hash_max	DSC	running state check	br_dev	Denial of service	S
Net	br_multicast.c	br_ip4_multicast_igmp3_report	DSC	running state check	skb	Denial of service	S
Net	wext-core.c	ioctl_standard_iw_point	DSC	bound check	len	Out-of-bound access	S
Net	xgmac.c	xgmac_change_mtu	DSC	bound check	new_mtu	Out-of-bound access	S
Net	via-velocity.c	velocity_init_registers	DVI	device initialization	vptr	Denial of service	S
Net	nf_conntrack_l3proto_ipv4.c	ipv4_conntrack_local	DSC	bound check	skb->len	Out-of-bound access	S
Net	af_econet.c	econet_sendmsg	DSC	bound check	len	Out-of-bound access	S
Net	main.c	b43_wireless_core_init	DVI	device initialization	dev->wl	Denial of service	S
Net	netdev.c	bnep_net_setup	DVI	memset	dev->broadcast	memory lekage	C
Net	tty.c	rfcomm_wmalloc	DSC	bound check	&dev->wmem_alloc	Out-of-bound access	C
Net	raw.c	do_rawv6_setsockopt	DSC	running state check	sk	Denial of service	S
Net	tcp.c	tcp_init_sock	DVI	variable initialization	&tp->tsq_node	Denial of service	S
Net	l2cap_sock.c	l2cap_sock_accept	DSC	running state check	&sk->sk_state	Double free	S
Net	l2cap_core.c	l2cap_ertm_init	DVI	variable initialization	&chan->busy_q	Denial of service	C
Net	l2cap_core.c	l2cap_ertm_init	DVI	variable initialization	l2cap_busy_work	Denial of service	C
Net	sme.c	__cfg80211_connect	DSC	running state check	wdev->sem_state	Denial of service	S
Net	soc.c	sco_sock_connect	DSC	bound check	alen	Out-of-bound access	C
Net	nl80211.c	nl80211_key_allowed	DSC	NULL pointer check	wdev->current_bss	Denial of service	S
FS	smb2pdu.c	SMB2_sess_setup	DRR	kfree	ses->auth_key.response	memory lekage	S
FS	messenger.c	ceph_msg_data_destroy	DSC	NULL pointer check	data	NULL pointer dereference	S
ARCH	c-r4k.c	local_r4k_flush_cache_page	DSC	NULL pointer check	vaddr	NULL pointer dereference	C
Security	keyctl.c	SYSC_add_key	DSC	NULL pointer check	_payload	NULL pointer dereference	S

```
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.