Detecting Callback Related Deep Vulnerabilities in Linux Device Drivers

Tuba Yavuz
ECE Department
University of Florida
Gainesville, FL, USA
tuba@ece.ufl.edu

***Abstract*—Extensibility is an important design goal for software frameworks that are expected to evolve in a variety of dimensions. Callback mechanism is utilized extensively in large frameworks to achieve extensibility. However, callback mechanism introduces implicit control-flow dependencies that make program comprehension and analysis difficult. This paper presents an automated approach for detecting deep bugs/vulnerabilities that involve callbacks. Our approach consists of several stages to balance scalability and precision. Specifically, it uses a light-weight static analysis for extracting callback related interactions between the application modules and the framework modules. This information is used to extend the basic call graph of the application modules to incorporate implicit call chains due to callbacks. The second stage, summary mode, summarizes bug relevant data-flow facts for paths that start at callbacks. The third stage, summary-aware mode, uses the extended call graph to incorporate data-flow facts due to implicit paths that lead to the callbacks and detects deep bugs. We have implemented the presented model extraction and bug detection approach in a framework called MOXCAFE and applied it to Linux device drivers. Using our approach, we could detect several deep vulnerabilities.**

***Index Terms*—Deep bugs, API misuse, callbacks, static analysis, inter-procedural, path-sensitive, model extraction, Linux kernel, vulnerability.**

I. Introduction

Large frameworks use the callback mechanism to support extensibility. A typical application module[1] defines a set of callbacks and communicates with the framework through a well-defined Application Programming Interface (API), which provides functionalities such as registering the callbacks and requesting various services. An important aspect of the callback mechanism is introduction of implicit control-flow and data-flow dependences. As any implicit programming mechanism, callbacks make program comprehension and automated analysis difficult, which can potentially give rise to *deep bugs*. We can broadly define deep bugs as those that may be triggered by a specific input or system state and cannot be easily spotted via code reviews due to various types of indirection, e.g., long call chains, or implicit dependencies, e.g., callback execution.

Recent studies on API usability [1]–[7] report various difficulties faced by the developers when using APIs and how API misuses may lead to vulnerabilities. The focus in these studies has been mostly on the misuse of the cryptographic APIs.

In this paper, we analyze a new class of API misuses that involves callbacks. Our experiments on the Linux drivers indicate that kernel developers may not be fully aware of the side effects of the kernel API functions in terms of the callback functions that may get executed. In some cases, the call chain that starts from an API function and ends at a driver callback function may be quite long as in the case of the double-free vulnerability that was found in the USB midi driver [8], which we explain in detail in Section II as a motivating example. The misuse of such API functions may lead to deep vulnerabilities, which involve execution of application specific callbacks by the API functions and result in an undesirable interaction with what gets executed before or after the API function, e.g., double-free.

The Linux operating system provides a callback based framework and, therefore, may be susceptible to deep bugs and vulnerabilities. However, there has been relatively little interest [9], [10] w.r.t. callbacks in the context of the Linux kernel. A major concern with regards to Linux is the existence of vulnerabilities that can be exploited to compromise the security of the host and the peripheral devices. Considering the role of Linux in the mobile domain and in the emerging world of Internet of Things, it is imperative to detect deep bugs and vulnerabilities in its components.

Static analysis techniques developed for detecting bugs in the Linux kernel do not handle function pointers properly [9], [11]–[23], which prevents them from finding deep bugs that involve callbacks. Although, in theory, one can use a general points-to analysis to resolve targets of function pointers, it is challenging to strike a balance between precision and scalability. Precise points-to analyses that have been shown to scale [15], [24], [25] often make various engineering decisions to achieve scalability and provide sufficient precision only for the specific types of client analyses that they get evaluated for such as code optimization. We are not aware of any scalable and precise points-to analyses that have been evaluated for the

This work was partially funded by the National Science Foundation under grant CNS-1815883 and by the Semiconductor Research Corporation.

[1]We use the term application module to denote any component that implements a customized functionality in the context of a framework. Depending on the framework, an application module can be a user-level or a kernel-level component.

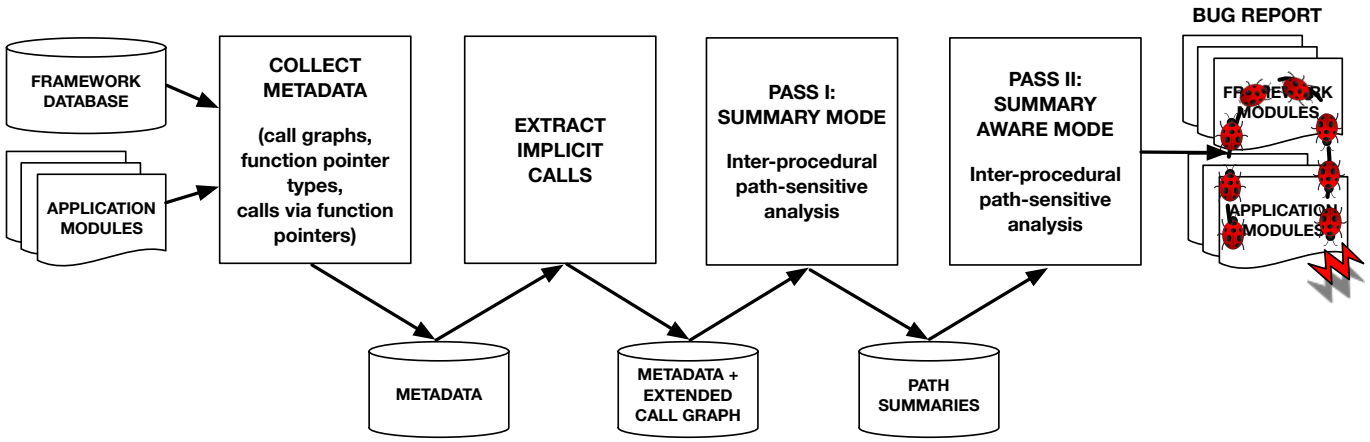

Fig. 1: The architecture of MOXCAFE: **Mo**del E**x**traction for **Ca**llback based **F**rame**w**orks. Arrows denote data-flow.

precision of call graphs for the C language. In this paper, we show that, for detecting deep bugs, a call graph construction algorithm that focuses only on the function pointers performs better than the call graph construction approach by Lattner et al. [15], which uses a context-sensitive points-to analysis (see Section IV-C).

In this paper, we present a staged static analysis approach to detecting deep bugs that involve callbacks. Our approach, as shown in Figure 1, consists of several stages to balance scalability and precision. Specifically, it uses a light-weight static analysis for extracting callback related interactions between the application modules and the framework modules. This information is used to extend the basic call graph of the application modules to incorporate implicit call chains due to callbacks. We show that a general points-to analysis is not necessary for extracting an extended call graph to guide the detection of deep bugs.

We use the extended call graph to guide the second and third stages of our approach, which use inter-procedural path-sensitive analyses to detect specific types of deep bugs. The second stage, summary mode, summarizes bug relevant data-flow facts for implicit paths that start at the callbacks reachable from the API functions. The third stage, summary-aware mode, uses the extended call graph to incorporate data-flow facts due to the implicit paths and detects deep bugs. Our contributions can be summarized as follows:

- A scalable algorithm for generating extended call graphs of callback-based framework applications,
- A generic two-pass static program analysis algorithm that uses an extended call graph to detect deep bugs and its instantiation for double-free and double locking, which may be exploited to execute arbitrary code and to cause denial of service, respectively,
- Discussion of the detected real deep vulnerabilities (4 new, including CVE-2017-17975, CVE-2018-20961, and CVE-2019-14763, out of a total of 6) with recommendations for the developers and API designers of extensible frameworks.

The rest of the paper is organized as follows. Section II presents the details of a callback related deep vulnerability as a motivating example. Section III presents the technical details of extracting an extended call graph and using that extended call graph in callback-aware analysis for detection of deep bugs. Section IV evaluates our approach in terms of its effectiveness in extracting extended call graphs as well as in detecting deep bugs in Linux device drivers. Section V provides recommendations for secure development for frameworks that use the callback mechanism. Section VI discusses related work. Finally, Section VII provides a conclusion and directions for future work.

II. Motivating Example

In this section, we present a double-free vulnerability CVE-2016-2384 [8] found in the Linux USB midi driver. Figure 2 shows a snippet of code from the driver. The bug gets manifested inside the function `snd_card_free` (callsite at line 23) that gets executed as part of handling the hot-plug event of the audio device by the function `usb_audio_probe`. The `umidi` object is freed inside the function `snd_usbmidi_free` (line 62) and both the first and second free are performed by this function. The bug is difficult to identify via code reviews as the function `snd_usbmidi_free` seems to be executed from the `usb_audio_probe` (via `snd_usb_create_streams`) just once (via the call chain line 17 $\rightarrow$ line 29 $\rightarrow$ line 33). However, as seen in Figure 3, `snd_usbmidi_free` also gets called from the function `snd_card_free` through a mixed series of explicit and implicit calls as illustrated by Figure 3.

It is interesting to note that the function `snd_usbmidi_free`, which performs memory deallocation, is not even a callback function. However, it gets called by the callback function `snd_usbmidi_rawmidi_free` (line 53). In Figure 3, all functions except `snd_usbmidi_free` and `snd_usbmidi_rawmidi_free` are framework functions belonging to various kernel layers.

```
1  // in sound/usb/midi.c
2  // assigns the callback function
3  int snd_usbmidi_create_rawmidi(...)
4  {
5      ...
6    rmidi->private_free = snd_usbmidi_rawmidi_free;
7    ...
8  }
9
10 // in sound/usb/card.c
11 // 1st free by snd_usb_create_streams
12 // 2nd free by snd_card_free
13 int usb_audio_probe(struct usb_interface ...)
14 {
15   ...
16   if (err > 0) {
17      err = snd_usb_create_streams(chip,...);
18       if (err < 0) goto __error;
19         ...
20  __error:
21         if (chip) {
22                 if (!chip->num_interfaces)
23                     snd_card_free(chip->card);
24         ...
25 }
26
27 // in sound/usb/midi.c
28 // called by snd_usb_create_streams
29 int snd_usbmidi_create(...)
30 {
31       ...
32       if (err < 0) {
33           snd_usbmidi_free(umidi);
34           return err;
35       }
36       ...
37 }
38
39 // called by snd_card_free in a chain of function calls
40 // executes the callback
41 static int snd_do_card_free(struct snd_rawmidi *rmidi)
42 {
43     ...
44     rmidi->private_free(rmidi);
45     ...
46 }
47
48 // in sound/usb/midi.c
49 // the callback function
50 int snd_usbmidi_rawmidi_free(struct snd_rawmidi *rmidi)
51 {
52    struct snd_usb_midi *umidi = rmidi->private_data;
53    snd_usbmidi_free(umidi);
54    ...
55 }
56
57 // in sound/usb/midi.c
58 // helper function performing the deallocation
59 void snd_usbmidi_free(struct snd_usb_midi *umidi)
60 {
61         ...
62         kfree(umidi);
63 }
```

Fig. 2: Code snippets from the Linux USB midi driver (Linux kernel v4-4) that get executed as part of handling the hot-plug event. Memory deallocation at line 62 is executed twice by `usb_audio_probe`: first via `snd_usb_create_streams` (line 17) and then via `snd_card_free` (line 23).

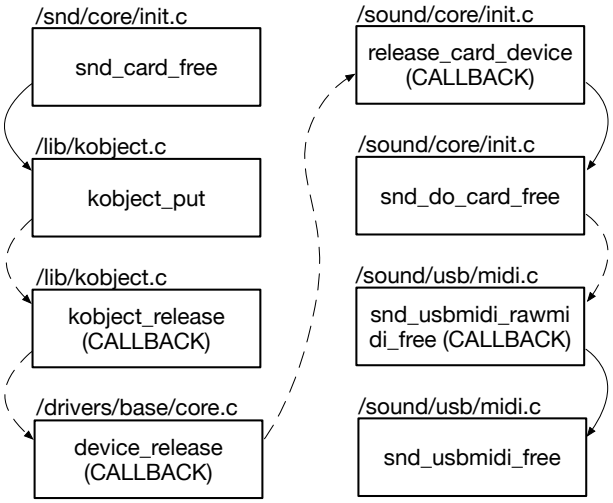

Fig. 3: A partial call graph of the Linux USB midi driver that relates to the double-free bug. The dashed arrows denote calling callback functions via function pointers while the solid arrows denote explicit calls.

The main challenge for static analysis tools in dealing with scenarios like this is the incomplete call graph due to callbacks. For instance, the dashed arrows in Figure 3 would be absent in the basic call graph, which does not include call dependencies due to function pointers. Therefore, any static analysis tool that considers the basic call graph only would miss the second deallocation that is triggered at line 23 in Figure 2, and, hence, would not be able to detect the double-free vulnerability. This vulnerability was initially detected by a combination of two tools: a dynamic memory error detection tool, KASAN [26], and a USB fuzzer, vUSBf [27]. Although fuzzing can be used to generate inputs that may trigger such bugs, it requires access to the device or to an emulation environment. However, static analysis requires neither of these and can be easily incorporated to the development environment for automated regression analysis.

The goal of this paper is to present a model extraction technique that discovers implicit edges in the call graph of an application component that is embedded in a framework so that potential client static analysis on the application component can leverage these inferred edges to find deep bugs that involve callbacks as in the case of the USB midi double-free vulnerability.

III. Approach

In this section, we present the technical details of our approach using the USB Midi double-free vulnerability from Section II as an example. Our approach uses component-

level analysis. It first summarizes the interaction between the application modules and the framework modules in what we call an *extended call graph*. Figure 4 shows the extended call graph for the code snippet given in Figure 2. An extended call graph extends a basic call graph by adding special edges, denoted by the dashed arrows, from the API functions to the callbacks of the application modules. As an example, the `snd_card_free` API function may call the `snd_usbmidi_rawmidi_free` callback function of the USB midi driver as shown in Figure 3. In an extended call graph, we summarize this type of side effect by creating an edge from the API function to the callback of the application module. The idea is to embed sufficient information in the extended call graph to enable a precise and a scalable deep analysis; use path-sensitive analysis on the application modules only while leveraging the implicit dependencies recorded in the extended call graph. Specifically, we compute the side effects of the callback functions and store them as *path summaries*. As we analyze the application modules to detect bugs, we utilize the dashed edges in the extended callgraph and the path summaries to simulate the side effects of callback functions at API callsites. We present our extended call graph generation algorithm in Section III-A and the details of our path-sensitive static analysis in Section III-B.

A. Extended Call Graph Generation

In this section, we will explain the *Collect Metadata* and the *Extract Implicit Calls* stages shown in Figure 1 using the example code given in Figure 5 and the case study presented in Section II. In the *Collect Metadata* stage both the application and the framework modules are parsed. Metadata per module M includes the basic call graph, $M.CG$, the functions, $M.F$, and the callback types. We identify the type of callbacks in two ways: 1) By the type of the data structure that points to callbacks, *callback type* ($Type$), 2) By the signature of the callback functions, *callback signature* (Sig).

It is very common in frameworks to define callback types as part of the interface. In the Linux kernel this is achieved by defining function pointer fields of `structs`. So, the first way is represented by a tuple $(type, field)$, where $type$ denotes the type of the data structure and $field$ denotes the identifier for the field of the data structure that represents a function pointer. An example for this type of metadata is `(struct snd_rawmidi, private_free)`, where the field `private_free` of `struct snd_rawmidi` represents a callback, which is used in Figure 2.

Another type of metadata for this type of callbacks is the bindings of actual functions to the callback types. We collect this information by processing the assignment statements in the data structure initializations, e.g., line 2, and inside the function bodies, e.g., line 16, as shown in Figure 5a. The callback types and their targets are listed in the top table (the first three rows) in Figure 5b. The callback type `(T,f1)` is bound to {`cb1`} as `cb1` is the only function that is assigned to this callback type whereas `(T,f2)` is bound to the set {`cb2,cb5`} due to the assignments on lines 3 and 16.

The second way of using signatures is needed as callbacks may be represented by their signatures, i.e., a combination of return type and the parameter types, and may not be embedded inside a `struct` type. As an example, the `kobject_release` function in Figure 3 is a callback that is passed to the `kref_put` function as a parameter. To track such callbacks, we parse callback sites that pass function pointers and record the function names that are passed as actual parameters. The top table (the last three rows) in Figure 5b also shows the callback signatures and their targets, which are collected from the call sites at lines 49, 50, and 54. Another type of metadata that is recorded is the information about call sites at which function pointers are passed and denoted by SP. As an example, `(a3,cb4)` is included in SP to reflect the call site on line 49.

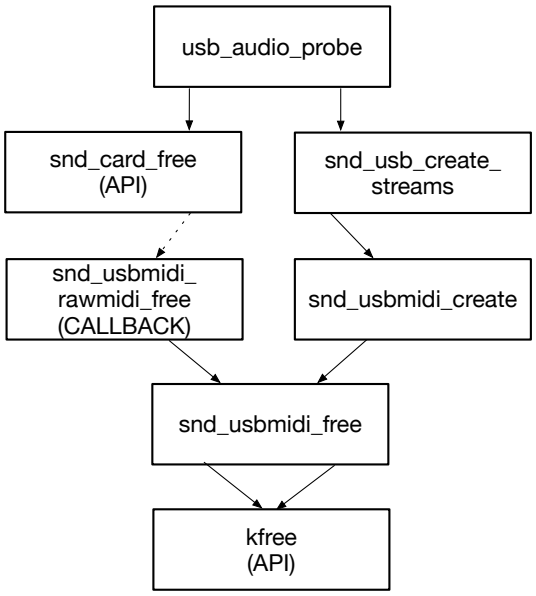

Fig. 4: Part of the extended call graph for the example given in Figure 2. The dashed edge does not exist in the basic call graph.

Algorithm 1 implements the *Extract Implicit Calls* stage in Figure 1 and gets as input an application module, A, a set of framework modules, FM, and a bound N. It traverses the global call graph and propagates the function pointer calls to identify the call chains that include callbacks. The global call graph, CG, is a combination of the application call graph, $A.CG$, and the call graphs of the framework modules, $\bigcup_{fm \in FM} fm.CG$, and is constructed at line 3. The other steps of the initialization captures the direct calls to function pointers and initializes the call chains for the callback types, TC, (line 4) and for the callback signatures, SC, (line 5). A tuple (a, b) in TC (SC) denotes the fact that a may call a callback of (signature) type b in a sequence of calls of length one, i.e., calls directly, or more. For the example in Figure 5, the tuple `(a0,(T,f1))` gets included in SC and the tuple `(a1,void (*)())` gets added to TC at this stage.

The loop (lines 6-14) propagates the discovered call chains

```
1  struct T t1 = {
2    .f1 = cb1,
3    .f2 = cb2,
4    .f3 = 0,
5  };
6  void a0(struct T *t0)
7  {
8     t0->f1(...);
9  }
10 void a1(void (*p)())
11 {
12    p();
13 }
14 void a2(struct T *t2)
15 {
16    t2->f2 = cb5;
17 }
18 void a3(void (*p)())
19 {
20    a1(p);
21 }
22 void a4(struct T *t7,
23      void (*p)(char))
24 {
25      t7->f3(p);
26 }
27 void a5(void (*p)(int),
28      struct T *t2) {
29   t2->f2 = p;
30 }
31 void a6(struct T *t5)
32 {
33   t5->f2();
34 }
35 void a7(void (*p)(int),
36      struct T *t7) {
37   a5(p,t7);
38 }
39 void cb3(void (*p)..)
40 {
41    p(...);
42 }
43 void cb4() {
44      t1.f1(...);
45 }
46 int main(...) {
47      t1.f3 = cb3;
48      a0(&t1);
49      a3(cb4);
50      a4(&t1,cb6);
51      a6(&t1);
52      a2(&t1);
53      a6(&t1);
54      a7(cb7,&t1);
55      a6(&t1);
56 }
```

(a) An example code that uses function pointers.

CB Type/Sig	Targets
(T,f1)	{cb1}
(T,f2)	{cb2,cb5}
(T,f3)	{cb3}
void (*)()	{cb4}
void (*)(char)	{cb6}
void (*)(int)	{cb7}

API (source)	Callbacks (destination)	
	$N=1$	$N=2$
a0	{cb1}	{cb1}
a2	$\emptyset$	$\emptyset$
a3	{cb4}	{cb4,cb1}
a4	{cb3}	{cb3}
a5	$\emptyset$	$\emptyset$
a6	{cb2,cb5}	{cb2,cb5}
a7	$\emptyset$	$\emptyset$

(b) Metadata for the callbacks of the sample code (the top table) and the inferred call graph edges (the bottom table). N denotes the number of iterations performed by Algorithm 1.

Fig. 5: An illustration of inferring implicit edges of the extended call graph.

Algorithm 1 An algorithm for extending the call graph of an application module $\mathcal{A}$ with edges that represent implicit call sequences to the callbacks of A.

1: **ExtendCallGraph**(A: *MODULE*, FM: $\mathcal{P}$(*MODULE*),N:$\mathcal{Z}$)
2: **Output**: An extended call graph that includes edges from API functions to the callbacks of A
3: $CG \leftarrow A.CG \cup \bigcup_{fm \in FM} fm.CG$
4: Let TC denote the set of all tuples (a, b) s.t. a directly calls a callback of type b
5: Let SC denote the set of tuples (a, b) s.t. a directly calls a callback of signature type b
6: **for** i: 1 to N **do**
7: Let Π denote all possible simple paths in CG
8: $TC' \leftarrow \{(c, b) \mid (a, b) \in TC \ \wedge \ (c, Type(a)) \in TC\}$
9: $TC'' \leftarrow \{(c, b) \mid (a, b) \in TC' \ \wedge \ \exists (c, a) \in \Pi\}$
10: $SC' \leftarrow \{(c, b) \mid (a, b) \in SC \wedge (c, Sig(a)) \in SC \wedge (c, a) \in SP\}$
11: $SC'' \leftarrow \{(c, b) \mid (a, b) \in SC' \ \wedge \ \exists (c, a) \in \Pi\}$
12: $CG' \leftarrow CG \ \cup \ \{(a, b) \mid (a, Sig(b)) \in SC \ \wedge \ (a, b) \in SP\}$
13: $(TC, SC, CG) \leftarrow (TC', SC'', CG')$
14: **end for**
15: **return** $A.CG \ \cup \ \{(a, b) \mid \exists c.(c, a) \in A.CG \ \wedge \ b \in A.F \ \wedge \ ((a, Type(b)) \in TC \vee ((a, Sig(b)) \in SC \ \wedge \ (a, b) \in SP)\}$

across every function that is used as a callback (lines 8 and 10) as well as across the call graphs (lines 9 and 11). These two propagation steps are performed for the callback types (lines 8 and 9) as well as for the callback signatures (lines 10 and 11) until the input bound, N, is reached.

Line 8 propagates the callback chains that start at a function that is a callback to its implicit callers. For the example in Figure 3, at some point in the computation, TC will include `(release_card_device,(snd_rawmidi, private_free)` and `(kobject_release, (device,release))`. Since the function `release_card_device` is bound to the callback type `(device, release)`, it will infer the callback call chain `(kobject_release, (snd_rawmidi, private_free))`, which will be added to TC. Line 10 performs a similar callback chain propagation for the callbacks that are identified by the signature types.

Lines 9 and 11 propagate the callback chains in TC and SC, respectively, across the call graph. For the example in Figure 3, line 9 will use the tuple `(snd_do_card_free, (snd_rawmidi, private_free))` in TC and the fact that `release_card_device` calls `snd_do_card_free` to infer the tuple `(release_card_device,(snd_rawmidi, private_free))`, which will be added to TC. For the code in Figure 5a, line 11 will use the tuple `(a1,void(*)())` in SC and the fact that `a3` calls `a1` to infer the tuple `(a3,void(*)())`, which will be added to SC.

An important step in each iteration of the loop (line 12) is updating the global call graph, CG, based on the call sites that pass function pointers that are identified by their signatures, which are recorded in the metadata SP. So $(a, b) \in SP$ means b is passed to a at some call site as an actual parameter of a function pointer type. As an example, `kobject_release` is passed as a function pointer to the `kref_put` function inside the `kobject_put` function, i.e., `(kref_put, kobject_release)` in SP. Since `kref_put` has also been found to call a callback of signature type equivalent to the signature of `kobject_release`, i.e., `(kref_put, void(*)(struct kref*))` in SC, an edge from `kref_put` to `kobject_release` is added in the global call graph, CG. So the updated global call graph helps construct implicit call chains that may involve

callbacks that are defined as fields of `struct` types, e.g., `snd_usbmidi_rawmidi_free`, as well as those that are identified by their signatures, `kobject_release`, as in the case of the USB midi example.

The output of Algorithm 1 is the extended call graph for the application module, in which implicit calls to callbacks via API functions are made explicit. On line 15, the call chains from the API functions to the callback types and to the callback signatures are used to infer the implicit edges of the extended call graph. It is important to note that the destinations of these implicit edges are the callbacks of the application module. So, for the USB midi driver, this means that an edge from the node for the `snd_card_free` function to the node for the `snd_usbmidi_rawmidi_free` function is added to the basic call graph of the USB midi driver. Note that we bypass `snd_do_card_free` as there is no direct call from the driver to this function whereas `snd_card_free` is called as a framework API function (line 23, Figure 2).

For the code in Figure 5a, Algorithm 1 infers the edges shown by the bottom table in Figure 5b. It infers `(a3,cb4)` in the first iteration whereas it infers `(a3,cb1)` in the second iteration. Our approach avoids some of the potential false positives by 1) restricting the scope of the analysis to the application module, A, and the framework modules, FM, that A depends on and 2) binding a callback c to a function pointer signature as a possible target only if c has been passed as an actual parameter to a function that accepts such a signature by checking the SP metadata.

B. Callback-aware Analysis

After collecting the metadata and extracting an extended call graph for the application modules, we perform inter-procedural path-sensitive program analysis on the application modules only. This is performed in two passes: 1) *Summary Mode*, 2) *Summary-Aware Mode*. The pass in Summary Mode performs program analysis on the basic call graph to summarize paths that start with callback functions. The pass in Summary-Aware Mode considers the extended call graph along with the exact program locations that happen to be the callback callsites and uses path summaries to detect bugs and to propagate summaries on the paths that involve callbacks. So for the USB midi example, the first pass would record the fact that a path starting with the `snd_usbmidi_rawmidi_free` function would deallocate a memory object of type `snd_usb_midi` without a prior allocation. The second pass would realize that a path on which a memory object of type `snd_usb_midi` has already been freed (due to `snd_usb_create_streams` on line 17) frees a memory object of type `snd_usb_midi` a second time due to a chain of calls that is initiated by `snd_card_free` at line 23 and end up executing the `snd_usbmidi_rawmidi_free` callback.

In our MOXCAFE tool, we have used the Clang static analyzer [28][2], which implements the inter-procedural path-sensitive analysis algorithm presented in [29] using a region-based memory model [30]. So, the elements of aggregate types such as the elements of arrays and the fields of structures can be tracked precisely on each path explored by the analysis. The challenge in our approach is that in the summarized call chains we abstract away the data-flow from the API functions to the callbacks. Although the underlying inter-procedural path-sensitive analysis is context-sensitive, we are not able to preserve context-sensitivity at API call sites at the precision level of the region based memory model. This is because we do not analyze the API functions as the framework modules are excluded from these stages of the analysis. So, we cannot relate the memory objects that are used by a callback function to the memory objects that are passed to the API function. We handle this problem by using the type information, $Type(o)$, for bug-relevant objects, o, in the path summaries. So, in the summary-aware mode when a bug-relevant operation, e.g., `kfree`, is encountered at a callsite, the type of the object, rather than the region-based memory representation, is used to update the bug relevant metadata.

To demonstrate the utilization of the extended call graphs in deep bug detection, we focus on two types of bugs: Double-free and Double-locking. Our approach extends an inter-procedural path-sensitive analysis by adding bug relevant metadata, $\mathcal{E}$, to the generic state representation, $\mathcal{IPS}$, which stores a representation of the memory locations that belong to the global scope, the stack, or the heap, and the concrete or symbolic values that these memory locations are associated with. Table I presents the description of the the metadata that is kept track of during the inter-procedural path-sensitive analysis.

The main idea in our extension is to record two types of data-flow facts. The first type is related to a specific operation such as $\mathcal{AL}$ for memory allocation and $\mathcal{ACQ}$ for lock acquires and represents the state on a given code location on the current path. The second type relates an operation with its reverse operation such as $\mathcal{ALR}$ that records allocations without a prior freeing and $\mathcal{ACQR}$ that records acquires without a prior release. The first type is fundamental to keeping track of data-flows w.r.t. specific bug-related operations on a path, whether in the summary-mode or summary-aware mode. However, the second type is important for detecting deep bugs that get manifested at API call sites as metadata in the path summaries such as $\mathcal{FRA}$ and $\mathcal{ACQR}$ precisely reveal conflicting operations, e.g., consecutive freeing of the same memory object, from the part of the path explored in the summary-aware mode and the part explored in the summary-mode.

Figure 6 shows the rules of the analysis that are common to both modes. Here, the novelty of our approach is keeping

[2]Our deep bug detection depends on the Clang static analyzer's handling of loops and recursion. For loops, it sets a bound on the maximum number of iterations (set to 10 in our experiments). There is no special treatment of recursive calls. However, it enforces the maximum number of nodes (set to default in our experiments) in the exploded super graph, which may bound recursive calls that add new nodes to this graph.

TABLE I: Description of metadata that is kept track of during the Summary and Summary-Aware Modes.

Metadata	Description
$\mathcal{AL}$	Type of objects that have been allocated and not freed
$\mathcal{FR}$	Type of objects that have been freed
$\mathcal{FRA}$	Type of objects that have been freed without any prior allocation
$\mathcal{ACQ}$	Type of objects acquired and not released
$\mathcal{ACQR}$	Type of objects acquired without a prior release
$\mathcal{RL}$	Type of objects released

track of $\mathcal{ALR}$ and $\mathcal{FRA}$ to be able to detect conflicts with data-flow in the implicit paths. To avoid the clutter, we did not show the components of $\mathcal{E}$ that remain the same, e.g., $\mathcal{E}'.\mathcal{ACQ} = \mathcal{E}.\mathcal{ACQ}$ for `alloc`. Figure 7 shows an additional rule of the analysis for the Summary Mode, in which we analyze paths that start from a callback function recorded in $Callback$. If any action that is relevant to a specific bug has been performed on such a path, we record this bug relevant metadata along with the callback identifier in what we call a *path summary*. All path summaries are stored in $\mathcal{PS}$. In Figure 7, $return\ exp$ denotes the `return` statement of the callback function and $Context(return\ exp)$ denotes the name of the callback function.

Figure 8 shows the rules of the analysis exclusive to the Summary-Aware Mode. In this mode, while analyzing a path in the program, we treat call sites that may start a chain of calls to a callback in a special way as specified by the **callback-aware** rule in Figure 8. Specifically, for call statements, $fexp(args)$, there are two possibilities that need special treatment: a) The callee is an API function and the corresponding node has an outgoing edge to the node of a callback function in the extended call graph, $\mathcal{CG}_{extended}$, ($\mathcal{C}_1$) , b) The callee is represented by a function pointer that corresponds to a callback type ($\mathcal{C}_2$) or a callback signature ($\mathcal{C}_3$) and the caller or context of the callsite has an outgoing edge to a matching callback function in the extended call graph. We use $\mathcal{C}_{1,2,3}$ to represent $\mathcal{C}_1 \vee \mathcal{C}_2 \vee \mathcal{C}_3$. Note that $\mathcal{C}_{1,2,3}$ evaluated on the call expression, $fexpr$, and the extended call graph. We use $CB(fexp)$ to denote the set of callback functions that may get executed according to the extended call graph. As an example, in Figure 5 for the callsite `a0(&t1)` at line 48, $CB($`a0`$)$ consists of `cb1` as the API function `a0` has an outgoing edge to the callback function `cb1` in the extended call graph generated by Algorithm 1.

A key step in the callback-aware analysis is propagating the metadata of every qualifying path summary to the current path (see the definition of *propagate* in Figure 8). Ideally, this requires precise knowledge of whether the callback has been executed on the current path. Our analysis cannot precisely decide this as we do not analyze the API function bodies. However, assuming that the callbacks perform some bug relevant operations and, hence, are associated with some path summaries, we need to incorporate this implicit data-flow to the data-flow tracked at the application component level. We do this nondeterministically in the *callback-aware* rule by also considering to keep the metadata unchanged (see the definition of *ignore* in Figure 8) to model the fact that the callback may not be executed at the given call site.

If one of the conditions $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ in Figure 8 holds and we consider to propagate the data-flow in the path summary, we consider each path summary of the qualifying callback

Fig. 6: Rules common to the Summary Mode and the Summary-Aware Mode in Callback-Aware Analysis.

$$\frac{\langle stmt; S, \mathcal{IPS}\rangle, stmt \text{ is the first statement}}{\langle stmt; S, \mathcal{IPS}, \mathcal{E}\rangle},$$
$$\mathcal{E}.\mathcal{AL} = \mathcal{E}.\mathcal{FR} = \mathcal{E}.\mathcal{FRA} = \emptyset$$
$$\mathcal{E}.\mathcal{ACQ} = \mathcal{E}.\mathcal{RL} = \mathcal{E}.\mathcal{ACQR} = \emptyset$$

$$\frac{\langle o = alloc; S, \mathcal{IPS}\rangle \longrightarrow \langle S, \mathcal{IPS}'\rangle}{\langle o = alloc; S, \mathcal{IPS}, \mathcal{E}\rangle \longrightarrow \langle S, \mathcal{IPS}', \mathcal{E}'\rangle}, \text{where}$$
$$\mathcal{E}'.\mathcal{AL} = \mathcal{E}.\mathcal{AL} \cup \{Type(o)\},$$

$$\frac{\langle free\ o; S, \mathcal{IPS}\rangle \longrightarrow \langle S, \mathcal{IPS}'\rangle}{\langle free\ o; S, \mathcal{IPS}, \mathcal{E}\rangle \longrightarrow \langle S, \mathcal{IPS}', \mathcal{E}'\rangle}, \text{where}$$
$$\mathcal{E}'.\mathcal{AL} = \mathcal{E}.\mathcal{AL} \setminus \{Type(o)\}, \mathcal{E}'.\mathcal{FR} = \mathcal{E}.\mathcal{FR} \cup \{Type(o)\},$$
$$\mathcal{E}'.\mathcal{FRA} = \textbf{if } Type(o) \in \mathcal{E}.\mathcal{AL} \textbf{ then } \mathcal{E}.\mathcal{FRA} \textbf{ else } \mathcal{E}.\mathcal{FRA} \cup \{Type(o)\}$$

$$\text{(Double-free) } \mathcal{E}'.\mathcal{DF} = \textbf{if } Type(o) \in \mathcal{FR} \textbf{ then } \mathcal{E}.\mathcal{DF} \cup \{(t, free\ o)\} \textbf{ else } \mathcal{E}.\mathcal{DF}$$

$$\frac{\langle acquire\ o; S, \mathcal{IPS}\rangle \longrightarrow \langle S, \mathcal{IPS}'\rangle}{\langle acquire\ o; S, \mathcal{IPS}, \mathcal{E}\rangle \longrightarrow \langle S, \mathcal{IPS}', \mathcal{E}'\rangle}, \text{where}$$
$$\mathcal{E}'.\mathcal{ACQ} = \mathcal{E}.\mathcal{ACQ} \cup \{Type(o)\},$$
$$\mathcal{E}'.\mathcal{ACQR} = \textbf{if } Type(o) \in \mathcal{E}.\mathcal{RL} \textbf{ then } \mathcal{E}.\mathcal{ACQR} \textbf{ else } \mathcal{E}.\mathcal{ACQR} \cup \{Type(o)\}$$

$$\text{(Double-locking) } \mathcal{E}'.\mathcal{DL} = \textbf{if } Type(o) \in \mathcal{ACQ} \textbf{ then } \mathcal{E}.\mathcal{DL} \cup \{(t, acquire\ o)\} \textbf{ else } \mathcal{E}.\mathcal{DL}$$

$$\frac{\langle release\ o; S, \mathcal{IPS}\rangle \longrightarrow \langle S, \mathcal{IPS}'\rangle}{\langle release\ o; S, \mathcal{IPS}, \mathcal{E}\rangle \longrightarrow \langle S, \mathcal{IPS}', \mathcal{E}'\rangle}, \text{where}$$
$$\mathcal{E}'.\mathcal{ACQ} = \mathcal{E}.\mathcal{ACQ} \setminus \{Type(o)\}$$
$$\mathcal{E}'.\mathcal{RL} = \mathcal{E}.\mathcal{RL} \cup \{Type(o)\}$$

Fig. 7: An additional rule for the Summary Mode in Callback-Aware Analysis.

$$\frac{\langle return\ exp; S, \mathcal{IPS}\rangle \Downarrow \langle S, \mathcal{IPS}'\rangle}{\langle return\ exp; S, \mathcal{IPS}, \mathcal{E}\rangle \Downarrow \langle S, \mathcal{IPS}', \mathcal{E}'\rangle}, \text{where}$$
$$\mathcal{PS}' = \textbf{if } Context(return\ exp) \in Callback \textbf{ then } \mathcal{PS} \cup \{(Context(return\ exp), \mathcal{E}')\} \textbf{ else } \mathcal{PS}$$

Fig. 8: Rules exclusive to the Summary Aware Mode in Callback-Aware Analysis.

callback-aware:

$$\frac{\langle fexp(args); S, \mathcal{IPS}\rangle \Downarrow \langle S, \mathcal{IPS}'\rangle,\ \mathcal{C}_{1,2,3} \wedge \exists cb.\ cb \in CB(fexp) \wedge \exists ps \in \mathcal{PS}.\ ps = (cb, \mathcal{E}_1)}{nondet\_choose(\textbf{propagate}, \textbf{ignore})}$$

, where

$$propagate \equiv \langle fexp(args); S, \mathcal{IPS}, \mathcal{E}\rangle \Downarrow \langle S, \mathcal{IPS}', \mathcal{E}'\rangle, \text{ where}$$
$$\mathcal{E}'.\mathcal{AL} = \mathcal{E}.\mathcal{AL} \cup ps.\mathcal{E}.\mathcal{AL}$$
$$\mathcal{E}'.\mathcal{FR} = \mathcal{E}.\mathcal{FR} \cup ps.\mathcal{E}.\mathcal{FR}$$
$$\mathcal{E}'.\mathcal{ACQ} = \mathcal{E}.\mathcal{ACQ} \cup ps.\mathcal{E}.\mathcal{ACQ}$$
$$\mathcal{E}'.\mathcal{RL} = \mathcal{E}.\mathcal{RL} \cup ps.\mathcal{E}.\mathcal{RL}$$

(Double-free) $\mathcal{E}'.\mathcal{DF} = \textbf{if } \exists t.\ t \in \mathcal{FR}.\ t \in ps.\mathcal{E}.\mathcal{FRA} \textbf{ then } \mathcal{E}.\mathcal{DF} \cup \{(t, fexp(args))\} \textbf{ else } \mathcal{E}.\mathcal{DF}$

(Double-locking) $\mathcal{E}'.\mathcal{DL} = \textbf{if } \exists t.\ t \in \mathcal{ACQ}.\ t \in ps.\mathcal{E}.\mathcal{ACQR} \textbf{ then } \mathcal{E}.\mathcal{DL} \cup \{(t, fexp(args))\} \textbf{ else } \mathcal{E}.\mathcal{DL}$

$$ignore \equiv \langle fexp(args); S, \mathcal{IPS}, \mathcal{E}\rangle \Downarrow \langle S, \mathcal{IPS}', \mathcal{E}\rangle$$

function to check for a manifestation of a deep bug by combining it with the metadata collected so far on the current path as explained in the bug detection section (see labels Double-free and Double-locking) in Figure 8. Note that our algorithm also handles bugs manifested as a result of a conflicting operation after a callback gets executed as in the case of the `usbtv` double-free vulnerability found by MOXCAFE and presented in Section IV. Such bugs are detected by the checks shown in Figure 6 during the Summary-Aware Mode.

Considering the running example given in Figure 2, we have an edge from the `snd_card_free` API function to the driver's `snd_usbmidi_rawmidi_free` callback function in the extended call graph for the USB midi driver. In summary mode, our algorithm analyzes the paths that start from `snd_usbmidi_rawmidi_free` and checks for bug-relevant operations. It realizes that on such a path `kfree` is called on a `snd_usb_midi` type object (line 62). So a path summary that is associated with `snd_usbmidi_rawmidi_free` gets stored with both $\mathcal{FR}$ and $\mathcal{FRA}$ set to $\{$`snd_usb_midi`$\}$. In summary-aware mode, on the error path that traverses the lines 16, 17, 18, 20, 21, 22, and 23 the algorithm has recorded that `snd_usb_midi` is in $\mathcal{FR}$ of the current path. At line 23, it realizes that `snd_card_free` is an API function that has an outgoing edge to the `snd_usbmidi_rawmidi_free` callback function according to the extended call graph. When it applies the $propagate$ rule and incorporates the data-flow facts in the path summary, ps, that starts at `snd_usbmidi_rawmidi_free`, it detects the double-free by finding out that `snd_usb_midi` appears in both $\mathcal{FR}$ of the current path and the path summary's $\mathcal{FRA}$, $ps.\mathcal{E}.\mathcal{FRA}$.

TABLE II: Time (in secs) taken by each stage of Extended Call Graph generation.

Stages	Min	Max	Median
Metadata Collection	9.65	749.30	207.53
Extract Implicit Calls	0.06	19.40	3.62

IV. Evaluation

We have applied the presented approach to a diverse set of Linux drivers with the goal of detecting deep bugs/vulnerabilities that involve callbacks. We have implemented the presented approach using the LLVM compiler framework [31], version 5.0.0. MOXCAFE has a total of 6.5K SLOC. Our approach works on the source code. We have used LLVM's AST Analyzer component to implement the metadata collection and its static analyzer [28] to implement detection of the two types of deep bugs we considered. We have used version v4.14-rc2 of the Linux kernel as recommended by the clang-kernel-build project [32] to benefit from the provided patches that made compilation of Linux possible.

We have used 40 Linux drivers to evaluate our approach. We have included a total of 12 framework components/layers that these drivers use. These framework components belong to the following kernel layers: `video` (v) , `sound` (s), `gadget` (g), `dwc3` (d), `serial` (r), `tty` (t), `block` (b), `scsi` (c), `network` (n), `usb core` (u), `input` (i), and `hid` (h). In this section, we summarize the experimental results and refer the reader to Table IV in the Appendix for more details.

Table II summarizes the timing results for Extended Call Graph generation. As the data shows, much more time is spent on the metadata collection than on the extraction of implicit calls. This is especially more significant for larger frameworks such as the sound layer. However, metadata collection for framework modules needs to be performed only once and can be reused each time the call graph inference algorithm needs to be run for a new driver.

To determine the feasibility of the inferred paths, we used the capability of our tool to list all possible simple paths from each API function to the callback it may call. For each inferred edge, we have gone through the list of possible paths and analyzed the implementation of each function on the chain to determine the feasibility. Once we found a path that we thought was feasible, we skipped the other possible explanations for the inferred edge and continued analyzing the next inferred edge, and so on. When the code was complicated, e.g., it was difficult to trace the data-flow dependencies, we conservatively concluded as infeasible.

The false positive rate for our extended call generation is 18% on average. The major reason for false positives is due to infeasibility of data-flow constraints on the inferred call chains. Some of these data-flow constraints are due to consolidation of a variety of functionalities inside the same function and using

function parameters, e.g., states or configuration information, to implement the relevant functionality. In our experience the longer the callback chain the higher the chance of it being a false positive. Another source of infeasible data-flow constraints is functions that involve the generic `struct device` and inferring call chains by mistaking the parent device as the device being manipulated by the device driver.

We were able to detect real bugs/vulnerabilities in 6 of the benchmarks. 3 of these were double-free (`usbtv`, `f_loopback`, and `f_midi`) and 3 of them were double-locking (`f_hid`, `f_printer`, and `renesas_usb3`). 4 of these were new[3], denoted with an * appended to the driver name, and 2 of them were recently fixed in the newer versions. Table III shows the unique API to generic callback function call paths that have been detected by our extended call graph generation algorithm, Algorithm 1, and have been found to involve in an API misuse bug as detected by our callback-aware analysis. Specifically, $AC1$ got involved in the `usbtv` double-free vulnerability, $AC2$ got involved in the `f_loopback` double-free vulnerability, $AC3$ got involved in the `f_midi` double-free vulnerability, the `f_hid` double-locking bug, and the `f_printer` double-locking bug, and $AC4$ got involved in the `renesas3` double-locking bug. What is common to these API to callback paths is that a framework callback, as denoted by FCB, also gets involved. So, any static analysis that targets callback related API misuse bugs must also resolve function pointers in the framework code. Although the `f_printer` and the `renesas3` double-locking bugs were previously known and fixed, the API to callback chains were not explicitly provided in the relevant discussion forums. So, to our knowledge, MOXCAFE is the first static analysis tool to reveal these four API to callback chains for which the developers should guard their code and prevent a potential API misuse.

A. False Positive Analysis

MOXCAFE reported an average of 0.35 false positives per benchmark in our data set and achieved an average false positive rate of 60%. Although the false positive rate is high, it should be evaluated in the context of the framework and the applications we targeted: the Linux kernel subsystems and the drivers. Other static analysis work that detect vulnerabilities in the Linux kernel report high false positive rates, either explicitly as in [33] (a false positive rate of 76.4%-83.30%) or implicitly as in K-Miner [34] (29 true positives out of 539) and APISan [22] (54 true positives out of 445). We analyzed the false positives reported by MOXCAFE and identified three main root causes as we explain below.

a) Missing context information: API functions have additional side effects that need to be captured to precisely reason about whether an API function ends up executing a callback. For instance, some API functions, e.g., `video_register_device`, may register certain framework callbacks, e.g., `v4l2_device_release`, only when the API function returns a success return value. Such side effects determine whether the callback gets called by another related API function, e.g., `v4l2_device_unregister`. To eliminate this type of false positives, more data-flow facts must be captured from the API functions and incorporated into the extended call graph generation as well as to the application-level bug detection. 70% of the false positives falls into this category.

b) Type-based alias analysis: The summary information for callbacks are represented using types of the manipulated data structures. During summary-aware bug detection, conflicting operations were matched based on the type. This type of false positives can be eliminated by performing path-sensitive analysis for the the callbacks and incorporating the precise context information from the API call sites. 30% of the false positives falls into this category.

TABLE III: The API to callback paths that are detected by Algorithm 1 and are involved in the deep bugs detected by our callback-aware analysis. FCB and CB represent framework callbacks and driver callbacks, respectively.

Implicit Path	**API to Callback Chain**
AC1	v4l2_device_put kref_put v4l2_device_release (FCB) (struct v4l2_device, release) (CB)
AC2	usb_ep_disable dwc3_gadget_ep_disable (FCB) __dwc3_gadget_ep_disable dwc3_remove_requests dwc3_gadget_giveback usb_gadget_giveback_request (struct usb_request, complete) (CB)
AC3	usb_ep_queue dwc3_gadget_ep_queue (FCB) __dwc3_gadget_ep_queue __dwc3_gadget_kick_transfer dwc3_gadget_giveback usb_gadget_giveback_request (struct usb_request, complete) (CB)
AC4	usb3_disconnect composite_disconnect (FCB) reset_config afunc_disable free_ep usb_ep_dequeue (struct usb_ep_ops, dequeue) (CB)

B. Discussion

The deep bugs we have found using MOXCAFE suggest that the developers may not be fully aware of the driver callback functions that may get called from the kernel API

[3]These bugs have been confirmed by the kernel developers. For some we have provided the patches and for others we have participated in the discussions for the right patch.

functions. We think that this would be an issue with any large framework that uses callbacks. Some of the API function to callback chains in our examples were of length 7. We believe that this much complexity is challenging for developers to understand the side effects of every single API function they need to use by reviewing the framework code.

```
err = usb_ep_queue(midi->out_ep, ...); // 1st free
if (err) {
    ERROR(midi, ...);
    free_ep_req(midi->out_ep, req); // 2nd free
    return err;
}
```

Fig. 9: The double-free found in the `f_midi` driver.

We recommend the API designers and developers to provide sufficient documentation on the behavior of the API functions regarding the callbacks. One thing we have observed by reading the patch discussions and reviewing the code is that the kernel developers have better awareness on the side effects of certain API functions such as the `usb_gadget_giveback_request` function, which acquires a lock before calling a callback and appears deep in some of the call chains, e.g., in $AC2$ and $AC3$ in Table III. This awareness develops as bugs get manifested. However, developing an understanding of a complex framework as bugs get manifested leaves the system at risk for long periods of time. Also, developing awareness on certain API functions does not prevent similar bugs from being manifested using different API chains, e.g., the double-free shown in Figure 9. In this one, on an error path in the `f_midi_set_alt` function, the API function `usb_ep_queue` fails to queue a request and may potentially free the request buffer via the `f_midi_complete` callback of the driver if the underlying device controller is implemented as in `dwc3` (see AC3 in Table III for the API to callback chain). In the error handling code (inside the `if` statement), the request buffer is freed this time via the helper function `free_ep_req`. This vulnerability was introduced to fix a memory leak regarding the request buffer [35]. We think that tools like MOXCAFE can help developers to detect deep bugs early on in the development stage.

C. Comparison with DSA

We have compared our approach with a points-to analysis, Data Structure Analysis (DSA) [15], that is context-sensitive, flow-insensitive, and field-sensitive. DSA has been used to aid compiler optimization including memory allocation for heap-based data structures. As a by-product of its analysis, DSA also provides a call graph with concrete callees for call sites that involve function pointers. We have chosen DSA as it has been previously applied to the Linux kernel and has been shown to scale to such large code bases. We have used DSA's implementation [36] for LLVM 3.7.

Figure 10 shows the timing results of the summary mode (Pass I) and the summary-aware mode (Pass II) of our staged static analysis. We ran Pass I and Pass II with the extended call graph generated by Algorithm 1 as well as with the extended call graph derived from the call graph generated by DSA. As the results show, the performance with these two types of extended call graphs are comparable. However, when we run Pass I and Pass II using DSA's call graph, we could only detect two of the deep bugs that were found by MOXCAFE.

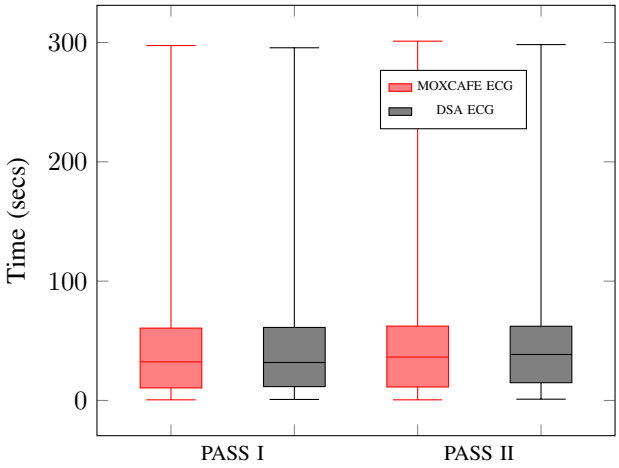

Fig. 10: Comparison of using the Extended Call Graph generated by Algorithm 1 with using that derived from DSA's Call Graph during staged static analysis.

We think that there are two main issues related to the call graphs generated by DSA: imprecision and incompleteness. The imprecision concerns DSA's field sensitivity being preserved only for type homogenous cases. So this causes pointer assignments that involve type casting to collapse the associated nodes and stop tracking individual fields, practically losing field sensitivity in such cases (see the explanation for the collapsed flag in [15]) and leading to spurious cycles and SCCs. DSA generates more targets for function pointers than our inference algorithm. We are not sure for the reason for the incompleteness. However, we suspect that DSA is not able to handle the `container_of` macro used in the Linux kernel. This macro uses a pointer to an embedded data structure to derive a pointer to the embedding data structure using pointer arithmetic. It basically subtracts an offset from the given address. We think that any general points-to analysis algorithm has to handle this construct properly to generate a complete call graph for the Linux kernel. Our inference algorithm works at the source code level instead of the bitcode level. Therefore, it can precisely extract the type information of callbacks at the callsites that use function pointers.

V. Secure Development with Callbacks

Secure software development requires incorporating security at the design stage. This should be the case for the development of applications as well as for the development of the software framework. In this section, we would like to discuss several ways through which the callback mechanism can be made more secure.

a) Documentation: We would like to note that ideally sufficient documentation should be provided for every function

whether it executes a callback or not. However, due to the implicit nature of callbacks, it is even more critical to provide documentation for any function that gets called on a chain that involves callbacks and, hence, may execute conflicting operations with those performed by the callbacks.

As we mentioned in Section IV-B, kernel developers seem to be sensitive to the known side effects of API functions and are relatively more effective in using such API correctly, e.g., in contexts that use the `usb_gadget_giveback_request` as discussed in Section IV-B. For all the API misuse vulnerabilities we detected, there was a lack of documentation w.r.t. callback related side effects in the source files of the kernel subsystems. We think that any API function that executes a callback must be accompanied with proper documentation on this aspect. At a minimum, the documentation must specify the type of callback, either as a type and field combination or as a function signature. Ideally, the documentation should also specify the specific cases and conditions in which the callback gets executed, e.g., only on success paths, for specific input values, while holding a specific lock, etc.

The applications, on the other hand, should provide documentation for the callbacks they implement and for the functions that end up executing callbacks. For callback functions, specifying the pre and post conditions and the API functions from which they may get executed would help identify specific API misuse issues. For the functions that may execute callbacks, information on the called API functions with callback side effects should be specified, e.g., acquires lock X due to execution of callback Y. This information can help developers better understand the impact of their changes.

b) Secure Interface and Data Structure Design: The major challenge with callbacks is that it is an implicit mechanism. However, it would be possible to minimize the security risk associated with callback execution if callbacks could be implemented in a defensive way; that is secure programming practices can be applied for the callbacks. As an example, it is a secure programming practice to set a pointer to NULL after deallocating the memory pointed by that pointer. As an example, a callback that deallocates memory should ideally set the pointer(s) to the deallocated memory to NULL so that use-after-free and double-free vulnerabilities can be prevented. However, since callbacks are not designed with security in mind, typically they are provided with the address of the memory to be deallocated and not with the address of the pointers to such memory. As an example, it would be ideal for the `snd_usbmidi_rawmidi_free` callback in Figure 2 to be able to set the pointer to NULL after `kfree`. For this specific case, passing a double pointer does not work as the `snd_usbmidi_rawmidi_free` function uses the argument pointer `rmidi` to derive the address of the memory to be deallocated (`umidi`) (line 52 in Figure 2). A possible solution would be to change the type of `private_data` field in `struct snd_rawmidi` to a double pointer to the `snd_usb_midi` type. However, there may be several pointers for the same memory location and possibly different sets of aliases depending on the context. One way to deal with this type of complication is to define an auxiliary data structure to hold callback relevant data and pass such auxiliary data to the API functions that calls the relevant callback function. So the auxiliary data needs to be passed down the complete call chain. Another way is to store a pointer to the auxiliary data in the data structure that the callback will manipulate. Either of these cases requires incorporating security at the design stage and baking security into the interfaces of the APIs as well as the data structures used by such APIs. Otherwise, it would simply be impractical to change the existing code base to minimize the security risk.

VI. Related Work

Applications of model checking to system software [12]–[14], [18], [20], [37] expect the user to provide an environment model that includes the life-cycle of a component and analyzes the component behavior in the context of this environment model. Similar to SDV [13], the Linux Driver Verification (LDV) framework [20] statically verifies device drivers by combining a manually specified environment model with the device driver source code and using software model checkers [14], [18]. Unlike SDV [13] and LDV [20], our analysis handles function pointers defined and/or used by the framework components and incorporates callback chains into the analysis. We believe that our call graph inference algorithm can be used to support approaches as in [12]–[14], [18], [20] with an automatically generated environment model.

Our results support the empirical evidence provided by Milanova et al. in [38] that for the purposes of call graph construction in the presence of function pointers, inexpensive pointer analyses, i.e., flow and context insensitive, can provide very good precision. In this paper, we further showed that for detecting callback related deep bugs, a general points-to analysis is not needed and a field-based, flow-insensitive, and context-insensitive analysis that handles only function pointers may work sufficiently well.

DSA [15] is a scalable points-to analysis algorithm that is context-sensitive, field-sensitive, and flow-insensitive. DSA loses field sensitivity when it finds that a type is not used consistently and collapses all the fields, which is performed to achieve scalability as mentioned in [15]. As we have shown in this paper, our approach provides better precision than DSA for call graph inference as it maintains field sensitivity. Also, our approach handles function target resolution correctly even in the presence of cyclic dependencies in the call graph.

Function pointers are considered by Gunawi et al. [16] and by Rubio-González et al. [17], which use static analysis to detect incorrect error propagation in file systems. Function pointers are identified by handling initializations of global structures and assignments performed inside the function bodies. However, unlike our approach, they do not consider indirect chain of calls due to function pointers, which requires computing a transitive closure as we presented in this paper.

Detection of misuse of error codes that represent pointer values in Linux file systems is presented in [19]. Relying on the assumption that function pointers are used in a fairly

restrictive manner in the file system, they do not perform points-to analysis and instead hard-code possible candidates. It is reported that 80% of the function pointers could be resolved this way, which indicates that errors that involve the remaining 20% of the function pointers might be missed.

A technique for detecting race conditions in drivers is presented in [9]. The entry points of Linux device drivers are extracted using a tool called Chauffeur [39], which is used to identify threads that may run concurrently. So they consider identification of asynchronous callbacks only. As we show in this paper, synchronous callbacks, which get executed from the API functions, do play an important role in the formation of a complete call graph and, hence, their incorporation into the analysis would potentially detect more races.

An inter-procedural path-sensitive program analysis is used in [21] to detect deep semantic bugs on different implementations of file systems. Their treatment of function pointers is limited to identification of entry points to the analyzed file system implementations.

Tree-adjoining language reachability is used in [40] to summarize library calls in the presence of callbacks. The goal is to be able to perform precise context-sensitive data-dependence analysis that incorporates data-flow through the library code. Our approach abstracts away the data-flow through the library code and summarizes APIs in terms of the callback functions they may execute.

Ramos et al. [33] use under-constrained symbolic execution to detect deep bugs that get manifested in functions deep within a program. Their approach enables analysis of functions by abstracting the specific calling contexts. Our approach, on the other hand, abstracts the data-flow in API functions while keeping track of callback related side effects.

Apisan [22] detects bugs that are due to incorrect usage of APIs in large code bases. It uses relaxed symbolic execution to infer semantic beliefs for API usage and reports bugs when a deviation from the inferred beliefs is detected. We believe that this work is complementary to our work as stronger and richer semantic beliefs can be inferred when implicit paths are incorporated to the analysis. Also, our approach does not rely on any generalization, which may not be applicable or available for certain bugs.

DR. CHECKER [23] presents a scalable static analysis approach that, similar to our work, focuses on the analysis of drivers. Similar to our approach, their function pointer target resolution is type-based. Unlike our approach, their function pointer resolution is performed at the scope of the driver only, and, hence, their analysis does not consider implicit driver code paths due to interaction with the kernel layers.

DIFUZE [41] uses a combination of static analysis and fuzzing to analyze device drivers as they execute on the device. It automatically recovers data structures to craft tests that exercise the `ioctl` entry points. Our approach can provide feedback to tools like DIFUZE and fuzzers such as vUSBf [27] to identify the entry points of the driver that may host specific type of vulnerabilities.

K-Miner [34] detects memory corruption vulnerabilities in the Linux kernel by partitioning the kernel code around the system call interface and using inter-procedural static analysis. MOXCAFE also uses inter-procedural analysis. However, it focuses on callback related API misuses and achieves scalability by abstracting the way drivers interact with the kernel.

Although our staged analysis uses LLVM framework's implementation of the algorithm presented in [29], it can be based on any path-sensitive analysis including those presented in [42] and [43].

VII. Conclusions

We have presented a multi-stage static analysis approach for detecting deep vulnerabilities that involve callbacks. Our approach leverages the programming model of frameworks that use the callback types and the signatures for handling callbacks. Scalability is achieved by summarizing the interaction of application modules with the framework components through implicit edges in the extended call graph of the application module. We have shown the effectiveness of the approach by applying it to several Linux drivers and kernel layers. Using the presented approach, we were able to detect two known and four new callback related deep bugs/vulnerabilities. In future work, we are planning to explore more types of bugs and incorporate more precise context information into the analysis.

References

[1] Y. Acar, M. Backes, S. Fahl, S. L. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 154–171.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of API-Misuse Detectors," *CoRR*, vol. abs/1712.00242, 2017.

[3] B. A. Myers and J. Stylos, "Improving API Usability," *Commun. ACM*, vol. 59, no. 6, May 2016.

[4] S. Indela, M. Kulkarni, K. Nayak, and T. Dumitraş, "Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward!, 2016.

[5] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016.

[6] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL Development in an Appified World," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013.

[7] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[8] "CVE-2016-2384: exploiting a double-free in the usb-midi linux kernel driver," https://xairy.github.io/blog/2016/cve-2016-2384.

[9] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15*, 2015, pp. 166–177.

[10] F. Fowze and T. Yavuz, "Specification, verification, and synthesis using extended state machines with callbacks," in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE'16*, 2016, pp. 95–104.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01, 2001, pp. 57–72.

[12] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," in *5th Symposium on Operating System Design and Implementation (OSDI'02)*, 2002.

[13] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06, 2006, pp. 73–85.

[14] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[15] C. Lattner, A. Lenharth, and V. Adve, "Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, 2007, pp. 278–289.

[16] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit, "EIO: Error Handling is Occasionally Correct," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08, 2008, pp. 14:1–14:16.

[17] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, pp. 270–280.

[18] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *Computer Aided Verification - 23rd International Conference, CAV'11. Proceedings*, 2011, pp. 184–190.

[19] C. Rubio-González and B. Liblit, "Defective Error/Pointer Interactions in the Linux Kernel," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 111–121.

[20] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov, "Configurable toolset for static verification of operating systems kernel modules," *Programming and Computer Software*, vol. 41, no. 1, pp. 49–64, 2015.

[21] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking Semantic Correctness: The Case of Finding File System Bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015, pp. 361–377.

[22] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API Usages through Semantic Cross-Checking," in *25th USENIX Security Symposium, USENIX Security'16*, 2016, pp. 363–378.

[23] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers," in *26th USENIX Security Symposium, USENIX Security'17*, 2017, pp. 1007–1024.

[24] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'09*, 2009, pp. 226–238.

[25] ——, "Flow-sensitive Pointer Analysis for Millions of Lines of Code," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011, pp. 289–298.

[26] "The Kernel Address Sanitizer (KASAN)," https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html.

[27] "vusbf-Framework," https://github.com/schumilo/vUSBf.

[28] "Clang static analyzer," http://clang-analyzer.llvm.org/.

[29] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, 1995, pp. 49–61.

[30] Z. Xu, T. Kremenek, and J. Zhang, "A Memory Model for Static Analysis of C Programs," in *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ser. ISoLA'10, 2010.

[31] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[32] A. Potapenko, "clang-kernel-build," https://github.com/ramosian-glider/clang-kernel-build.

[33] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, 2015, pp. 49–64.

[34] D. Gens, S. Schmitt, L. Davi, and A. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[35] "usb: gadget: f_midi: fix leak on failed to enqueue out requests," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/drivers/usb/gadget/function/f_midi.c?id=ad0d1a058eac46503edbc510d1ce44c5df8e0c91.

[36] "Data Structure Analysis - LLVM ," https://github.com/jtcriswell/llvm-dsa.

[37] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using Model Checking to Find Serious File System Errors," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, Nov. 2006.

[38] A. Milanova, A. Rountev, and B. G. Ryder, "Precise Call Graph Construction in the Presence of Function Pointers," in *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, 2002, pp. 155–162.

[39] "Clang AST frontend for Linux device driver analysis," https://github.com/mc-imperial/chauffeur.

[40] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, 2015, pp. 83–95.

[41] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface Aware Fuzzing for Kernel Drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS'17*, 2017, pp. 2123–2138.

[42] T. Reps, S. Schwoon, and S. Jha, "Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03, 2003, pp. 189–213.

[43] I. Dillig, T. Dillig, and A. Aiken, "Sound, Complete and Scalable Path-sensitive Analysis," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, 2008, pp. 270–280.

APPENDIX
DETAILED EXPERIMENTAL RESULTS

Table IV shows our experimental results on extended call graph generation and callback-aware analysis for deep bug detection. **FT**, **#F** , **N**, **MT**, and **EIC** denote the type of the framework modules, the number of framework modules, the number of iterations after which the extended call graph stabilized (maximum # iterations was set to 15), the time for metadata collection, and the time for generating the extended call graph using the metadata, respectively.

The columns **F** and **T** in the **Extended CG Inference** section of Table IV shows the number of false and true inferences, respectively.

MOXCAFE columns in Table IV represent the data for our callback-aware analysis that uses the extended call graph generated using Algorithm 1. Real bugs/vulnerabilities are shown in bold. **S**, **#P**, **P I**, and **P II** denote the number of path summaries, the number of paths explored, the time for the summary mode and the time for the summary-aware mode, respectively. The **Bugs** column presents the # of false, **F**, and the # of real, **R**, deep bugs detected. **DSA CG Extended + Callback Aware** column of Table IV shows the results of callback-aware analysis that uses the call graph generated by

TABLE IV: Extended Call Graph Generation and Callback-aware bug detection for various Linux drivers. Driver names in bold represent the drivers in which real deep bugs have been detected.

Driver	Extended CG Inference							MOXCAFE						DSA CG Extended + Callback Aware					
	FT	#F	N	Time (s)		Inferred		S	#P	P I	P II	Bugs		S	#P	P I	P II	Bugs	
				MT	EIC	F	T					F	R					F	R
usbtv*	v,s	72	4	255.96	4.95	22	24	3	140	21.57	23.56	0	**1**	1	2490	21.97	30.52	0	0
airspy	v	33	3	134.42	1.79	0	8	3	153	10.12	10.15	1	0	0	153	11.75	11.80	0	0
em28xx	v,s	72	3	252.75	4.40	4	7	9	1022	297.46	301.14	4	0	2	786	295.60	298.27	0	0
hackr	v	33	3	133.06	2.60	2	7	3	220	11.08	11.20	2	0	2	220	11.72	11.77	1	0
hdpvr	v	33	3	155.79	19.40	1	4	1	305	26.19	26.57	0	0	2	304	25.72	26.75	0	0
stkweb	v	33	3	136.70	4.67	0	3	1	633	47.45	47.86	1	0	9	3519	47.20	48.13	1	0
zr364	v	33	3	129.85	4.31	2	31	3	396	16.78	16.92	2	0	3	609	17.20	17.92	2	0
f_acm	d,g	45	3	207.53	4.27	5	12	5	64	0.71	0.72	0	0	4	4183	0.95	8.17	0	0
f_eem	d,g	45	3	207.53	3.60	0	5	1	656	8.03	8.24	0	0	6	1648	8.48	9.92	0	0
f_fs	d,g	45	2	207.53	3.64	4	23	8	1284	103.39	103.60	0	0	8	7549	100.64	122.51	0	0
f_hid*	d,g	45	3	207.53	3.64	0	8	5	3220	10.81	18.03	2	**1**	7	446	10.41	10.48	0	1
f_loo*	d,g	45	3	207.53	3.60	0	6	3	3923	0.79	9.90	0	**1**	8	15224	1.15	11.49	0	0
f_mass.	d,g b, c	98	3	749.30	9.70	1	21	7	469	112.93	114.22	0	0	12	9169	110.06	122.30	0	0
f_mid*	d,g s	82	3	255.94	6.31	6	11	4	3029	30.06	37.02	0	**1**	6	30962	30.93	43.19	0	0
f_ncm	d,g	45	3	207.53	3.65	0	14	5	159	11.13	11.29	0	0	6	4728	11.55	22.04	0	0
f_obex	d,g	45	2	207.53	3.70	0	4	1	34	0.54	0.54	0	0	4	212	0.87	1.06	0	0
f_phon.	d,g	45	3	207.53	3.59	0	7	5	274	17.99	19.58	0	0	6	3366	18.34	33.17	0	0
f_print.	d,g	45	2	205.30	3.22	0	7	5	884	40.02	48.23	0	**1**	10	10867	40.55	55.39	0	1
f_seri.	d,g r, t	59	2	244.90	4.40	0	1	1	28	0.52	0.52	0	0	4	266	0.82	1.12	0	0
f_tcm	d,g	45	3	207.53	3.78	2	43	7	11942	10.32	20.02	0	0	13	56907	11.88	50.46	0	0
renes.	g	37	6	203.72	1.95	13	26	9	613	34.06	45.59	1	**1**	6	762	32.70	54.01	1	0
vudc	g	37	4	209.86	6.57	14	18	11	298	5.09	12.95	0	0	12	404	5.33	22.63	0	0
6fire	s	39	4	129.78	2.49	9	7	6	236	41.75	42.06	0	0	8	236	41.21	41.49	0	0
caiaq	s	39	2	129.95	2.27	2	2	2	143	68.33	68.57	0	0	3	143	67.40	67.70	0	0
hiface	s	39	4	129.07	2.70	2	7	4	1727	3.78	10.10	0	0	6	5105	4.02	10.00	0	0
midi	s	39	4	169.08	3.16	13	123	3	1454	206.90	209.30	0	0	9	971	207.14	208.98	0	0
usx2y	s	39	4	129.85	3.53	5	28	4	8833	107.10	119.45	0	0	7	5408	107.04	130.26	0	0
mass.	b,c	53	2	243.06	4.70	11	8	3	155	110.23	111.30	0	0	1	194	44..83	45.09	0	0
uas	b,c	53	2	222.35	3.32	3	3	3	486	46.11	107.51	0	0	1	472	106.61	107.83	0	0
io_edg.	r,t	14	3	43.82	1.19	3	8	5	893	63.23	63.42	0	0	11	1405	61.24	62.49	0	0
mxupo.	r,t	14	2	43.84	0.54	0	2	1	303	35.40	35.70	0	0	9	303	34.24	36.23	0	0
usbser	r,t	14	2	43.80	0.83	3	16	3	203	37.73	37.87	1	0	0	203	37.80	38.54	0	0
cdcacm	t	11	2	28.22	0.42	5	12	5	462	30.62	30.64	0	0	8	454	29.58	29.83	0	0
hso	n	46	3	337.06	5.97	4	7	2	644	60.41	60.68	0	0	10	4447	62.69	70.12	0	0
pegasus	n	47	2	344.40	6.46	1	3	1	1492	10.56	10.09	0	0	3	118	10.73	10.89	0	0
r8152	n	47	2	344.40	8.89	4	28	5	2017	79.22	79.82	0	0	9	2017	79.73	80.30	0	0
usbnet	n	49	2	359.05	10.38	0	2	1	823	53.62	54.13	0	0	0	820	53.70	54.25	0	0
hdc	u	23	2	99.34	1.08	0	2	1	783	61.24	61.93	0	0	0	781	61.10	61.86	0	0
hidco.	i,h	10	3	36.14	0.34	0	9	3	1201	44.89	44.88	0	0	0	303	35.58	35.76	0	0
usbkbd	i	2	2	9.65	0.06	1	3	1	24	7.54	7.65	0	0	-	-	-	-	-	-

DSA, which could detect only two of the deep bugs that were found by MOXCAFE.