Noname manuscript No.
(will be inserted by the editor)

Analyzing System Software Components using API Model Guided Symbolic Execution

Tuba Yavuz · Ken (Yihang) Bai

Received: date / Accepted: date

Abstract Analyzing real-world software is challenging due to complexity of the software frameworks or APIs they depend on. In this paper, we present a tool, PROMPT, that facilitates the analysis of software components using *API model guided symbolic execution*. PROMPT has a specification component, PROSE, that lets users define an API model, which consists of a set of data constraints and life-cycle rules that define control-flow constraints among sequentially composed API functions. Given a PROSE model and a software component, PROMPT symbolically executes the component while enforcing the specified API model. PROMPT has been implemented on top of the KLEE symbolic execution engine and has been applied to Linux device drivers from the video, sound, and network subsystems and to some vulnerable components of BlueZ, the implementation of the Bluetooth protocol stack for the Linux kernel. PROMPT detected two new and four known memory vulnerabilities in some of the analyzed system software components.

This work was partially funded by the National Science Foundation under awards CNS-1815883 and CNS-1942235 by the Semiconductor Research Corporation. We would like to thank the anonymous reviewers for their feedback. We would like to thank Joshua Nelson for helping with the PROSE parser as an undergraduate researcher. This is a pre-print of an article published in Automated Software Engineering. The final authenticated version is available online at: https://doi.org/10.1007/s10515-020-00276-5

T. Yavuz
Benton 321, ECE Department
University of Florida
Gainesville, FL 32611
Tel.: +1-352-846-0202
E-mail: tuba@ece.ufl.edu

Y. Bai
Larsen 234, ECE Department
University of Florida
Gainesville, FL 32611

Keywords Symbolic execution · API modeling · Specification.

1 Introduction

Analyzing real-world applications requires modeling of the environment including the Application Programming Interface (API) of the underlying software framework. While software frameworks are designed to enable faster development, modularity, and extensibility, incorrect use of their APIs creates reliability issues. Recent studies on API usability [7,19,20,23,27,28] report various difficulties faced by the developers when using APIs and how API misuses may lead to vulnerabilities. Although the focus in these studies has been mostly on the misuse of cryptographic APIs, API misuse is a potential problem for any complex framework.

Due to the complexity of software frameworks, precise analysis of software components along with the framework code is not feasible. A typical solution is to analyze components using an environment model. However, manually generating environment models is error-prone. Depending on the goal of the analysis, it may require an extensive engineering effort. This challenge has recently inspired researchers to automatically synthesize API models in the form of implementations or usage rules [22, 29, 30, 34]. Although the results of these studies are promising, they rely on the existence of run-time data or sample user-space applications that can be executed to exercise the APIs of interest. However, setting up the right execution environment is challenging for systems that interact with hardware, e.g. device drivers, and those that involve complex API, e.g., cryptographic libraries.

Symbolic execution [25] has emerged as a test generation technique and has also become an important program analysis technique for finding bugs and vulnerabilities. Dynamic symbolic execution [15] can mix concrete and symbolic execution. Therefore, it provides a precise memory model and is effective in detecting memory related errors. However, symbolic execution is not scalable due to the well-known path explosion problem and so it cannot be applied to the analysis of a software framework. Therefore, effective symbolic execution of application components requires a precise model of the software framework.

In this paper, we present a modeling language, PROSE, for specifying API models and a symbolic execution based tool, PROMPT, that performs symbolic execution on a software component while enforcing the specified PROSE API model. Users can also implement API function models in the C language and leverage the metadata handling interface provided by PROMPT. Additionally, PROSE enables modeling of programming idioms, e.g., `container_of` macro in the Linux kernel, that are common in systems code, which can be used to guide PROMPT for a more precise analysis at the component level. Our approach facilitates analysis of system software components by avoiding the need for developing a test harness or changing and recompiling the underlying code base.

We have modeled the registration, setup, and cleanup APIs of the `video`, `sound`, and `network` subsystems of the Linux kernel. We have applied PROMPT to 57 Linux device drivers and devised API models to enable precise symbolic execution of these drivers. We were able to cover success as well as failure paths of the setup and teardown functions of the drivers in our evaluation set. We also analyzed some of the vulnerable components in BlueZ, an implementation of the Bluetooth protocol stack for the Linux kernel, and detected some vulnerabilities that require considerable testing effort. We also detected several real memory bugs in some of the device drivers.

Our contributions can be summarized as follows:

- We present a modeling language, PROSE, that can be used to specify API models, which incorporate the life-cycle of an event-based system and the data constraints.
- We present an open-source analysis tool[1], PROMPT, that is developed on top of the KLEE execution engine. PROMPT features model guided lazy initialization, precise simulation of life-cycle models, and metadata tracking.
- We have modeled the registration, teardown, and setup API of three different subsystems, `video`, `sound`, and `network`, in the Linux kernel and validated them using PROMPT.
- We have applied PROMPT to various components in the Linux kernel. The first case study reports on the analysis of 57 Linux device drivers using the PROSE API models for the three subsystems. The second case study reproduces some known vulnerabilities in BlueZ including one of the BlueBorne [1] vulnerabilities: CVE-2017-1000251. We detected a total of six memory related bugs, two new and four known.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of PROSE and PROMPT on a real use-after-free vulnerability. Section 4 introduces the PROSE API modeling language. Section 5 explains the API model guided symbolic execution as implemented in the PROMPT tool. Section 6 presents the details of our case studies. Section 7 presents an evaluation of our approach. Finally, Section 8 concludes with directions for future work.

2 Related Work

Environment modeling Analysis of real-world applications requires the existence of an environment model. In the context of symbolic execution, modeling of low-level system call API [13] and modeling of POSIX API [12] have been considered to enable analysis of code that uses these API. However, these

[1] PROMPT will be released as an open source project and the github link will be provided in the camera-ready version. We will also publicly release the device driver benchmarks and their models.

efforts provide predefined models and do not provide the users with mechanisms to use custom defined models. DART [21] analyzes the component under test to automatically extract external function models that return random values of the correct return type, which get combined with the component for concolic execution. In the context of model checking, environment models of Windows Framework Drivers [9], Linux device drivers [37], the Java Swing Library [26], and the Android OS [8] have been used. In [9] the API functions were modeled through stubs that nondeterministically return possible return values or allocate memory and return the address. However, semantics of the API functions were also specified using rules written in the SLIC specification language [10]. In [37], the environment models are encoded in the C code and enriched with nondeterministic choice directives. In [26], event sequences are specified in a user script, which get simulated by model Java implementations of some of the framework classes. In [8], mock-ups of OS functionalities are implemented in Java by leveraging the original Android implementation. The LDV toolset [41] enables specification of environment models by weaving the correctness rules and the API models expressed in the C language to the source code of the component under analysis and uses a reachability checker under the hood such as [11] to detect violations and memory errors. *PROMPT allows users to specify custom API models and enables symbolic execution of the component under analysis within the context of these models while simulating the custom data constraints and control-flow rules specified as part of the API model.*

Symbolic Execution for System Code S2E [18] uses selective symbolic execution to analyze binaries. S2E can be directed to restrict symbolic execution to the specific parts of the code and manage the transitions between symbolic and concrete execution modes. While S2E targets errors that may get manifested in any layers of the software stack, *PROMPT targets errors inside the component under analysis while using a model for the environment.* Since S2E mixes concrete and symbolic execution, it needs hardware emulation to analyze device drivers or any component that directly interacts with the hardware. However, PROMPT is based on symbolic execution only and, therefore, does not require hardware emulation. SymDrive [33] uses S2E and symbolic hardware models and analyzes driver code that is instrumented with checkers. SymDrive is specialized for driver analysis whereas PROMPT provides a generic modeling framework for analyzing components of system code. Apisan [40] 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. Due to the precise memory model used by the underlying symbolic execution engine, *PROMPT can check whether such deviating behaviors are correct or not without a belief model as long as it is provided with sufficiently precise API models.*

Lazy Initialization Lazy initialization for symbolic execution has been presented in [24] and was implemented as an extension to the Java Path Finder model checking tool [35]. In this work, the motivation for lazy initialization was to represent unbounded data structures symbolically and to check concurrent data structure implementations in an exhaustive way. Therefore, when a symbolic field needed to be concretized all possible candidates among the existing compatible memory objects and the null value would be considered non-deterministically. Lazy initialization capability was added to the KLEE symbolic execution engine in UC-KLEE [31] to reach deep parts of the system code and libraries. The modeling constructs provided by UC-KLEE get weaved into the analyzed code with the goal of filtering out some of the false positives. *PROMPT enables customization of the analysis environment through API model guided lazy initialization and symbolic execution, which achieves scalability in addition to achieving a lower false positive rate.*

3 Overview

In this section, we present an overview of our API modeling approach and introduce its salient features. The overall goal is to be able to analyze implementations of software components independently from the implementations of the APIs they interact with. However, users should also be able to specify a model of such API based on their analysis goals. This would have the benefit of reducing the footprint of the analyzed code and the number of paths explored. The users can design the models based on domain expertise and a specific goal, e.g., detecting memory related errors.

Another goal is usability of the approach. We would like to minimize the manual effort that would be needed for API modeling, which needs to be done once for every version of the software framework [2]. We achieve this by eliminating the need to write a test harness for the component to be analyzed using the lazy initialization approach [24, 31]. Lazy initialization eliminates the need for creating and initializing dynamic data structures. Plain lazy initialization has been shown to be effective for the analysis of abstract data type implementations [24]. However, it leads to a high false positive rate for software frameworks, such as the Linux kernel [31]. This is due to the inability to capture the rules of the underlying API.

We have designed PROMPT to perform API model guided symbolic execution on a software component and the API models it interacts with. PROMPT gets the code for the software component under analysis and an API model specified in the PROSE modeling language. A PROSE model consists of the C implementation of the modeled API functions and the API models, which fall into one or more of the three categories: data models, function models, and life-cycle models.

[2] Models of the API functions can be reused across different versions as long as the modeled aspects do not change.

3.1 A Motivating Example

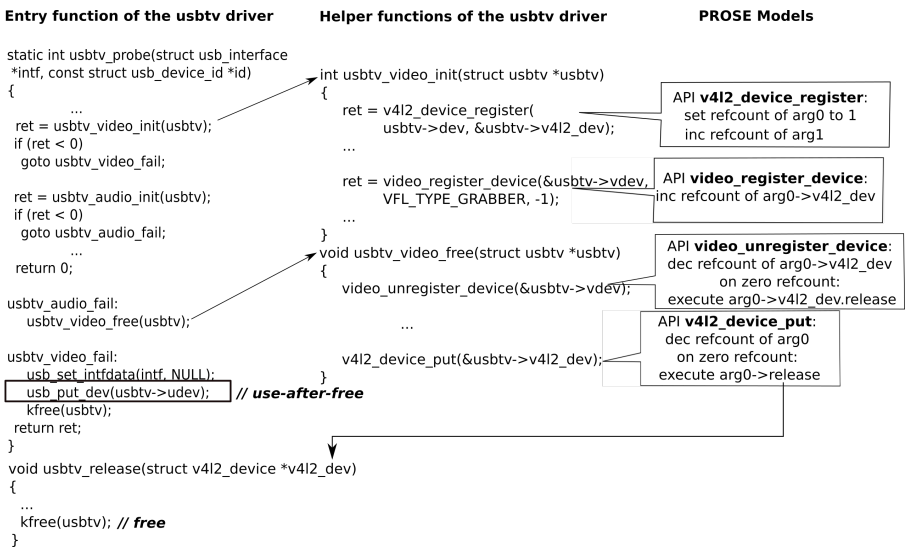


Fig. 1: The usbtv driver (the component under analysis, on the left and on the middle) and the summarized PROSE API models (on the right). The use-after-free vulnerability can be detected as PROMPT precisely simulates the environment for the usbtv driver using the video API models and other rules of the API modeling.

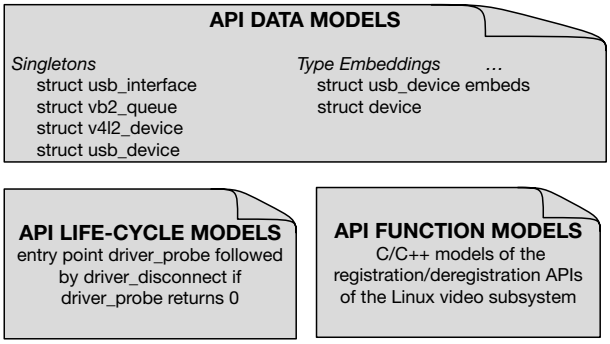


Fig. 2: The components of a PROSE API model specification as defined for the Linux drivers from the video subsystem.

In this section, we motivate the need for modeling the environment of a software component for a precise and scalable analysis using symbolic execu-

```
int v4l2_device_register_PROSE(struct device *dev,
          struct v4l2_device *v4l2_dev)
{
   int dev_refcount;
   int return_value;

   klee_make_symbolic(&return_value,
          sizeof(return_value),
          "v4l2_device_register_return_value");

   if (return_value >=0 ) {

      // kref_init(&v4l2_dev->ref);
      klee_set_metadata(v4l2_dev, 1);

      // get_device(dev);
      dev_refcount = klee_get_metadata(dev);
      klee_set_metadata(dev, ++dev_refcount);

      v4l2_dev->dev = dev;

      // if (!dev_get_drvdata(dev))
      if (!dev->driver_data)
        // dev_set_drvdata(dev, v4l2_dev);
        dev->driver_data = v4l2_dev;
   }
   return return_value;
}
```

Fig. 3: The C implementation of the PROSE model for the `v4l2_device_register` function. Commented lines denote the original program statements that are modeled. The metadata handling functions `klee_set_metadata` and `klee_get_metadata` are PROSE extensions to KLEE.

tion. Although our approach can be applied to a variety of kernel components, e.g., protocol stack implementations, and to other software frameworks, we use a Linux device driver with a known use-after-free vulnerability as a running example. Figure 1 shows a function from the `usbtv` driver, a Linux driver for a USB tuner, which is used as the entry point for symbolic execution.

Figure 2 shows some of the components of the API model specification for the `usbtv` driver. A Linux device driver that supports the hot-plug events implements a function, `probe`, to be called when the device gets plugged in and implements another function, `disconnect`, to be called when the device gets unplugged. According to the life-cycle rules, the disconnect function executes after the probe function and only if the probe function returns a success value. According to the data constraint rules, some of the data structures, such as `struct usb_interface`, have only one instance. As an example, for USB

drivers the single instance of the `struct usb_interface` type gets passed as a parameter to both the probe and the disconnect functions. [3]

Another data constraint rule involves specifying embedding of one data structure type in another one. In the Linux kernel, polymorphism is achieved by embedding generic data structures, e.g., `struct device`, within specialized data structures, e.g., `usb_device`. Also, the address of the generic data structure instance is used to compute the address of the specialized data structure. This is implemented by the `container_of` macro and involves pointer arithmetic. Basically, the address of a specialized data structure instance is computed by subtracting the static offset of an embedded data structure from the address of an embedded data structure instance. Note that this type of pointer arithmetic that goes outside the boundaries of a data object is not supported by standard points-to analyses [39], which led to the design of specialized techniques such as [17, 36]. Symbolic execution has a precise memory model and can deal with such non-standard pointer arithmetic as long as it is provided with sufficient context, e.g., the embedding object has also been created. However, in component-level analysis such context may not be available. *PROSE allows specification of such data structure dependencies and PROMPT applies them during lazy initialization to provide proper context during component-level analysis.* Such pointer arithmetic exists in low-level system software, e.g., in dynamic data structure implementations [17], and, hence, we expect the embedding rule to be effective in the analysis of system code other than the Linux kernel as well.

In Figure 1, the boxes on the right-hand side show summarized PROSE models of the API functions that called from the driver code, e.g., `usbtv_video_init` calls the `v4l2_device_register` and the `video_register_device` API functions. Figure 3 shows the C implementation of the PROSE model for `v4l2_device_register`. This driver interacts with the `video` subsystem of the Linux kernel. The `usbtv_probe` function is the entry function and it gets executed when the USB device gets plugged in. The driver calls some video API functions inside the `usbtv_video_init` function of the driver to setup video related data structures and to register them with the kernel. Among other things, some of these API functions keep track of the reference counts of video data structures. In the Linux kernel, two different APIs are used for reference counting: `kref` and `kobject`. These APIs differ in how they specify the cleanup function. We abstract away the details of these two different APIs by providing a metadata tracking and update mechanism in PROMPT. Basically, we implemented two handlers: `klee_set_metadata` and `klee_get_metadata`. The former function sets the metadata for the first argument, which represents an address in the analyzed component, and the latter returns the existing metadata. The metadata API keeps the mapping between the address and the metadata specific to the current symbolic execution state in which the call gets executed. Currently, we support `int` as a type for the metadata and we will extend it with additional types in the future.

[3] This singleton rule applies to other bus types including the PCI and I2C.

Using the metadata interface we mention above, we could abstract the reference counting APIs for the `v4l2_device_register` PROSE model. For instance, as shown in Figure 3, the `v4l2_device_register` function sets the reference count of a `v4l2_device` type object to 1 at line 14. Line 13 shows the commented out original program statement that achieves this operation using the `kref` API. Lines 17-18 show the reference count increment operation for the `device` type object. Line 16 shows the commented out original program statement that achieves this operation using the `kobject` API.

PROMPT detected this known use-after-free vulnerability, CVE-2017-17975, in the `usbtv` driver. The bug gets manifested on an error path and the use-after-free happens due to accessing the driver data structure pointed by the `usbtv` variable, denoted by the rectangle shown in Figure 1, after freeing of the object. The error path is executed due to the `usbtv_audio_init` function returning an error value. So, the `usbtv_video_free` function gets executed next. Inside this function, the reference count of the `v4l2_device` object is decremented as indicated by the PROSE models of the `video_unregister_device` and `v4l2_device_put` functions. When the reference count becomes zero, a callback function gets called according to the model. The address of this callback function is represented by the expression `arg0->release` and it turns out to be the `usbtv_release` function of the driver, which gets registered/set inside the `usbtv_video_init` function of the driver. The callback frees the driver data structure, `usbtv`, using the `kfree` function. Eventually, the control-flow on the error path moves to the `usb_put_dev` callsite, at which point the use-after-free is detected by PROMPT.

We note that the actual PROSE model we used for the `video API` to detect this vulnerability is more detailed than the one shown in Figure 1[4]. Without modeling, i.e., using the full kernel code, initialization of the global state took approximately two hours while using a 32GB of memory on an Ubuntu 16.04 machine with 256GB of RAM (see Section 7.2). However, using the PROSE models and the driver code as system under analysis, PROMPT could detect the use-after-free vulnerability shown in Figure 1 within 6 seconds as reported in Section 7.6.

4 The PROSE API Modeling Language

Figure 4 shows the grammar of the PROSE API modeling language. One of the goals of PROSE is to minimize modeling effort by providing a way to specify rules that can be applied globally. However, this may lead to an imprecise analysis. So, users can also specify rules that involve specific data types and functions. In that case, type and function specific modeling rules take precedence over the globally specified modeling rules. Below, we explain major components of the PROSE API modeling language.

[4] The PROSE models of our benchmarks will be released along with the PROMPT tool.

⟨*API model*⟩ ::= 'Global Settings:' ⟨*global settings*⟩ 'Data Models:' ⟨*data models*⟩
| 'Function Models:' ⟨*function models*⟩ 'Life-cycle Model:' ⟨*life-cycle models*⟩

⟨*global settings*⟩ ::= ⟨*global setting*⟩ ; ⟨*global settings*⟩ | ⟨*global setting*⟩

⟨*global setting*⟩ ::= 'array size' ⟨*number*⟩ | 'NULL return' ⟨*choice*⟩ | 'init funcptrs to NULL' ⟨*choice*⟩ | 'symbolize inline asm' ⟨*choice*⟩ | 'model funcs with asm' ⟨*choice*⟩ ['except' ⟨*regexpList*⟩] | 'skip havocing singletons' ⟨*choice*⟩

⟨*choice*⟩ ::= 'ON' | 'OFF'

⟨*data models*⟩ ::= ⟨*data model*⟩ ; ⟨*data models*⟩ | ⟨*data model*⟩

⟨*data model*⟩ ::= ⟨*type embedding*⟩ | 'singleton' ⟨*ident*⟩ | ⟨*bound constraint*⟩

⟨*bound constraint*⟩ ::= ⟨*expression*⟩ 'where' ⟨*bindings*⟩

⟨*expression*⟩ ::= constant | ⟨*expression*⟩ binaryOp ⟨*expression*⟩ | ⟨*unaryOp*⟩ ⟨*expression*⟩

⟨*bindings*⟩ ::= ⟨*binding*⟩ , ⟨*bindings*⟩ | ⟨*binding*⟩

⟨*binding*⟩ ::= ⟨*ident*⟩ 'is' ⟨*entity*⟩

⟨*entity*⟩ ::= ⟨*ident*⟩ 'field' ⟨*number*⟩ | 'size_of' ⟨*ident*⟩ 'field' ⟨*number*⟩ | ⟨*ident*⟩ 'arg' ⟨*number*⟩ | 'return_of' ⟨*ident*⟩

⟨*type embedding*⟩ ::= ⟨*ident_1*⟩ 'embeds' ⟨*ident_2*⟩ ['field' ⟨*number*⟩]

⟨*function models*⟩ ::= ⟨*function model*⟩ ; ⟨*function models*⟩ | ⟨*function model*⟩

⟨*function model*⟩ ::= ⟨*ident_1*⟩ 'modeled by' ⟨*ident_2*⟩ | 'returnOnly' ⟨*ident*⟩
| ⟨*bound constraint*⟩ | 'havoc args' ⟨*number_list*⟩ 'of' ⟨*ident*⟩
| 'alloc' ⟨*ident*⟩ ⟨*alloc_mem_pos*⟩ 'size_arg' ⟨*number*⟩ 'init_zero' ⟨*bool*⟩ 'symbolize' ⟨*bool*⟩ | 'free' ⟨*ident*⟩ 'mem_arg' ⟨*number*⟩

⟨*alloc_mem_pos*⟩ ::= 'mem_arg' ⟨*number*⟩ | 'mem_return'

⟨*life-cycle models*⟩ ::= ⟨*life-cycle sequence*⟩ | 'entry-point' ⟨*ident*⟩

⟨*life-cycle sequence*⟩ ::= ⟨*life-cycle entry*⟩ ; ⟨*life-cycle sequence*⟩ | ⟨*life-cycle entry*⟩

⟨*life-cycle entry*⟩ ::= ⟨*ident*⟩ ['continue if ' ⟨*bound constraint*⟩] | ⟨*ident*⟩ '[' ⟨*number*⟩ ']' | ⟨*ident*⟩

Fig. 4: The grammar of the PROSE API modeling language that gets interpreted by the PROMPT tool to simulate an environment model for the component under analysis.

4.1 Global Settings

In lazy initialization, an important modeling decision involves the size of memory region to be allocated for a pointer. It is possible that the pointer refers to a single object or an array of objects. So, in general an array of the base type with size greater than or equal to one needs to be allocated. If the array size is too small, this would lead to false positives. On the other hand, if the array size is too large, it would lead to false negatives and to high memory overhead during symbolic execution. With these caveats, the user can specify an array size to be applied to any pointer that gets lazily initialized using the `array size <number>` rule, where `<number>` denotes a positive integer. Users can overwrite this rule for a specific type as explained in Section 4.2.

In PROSE, a function can be modeled either via another function that matches the signature and is implemented in C or by specifying how to handle the arguments and the return value, which we call the *generic approach*. If the function is modeled using the generic approach, the return value is automatically symbolized. However, if a modeled function returns a pointer-type, the users can specify whether the `NULL` value should be considered as a possible return value by using the `NULL return <choice>` rule, where `<choice>` can be `ON` or `OFF`. Users can overwrite this rule for specific functions as explained in Section 4.2.

Another important aspect of lazy initialization of data structures is how to deal with the function pointers. In some cases, the user needs to specify which function to use for a specific field of a type. However, in some cases setting a function pointer field to `NULL` works if it is about an optional functionality that can be abstracted away. So, users can specify whether the function pointers in lazily initialized data structures can be set to `NULL` using the `init funcptrs to NULL` rule. Users can overwrite this rule for a specific type as explained in Section 4.2.

System code often come with inline assembly code. Although users can deal with inline assembly via automated assembly lifters such as [32], PROSE provides two ways to model inline assembly, if needed. The first way is to model the assembly instruction as a side-effect free operation and symbolizing the return value, if any, using the `symbolize inline asm ON` rule. The other way is to automatically modeling functions that have inline assembly using the `model funcs with asm <choice>` rule. The optional `except <regexpList>` specifies exceptions to this setting for functions with names matching the list of regular expressions `<regexpList>`. The details of modeling such functions are subject to other global and type and function specific rules as specified in the PROSE API model.

Users can also specify a global approach to dealing with the pointer arguments of functions that are modeled using the generic approach. As mentioned above and detailed in Section 4.3, such arguments can be chosen to be havoced (symbolized) to model the side-effect of the function in the most abstract sense. Havocing process marks the object pointed by the argument as symbolic. However, users can choose to skip or turn on this havocing operation

for functions modeled using the generic approach with the `skip havocing singletons <choice>` rule, which can be overwritten for specific functions and specific arguments as explained in Section 4.3.

4.2 Data Modeling

An important aspect of providing a precise context for a component under analysis is describing the embedding relationship between data structures. System software utilizes the embedding of one struct in another one to achieve polymorphism and code reuse [17]. This assumption may be leveraged to use the address of an embedded object to derive a pointer for the embedding object, e.g., the `container_of` macro in the Linux kernel. If a proper context is not provided, analysis of such code may lead to false positives and low coverage of the code. PROSE allows users to specify such embedding relationship using the `<ident_1> embeds <ident_2>` rule, where `<ident_1>` denotes the name of the embedding type and `<ident_2>` denotes the name of the embedded type. Optionally, users can specify the index of the field of `<ident_1>` at which `<ident_2>` is embedded using the extension `field <number>`. When the field index is not specified and if there are multiple fields of `<ident_1>` of type `<ident_2>` then the one with the lowest index is assumed for lazy initialization purposes.

Another type of data modeling has to do with whether the lazy initialized struct type has a single instance within the context of the analyzed components. If so, using the same instance instead of creating a new instance improves the precision of the analysis. Users can specify whether a data type should be treated as a singleton using the `singleton <ident>` rule, where `<ident>` is the type name.

Finally, users can specify constraints about the fields of struct types. Such constraints consist of two parts: the expression and the binding. In the expression part, users can use model variables in constraints that involve arithmetic (`+, -, *,/`), boolean (`!`), and relational operators ($>, \geq, <, \leq, =, !=$). These model variables need to be bound to some entities in the code using comma separated `<ident> is <entity>` clauses. The types of entities include the fields of `struct` types (`<ident> field <number>`), the arguments (`<ident> arg <number>`) or the return values (`return_of <ident>`) of functions, and the sizes of arrays or pointer fields (`size_of <ident> field <number>`). The field and argument numbers start at 0 and consecutive numbers are used based on the order of their declaration in the function signature or in the type definition. These constraints get enforced on the relevant objects during lazy initialization, handling of callsites for functions modeled using the generic approach, or handling of return instructions that cause transitioning from one life-cycle entry to another.

4.3 Function Modeling

When a software component is analyzed, it is important to capture the key interactions with some of the API functions that are critical for achieving the analysis goal. Some of the API functions, on the other hand, will not be that critical. So, PROSE provides various ways of modeling an API function with different levels of detail.

One way is to implement the model as a C function and specify the modeling relationship with the original one using the `<ident_1> modeled by <ident_2>` rule, where `<ident_1>` and `<ident_2>` denote the names of the original function and the model function, respectively.

Another way of function modeling in PROSE is by specifying the side effects of the function in terms of its return value and the pointer type arguments, which we call the generic approach. Return values of functions modeled in this way are always symbolized. However, the return value can be constrained as explained in Section 4.2. The pointer arguments can be explicitly specified to be havoced by providing the indices of the arguments using the rule `havoc args <number_list> of <ident>`. Another option in the generic approach is to symbolize the return value only and skip havocing of all pointer arguments, which can be specified using the `'returnOnly' <ident>`.

An important class of API functions involve memory allocation and deallocation. Such API can be modeled in PROSE to abstract away framework specific details. An allocation function may either return a pointer to the allocated memory or store the address in a pointer argument. Users can use the `alloc <ident> <alloc_mem_pos> size_arg <number> init_zero <bool> symbolize <bool>` to specify how the address of the allocated memory is returned (`<alloc_mem_pos>`), the argument index (`<number>`) that specifies the size of the memory allocation, whether the allocated memory would be initialized with zeros, and whether the memory would be symbolized. To model a deallocation function, one needs to specify the argument that holds the address of the memory region to be deallocated using the `free <ident> mem_arg <number>` rule, where `<number>` denotes the argument index.

4.4 Life-cycle Modeling

In a PROSE model, the components under analysis can be specified using a life-cycle model. If there is only one component to be analyzed then this can be specified using the `entry-point <ident>` rule, where `<ident>` denotes the name of the function to analyze. If there are multiple functions to analyze in a sequential order then the sequential composition can be specified as a semicolon separated list of life-cycle entries. A life-cycle entry can be just a function name, which means that either the return type of the function is void or that regardless of the return value the execution will continue with the next function in the sequence. If the execution should continue only for certain cases of return values, e.g., success cases, then the `continue if <bound`

`constraint>` clause should be specified after the function name. The bound constraints that are valid in this context are those that involve the return values of the functions (`return_of <ident>`) in the life-cycle sequence. Alternatively, users can choose to use the `<ident> [ <number> ]` rule to specify the specific return value, `<number>`, for which the execution continues. Finally, just specifying the name of the function `<ident>` indicates that the execution must continue with the next life-cycle step regardless of the return value, if any, as long as the path has not terminated.

5 API Model Guided Symbolic Execution

In this section, we explain the details of our approach for API model guided symbolic execution as implemented in the PROMPT tool. PROMPT extends the KLEE symbolic execution engine, which follows an Execution-Generated Testing (EGT) approach to dynamic symbolic execution [14]. Algorithm 1 shows how the basic EGT approach to symbolic execution works. Starting from an initial state that consists of the dynamic and static memory, *Mem*, the stack, *Stack*, the path condition, PC, the next instruction to execute, *nextInst*, and the termination status, *term*, it generates a tree of states as the component under analysis, C, gets executed symbolically. By designating some of the inputs as symbolic, which is invisible in Algorithm 1, the program instructions get executed by computing symbolic expressions when the operands involve symbolic values. So, both *Mem* and *Stack* have a combination of memory locations with concrete values and memory locations with symbolic expressions. The PC, known as the path condition, represents the constraints on the symbolic inputs on a given execution path. Computation of the PC in Algorithm 1 is implicitly handled by line 8. Each branch instruction generates the children of the current state such that the path condition of each child, PC_i, restricts the path condition of the parent with the symbolic branch condition, *cond*, that is true on that path, i.e., $PC_i \equiv PC \ \wedge \ cond$. For non-branching instructions, the set *succs* is empty if *inst* is the last instruction of C, in which case the *state* gets terminated. For non-terminating non-branching instructions, the set *succs* is a singleton and the PC is an exact copy of that of *state*, i.e., the predecessor.

Algorithm 1 Basic symbolic execution.

```
1: BSE(C: Component, τ: time out)
2: Let (Mem, Stack, PC, nextInst, term) represent a symbolic execution state
3: Let S_0 represent the initial state
4: Let states ← {S_0}
5: while states not empty and timeout τ not reached do
6:     state ← Choose(states)
7:     inst ← state.nextInst
8:     succs ← ExecuteInstruction(state, inst)
9:     states ← states ∪ succs
10: end while
```

Basic symbolic execution (BSE) does not scale when the component under analysis is large. So, we present an A**PI** **M**odel **G**uided **S**ymbolic **E**xecution (PMGSE) to enable the analysis of software components of large frameworks. The challenge in analyzing a software component is to come up with a precise model of the environment it interacts with so that symbolic execution can be performed on the software component and the model only, which would scale better than analyzing the component in the context of the large software framework.

We assume that depending on the goal of the analysis the framework API models have been implemented in C and get linked with the component. However, replacing the API functions with their model implementations does not provide a precise analysis. The rules of the API also need to be enforced. Otherwise, either some of the important paths will not get explored or too many false positives will be generated. In Section 4, we have presented the API modeling rules as defined in the PROSE language. In this section, we explain how the rules that are specified in a PROSE model get enforced by PMGSE as implemented in the PROMPT tool.

Algorithm 2 shows how PMGSE extends the basic symbolic execution given in Figure 1 to enforce a given API model, M, which consists of the global settings, the singleton types, SG, the type embedding relations, EM, the data constraints, DC, to be used during lazy initialization, the function argument, FAC, and return value, FRC, constraints to be used to simulate the life-cycle rules, LC, the modeled allocation, AF, and deallocation, DF, functions, the functions that are modeled by other functions, FM, the exceptional cases, AE, for modeling functions with inline assembly, and the rules about havocing arguments, HV. We refer to the relevant elements of the API model M in Algorithms 2-8 to explain how PROMPT implements PMGSE.

PMGSE extends the symbolic execution state with metadata, MT, that gets manipulated by the API models. The main extensions of Algorithm 2 to Algorithm 1 are 1) extending the state representation with metadata, 2) collecting type embedding information by traversing all the types in the component under analysis, 3) keeping track of a type to address mapping, TA, and 4) extending the handling of several instruction types to enforce the API model. We skipped some details about keeping track of TA such as including static allocations and bitcast instructions that cast `void` pointers to a specific type. In what follows, we use *executeInstruction* and *allocate* to represent the algorithms for symbolically executing an instruction and allocating memory for a given type, respectively, as performed in BSE. We mark all `struct` types and all primitive types that are used as pointer fields in the `struct` types to be lazily initialized and do so if a pointer to one of these types actually take a symbolic address.

Algorithm 3 handles the load and store instructions. Since KLEE executes LLVM bitcode, we would like to provide some details in the context of LLVM. In LLVM, a load/store instruction may have a pointer type or a double pointer type address operand. Those with the latter refer to the memory locations that store memory addresses. So, if a symbolic expression is stored in the address

Algorithm 2 API model guided symbolic execution as implemented in the PROMPT tool.

PMGSE(C: Component, M: API Model, τ: time out)
Let $M = (SG, EM, DC, FAC, FRC, LC, AF, DF, FM, AE, HV)$
Let $(Mem, Stack, PC, nextInst, MT, TA)$ represent an extended symbolic execution state
Let S_0 represent the initial state
Let *entry* denote a function in C specified as the first life-cycle entry in $M.LC$
$S_0.Stack.push(entry)$
ApplyFunctionConstraint(st, f, arg, $false$, $entry.firstInst$, M)
if $s.term = false$ **then** ▷ Consistent data constraints on arguments
    $S_0.nextInst \leftarrow entry.firstInst$ ▷ Start the execution from the first function in $M.LC$
    Make $entry.args$ symbolic in S_0
    Collect embedding info into **global** ER s.t. $(t_1, t_2) \in ER$ iff t_1 embeds t_2
    Let $states \leftarrow \{S_0\}$
    while $states$ not empty and timeout τ not reached **do**
        $state \leftarrow Choose(states)$
        $inst \leftarrow state.nextInst$
        if $inst$ is a load or store **then**
            $succs \leftarrow HandleLoadStore(state, inst, M)$
        else if $inst \equiv call\ f()$ **then**
            $succs \leftarrow HandleCall(C, state, inst, M)$
        else if $inst \equiv return\ value$ **then**
            $succs \leftarrow HandleReturn(state, inst, M)$
        else
            $succs \leftarrow ExecuteInstruction(state, inst)$
        end if
        $states \leftarrow states \cup succs$
    end while
end if

provided in a load/store instruction and the operand is a double pointer type, we allocate the memory object of that type and store the address of that allocated memory into the relevant memory location. We represent an allocated memory region with a memory object $mo = (A, T, C)$, where A denotes the base address, T denotes the base type, and C denotes the number of T objects stored in mo, i.e., size of mo is $C \times sizeof(T)$. So, given a state st and a possibly symbolic address A, $Resolve(st, A)$ returns the set of memory objects that A may fall into by forking at the instruction i and generating a copy of state st in each st'. For each memory object mo, the algorithm checks if the address value stored at address A is a symbolic value. If so, it calls Algorithm 4 to lazily create an object and copies the address of the object, $address_2$, to address A. The size of the lazily initialized memory region is inferred from the API model; if there is a type specific rule then the specified constant is retrieved from the data constraint, $M.DC$, or from the singletons $M.SG$, otherwise, the global setting on the array size is used.

After the memory gets created, PROMPT applies any data constraint that is related to the size of the generated memory object by executing the **ApplyDataConstraint** algorithm, which is explained below. If the applied constraints contradict the path condition then the path gets terminated. Otherwise, the load/store instruction is executed as in regular basic symbolic ex-

ecution, which would now use a concrete address, $address_2$, to execute the load/store instruction. Memory allocation is shown in Algorithm 4 and performs additional tracking for lazy initialization.

Algorithm 3 Special handling of the *load* and *store* instructions. Symbolic addresses trigger lazy initialization.

```
1: HandleLoadStore(st: State, i: Instruction, M: API Model): P(State)
2: result ← ∅
3: Address A ← AddressOperand(i)
4: P(MemoryObject, State) set ← Resolve(st, A), where MemoryObject = (Address, Type, N)
5: for each (mo, st') ∈ set s.t. mo = (address_1, T_1, size) do
6:     if A is within bounds of mo then
7:         lazyinit ← false
8:         if i has a double pointer address operand then
9:             value ← st'.Mem[A] ; lazyinit ← true
10:            Let T_2 denote the non-pointer base type of i's address operand
11:            if value is a symbolic expression then ▷ Lazy initialize the symbolic address
12:                count ← InferArraySize(T_1, T_2, M)
13:                (address_2, mo') ← HandleAllocate(st', T_2, true, count, M)   ▷ Allocate memory of base type
14:                Let k s.t. k * size(T_1) ≤ A − address_1 < (k + 1) * size(T_1)
15:                ApplyDataConstraint(st', T_1, address_1 + k * size(T_1), true, A, M.arraySize, M)
16:                if st'.term = false then
17:                    st'.Mem[A] ← address_2                    ▷ Initialize the pointer
18:                    PerformLoadStore(st', i)                  ▷ Now do the actual load/store
19:                    result ← result ∪ {st'}
20:                end if
21:            end if
22:        end if
23:        if lazyinit = false then
24:            PerformLoadStore(st', i)
25:            result ← result ∪ {st'}
26:        end if
27:    end if
28: end for
29: if set = ∅ then
30:    st.term ← true                                            ▷ Report memory error
31:    return ∅
32: else
33:    for each st' ∈ set \ result do
34:        st'.term ← true                                       ▷ Report memory error
35:    end for
36: end if
37: return result
```

Algorithm 4 performs some extra steps if it is called as part of the lazy initialization process, e..g, from the load/store instructions. Since it needs to enforce the embedding rules, it executes recursively to make sure that the outermost embedding object gets created as part of the allocation of the given type T. One side effect of this algorithm is to infer new singletons; if a type is specified to be a singleton then so must be its embedding type. When we

reach the base case of this recursive algorithm, i.e., there is no embedding type to consider according to the specified API rule, $M.EM$, we check if this is a singleton. If so, we should use the existing instance, if any. Otherwise, a new instance gets created. In our implementation, we create an array instead of a single cell when we lazily initialize pointers to primitive or non-singleton types. The size of the array can be tuned. We have used 20 for our evaluation.

Algorithm 4 Special handling of memory allocation.

1: **HandleAllocate**(st: State, T: Type, $lazy$: bool, $count$: $\mathcal{N}$, M: API Model) : ($Address$, $MemoryObject$)
2: **if** $lazy = true$ **then** ▷ to be lazily initialized
3: **if** exists some T' s.t. $(T', T) \in ER$ and $(T', f, T) \in M.EM$ **then**
4: **if** $T \in M.SG$ **then**
5: $M.SG \leftarrow M.SG \cup \{T'\}$ ▷ Infer embedding type being a singleton
6: **end if**
7: $(ea, mo) \leftarrow HandleAllocate(st, T', true, count, M)$
8: **return** $(ea + StaticOffset(T', f), mo)$;
9: **else**
10: **if** $T \notin M.SG$ or $st.TA[T] = \perp$ **then**
11: $address \leftarrow allocate(size(T) \times count)$
12: $mo \leftarrow (address, T, count)$
13: $st.TA[T] \leftarrow mo$
14: $makeSymbolic(address, size(T) \times count)$
15: **for** i 0 to $count - 1$ **do**
16: **ApplyDataConstraint**(st, T, $address + i \times size(T)$, $false$, $undef$, $undef$, M)
17: **end for**
18: **return** $(address, mo)$
19: **else**
20: Let $st.TA[T] = mo$
21: **return** $(mo.baseAddress, mo)$
22: **end if**
23: **end if**
24: **else**
25: $address \leftarrow allocate(size(T) \times count)$
26: $st.TA[T] \leftarrow (address, T, count)$
27: **return** $(address, mo)$
28: **end if**

Algorithm 5 applies data constraints in the API model, $M.DC$. If *sizeof* parameter is set to true then this algorithm gets called from Algorithm 3 to apply constraints on the size of a dynamic array, e.g., `x < y, where x is A field 0, y is size_of A field 1`, where field 0 represents the length and field 1 represents the dynamic array in `struct A = {int length; int *data}`. The algorithm finds the relevant constraint by locating the relevant field based on the base address b of the object of type T and the address a of the dynamic array field. If *sizeof* parameter is set to false then this algorithm gets called from Algorithm 4 to make sure that primitive fields of the lazy initialized object of type T gets constrained as specified. In both cases, the abstract syntax trees that corresponds to the specified constraints get translated into expressions on the symbolic names of the relevant memory regions.

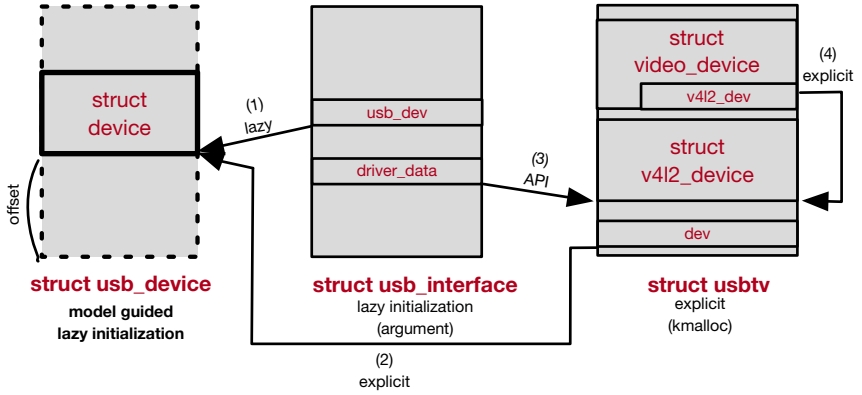


Fig. 5: Major data structures manipulated by the `usbtv` driver. The region with a bold border shows an example of an embedded region, where the embedding type is `usb_device` (with the dashed border), which gets created to enforce the modeling rule on type embedding. The labels on the arrows represent the source of the updates and those under the objects show how they get created.

The symbolic expressions are checked against the path condition and the state gets terminated if a contradiction is detected. The algorithm is executed recursively to handle the struct type fields of T to apply the constraints that involve any of the embedded types.

Figure 5 illustrates some of the data structures used by the `usbtv` driver given in Figure 1 and how they get created. We use the **explicit** label (positioned below the objects) to represent memory allocation due to an allocation callsite such as calling `kmalloc` or `kzalloc` and we use `lazy` to denote that the object gets created as part of the lazy initialization. As an example, the `usbtv` driver creates a `struct usbtv` object inside the `usbtv_probe` function by calling `kzalloc` and, therefore, this objects get created explicitly during symbolic execution. However, both the `struct usb_interface` type and the `struct usb_device` type objects get lazily initialized. The lazy initialization of the former is due to being a parameter of some entry function, e.g., `usbtv_probe` or `usbtv_disconnect`, whereas that of the latter is due to an access of the `usb_dev` field on the lazily initialized `struct usb_interface` object, denoted by the **lazy** label on arrow (1) from the `usb_dev` field to the `usb_device` object.

The reason for the `device` object being created as embedded inside a `usb_device` object instead of as being a standalone object is that the API rule on embedding was specified as `usb_device` embeds `device`. It is possible that some symbolic pointers get later in the execution set to concrete values, examples of which are shown in Figure 5 and are denoted by the arrows labelled with either **explicit** or **API**. The **explicit** label indicates that the pointer

Algorithm 5 Application of data constraints to lazily created objects.

1: **ApplyDataConstraint**(st: State, t: Type, b: Address, $sizeof$: bool, a: Address, $size$: $\mathbb{N}$, M: API Model)
2: **for** each $(exp, binding) \in M.DC$ s.t. $\exists\ v.f.$ s.t. $binding[v] = (t, f)$ or $binding[v] = size\_of(t, f)$ **do**
3: **if** $sizeof$ is true **then**
4: **if** $binding$ does not have a size_of clause **then**
5: continue
6: **else if** there exists a size_of clause (t, f) in $binding$ s.t. $StaticOffset(t, f) \neq a - b$ **then**
7: continue
8: **end if**
9: **else if** $binding$ has a size_of clause **then**
10: continue
11: **end if**
12: **if** $exp \equiv v = constant$, where $binding[v] = (t, f)$ **then**
13: $st.Mem[b + StaticOffset(t, f)] \leftarrow constant$
14: **else**
15: Let V denote the variables $v_1, v_2, ..., v_n$ in exp
16: Let $V' = v'_1, v'_2, ..., v'_n$
17: **for** each v_i **do**
18: **if** $binding[v_i] \equiv (t, f)$ **then**
19: $v'_i \leftarrow st.Mem[b + StaticOffset(t, f)]$
20: **else if** $binding[v_i] \equiv size\_of(t, f)$ **then**
21: $v'_i \leftarrow size$
22: **end if**
23: Let $exp' \leftarrow exp[V/V']$
24: **if** $st.PC \ \wedge \ exp'$ is SAT **then** ▷ Is the data constraint consistent with the path constraint?
25: $st.PC \leftarrow st.PC \wedge exp'$ ▷ Apply the data constraint to the state
26: **else**
27: $st.term \leftarrow true$ ▷ Terminate the state
28: **return**
29: **end if**
30: **end for**
31: **end if**
32: **end for**
33: **for** each field f of t that is a struct type **do** ▷ Handle embedded types
34: $b' \leftarrow b + StaticOffset(t, f)$
35: **ApplyDataConstraint**(st, $Type(t, f)$, b',$sizeof$,a,$size$,M)
36: **end for**

has been set due to an assignment instruction in the component whereas the **API** label indicates the pointer having been set inside the API model. As an example, the link denoted by arrow (3) in Figure 5 is generated due to line 25 of the `v4l2_device_register` function model shown in Figure 3.

In Algorithm 6, we handle certain callsites in a special way to 1) track and check consistency of declared and inferred singleton types (line 3-13), 2) perform metadata handling (lines 14-17), and 3) to handle the API functions in a special way. PROMPT consults the API model M to determine how to handle a callsite, i.e., to execute the original function or perform some modeling. If the function is modeled by another function then we execute that model function instead of the original function (lines 19-20). Otherwise, we

check if the function can be modeled in a generic way, i.e., by symbolizing its arguments and the return value, if any. External functions are handled in a generic way. A function that has a definition may also be modeled in a generic way if it has inline assembly and the global setting on modeling functions with inline assembly, $M.modelwithassembly$, is set and the function name does not match any of the exceptional cases, $M.AE$. The significance of assembly-level instructions is due to the inability of the underlying symbolic execution engine, KLEE, to handle them, which leads to the immediate termination of the path, and, hence, prevents coverage of the symbolic execution tree beyond that instruction. In general, PROMPT models a function in a generic way by havocing (symbolizing) its arguments and the return value, if any, (lines 22-44) However, havocing of an argument would still be skipped if the global setting on skipping havocing singletons is set, the argument type is a singleton, and havocing of the argument is not specified in $M.HV$. Otherwise, the argument gets havoced. The return value is symbolized and if the global setting for NULL return is set then that case is also considered. Algorithm 7 gets executed to apply any constraint on the return value on the original state and, if applicable, on the cloned one in which the return value is set to NULL. For the API functions for which PROSE models are available, the return values can be explicitly made symbolic inside the models as shown in Figure 3 in lines 7-9. So, for the `v4l2_device_register` function, the side-effects specified on lines 13-25 are are only modeled for the success cases, i.e., $return\_value \geq 0$. This is because on failure cases, the API functions typically revert back the side effects that they may have performed, e.g., allocated memory gets deallocated and the reference count operations get reverted.

Functions that have inline assembly would end up being executed like other unmodeled functions if they correspond to some exceptional cases (line 46). If the global setting on `symbolize inline assembly` is set to true then such instructions would be abstracted away by symbolizing their return values, which is not explicitly shown in the algorithms. This would avoid an error on such a path and enable further exploration.

Algorithm 7 applies constraints on the function arguments or function return values to the given state by transforming the abstract syntax trees of the constraints into symbolic expressions similar to Algorithm 5, which applies type-specific data constraints. For return values, there are two types of constraints depending on whether they relate to the life-cycle or not. Those that relate to the life-cycle determine whether the execution can continue with the next life-cycle entry or terminates. Algorithm 8 calls Algorithm 7 on line 6 for such cases. Those that do not relate to the life-cycle just constrain the return value for that path. Algorithm 6 calls Algorithm 7 line 34 for such cases. The argument constraints are always related to the life-cycle model, i.e., they determine how the arguments should be constrained when a life-cyle entry gets executed. This algorithm gets called by Algorithm 2 (line 7) when the life-cycle starts and by Algorithm 8 (line 13), which we explain next.

Finally, we handle the *return* instructions to enforce the life-cycle rules as shown in Algorithm 8. Recall that in Algorithm 2, we start the execution from

Algorithm 6 Special handling of callsites.

HandleCall(C: Component, st: State, i: Instruction, M: API Model) : $\mathcal{P}(State)$
Let $i \equiv f(args)$
if $(f, mem\_arg) \in M.AF$ **then**
  $(address, mo) \leftarrow HandleAllocate(st, Type(args[mem\_arg]), false, M)$
  if $T \in M.SG$ and $st.TA[T] \neq \bot$ **then**
    print(error: T is not a singleton)
    $st.term \leftarrow true$
    return $\emptyset$
  end if
  $st.TA[T] \leftarrow address$ ▷ Record type to address mapping
else if $(f, arg\_no) \in M.DF$ **then**
  $st.TA[Type(args[arg\_no])] \leftarrow \bot$ ▷ Record type to address mapping
  return $executeInstruction(st, i')$, where $i' \equiv free(args[arg\_no])$
else if $f =$ klee_set_metadata **then**
  $st.MT[args[0]] \leftarrow args[1]$
else if $f =$ klee_get_metadata **then**
  $i.result \leftarrow st.MT[args[0]]$
else if $(f, pf) \in M.FM$ **then** ▷ Use the model function, if exists
  Let $i' = pf(args)$
  return $executeInstruction(st, i')$
else
  if f has assembly and $M.modelwithassembly$ is set and f does not match any of the exception cases in $M.AE$ or $f \notin C.DefinedFunctions$ **then**
    for each arg in $args$ **do**
      if arg is a pointer **then**
        if $M.skiphavocsingletons$ is ON and arg is or reachable from a singleton type and $(f, arg) \notin M.HV$ **then**
          Keep object pointed by arg unchanged
        else
          Make object pointed by arg symbolic
        end if
      end if
    end for
    if return type of f is not void **then**
      Make the return value in i symbolic
      $set \leftarrow \{st\}$
      if $M.nullReturn$ is ON **then**
        $st' \leftarrow st$, where the return value in i is set to NULL in st'
        $set \leftarrow set \cup \{st'\}$
      end if
      $result \leftarrow \emptyset$
      for $st'' \in set$ **do**
        $result \leftarrow result \cup$ **ApplyFunctionConstraint**(st'', f, $return$, $false$, i, M)
      end for
      return $result$
    end if
  else
    return $executeInstruction(st, i)$
  end if
end if
return $\{st\}$

the first entry point specified in the life-cycle rule, $M.LC$, which includes a

Algorithm 7 Application of function constraints on arguments and return values.

```
1: ApplyFunctionConstraint(st: State, f: Function, mtype: {arg, return}, lifecycle:
   boolean, i: Instruction, M: API Model)
2: exp ← true
3: if mtype = arg then
4:     for each (aexp, abinding) ∈ M.FAC[f] do
5:         for each v s.t. abinding[v] = (f, arg_i) do
6:             aexp' ← aexp'[v/st.Mem[Address(arg_i)]]
7:         end for
8:         exp ← exp ∧ aexp'
9:     end for
10: else
11:     if lifecycle = true then (rexp, rbinding) ← M.LC.FRC[f]
12:     else (rexp, rbinding) ← M.FRC[f]
13:     end if
14:     Let v s.t. rbinding[v] = (f, return)
15:     exp ← rexp[v/st.Mem[Address(i.return)]]
16: end if
17: if st.PC ∧ exp is SAT then
18:     st.PC ← st.PC ∧ exp
19:     return {st}
20: else st.term ← true return ∅
21: end if
```

list of function/entry point names along with the constraints on the return values for continuing with the next step. When we return from a function, f, that happens to be one of the life-cycle entries in $M.LC$, we check if the return value satisfies the constraints for continuation of the life-cycle sequence as long as it is not the very last life-cycle entry. If so, the execution continues with the next entry point from the life-cycle sequence by updating the execution stack. Otherwise, the path gets terminated.

6 Case Studies

6.1 Linux Device Drivers

We have used PROSE to model memory related API of three subsystems in the Linux kernel: `video`, `sound`, and `network`. Below, we briefly describe our modeling experience for each subsystem.

The `video` *subsystem* This subsystem creates and manages two main data structures, `video_device` and `v4l2_device`. Typically both objects get embedded in the custom driver data structure and reference counts are kept for each object. Either one of the object types get assigned a `release` callback, which gets called when the reference count of the respective object gets to zero. The `release` callback gets assigned inside the device driver. The `v4l2_device_register` function also sets some pointer fields of the parameters (if not already set) to be used later to drive a pointer to the `v4l2_device`

Algorithm 8 Special handling of *return* instructions to enforce life-cycle rules.

```
1: HandleReturn(st: State, i: Instruction, M: API Model) : P(State)
2: Let i ≡ return exp
3: Let f ← Function(i)
4: if f ∈ M.LC.F then                                          ▷ A life-cyle element
5:     if ReturnType(i) ≠ void then
6:         ApplyFunctionConstraint(st, f, return, true, i, M)
7:         if st.term = false and f is not the last life-cycle entry function then
8:             Let fs ← M.LC.Succ(f)
9:             st.nextInst ← fs.firstInst
10:            st.Stack.pop(f)
11:            st.Stack.push(fs)
12:            Make fs.args symbolic          ▷ Make the arguments symbolic and apply
    constraints on them
13:            return ApplyFunctionConstraint(st, f, arg, false, i, M)
14:        else
15:            st.term ← true                                  ▷ Terminate life-cycle sequence
16:            return ∅
17:        end if
18:    end if
19: end if
20: return executeInstruction(st, i)
```

object from a given `device` object. We modeled 11 API functions for the `video` subsystem. The model consist of 150 SLOC.

The `sound` *subsystem* This subsystem creates a `snd_card` data structure and some auxiliary data structures such as `rawmidi` and `pcm`. Some drivers create a `snd_device` object. Unlike the `video` subsystem, the private/custom data structure of the driver may get allocated by the `snd_card_new` function if a non-zero size for the private data structure is provided as the 5th argument. If so, the `snd_card` object and the private data structure is created as a single blob. The `private_data` field of the `snd_card` object is set to the private data by the `snd_card_new` function. This enables the driver to access the created private data structure via this field. Drivers free `sound` subsystem related data structures using the `snd_card_free` function and are not supposed to free the `card` object or the private data structure explicitly. Other other hand, if the private data structure has not been created via `snd_card_new` then it is the driver's responsibility to free it. We modeled 17 API functions for the `sound` subsystem. The model consists of 342 SLOC.

The `network` *subsystem* In this subsystem, `net_device` is the core data structure. However, as in the `sound` subsystem, it may get allocated as a single blob with the private data structure by the allocation API functions `alloc_netdev` and `alloc_etherdev`. Both of these functions use the first parameter as the size of the private data structure. However, they differ in terms of the `setup` callback they use. The `alloc_netdev` function gets the `setup` callback from the driver whereas the `alloc_etherdev` function uses a specific callback function, `ether_setup`, defined by the `ethernet` subsystem.

The significance of the `setup` callback and how it initializes the fields of the `net_device` object comes into play at cleanup time. The cleanup is generally performed by the `free_netdev` function, which checks a flag field, `reg_state`, of the `net_device` object to check for the registration status. If the object is not registered, it performs an explicit free of the blob. Otherwise, it decrements a reference count, which triggers a generic callback function that frees the object containing the reference count. However, if the `setup` callback sets the `destructor` function pointer, that callback gets executed inside the `unregister_netdev` API function. We modeled 13 API functions for the `network` subsystem. The model consists of 215 SLOC.

We have studied the drivers for each subsystem and reviewed the implementations of the API functions to summarize their side effect in terms of reference counting of the subsystem specific data structures and their registration/deregistration logic. Each subsystem took 1 person day to do the modeling and its validation using PROMPT. In some cases, we got false positives that helped us understand what part of the model was not precise enough. We think that it may take less time to develop precise models for domain experts. As we report in Section 7, with these models we could analyze the setup and teardown entry points of 57 device drivers with considerable coverage and found some real bugs.

6.2 BlueZ: A Bluetooth Stack

In this section, we show how to detect some of the recent vulnerabilities in BlueZ [4] using PROMPT. BlueZ is the implementation of the Bluetooth protocol [2] for the Linux kernel. The implementations of the Bluetooth protocol form an important part of the attack surface for the Internet of Things (IoT) due to its critical role for short-range communications. Researchers have found a set of vulnerabilities, called BlueBorne [1], in various Bluetooth stack implementations. One of these is a stack overflow in the L2CAP layer of BlueZ. Figure 6 shows an excerpt from the `l2cap_config_rsp` function that is vulnerable to a stack overflow. The vulnerability is due to not checking the size of the buffer `buf` (line 16) while copying configuration data from the `rsp->data` buffer inside the `l2cap_parse_conf_rsp` function (line 18).

To analyze the `l2cap_config_rsp` function and to reproduce the Blueborne vulnerability, we needed to specify the type embedding relationship between the `list_head` struct and the `l2cap_chan` struct as shown in Figure 7. This is because before the configuration options get copied to the local buffer, the `l2cap_config_rsp` function retrieves a handle to the communication channel using the `l2cap_get_chan_by_scid` function, which gets a pointer to a `l2cap_conn` object and a channel no `scid` and performs a linear search on a linked list. Failing to specify the type embedding relationship causes a memory error due to a pointer arithmetic that tries to compute the address of the `l2cap_chan` object that encloses the `list_head` object shown with a rectangle with a bold border in Figure 7. This false positive also pre-

```
1 static inline int l2cap_config_rsp(struct l2cap_conn *conn,
2                                    struct l2cap_cmd_hdr *cmd, u16 cmd_len,
3                                    u8 *data) {
4         struct l2cap_conf_rsp *rsp = (struct l2cap_conf_rsp *)data;
5         struct l2cap_chan *chan;
6
7         int len = cmd_len - sizeof(*rsp);
8         chan = l2cap_get_chan_by_scid(conn, scid);
9         if (!chan)
10                return 0;
11
12        switch (result) {
13        case L2CAP_CONF_SUCCESS: ...
14        case L2CAP_CONF_PENDING: ...
15                if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
16                        char buf[64];
17
18                        len = l2cap_parse_conf_rsp(chan, rsp->data, len,
19                                                   buf, &result);
20                                                   ...
21 }
```

Fig. 6: A code excerpt from the `l2cap_config_rsp` function that hosts the Blueborne vulnerability, CVE-2017-1000251, on line 18.

vents symbolic execution from covering code after line 8 in Figure 6 and, hence, prevents component-level symbolic execution from detecting the vulnerability. Embedded linked lists are another source of non-standard pointer arithmetic that is known to exist in system code [17]. PROSE enables us to model this programming idiom through the type embedding relationship and PROMPT incorporates this type of modeling to its lazy initialization process.

Another vulnerability in BlueZ is a memory out of bounds read [3] that was recently found in the HCI layer. Figure 8 shows a code excerpt from the `hci_extended_inquiry_result_evt` function. The number of responses stored in `num_rsp` is read from the event packet. In a malformed event packet this field may be larger than the socket buffer length `skb->len`, which leads to a memory out of bounds error at line 16. For reproducing this vulnerability, our modeling efforts involved taming the path explosion through modeling all functions inside the `hci_extended_inquiry_result_evt` function except `bacpy` using the generic modeling approach.

7 Experiments

We have implemented our approach in a tool, called PROMPT, by extending the KLEE symbolic execution engine [13]. We have applied PROMPT to two case studies: Linux device drivers and BlueZ as explained in Sections 6.1 and 6.2, respectively. We have analyzed a total of 57 Linux device drivers: 18 video drivers, 19 sound drivers, and 20 network drivers. We chose the drivers that were developed for the x86 architecture and those that do not use the

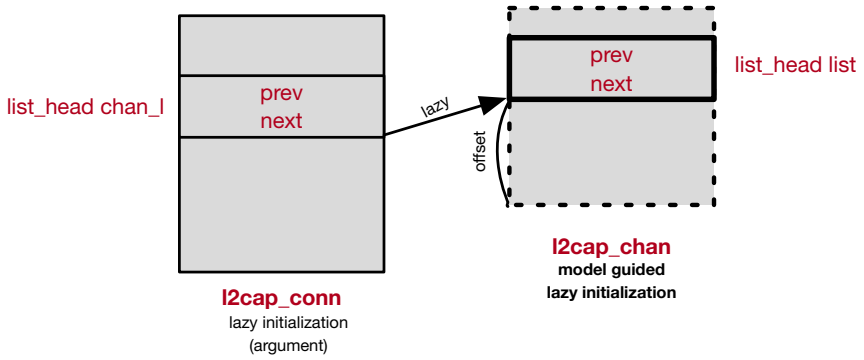


Fig. 7: Model guided lazy initialization of an embedded linked list in the L2CAP layer of BlueZ. The region with a bold border shows an example of an embedded region, where the embedding type is `l2cap_chan`, (with the dashed border), which gets created to enforce the modeling rule on type embedding. The labels on the arrows represent the source of the updates and those under the objects show how they get created.

```
1 static void hci_extended_inquiry_result_evt(struct hci_dev *hdev,
2                                             struct sk_buff *skb) {
3         struct inquiry_data data;
4         struct extended_inquiry_info *info = (void *) (skb->data + 1);
5         int num_rsp = *((__u8 *) skb->data);
6
7         if (!num_rsp)
8                 return;
9         ...
10
11        for (; num_rsp; num_rsp--, info++) {
12                u32 flags;
13                bool name_known;
14
15                bacpy(&data.bdaddr, &info->bdaddr);
16                ...
17 }
```

Fig. 8: A code excerpt from the HCI layer of BlueZ that hosts a memory out of bounds read at line 15.

firmware upload feature as we lacked the domain knowledge[5] needed to model the relevant API functions. For BlueZ, we focused on reproducing two vulnerabilities: one on the L2CAP layer and the other one on the HCI layer. We have used version v4.11-rc2 of the Linux kernel. Our experiments have been performed on an Ubuntu 16.04 rack server that features 4 processors with 16

[5] Note that the developers of such drivers typically have such domain knowledge and by modeling the relevant API functions they can analyze such drivers with the help of PROMPT.

cores, and 256GB memory. Section 7.1 presents statistics about the benchmarks and their models. Sections 7.2-7.6 present the experimental results to evaluate PROMPT w.r.t. five important research questions.

7.1 Benchmarks

The benchmarks we have used for our experiments are shown in Tables 1-3, which list the driver and vendor names, bus types, and bitcode sizes of the drivers along with the sizes of the models in terms of the number of functions specified as return only (**RO**), the number of singletons (**ST**), the number of modeled alloc/dealloc functions (**AL**), the number of embedding pairs (**EM**), and the number of data constraints (**DC**). The models mentioned in Tables 1-3 are in addition to the PROSE models of subsystems that are explained in Section 6.1. As shown in Algorithm 6, the functions that are specified as return value only are modeled by symbolizing the return value and interpreting them as side-effect free. Such functions constitute an important part of the modeling effort as they are identified in an iterative manner and as a response to various issues encountered during API model guided symbolic execution of the component under analysis. These issues include the following: 1) the underlying symbolic execution engine, KLEE, terminating a path upon encountering an assembly-level instruction, 2) a memory out-of-bounds error due to an under-constrained symbolic value that gets used as an array index, and 3) a symbolized object pointed by a pointer argument leading to a false positive as a result of imprecise data-flow in subsequent instructions. The return value only model helps achieve more coverage in cases 1) and 2) by modeling the function in which the error occurs and avoiding the error cases and in case 3) by avoiding symbolization of the arguments. We talk more about this modeling overhead and how PROMPT manages to eliminate it in Section 7.3. The majority of singleton types (**ST**) are the same for the drivers in the same subsystem. The number of unique singleton types from various subsystems and the number of unique driver specific singleton types within the benchmark set of each subsystem are as follows: video: (12,18), sound: (9,19), network: (11,20). The number of unique (de)alloc functions modeled within the benchmark set of each subsystem is as follows: video: 7, sound: 4, network: 5. The number of unique embedding types within the benchmark set of each subsystem is as follows: video: 3, sound: 2, network: 4. The number of unique data initializations within the benchmark set of each subsystem is as follows: video: 2, sound: 0, network: 2.

7.2 RQ1: What is the advantage of modeling and API Model Guided Symbolic Execution?

An obvious question is whether we could do the analysis on the whole framework and avoid modeling. We created the full bitcode for the Linux kernel to

Table 1: Drivers from the Linux video subsystem. **RO**, **ST**, **AL**, **EM** , and **DC** denote the number of functions specified as return only, the number of singletons, the number of modeled (de)alloc functions, the number of embedding pairs, and the number of data constraints, respectively.

Driver	**Device Vendor**	**Bus Type**	**Size (KB)**	**Model Size**				
				RO	**ST**	**AL**	**EM**	**DC**
airspy	AirSpy	USB	58	18	5	5	1	1
dsbr100	D-Link	USB	33	11	5	5	1	0
go7007	Micronas	USB	242	11	5	5	1	0
hackrf	SparkFun	USB	69	20	5	5	1	1
hdpvr	Hauppauge	USB	92	12	5	5	1	1
radio-keene	Keene	USB	34	13	5	5	1	0
radio-ma901	MasterKit	USB	33	13	5	5	1	0
radio-maxiradio	Guillemot	PCI	26	22	5	5	1	1
radio-mr800	Avervideo	USB	38	13	5	5	1	1
radio-raremono	Cisco	USB	32	14	5	7	1	1
radio-shark2	Griffin	USB	33	13	5	5	1	0
radio-shark	Griffin	USB	34	13	5	5	1	0
radio-tea5764	NXP	I2C	35	13	5	5	1	0
saa7706h	Philips	I2C	27	15	7	5	1	0
stkwebcam	Syntek.	USB	80	11	5	5	1	0
tef6862	Philips	I2C	25	16	5	5	1	0
usbtv	Fushicai	USB	68	11	5	5	0	0
zr364xx	Zoran	USB	74	12	5	5	1	0

analyze each driver without any models. On an Ubuntu 16.04 machine with 256GB of RAM, running PROMPT on the full kernel bitcode without any modeling took 2 hours to initialize the global state with around 86K global object allocations and a total memory usage around of 32GB virtual memory.

Another relevant question is how easy it would be to detect the vulnerabilities using testing instead of API model guided symbolic execution. To answer this question we focus on the two vulnerabilities found in BlueZ from Section 6.2 as we have some evidence about the difficulty of detecting these vulnerabilities. The first vulnerability we consider is the stack overflow vulnerability in BlueZ and is part of the BlueBorne family of vulnerabilities [1]. We quote from the report [1] that comments on the difficulty of detecting this stack overflow vulnerability: ‘ ‘It should be mentioned that testing and triggering this vulnerability was not an easy task, and required direct use of the ACL layer to send malformed L2CAP packets. Since no Bluetooth stack provides this to the user a minimal stack implementing the HCI, ACL and L2CAP layers had to be created. The high barrier of entry for testing highly exposed kernel code paths is also detrimental to security”. We were able to reproduce this stack overflow vulnerability with PROMPT within 5 minutes. Modeling the embed-

Table 2: Drivers from the Linux sound subsystem. **RO**, **ST**, **AL**, **EM** , and **DC** denote the number of functions specified as return only, the number of singletons, the number of modeled (de)alloc functions, the number of embedding pairs, and the number of data constraints, respectively.

Driver	Device Vendor	Bus Type	Size (KB)	Model Size				
				RO	ST	AL	EM	DC
6fire	TerraTec	USB	85	26	7	4	1	0
ad1889	A.D.	PCI	48	31	6	3	1	0
ali5451	ALi	PCI	75	33	6	3	1	0
aw2	Emagic	PCI	42	33	6	3	1	0
bcd2000	Behringer	USB	31	27	8	4	1	0
ca0106	S.B.	PCI	118	38	6	3	1	0
caiaq	Caiaq	USB	118	24	7	4	1	0
card	-	USB	88	23	7	4	1	0
cs46xx	Cirrus L.	PCI	218	41	6	3	1	0
cs5535audio	-	PCI	50	33	6	3	1	0
hiface	M2Tech	USB	44	27	8	4	1	0
lx6464es	Digigram	PCI	73	34	6	3	1	0
misc	Edirol	USB	28	27	8	3	1	0
nm256	NeoMagic	PCI	113	35	6	3	1	0
oxygen	C-video	PCI	240	39	6	3	1	0
riptide	Conexant	PCI	79	35	6	3	1	0
usx2y	Tascam	USB	28	27	8	4	1	0
vx222	Digigram	PCI	45	33	6	3	1	0
ymfpci	Yamaha	PCI	107	36	6	3	1	0

ding relationship mentioned in Section 6.2 enabled component-level symbolic execution to move beyond the callsite for the `l2cap_get_chan_by_scid` at line 8 in Figure 6 and to reach the location of the memory error. We also needed to restrict the type of configuration option to `MTU` by implementing a model function for the `l2cap_get_conf_opt` option to deal with the state explosion.

The other vulnerability we consider has been detected recently in the HCI layer of BlueZ [3] using a coverage based fuzzing tool, Syzkaller [6]. Analyzing the stack trace [5] indicates the difficulty of preparing a test environment. There are a total of 29 kernel functions listed in the stack trace excluding those functions from KASAN, the kernel sanitizer. Only five of these functions are from BlueZ. We have detected this memory out of bounds vulnerability within a minute by modeling all functions inside the loop in Figure 8 except `bacpy` using the generic approach.

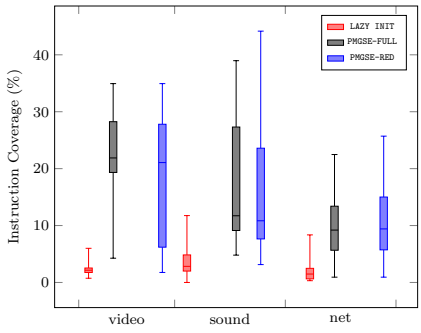
As we present in the following subsections, *modeling reduces the memory footprint and the analysis time of the system under analysis. It also helps detect deep vulnerabilities that may require considerable testing effort.*

Table 3: Drivers from the Linux network subsystem. **RO**, **ST**, **AL**, **EM** , and **DC** denote the number of functions specified as return only, the number of singletons, the number of modeled (de)alloc functions, the number of embedding pairs, and the number of data constraints, respectively.

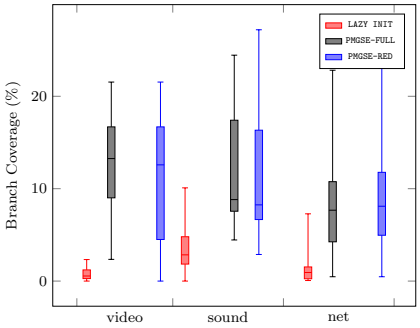
Driver	**Device Vendor**	**Bus Type**	**Size (KB)**	**Model Size**				
				RO	**ST**	**AL**	**EM**	**DC**
amazon	Amazon	PCI	298	21	5	5	1	0
amd	Amd	PCI	85	27	3	5	1	0
axnet_cs	-	PCI	67	28	4	3	1	0
catc	CATC	USB	56	30	4	3	1	0
cdc-phonet	Nokia	USB	43	27	4	3	1	0
cisco	Cisco	PCI	320	21	3	5	1	0
dlink	DLink	PCI	90	21	3	5	1	0
e100	Intel	PCI	126	22	3	5	1	0
fujitsu	Fujitsu	PCI	55	42	3	5	1	0
hso	Option	USB	119	25	7	3	1	1
ipheth	Apple	USB	49	24	4	3	1	0
kaweth	-	USB	61	25	4	3	1	0
lan78xx	Microchip	USB	159	25	4	3	1	0
forcedeth	Nvidia	PCI	206	21	3	5	1	0
qlogic	QLogic	PCI	143	28	3	5	1	0
qmi_wwan	Huawei	USB	70	26	6	3	1	0
r8152	Realtek	PCI	175	24	4	3	2	0
r8169	Realtek	PCI	249	24	4	5	2	1
typhoon	3Com	PCI		24	3	3	1	0
usbnet	-	USB	109	24	3	3	1	0

7.3 RQ2: How does PROMPT perform when some of the modeling effort is reduced through automation?

As mentioned in Section 7.1, it is important to identify the functions to be modeled as the return value to be symbolized only to improve coverage during API model guided symbolic execution. However, the manual effort in the identification of such functions dominates the modeling effort for each benchmark. We have implemented two features that are shown in Algorithm 6 to eliminate this manual overhead: 1) automatically identifying functions with assembly-level instructions and handling them by symbolizing the return value only and 2) not symbolizing the arguments of external functions that point to singleton type objects or point to objects reachable by singleton type objects. To evaluate the effectiveness of these two automated modeling heuristics, we compared PROMPT with a manually identified set of functions that are handled by symbolizing the return value only , which we call `PMGSE-FULL`, with PROMPT not using any such manually identified set of functions, i.e., the **RO**



(a) Comparison in terms of instruction coverage.



(b) Comparison in terms of branch coverage.

Fig. 9: Comparison of lazy initialization only (`LAZY INIT`) and programming model guided symbolic execution with full (`PMGSE-FULL`) and reduced (`PMGSE-RED` models.) models.

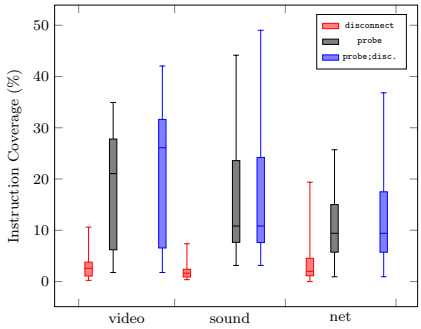
column in Tables 1-3 being 0 for each benchmark, which we call `PMGSE-RED`. We based our comparison in terms of the effectiveness of each configuration with respect to instruction coverage and branch coverage. As Figures 9a and 9b show `PMGSE-RED` achieves comparable coverage while having less manual effort for modeling.

7.4 RQ3: Is programming model guided symbolic execution more effective than symbolic execution with lazy initialization?

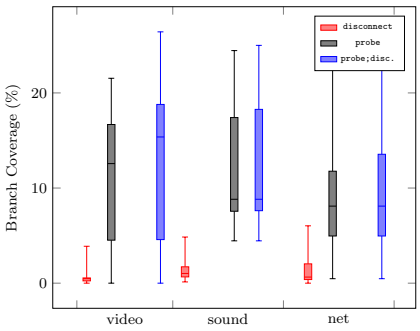
We would have liked to compare PROMPT with the lazy initialization implementation in [31]. However, due to the unavailability of this code, we ran PROMPT in a mode without an API model and used lazy initialization only (`LAZY INIT`). Our goal was to understand in what ways programming model guided symbolic execution improves over pure lazy initialization. We have analyzed the `probe` functions of the drivers using both modes. Figures 9a and 9b compare lazy initialization only with the two modes of PMGSE in terms of instruction coverage and branch coverage, respectively. We also measured the percentage of error paths and present it in Table 4. API model guided execution, both `PMGSE-FULL` and `PMGSE-RED`, significantly improves lazy initialization both in terms of coverage and in terms of false positives as all the errors generated by the `LAZY INIT` mode were false positives and in that mode PROMPT could not detect any of the six real bugs that it could detect when executed in API model guided mode.

Table 4: Comparison of lazy initialization with programming model guided symbolic execution (`PMGSE-FULL` and `PMGSE-RED`) in terms of error rate.

%	PMGSE-FULL			PMGSE-RED			LAZY INIT		
	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX	AVG
video	0	5.88	0.33	0	100	11.44	0	100 .00	78.70
sound	0	16.67	4.44	0	25.00	7.92	0	50.00	23.69
net	0	1.27	0.08	0	1.33	0.17	0	100.00	61.47



(a) Comparison in terms of instruction coverage.



(b) Comparison in terms of branch coverage.

Fig. 10: Comparison of analysis of individual functions with the life-cycle model of `probe;disconnect`.

7.5 RQ4: Is simulation of the life-cycle more effective than the individual analysis of life-cycle functions?

Finally, we wanted to check whether simulation of the life-cycle as a sequence of function executions have any benefit over analyzing the life-cycle functions individually. We ran PROMPT in API model guided mode in three configurations: 1) executes the probe function only, 2) executes the disconnect function only, and 3) executes probe followed by disconnect, denoted by `probe ; disconnect`. As shown in Figure 10, life-cycle model does not improve coverage as the sum of coverage of individual cases is almost equal to the coverage of the life-cycle model. This is because the `disconnect` functions are typically small in size and mostly without any branch instructions. However, as Tables 5 and 6 show, the percentage of error paths for the `disconnect` case is significantly higher than the `probe ; disconnect` case. This is because when `disconnect` is executed without the proper setup that is normally performed by the `probe` function, errors that are due to improper setup get manifested. This also prevents the detection of real errors, e.g., a memory leak when the device gets unplugged.

Table 5: Comparison of enforcing the life-cycle model `probe ; disconnect` vs executing `probe` and `disconnect` alone in terms of the error rate for `PMGSE-FULL`.

%	DISCONNECT			PROBE			PROBE;DISC.		
	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX	AVG
video	0.00	100.00	88.89	0.00	5.88	0.33	0.00	3.23	0.18
sound	0.19	100.00	47.90	0.00	16.67	4.41	0.00	16.67	4.43
net	0.00	100.00	81.58	0.00	1.27	0.09	0.00	1.28	0.09

Table 6: Comparison of enforcing the life-cycle model `probe ; disconnect` vs executing `probe` and `disconnect` alone in terms of the error rate for `PMGSE-RED`.

%	DISCONNECT			PROBE			PROBE;DISC.		
	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX	AVG
video	0.00	100.00	80.56	0.00	100.00	11.43	0.00	100.00	11.29
sound	0.19	100.00	47.09	0.00	25.00	7.92	0.00	25.00	7.92
net	0.00	100.00	81.58	0.00	1.33	0.17	0.00	2.13	0.29

7.6 RQ5: How effective is PROMPT in detecting memory errors?

In addition to the two BlueZ vulnerabilities presented in Section 6.2, PROMPT was able to detect four vulnerabilities in the Linux device drivers. Two of them were previously known use-after-free vulnerabilities in the `usbtv` (also shown in Section 3) and the `stkwebcam` drivers from the video subsystem. The new bugs discovered by PROMPT consist of a double-free in the `hso` driver (network subsystem) and a NULL pointer error in the `cs46xx` driver (sound subsystem). Table 7 presents the time and maximum amount of memory[6] usage of PROMPT for both `PMGSE-FULL` and `PMGSE-RED` modes as reported by the `klee-stats` tool. In terms of the bug detection capability, `PMGSE-RED` was able to detect four out of six bugs. The $\checkmark^*$ in `stkwebcam` and `cs46xx` refers to the fact that we needed to enforce usage of the original implementations for the functions of the driver even if they may have inline-assembly (see the formal condition on line 22 in Algorithm 6), which required a simple pattern for such functions to be defined: `*stk*` and `*cs46xx*` for the `stkwebcam` and the `cs46xx` drivers, respectively. So, as a result any function in the `stkwebcam` (`cs46xx`) driver that includes the string `stk` (`cs46xx`) in its name would not be modeled as return only and the original implementation would be used in the analysis even if it has inline assembly.

`PMGSE-RED` took much longer to detect the stack overflow in BlueZ compared to `PMGSE-FULL` as not all functions that contributed to path explosion included inline assembly. This is because of the path explosion problem ob-

[6] klee-stats reports the amount of heap memory created via malloc.

served in these components. In the case of memory out of bounds in the HCI layer, `PMGSE-RED` could not detect the vulnerability within 12 hours. For the `hso` case, `PMGSE-RED` could not detect the bug because it gets masked by a memory error due to an under-constrained symbolic value that gets used as an array index in the `hso_get_config_data` function. However, specifying this function to be modeled as symbolizing the return value only, enables `PMGSE-RED` also detect this bug, which is still an improvement in terms of modeling effort as the size of **RO** is 1 compared to the **RO** size of 25 for the `PMGSE-FULL` mode. The bug in the `cs46xx` driver got fixed [16] after we reported it and we provided a patch for the bug in the `hsobug` driver [38].

Table 7: Time and memory usage of PROMPT for finding various bugs with full (`PMGSE-FULL`) and reduced (`PMGSE-RED`) models. ✓ (✓*) means the bug could be detected (while enforcing originals of certain functions with inline assembly), and - denotes the bug could not be detected.

COMPONENT	**BUG**	**PMGSE-FULL**			**PMGSE-RED**		
		Time (secs)	Mem. (MB)	Det.?	Time (secs)	Mem. (MB)	Det.?
L2CAP (BlueZ)	stack overflow	259.93	155.65	✓	9480.00	1250.97	✓*
HCI (BlueZ)	out of bounds	41.42	133.20	✓	-	-	✗
usbtv	use-after-free	5.93	7.78	✓	6.67	8.42	✓
stkwebcam	double-free	7.64	6.68	✓	7.74	6.68	✓*
cs46xx	null pointer	10.68	22.12	✓	10.68	22.08	✓*
hso	double-free	305.10	132.53	✓	-	-	✗

8 Conclusions

We have presented an API model guided symbolic execution tool, PROMPT, to detect memory related bugs in system code. Our work identifies the major aspects of API modeling and provides an API modeling language, PROSE, and presents PROMPT, a component-level symbolic execution algorithm that enforces the specified API model on top of the KLEE symbolic execution engine. PROSE facilitates analysis of system components by eliminating the need for developing a test harness, recompilation of the underlying code base, and changing the underlying symbolic execution engine. Additionally, PROSE enables modeling of programming idioms that are common in systems code, which can be used to guide PROMPT for a more precise analysis at the component level. We demonstrated the effectiveness of our approach by modeling the registration and cleanup APIs of the `video`, `sound`, and `network` subsystems of the Linux kernel and by analyzing 57 device drivers. We also applied our approach to reproduce some critical vulnerabilities in BlueZ. PROMPT

could detect two new and four known memory vulnerabilities in the Linux kernel. PROMPT can also be used to validate models of API functions and to infer various rules on their usages. In future work, we are planning to extend PROMPT with automated API model synthesis capability.

References

1. BlueBorne. "https://info.armis.com/rs/645-PDC-047/images/BlueBorne%20Technical%20White%20Paper_20171130.pdf". Last accessed on August 1st, 2020
2. Bluetooth. "https://www.bluetooth.com/". Last accessed on August 1st, 2020
3. Bluetooth: Fix slab-out-of-bounds read in hci_extended_inquiry_result_evt(). "https://git.kernel.org/pub/scm/linux/kernel/git/bluetooth/bluetooth-next.git/commit/net/bluetooth?id=51c19bf3d5cfaa66571e4b88ba2a6f6295311101". Last accessed on August 1st, 2020
4. BlueZ Official Linux Bluetooth protocol stack. "http://www.bluez.org/". Last accessed on August 1st, 2020
5. KASAN: slab-out-of-bounds Read in hci_extended_inquiry_result_evt. "https://syzkaller.appspot.com/bug?id=4bf11aa05c4ca51ce0df86e500fce486552dc8d2". Last accessed on August 1st, 2020
6. syzkaller - kernel fuzzer. "https://github.com/google/syzkaller". Last accessed on August 1st, 2020
7. Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M.: A Systematic Evaluation of API-Misuse Detectors. CoRR **abs/1712.00242** (2017)
8. Bai, G., Ye, Q., Wu, Y., Botha, H., Sun, J., Liu, Y., Dong, J.S., Visser, W.: Towards model checking android applications. IEEE Trans. Software Eng. **44**(6), 595–612 (2018)
9. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM **54**(7) (2011)
10. Ball, T., Rajamani, S.: Slic: A specification language for interface checking (of c). Tech. rep. (2002). URL https://tinyurl.com/y8y4zkdy
11. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Computer Aided Verification - 23rd International Conference, CAV'11. Proceedings, pp. 184–190 (2011)
12. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011, pp. 183–198 (2011)
13. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings, pp. 209–224 (2008)
14. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. In: Proceedings of the 12th International Conference on Model Checking Software, SPIN'05 (2005)
15. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
16. Carpenter, D.: Alsa: cs46xx: Potential null dereference in probe. "https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/sound/pci/cs46xx?id=1524f4e47f90b27a3ac84efbdd94c63172246a6f"
17. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamaric, Z.: A low-level memory model and an accompanying reachability predicate. Int. J. Softw. Tools Technol. Transf. **11**(2), 105–116 (2009)
18. Chipounov, V., Kuznetsov, V., Candea, G.: The s2e platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1) (2012)
19. Fahl, S., Harbach, M., Perl, H., Koetter, M., Smith, M.: Rethinking SSL Development in an Appified World. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS '13 (2013)

20. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: 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, CCS '12 (2012)
21. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, pp. 213–223 (2005)
22. Heule, S., Sridharan, M., Chandra, S.: Mimic: Computing models for opaque code. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015 (2015)
23. Indela, S., Kulkarni, M., Nayak, K., Dumitraş, T.: 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, Onward! (2016)
24. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, pp. 553–568 (2003)
25. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976)
26. Mehlitz, P.C., Tkachuk, O., Ujma, M.: JPF-AWT: model checking GUI applications. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pp. 584–587 (2011)
27. Myers, B.A., Stylos, J.: Improving API Usability. Commun. ACM **59**(6) (2016)
28. Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16 (2016)
29. Park, J., Jordan, A., Ryu, S.: Automatic modeling of opaque code for javascript static analysis. In: Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, pp. 43–60 (2019)
30. Qi, D., Sumner, W.N., Qin, F., Zheng, M., Zhang, X., Roychoudhury, A.: Modeling software execution environment. In: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012, pp. 415–424 (2012)
31. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15, pp. 49–64 (2015)
32. Recoules, F., Bardin, S., Bonichon, R., Mounier, L., Potet, M.: Get rid of inline assembly through verification-oriented lifting. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pp. 577–589 (2019)
33. Renzelmann, M.J., Kadav, A., Swift, M.M.: Symdrive: Testing drivers without devices. In: C. Thekkath, A. Vahdat (eds.) 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012, pp. 279–292. USENIX Association (2012)
34. Shi, K., Steinhardt, J., Liang, P.: Frangel: Component-based synthesis with control structures. Proc. ACM Program. Lang. **3**(POPL) (2019)
35. Visser, W., Mehlitz, P.C.: Model checking programs with java pathfinder. In: Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings, p. 27 (2005)
36. Wang, W., Barrett, C.W., Wies, T.: Partitioned memory models for program analysis. In: Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10145, pp. 539–558. Springer (2017)
37. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA, pp. 501–504 (2007)

38. Yavuz, T.: [patch] net: hso: do not call unregister if not registered. "https://lists.openwall.net/netdev/2019/02/09/1"
39. Yong, S.H., Horwitz, S., Reps, T.: Pointer analysis for programs with structures and casting. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI '99, p. 91–103. Association for Computing Machinery, New York, NY, USA (1999)
40. Yun, I., Min, C., Si, X., Jang, Y., Kim, T., Naik, M.: APISan: Sanitizing API Usages through Semantic Cross-Checking. In: 25th USENIX Security Symposium, USENIX Security'16, pp. 363–378 (2016)
41. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software **41**(1), 49–64 (2015)