Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries

Haohuang Wen and Zhiqiang Lin, *The Ohio State University*

https://www.usenix.org/conference/usenixsecurity23/presentation/wen

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries

Haohuang Wen
The Ohio State University
wen.423@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

As one of the most popular C++ extensions for developing graphical user interface (GUI) based applications, Qt has been widely used in desktops, mobiles, IoTs, automobiles, *etc*. Although existing binary analysis platforms (*e.g.*, angr and Ghidra) could help reverse engineer Qt binaries, they still need to address many fundamental challenges such as the recovery of control flow graphs and symbols. In this paper, we take a first look at understanding the unique challenges and opportunities in Qt binary analysis, developing enabling techniques, and demonstrating novel applications. In particular, although callbacks make control flow recovery challenging, we notice that Qt's signal and slot mechanism can be used to recover function callbacks. More interestingly, Qt's unique dynamic introspection can also be repurposed to recover semantic symbols. Based on these insights, we develop QTRE for function callback and semantic symbol recovery for Qt binaries. We have tested QTRE with two suites of Qt binaries: Linux KDE and the Tesla Model S firmware, where QTRE additionally recovered 10,867 callback instances and 24,973 semantic symbols from 123 binaries, which cannot be identified by existing tools. We demonstrate a novel application of using QTRE to extract hidden commands from a Tesla Model S firmware. QTRE discovered 12 hidden commands including five unknown to the public, which can potentially be exploited to manipulate vehicle settings.

1 Introduction

Qt [12] is a cross-platform software development framework with rich software modules and libraries (*e.g.*, graphical widgets, networking, and database). It facilitates the development of various C++ programs, especially applications with graphical user interfaces (GUIs). So far, Qt has attracted millions of developers worldwide, and together they have produced numerous Qt applications across over 70 industries [13]. Today, Qt is ubiquitous not only in operating system interfaces (*e.g.*, Linux KUbuntu and BlackBerry mobile OS) and commercial software (*e.g.*, Adobe software, Teamviewer, and VirtualBox), but also embedded IoT devices (*e.g.*, LG smart TV WebOS). Qt has also been increasingly used in security-critical domains such as medical applications, industrial automation, and automotive systems [13] (*e.g.*, the in-vehicle infotainment systems in Mercedes and Tesla vehicles).

Since Qt plays an important role in many modern applications, it is imperative to ensure that Qt binaries are free of vulnerabilities and hardened. Fundamentally, one key approach to achieving these security goals is to enable reverse engineering (RE) of Qt. Although there have been numerous techniques and frameworks for binary RE (*e.g.*, BitBlaze [63], BAP [26], BinaryNinja [3], angr [61], and Ghidra [4]), which can certainly be applied to Qt binary analysis, they are not perfect and still need to address many fundamental RE challenges. For example, control flow graph (CFG) recovery is essential for any RE tasks but challenging due to the indirect control flow transfers such as callbacks [23, 27, 53, 67]. Meanwhile, although program symbols are extremely useful, they are often stripped in released binaries, making it non-trivial to recover them from the binary code [44, 45, 59, 62, 66]. Therefore, any advancement towards CFG and symbol recovery will tremendously help RE for Qt binaries.

Interestingly, new problems also come with new opportunities. We notice that the two unique features in Qt: (1) the signal and slot mechanism for implementing callbacks and (2) dynamic introspection for run-time class member query and update, can be *repurposed* to resolve the aforementioned two fundamental RE challenges for Qt binary analysis. Specifically, while the signal and slot mechanism provides a standard way for Qt programmers to register function callbacks, it also enables reverse engineers to identify function callback targets for CFG recovery. More interestingly, we find that the semantic symbols (*e.g.*, the names of variables and functions, and their corresponding addresses and types) can be recovered by repurposing Qt's dynamic introspection. Meanwhile, unlike existing approaches which cannot guarantee 100% accuracy for symbol recovery [38, 44, 45, 59, 62, 66], we can achieve this for Qt because fundamentally these symbols must be preserved in support of its dynamic introspection.

Based on these insights, we present QTRE, the first tool that leverages Qt's unique features for Qt binary RE. Although it may seem trivial to realize QTRE, we still have to solve two main challenges. First, identifying function callback targets requires us to not only resolve the callback function signatures, but also infer the correct class types due to the polymorphism in C++ [53]. Second, after extracting symbol strings from Qt's internal data structures and tables, we still need to map them (*e.g.*, property symbols) to the corresponding addresses, which are often relative and dynamically computed.

We have addressed these challenges through novel approaches including a source-aware class inference to resolve callbacks and a unit-level symbolic execution to compute symbol addresses. We have implemented QTRE atop the open-source binary analysis framework Ghidra [4]. To evaluate QTRE, we selected two suites of Qt software binaries: open-source KDE [6], and closed-source infotainment software from a Tesla Model S. Among the 123 binaries in total, QTRE additionally recovered 10,867 callback instances and identified 24,973 semantic symbols, which cannot be extracted through other static binary analysis tools such as Ghidra and angr. Through manual validation with 16 binaries from the open-source KDE suite, we successfully validated 598 callbacks and 1,161 symbols, and we did not find any false positives. However, we did observe 15% false negatives among the recovered callbacks and three false negatives in symbol recovery, which are caused by indirect function calls and memory aliasing [30, 46].

There could be multiple applications with QTRE, such as coverage-based fuzzing to identify vulnerabilities in GUI-rich Qt binaries. In this paper, we instead demonstrate a novel application of using input validation analysis to extract hidden commands from the Tesla infotainment software, based on the Qt-unique callbacks and symbols recovered by QTRE. To our surprise, among the 43 Tesla Qt binaries, QTRE detected 12 hidden commands that can be triggered by user inputs. Among them, five commands are unknown to the public, which can be potentially exploited to manipulate vehicle settings or leak sensitive user data of the vehicle.

Contributions. Our paper makes the following contributions:

- We are the *first* to propose effective techniques for CFG and symbol recovery in Qt binary RE (§3), by leveraging Qt's unique signal and slot as well as its dynamic introspection.
- We designed (§4) and implemented QTRE (§5), an open-source tool to facilitate Qt binary analysis. It uses a source-aware class inference to resolve indirect call targets of function callbacks, and a unit-level symbolic execution to recover semantic symbols.
- We evaluated QTRE on KDE and Tesla Model S firmware, where it additionally recovered 10,867 callbacks and 24,973 symbols (§6). From the Tesla firmware, it discovered 12 hidden commands with five new to the public (§7).

2 Qt Primer

Qt has many powerful features and attracted millions of developers [13]. In particular, Qt is cross-platform and supports various architectures (e.g., x86 and ARM) and operating systems (*e.g.*, Windows, Linux, MacOS, Android, iOS, and QNX). Meanwhile, Qt is cross-language and provides bindings for many widely used programming languages such as python, R, and Go [7]. In addition, Qt also offers a number of unique mechanisms that allow developers to easily implement functions not available in native C++, such as dynamic object introspection. Fundamentally, all of these important features are rooted on Qt's Meta-Object Compiler (MoC) [20]. Specifically, the MoC can automatically generate C++ code (*e.g.*, built-in functions and special data structures such as string tables) for Qt-based classes (*i.e.*, those inherited from the Qt base class `QObject`).

Prevalence. To understand the prevalence of Qt among numerous C++ programming frameworks, we conducted an empirical measurement by counting the Github code repositories developed based on different C++ frameworks. We provide their distributions in Table 8 in Appendix for reader's interest. According to the results, Qt is arguably the most dominant framework for developing C++ applications with over 45K repositories, nearly 3X of the second-place ROS. The details of the measurement study are presented in Appendix §A.

Application. Qt has been used by many applications. In particular, Tesla uses Qt to develop its in-vehicle system (IVS). Fundamentally, the Tesla IVS runs a Linux kernel-based Ubuntu OS with Nvidia Tegra CPUs (ARM architecture) to support high quality GUI [47]. At a high level, it can be divided into the front-end (for direct Qt-based GUI interactions) and the back-end (for internal network and control logic). On the front-end side, there is a central information display (CID), which is commonly known as the infotainment system, and an instrument cluster (IC). By default, the CID has many pre-installed Qt applications such as browser, navigation, and hands-free calls. On the back-end side, the parrot and sierra module handle network communications (*e.g.*, Wi-Fi, Bluetooth, and cellular network) including the Tesla cloud.

3 Overview

3.1 Objective

Reverse engineering (RE) of binaries is fundamental in computer security. In addition to binary code comprehension [48, 68], RE has been the fundamental building block of many security applications, including but not limited to vulnerability discovery [27, 28, 61, 71, 78], malware analysis [25, 33, 35, 73, 74], binary retrofitting [21, 29, 54, 55, 70, 72, 75] , and exploit generation [24, 36, 37, 61]. In this paper, we present QTRE, a static binary analysis tool to facilitate reverse engineering of Qt binaries. In particular, it

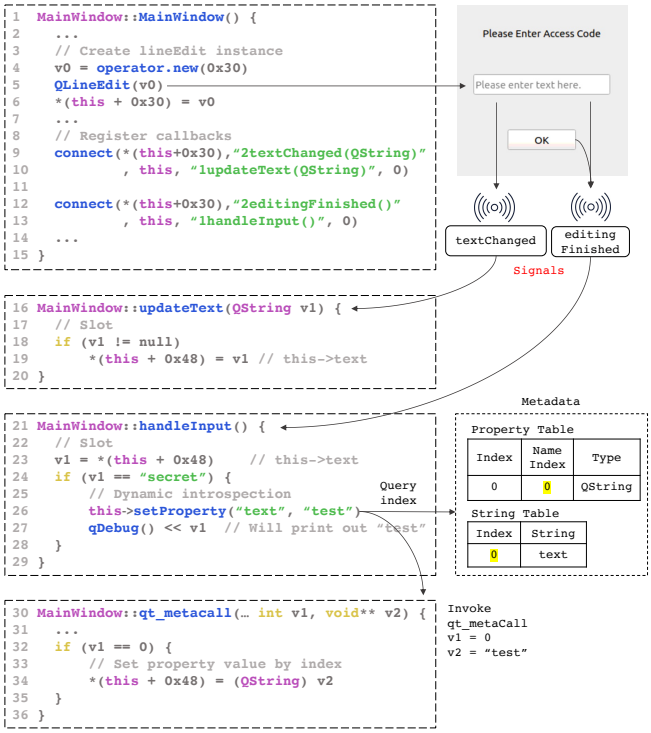

Figure 1: An example illustrating Qt binary internals.

attempts to address two fundamental challenges of RE: (1) control flow graph (CFG) recovery and (2) symbol recovery.

3.2 Key Insights

The recovery of CFG and symbols is fundamentally challenging due to indirect control flow transfer and code stripping [27, 53, 67]. Moreover, while existing analysis tools [3–5, 61] can be applied to Qt binary analysis, they will miss many Qt-specific callbacks and stripped symbols. Interestingly, we observe that two unique Qt mechanisms: (1) signal and slot, and (2) dynamic introspection can be leveraged for QTRE's objective, leading to two key insights. First, Qt's signal and slot mechanism provides a unique way to efficiently implement function callbacks, which can also be used to identify function callback targets for CFG recovery. Second, while Qt's dynamic introspection is for run-time variable query and update, we surprisingly find that such a process can be repurposed to recover symbols. In the following, we present the details of these two key insights.

Insight 1: Leveraging Qt's Signal and Slot. Function callbacks are extremely common in GUI applications due to the handling of UI events and asynchronous function calls. However, C++ provides neither standard APIs nor official guide for callback implementation, and thus programmers have to use function pointers to implement callbacks, which is error-prone and makes the code much less readable and maintainable. As such, Qt introduces the signal and slot mechanism [19] to address this issue. Essentially, signals and slots are functions defined with special macros (`signals` and `slots`), where a signal represents an event that an object fires, and a slot captures the event of its interest.

We further illustrate exactly how Qt's signal and slot work with a running example in Figure 1. At lines 9-13, the program registers two callback functions by invoking a Qt library function `connect`. Using the call site at line 9 as an example, the function takes five parameters as input, including the signal class object (a `QLineEdit` object), the signal function signature (`2textChanged(QString)`)[1], the slot class object (a `MainWindow` instance pointed by a `this` pointer), the slot function signature (`1updateText(QString)`), and the connection type (0, indicating that the callback is synchronous). After the connection is established, when a user enters some text in the `QLineEdit` UI widget, the slot function (lines 16-20) will be automatically notified to update the `text` variable. Fundamentally, the registered callbacks are stored in a `connection list` (i.e., a linked list) of the involved class objects. When a signal is emitted, the class object internally triggers a Qt library function `activate` to invoke the slot function from the connection list [16].

The above example shows that Qt's signal and slot can be used to recover function callbacks, which cannot be identified by other generic binary RE tools [4, 5]. Essentially, the task is to analyze the standard `connect` function and resolve the pair of the signal (*i.e.*, caller) and the slot (*i.e.*, callee) from the function parameters to establish the callback connection.

Insight 2: Repurposing Qt's Dynamic Introspection. Another distinctive feature of Qt is its dynamic introspection, a feature useful for run-time query and update of class attributes. To use dynamic introspection, a class member first needs to be registered as a *property*[2] by using a `Q_PROPERTY` macro [10]. Fundamentally, to support dynamic introspection, the MoC will collect the necessary meta-information and generate the corresponding code during compilation, which will be invoked at run-time for introspection [14, 20].

We use lines 21-29 of Figure 1 to illustrate how dynamic introspection works in Qt. First, the slot `handleInput` at line 21 is triggered by the signal `editingFinished` when the user finishes entering a string from the GUI. The slot then takes the input variable at memory location `this+0x48` (line 23), and compares it with a constant string "`secret`" (line 24). If the variable matches the string, `setProperty` is invoked to set the property value of `text` as "`test`" using dynamic introspection at line 26. Since such an update occurs internally in the Qt library (not directly visible to programmers) with complicated procedures, we explain it in a simplified manner. Specifically, the program first uses the property name "`text`" to query its index from the metadata tables. By associating the name index 0 from the string and property table, it

[1] Constants `1` and `2` are macros to indicate a slot or a signal function.
[2] A property is essentially a class member with additional features.

obtains the property index which is also 0. Next, the function `qt_metacall` is invoked along with the property index 0 and the updated value "`test`" as arguments, and the purpose is to store the value to the property's memory address at `this+0x48` (line 34). Finally, the program at line 27 will print out "`test`" on the console even though the input string is "`secret`", indicating that the variable has been updated.

Based on how dynamic introspection works, we notice that unlike C++ binaries, Qt binaries must preserve semantic symbols to support this feature, which makes the recovery of actual semantic symbols possible. More specifically, by repurposing the introspection process, we can reveal the semantic symbols from the special data structures (*e.g.*, metadata tables) and functions (*e.g.*, `qt_metacall`) in Qt. Compared with existing symbol recovery approaches [38, 44, 45, 59, 62, 66, 69, 70], our solution *recovers* the symbols instead of *inferring* them, as the recovered symbols are indeed the ones from the source code.

3.3 Scope and Assumption

We focus exclusively on Qt binaries and assume these binaries are not stripped, and no anti-RE techniques are deployed so that they can be disassembled using existing RE tools such as Ghidra. For CFG recovery, we aim to identify callbacks implemented by Qt's signal and slot as they are Qt-specific, whereas other CFG recovery challenges such as indirect function calls are handled by existing approaches [46, 53, 67] and are not unique to Qt. For symbol recovery, QTRE recovers symbols that can only be extracted using Qt's dynamic introspection.

4 QTRE Design

This section presents the design of QTRE. First, we illustrate how Qt's signal and slot are used to recover function callbacks (§4.1). Next, we describe how Qt's dynamic introspection is repurposed to recover semantic symbols (§4.2).

4.1 Identification of Function Callback Target

Challenges. Recall in §3.2, the essence of function callback identification is to resolve the signal and slot [19] from the `connect` function parameters. As shown in Table 1, the `connect` function has five parameters: (1) the signal class instance, (2) the signal function, (3) the slot class instance, (4) the slot function, and (5) the connection type. Although the signal and slot functions can be easily resolved as they are hardcoded strings that represent the function signatures, this is still not sufficient due to the polymorphism in C++ [53]. For instance, if class `A` has a function `foo`, then any class inherited from `A` can override function `foo`. Thus, only knowing the function signatures cannot accurately resolve the callback target, and we must use both the signature and class to uniquely identify the target. However, as shown in the running example, there are various ways to derive a class instance. For example, the signal class object at line 9 is initialized by a `new` operator (essentially a heap variable), whereas the slot class object is pointed by a `this` pointer.

Type	Name	Param.0	Param.1	Param.2	Param.3	Param.4
		Signal Class	Signal Sig.	Slot Class	Slot Sig.	Type
1	connect	QObject*	fptr*	QObject*	fptr*	int
2	connect	QObject*	char*	QObject*	char*	int

Table 1: Connect functions and argument types.

Solution. While there are many ways to derive a class object, we observe that there are a finite number of sources. In summary, there are six different sources, including (1) `this` pointer, (2) function parameter, (3) function return value, (4) global variable, (5) heap variable, and (6) stack variable. Therefore, to resolve the signal and slot classes, we first trace the use-def [22] chains of the corresponding parameters in the `connect` function. Next, based on the data definition of the class object, we use a set of source-aware inference rules to infer the class. In the following, we illustrate in greater detail how QTRE identifies function callbacks.

4.1.1 The `connect` Call Sites Identification

The first step of callback identification is to locate the `connect` function call sites. To achieve comprehensiveness, we exhaustively looked into the Qt's official documentations and found that there are only two types of `connect` functions (*i.e.*, type-1 and type-2) [19], as presented in Table 1. For both functions, the parameters 0 to 3 correspond to the signal class object, the signal function signature, the slot class object, and the slot function signature, respectively. The last integer parameter indicates the type of connection (whether the connection is synchronous or asynchronous). These two functions are different in parameters 1 and 3, as type-1 directly uses function pointers to denote the functions, whereas type-2 uses strings as function signatures. Note that type-1 is only available after Qt version-5 [19]. As a result, QTRE locates these two types of `connect` functions in the binary based on their signatures and then identifies their call sites.

4.1.2 Source-aware Class Inference

QTRE then resolves the signal and slot from the parameters of `connect`. Since there are two types of `connect` functions, QTRE uses two strategies accordingly: (1) for type-1 `connect`, by resolving the addresses to which the function pointer points, QTRE can determine the signal and slot. Therefore, it only needs to resolve parameters 1 and 3 in Table 1; (2) for type-2 `connect`, QTRE has to resolve all the pointer types (*i.e.*, parameters 0-3) including the class object pointers, since a callback target is determined by both the class and the function signature due to the polymorphism.

$$\textsc{ThisPointer}\ \frac{p = this}{this \mapsto class} \quad \textsc{FuncParam}\ \frac{p \mapsto v \quad v \in Parameter(f)}{Type(v)} \quad \textsc{FuncRetVal}\ \frac{p \mapsto v \quad v = f(...)}{ReturnType(f)}$$

$$\textsc{GlobalVar}\ \frac{p \mapsto v \quad v \in GlobalVariables}{Type(v)} \quad \textsc{HeapVar}\ \frac{p \mapsto HeapAlloc(v,size)}{Constructor(v)}$$

$$\textsc{StackVar}\ \frac{p \mapsto v \quad p \in Stack}{Type(v)} \quad \textsc{SigMatching}\ \frac{\exists! f,\ Signature(f) = signature}{Class(f)}$$

Figure 2: Formal representation of source-aware class inference rules for callback target identification.

However, it is not so straightforward to resolve these parameters, and QTRE has to perform a static analysis and trace back to the data definitions. Specifically, for each parameter, QTRE recursively traverses backward the use-def chains until the data definition is reached. As such, it easily obtains the signatures of the signal and slot functions of type-2 `connect` as most of the `char` pointers point to hardcoded strings. For the signal and slot class objects, QTRE further infers their classes based on the data definitions. In summary, there are six sources that can derive a class object according to our observation, and QTRE also has one additional signature matching inference rule (inspired by TypeArmor [67]) when it fails to infer the class from these six sources. The formal representations of the source-aware class inference rules are presented in Figure 2 and explained in the following.

- **This pointer**. If the class pointer p is a `this` pointer (*e.g.*, the slot class object at line 10 in Figure 1), then the corresponding class of the `this` pointer is used as the class type.
- **Function parameter**. If p points to a parameter of function f, the class type is inferred from the corresponding parameter type.
- **Function return value**. If p points to the return object of f, QTRE uses the return type of f.
- **Global variable**. If p points to an object located in data segments where global variables reside (*e.g.*, bss), QTRE traces the definition of the global variable to infer its type.
- **Heap variable**. If p points to a heap object v initialized through heap allocators such as `new` (*e.g.*, line 4 in Figure 1) or `malloc` family, QTRE searches the def-use chain of v for any constructor functions, since a heap variable must be initialized through a constructor (*e.g.*, line 5 in Figure 1). The class is thus inferred from the constructor name.
- **Stack variable**. If p is a pointer located on the stack, QTRE traces the definition of its pointed variable v and uses the above rules to recursively resolve the class type.
- **Signature matching**. If none of the above rules works (*e.g.*, the class object is the returned from a function that cannot be concretely resolved), QTRE uses an additional rule by matching the function signature, which is inspired by TypeArmor [67]. Specifically, if there exists only one function f that has exactly the same signature, the class of f is our target. Otherwise, if there are multiple fs, QTRE will not establish a callback to ensure soundness.

The type information is obtained directly from the symbols (e.g., the types of function parameters and return values) if available. If the binary is stripped, our approach still works by recursively applying the inference rules (e.g., tracing the definition of a function parameter at the call site). Additionally, the symbols can be contributed from QTRE's recovery approach (§4.2) as those cannot be fundamentally stripped. After QTRE infers the class types and function signatures using the inference rules, it constructs the concrete function callback targets (*i.e.*, signal and slot) by connecting the inferred class (*e.g.*, `QLineEdit`) with the function signature (*e.g.*, `textChanged(QString)`). Finally, to ensure the soundness of the results, QTRE establishes a callback connection only when both the signal and slot are resolved.

4.2 Semantic Symbol Recovery

Challenges. Next, QTRE recovers semantic symbols by repurposing Qt's dynamic introspection. First, as illustrated in our running example (Figure 1), the symbol strings are stored in metadata tables, and thus QTRE interprets them to extract the symbol strings, as demonstrated in an existing tool [14]. Afterwards, with the symbol strings extracted, we still need to map the property symbols to the corresponding memory addresses (*e.g.*, `text` is mapped to `this+0x48` in Figure 1). However, this is challenging for two reasons. First, the symbol addresses are not available from the metadata tables, and are hidden in deep program branches of the Qt binary code (*e.g.*, line 34 of Figure 1). Thus, we infer such a symbol-address association using a sophisticated binary analysis. Second, the symbol addresses are dynamically computed, which are derived from a base pointer `this` (lines 9 and 12 of Figure 1), making them non-trivial to solve statically.

Solution. Motivated by the running example, our key insight is that the *symbol to address* mapping must exist in the Qt library function `qt_metacall` to support the run-time query of class properties, which can be leveraged to compute the symbol addresses associated with the symbol strings. Specifically, in Figure 1 the `setProperty` function invokes `qt_metacall` which guides the program to update variable `text` at `this+0x48`. Although some static analysis approaches, such as data flow analysis [23] could be applied, it falls short due to the excessive number of branches in the `qt_metacall` function and dynamically computed values

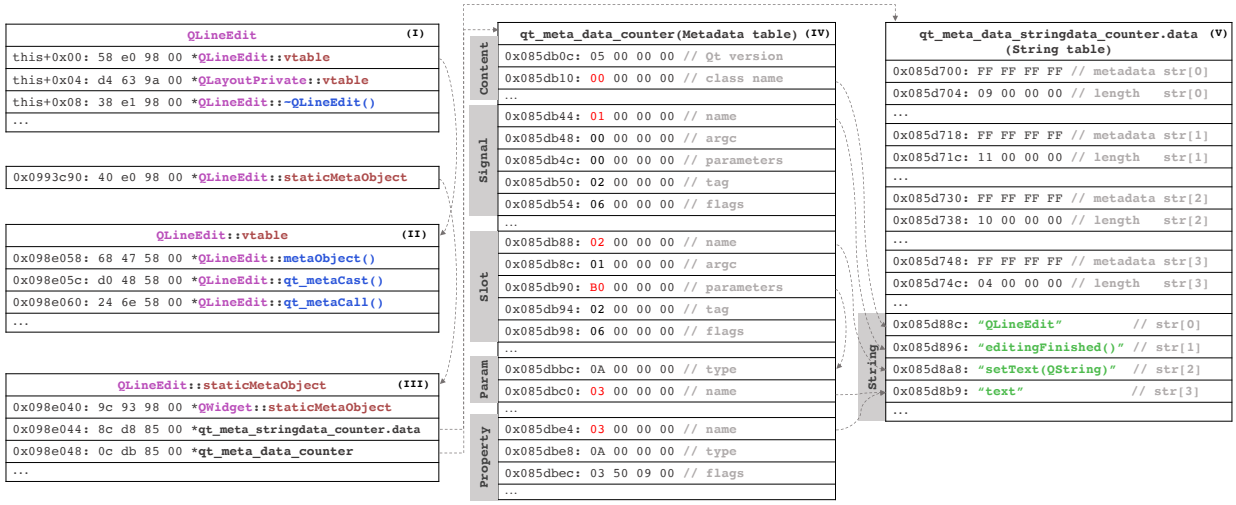

Figure 3: Tables and data structures that store the metadata information of the QLineEdit class in our running example.

(e.g., `this` pointer), making them difficult to design and implement. In contrast, we find that a light-weight unit-level symbolic execution [42] is well-suited for this problem, as we can leverage the code transition and arithmetic computation logic in function `qt_metacall` to efficiently compute the relative symbol addresses. Back to Figure 1, we apply this idea by executing `qt_metacall` with the property index 1 (`v1`) and a symbolic value assigned to the `this` pointer, which guides the execution to line 34 and computes the desired symbol address. In the following, we describe how QTRE extracts the symbol strings and computes the symbol addresses.

4.2.1 Symbol String Extraction

To extract the symbol strings, QTRE interprets 3 key data structures and 2 metadata tables. In Figure 3, we present a real-world example of the `QLineEdit` class to explain how its symbol strings are extracted from these five elements:

(I) **Object hierarchy**. For each instance of class `QLineEdit`, the hierarchy starts with its virtual function table pointer, followed by other pointers, functions, and member variables. Note that the class also has a pointer pointing to a static member `staticMetaObject`, which contains pointers pointing to the metadata tables of the class, as illustrated in Figure 3.

(II) **Virtual function table**. Whenever a class defines a virtual function (*i.e.*, a function defined at the base class but overridden in child classes), a virtual function table will be generated and shared among all class instances. All classes in Qt that inherit from the base class `QObject` will have a virtual table, which contains built-in Qt functions such as `qt_metacast` and `qt_metacall` [16].

(III) **Static meta object**. There is a static member named `staticMetaObject` shared among all instances of the same class. This member is created automatically by the MoC when a class instance inherits from `QObject` [11]. As shown, `staticMetaObject` is a data structure that consists of three pointers [16]. The first pointer points to the `staticMetaObject` of its parent class `QWidget`. The second pointer points to the data section of a structure called `qt_meta_stringdata_Counter`, which contains the metadata about the strings used by this class. The third pointer points to the `qt_meta_data_Counter` structure, which involves the metadata of its signals, slots, parameters, and properties in different data sections. For simplicity, we call the latter two string table and metadata table, respectively.

(IV) **Metadata table**. The metadata table consists of five sections [16] and stores the metadata of the class, signals, slots, parameters, and properties that are used in the class.

- **Content section** includes information about the Qt version, class name, number of methods, properties, and signals in the class, *etc.* For instance, the class name is represented by an index 0. By looking up the index 0 at the string table, we can know the class name is `QLineEdit`. Interpreting the content table is quite simple, as each element is of type `UInt`, and the size of the table is fixed.
- **Signal section** is right after the content section, which stores the metadata of the signals in the class. It consists of multiple signal entries, each of which has five `UInt` elements to describe a signal. For simplicity, we only show one entry in Figure 3 (the same as below). As shown, this entry has five elements, representing the signal name, argument count, parameter index, tag, and flags. Similarly, the signal name is represented by an index 1, which is interpreted as `editingFinished()` at the string table. In addition, if the signal function has parameters, the parameter entry offset (relative to the entry of the metadata table) will be specified. Since the signal does not have parameters, the parameter index is 0. The remaining two elements represent the tags and flags of the signal. Note that each signal is also indexed according to the order in which it appears (*e.g.*, `editingFinished` has an index of 0).

- **Slot section** is identical to the signal section, except that each entry represents a slot. Thus, QTRE uses the same strategy to interpret the slot table. As in the example, we can interpret the slot as `setText(QString)` with one parameter. In addition, the offset for parameter entry is `0xb0`, which refers to the entry `0x085dbbc` whose offset is `0xb0` from the entry of the metadata table at `0x085db0c`.
- **Parameter section** stores the metadata for the parameters of the signal and slot functions. Similarly, it also consists of entries for each parameter, but the size of the section depends on the number of parameters in the function. For instance, in the example, the entry at `0x085dbbc` has the name index 3 and type `0x0A`, which are interpreted as `text` and `QString` (based on the special type encoding in the Qt source code [16]). Combining with the extracted slot function, we can construct a complete function signature: `setText(QString text)`.
- **Property section** stores the metadata for the defined properties (*i.e.*, class members that have been registered as Qt properties). This section also has entries, and each of them has three `UInt` elements that denote the name, type, and flags, respectively. Using the same parsing strategy, we know that the property at `0x085dbe4` has the name `text` of `QString` type. The `flag` indicates the operational attributes, such as whether the property is readable. Note that each property is also uniquely identified by an index.

(V) **String table** consists of a list of strings used in the class and their metadata. Specifically, it starts with the string metadata and each entry contains 24 bytes (including attributes such as length). Next to these metadata there is the string section, which consists of a list of strings, and each of them is assigned an index for reference (*e.g.*, `editingFinished()` is indexed by 1). The string table can be easily located, as its pointer is the second element of `staticMetaObject`.

4.2.2 Symbol Address Computation

To understand how symbol addresses are computed, we present the code logic of `qt_metacall` to show how it handles the dynamic introspection of the `QLineEdit` class in Figure 4. Note that `qt_metacall` is generated automatically by the MoC in each Qt class with the same structure, and thus our unit-level symbolic execution handles it uniformly. In general, `qt_metacall` takes three parameters: an integer `call`, an integer `index`, and a pointer `a`. The first parameter `call` specifies the intention of this call, and there are three types as shown in the figure: 0 is for function invocation, 1 is for property value retrieval, and 2 is for property value assignment [16]. Based on the `index` parameter, this function further executes the corresponding branch to perform specific operations. For example, if we need to obtain the value of property 0 (*i.e.*, property `text` as shown in Figure 3), we invoke this function with `call` and `index` set to 1 and 0, respectively. As a result, this function will be directed to line 12 through a series of `if` and `switch` branches, and retrieves the property from a relative address (`*(*(*(this+4)+300)+8)`), and stores the value in `**a` (line 13). During the execution of this code logic, the address of the property `text` is revealed, since this function must retrieve the value from its address.

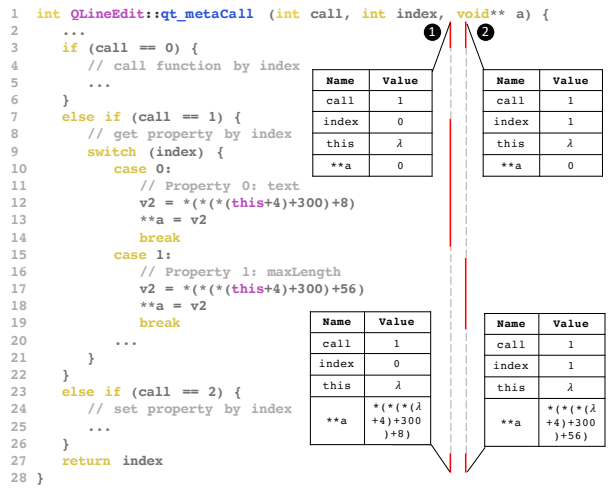

Figure 4: A typical example of `qt_metaCall` function and two example symbolic execution traces ❶ and ❷.

Based on the observations, we present a unit-level symbolic execution in algorithm 1 to compute the symbol addresses for a specific member at `index` in class c. Unlike normal symbolic execution, which aims to solve the constraints for reaching specific program states [42], our goal is to compute only the relative address expressions derived from the `this` pointer. To better understand how it works, we also provide two concrete symbolic execution examples (❶ and ❷) in Figure 4. To begin with, our algorithm initializes three parameters from line 2 to line 4. It assigns a symbolic value λ to `this` pointer, sets the `call` parameter as 1 and the `index` parameter as the target property index. For example, as shown in the top two snapshots in Figure 4, trace ❶ and ❷ set the `index` to 0 and 1 for properties `text` and `maxLength`, respectively. Starting from line 5, the algorithm starts to execute `qt_metacall`. For each instruction i to be executed, if the source operand s of i is symbolic (*i.e.*, calculated from λ), the algorithm needs to further compute the symbolic expression and propagate it to the destination operand d (line 9-line 15). For example, if the *opCode* of i is arithmetic (*e.g.*, *ADD*) or logical (*e.g.*, *AND*), we compute d based on the operation (*e.g.*, $\lambda \rightarrow \lambda + 4$ for *ADD*). Similarly, for other *opCode* (*e.g.*, data movement), the algorithm will compute the symbolic expression based on the instruction semantics (*e.g.*, $\lambda \rightarrow *\lambda$ for *LOAD*). However, if s is not symbolic, we take advantage of an emulator to execute instruction i (line 19) and proceed to the next instruction (line 23). Note that a variable could have multiple symbol addresses computed at different branches (*e.g.*, the branch condition is symbolic), and the algorithm

executes each of them to derive each address accordingly (line 17). Eventually, when i is the return instruction of `qt_metacall`, then the algorithm ends (line 21) and retrieves the relative symbol address (a symbolic expression) from `**a` and returns (line 25). As shown in the bottom two snapshots in Figure 4, when the execution ends, the value of `**a` is exactly the symbol address expressions `*(*(*(λ+4)+300)+8)` and `*(*(*(λ+4)+300)+56)` in trace ❶ and ❷.

5 Implementation

We have implemented a prototype of QTRE[3] with 5K lines of Java code based on Ghidra [4]. One key reason of building atop Ghidra is its cross-platform support due to its P-Code Intermediate Representation (IR) [8], which enables the analysis of both ARM (Tesla firmware) and x86 (KDE) binaries.

P-Code IR. P-Code is a register-transfer language that can translate processor instructions of different architectures into unified operational P-Code instructions [8]. For instance, function invocation instructions such as `CALL` in x86 and `BL` in ARM will both be translated to a `CALL` instruction in P-Code with a function address as its operand. Another key reason of choosing Ghidra is that it provides APIs to obtain the data def-use chain based on the P-Code, which is extremely useful for our static analysis, such as the source-aware class inference. Particularly in P-Code IR, each program variable (*e.g.*, a register and a memory address) will be translated into a built-in type `Varnode`, which will then be formed into an abstract syntax tree `VarnodeAST` that defines the def-use relations among all `Varnodes`. Using APIs such as `getDef()`, we can easily obtain the data-def of a `Varnode` object.

Symbolic Execution. While Ghidra does not directly support symbolic execution, there are many other alternatives such as Angr [61]. However, we still prefer to implement symbolic execution with Ghidra for two reasons. First, our proposed symbolic execution is light-weight and performs within the `qt_metacall` function, and thus is different from the existing symbolic execution and easier to implement. Existing symbolic executions can cause overhead, such as constraint solving, which is unnecessary for QTRE. Second, implementing symbolic execution on top of Ghidra allows it to be easily integrated with other components (*e.g.*, the callback recovery component) that are developed using Ghidra.

To implement our symbolic execution based approach for symbol address computation, we need to interpret each P-Code instruction semantics and execute them symbolically, which requires significant engineering work. Fortunately, Ghidra provides a built-in emulator that can automatically interpret and execute P-Code instructions, which serves perfectly as the building block for QTRE. Nevertheless, we still need to implement our emulator-based symbol address computation algorithm, as Ghidra does not support it. In particular, Ghidra's program emulator allows programmers to operate (*e.g.*, read, and write) on registers and memory during the execution of each instruction, by using APIs such as `writeRegister()`. For instance, we can specify the starting point of our symbolic execution by setting the value of the PC register to the entry address of our target function (*i.e.*, `qt_metaCall` for each class of our interests), where we initialize the function parameters by assigning values to the registers, such as assigning a symbolic value to the `this` pointer (*e.g.*, the `R0` register in ARM). To dynamically compute the relative address based on the symbolic values, we instrument each instruction logic to realize the computation with symbolic addresses. Finally, we retrieve the symbol address stored in the `**a` variable.

Algorithm 1: Symbol address computation.

```
   Input  : F: qt_metacall for class c, index: the property index
   Output : ADDR: the symbol address of property at index in c
1  Function
2      writeRegister(this, λ) ;                          //Init this as λ
3      writeRegister(param0, 1) ;                    //Init call param as 1
4      writeRegister(param1, index) ;                  //Init index param
5      while true do
6          i ← currentInstruction
7          s ← i.sourceOperand
8          d ← i.destinationOperand
9          if s.isSymbolic() then
10             if i.opCode is Arithmetic or Logical then
11                 d ← s < opCode > i.operand
12             else if i.opCode is DataMovement then
13                 d ←< opCode > s
14             else
15                 d ← compute based on instruction semantics
16         else
17             if i.opCode is Branch/Jump then
18                 Fork both branches for the execution
19             emulate(i)
20         if i == return instruction of F then
21             break ;                                   //Execution ends
22         else
23             Go to next instruction
24     ADDR ← **a
25     return ADDR
```

6 Evaluation

We seek to answer the following four research questions:

- **RQ1.** What is the false positive (FP) and false negative (FN) rate for callback and symbol recovery in QTRE?
- **RQ2.** How many callbacks and semantic symbols can be recovered by QTRE from Qt binaries?
- **RQ3.** How efficient is QTRE to analyze Qt binaries?
- **RQ4.** How is QTRE's performance compared to other existing RE tools?

To answer **RQ1**, we use the open-source KDE [6] code base as ground truth data to validate the results of QTRE (§6.1.1). To answer **RQ2**, we applied QTRE to both open-source KDE programs and closed-source infotainment software from a Tesla Model S (§6.1.2). To answer **RQ3**, we evaluate how

[3]The source code of QTRE is available at https://github.com/OSUSecLab/QtRE.

Binary Name	Source Repo.	# Ins.(K)	Time(s)	Callback Recovery				Symbol Recovery							
				# Total	# Recover	%	# Valid	# Prop.	# Signal	# Slot	# Param.	# Total	# Recover	%	# Valid
libKF5KHtml	khtml	104,234	965	192	74	38.5%	74	9	14	96	68	187	185	98.9%	185
libKF5GlobalAccel	kglobalaccel	63,560	244	4	1	25%	1	8	1	3	10	22	22	100%	22
libKF5KIOCore	kio	119,425	96	146	137	93.8%	137	6	42	9	46	103	103	100%	103
libKF5ItemModels	kitemmodels	14,906	26	51	42	82.4%	42	13	19	29	61	122	122	100%	122
libKF5IconThemes	kiconthemes	185,804	43	21	8	38.1%	8	3	4	13	9	29	29	100%	29
libKF5CompactDisc	libkcompactdisc	112,781	4	3	1	33.3%	1	0	9	17	17	43	43	100%	43
libKF5KDEGamesPrivate	libkdegames	185,791	34	46	37	80.4%	37	0	44	25	101	170	170	100%	170
libKF5Baloo	baloo	137,019	97	4	3	75%	3	20	10	0	0	30	30	100%	30
libKF5KDEGames	libkdegames	185,789	52	28	24	85.7%	24	25	17	16	20	78	78	100%	78
libKF5People	kpeople	112,808	10	9	6	66.7%	6	0	2	2	5	9	9	100%	9
libKF5ItemViews	kitemviews	152,312	13	27	11	40.7%	11	7	6	17	25	55	55	100%	55
libKF5DBusAddons	kdbusaddons	29,806	4	6	5	83.3%	5	0	8	2	11	21	21	100%	21
libKF5IdleTime	kidletime	119,421	2	3	1	33.3%	1	0	5	14	9	28	28	100%	28
libKF5GuiAddons	kguiaddons	56,404	6	8	8	100%	8	5	18	0	21	44	43	97.7%	43
libKF5Bookmarks	keditbookmarks	180,235	17	44	38	86.4%	38	0	10	6	16	32	32	100%	32
libKF5KIOFileWidgets	kio	185,799	199	204	202	99%	202	0	37	76	78	191	191	100%	191
Total	N/A	2,093,399	1,568	796	598	75.1%	598	96	246	325	497	1164	1161	99.7%	1161

Table 2: Validated result of function callback recovery and semantics recovery in KDE ground truth evaluation.

long QTRE takes to analyze the KDE and Tesla binaries (§6.2). To answer **RQ4**, we compare QTRE with two state-of-the-art RE tools GHIDRA [4] and ANGR [61] (§6.3).

Qt binary acquisition. KDE is an open-source Linux GUI desktop environment [6], and we use the KDE Plasma desktop image of version 21.04, which is based on Qt 5.79. We extracted 1,018 Qt binaries (including libraries) from the image, and further filtered them by scanning whether `connect` or `qt_metacall` function is used (as the binary code is not obfuscated), which confirms that each of them has at least one function callback or symbol. Finally, 80 binaries are selected for evaluation. The Tesla firmware was extracted from a real CID device (originally as a part of a Tesla Model S). The CID runs 2.52.22 (v8.0) firmware version based on Qt 4.7.2, which was released in 2017. The Tesla firmware contains 54 Qt binaries in total. We applied the same filtering strategy as KDE, and got 43 binaries for evaluation. In total, we obtain 123 binaries from the two binary suites. We notice that more KDE binaries were filtered due to the absence of Qt callbacks and symbols, compared with the Tesla binaries, because these two binary suites were engineered differently. Specifically, among the 1,018 binaries in KDE, most of them are generic C++ binaries and do not contain Qt classes, which are mainly for non-GUI functions (*e.g.*, back-end logic). In contrast, Tesla binaries tend to use Qt for both front-end and back-end logic.

Experiment environment. QTRE's analysis was conducted on an Ubuntu 18.04.4 LTS desktop. The machine is equipped with 12 Intel i7-8700 CPU cores and 32 GB RAM.

6.1 Effectiveness

6.1.1 Quantifying FP and FN with KDE Programs

Before presenting how many callbacks and symbols QTRE can recover, we first evaluate its effectiveness, *i.e.*, the false positive (FP) and false negative (FN) rates. As such, we use the open-source KDE binary suite [6] as our benchmark with 80 Qt binaries in total. While we wish to use all of them and develop an automatic validation tool, manual effort is unfortunately inevitable. The major challenge lies in the validation of symbol addresses that are only available in the binary code and are often derived through convoluted program logic such as nested function calls and branch statements, which is difficult to automate even with debug symbols (*e.g.*, DWARF) available. Therefore, we select a portion of the binaries for manual validation using the following criteria. First, we exclude standard Qt binaries (*e.g.*, libQt5Svg) and only include those directly compiled from the KDE source code. Second, we choose binaries that have both at least one callback and a recovered symbol, so that they can be used for both callback and symbol validations. By applying these two criteria, there are eventually 16 binaries selected for validation, which took us two weeks to complete. The results are reported in Table 2 and we detail them as follows.

Function callbacks validation. The validation of function callback is by comparing the recovered callback targets with the corresponding parameters of the `connect` function in the source code. According to the bottom row of Table 2, among the 16 binaries, QTRE successfully identified 598 callback instances in total, and *all of them are correct*. However, there are 796 callback instances in total among the 16 binaries (by counting call sites of the `connect` function), indicating that there are 198 callback instances that QTRE did not identify (*i.e.*, a 25% FN rate). However, later (§6.1.2) we show that the FN rate is significantly lower when we measure the FN rate for all 123 binaries in our dataset.

We further investigate the root causes of these FN cases, and find that QTRE failed to accurately infer either their signal or their slot classes, and thus did not construct callback relationships as described in §4.1. However, the function signatures in these cases were successfully recovered (as they are mostly hardcoded strings), which are still useful, as they can significantly narrow down the possible callback targets based on the signature strings. To summarize, there are two major causes that fail the source-aware class inference: (1) memory aliasing where QTRE could not find the data source (*e.g.*, QTRE resolves the pointer of a class object which is

Result	KDE		Tesla	
	#	%	#	%
# Total Binary	80	100%	43	100%
Callback Recovery				
├─# Total callback	3,972	100%	8,845	100%
└─# Recovered callback	3,323	83.7%	7,544	85.3%
├─# Type-1 connect	2,992	75.3%	0	0
└─# Type-2 connect	331	8.3%	7,544	85.3%
Source of Class Objects				
├─ # This pointer	275	4.1%	7,433	49.3%
├─ # Function parameters	113	1.7%	295	2.0%
├─ # Function return value	88	1.3%	1,371	9.1%
├─ # Global variable	83	1.2%	3,509	23.3%
├─ # Stack variable	5,984	90.0%	389	2.6%
├─ # Heap variable	88	1.3%	652	4.3%
└─ # Signature matching	15	0.2%	1,439	9.5%
Symbol Recovery				
└─# Total recovered symbols	4,362	100%	20,611	100%
├─# Property	817	18.7%	951	4.6%
├─# Signal	1,182	27.1%	9,266	45.0%
├─# Slot	841	19.3%	3,326	16.1%
└─# Function parameter	1,522	34.9%	7,068	34.3%

Table 3: Results of callback and semantics recovery.

initialized by another pointer pointing to it); (2) indirect calls where a class object to be inferred is the return value of a function that is indirectly called (*e.g.*, through a `CALL EAX` instruction). We note that these are common limitations in binary analysis (not unique for Qt binaries), and there have been many solutions such as aliasing analysis (*e.g.*, [30]) and Multi-Layer Type Analysis (*e.g.*, [41, 46]), which could be integrated in QTRE in future work. Therefore, *in short, there is no false positive in the function callback recovery among the* 598 *validated callbacks, while there exist* 25% *false negatives due to memory aliasing and indirect calls.*

Semantic symbols validation. The validation of semantic symbols is by comparing the symbols of signals, slots, parameters, and properties with those in the corresponding source code, including their names, types, and relative addresses. However, since the relative addresses are available only in binaries, we have to use both the source code and the decompiled binary code for the validation. As presented in Table 2, among the 16 binaries, there are 1,164 semantic symbols in total (by counting the entries at all symbol tables), and QTRE successfully recovered 1,161 of them, in which *all of them are correct, including the symbol names and relative addresses*.

For the remaining three (0.3%) cases, they are all property symbols, and QTRE did not successfully recover their relative addresses (though their symbol strings are still correctly recovered). The root cause is that the addresses of these three properties are returned from a virtual function call [32] where QTRE cannot precisely locate the function address as it is dynamically computed (derived from the `this` pointer). We consider these three cases as FNs since QTRE did not attempt to generate incorrect property addresses but instead left them unresolved. As these FN cases only account for a small portion of our results (three out of 96 properties in total), addressing the indirect call limitation is thus left to future work. On the other hand, there is no false positive as

Type	Symbol String
QString	text, key, ssid, plainText, carName, country, name, reason, response, message, location, keyword, command phoneNumer, debugMessage, errorMessage
QUrl	url, baseUrl, requestedUrl
QByteArray	data, userData
int	sec, id, pId, deviceId, securityType, canID, canData
bool	useCarLocation

Table 4: Selected symbols recovered from QTRE that potentially contain sensitive information.

the 1,161 symbols are all validated. *In summary, among the* 1,164 *validated symbols, there is no false positive and there are only three (0.3%) false negatives in the recovered symbol addresses due to indirect function calls.*

6.1.2 Real-World Qt Binaries

We evaluated QTRE using real-world binaries from both KDE and Tesla, as summarized in Table 3. In general, QTRE identified 10,867 callback instances and 24,973 semantic symbols from the 123 binaries in the two binary suites. According to row 1 of the table, there are 80 binaries from KDE and 43 from Tesla firmware. The detailed results of all binaries from KDE and Tesla are presented in Table 12 and Table 13 in Appendix, respectively, for readers of interests. In the following, we zoom in on the results of function callback target identification and semantic symbol recovery.

Recovered function callbacks. According to row 4 in Table 3, among the 80 binaries from KDE and 43 Tesla binaries, QTRE recovered 3,323 and 7,544 function callback instances, respectively. However, the recovered callbacks account for 83.7% and 85.3% among all identified callbacks in KDE and Tesla (obtained by counting the call sites of `connect`), and correspondingly the FN rates among all binaries are 16.3% and 14.7% respectively, which are much lower than the 25% FN rate of the validated callbacks in §6.1.1. These FNs are also caused by the same reasons as in our validation.

In terms of the callback types defined in Table 1, most of the callbacks in the KDE binaries use type-1 connect, while the Tesla binaries use type-2 connect because type-1 connect is only available after Qt5. Furthermore, to show that our source-aware class inference is useful, we further show the contribution of the seven sources (as described in §4.1) in §6.1.2. It can be inferred from the statistics that stack variable is the dominating source among the KDE binaries, and we find that it is frequently used by type-1 connects to store function pointers of signals and slots. In contrast, Tesla binaries tend to use `this` pointers to derive classes.

Recovered semantic symbols. According to row 14 in Table 3, QTRE recovered 4,362 symbols from 80 KDE binaries and 20,611 symbols from 43 Tesla binaries, showing that the Tesla binaries use Qt's metaobject system more frequently than KDE. The recovered symbols include four types: signals, slots, function parameters, and properties.

Result	KDE	Tesla	Total
# Total call graph edges			
├─# Recovered by ANGR [61]	791,907	1,395,093	2,187,000
│ └─# Callback recovered	0	0	
├─# Recovered by GHIDRA [4]	432,843	987,263	1,420,106
│ └─# Callback recovered	0	0	
└─# Recovered by GHIDRA [4] w/ QTRE	436,166	994,807	1,430,973
└─# Callback from QtRE	3,323	7,544	10,867
# Total symbols			
├─# Recovered by ANGR [61]	97,109	1,171,990	1,269,099
├─# Recovered by GHIDRA [4]	97,109	1,171,990	1,269,099
└─# Recovered by GHIDRA [4] w/ QTRE	101,471	1,192,601	1,294,072
└─# Symbol from QTRE	4,362	20,611	24,973

Table 5: Comparison of QTRE, GHIDRA and ANGR.

Note that one slot can be registered to receive multiple signals, which explains that the numbers of signals and slots are not always identical. To show that the recovered symbols are indeed useful for security analysis, we also present the semantic symbols selected manually that potentially contain sensitive information in Table 4. As shown, these symbols indicate sensitive information for confidentiality concerns, including personal identifiable information (*e.g.*, `id`), confidential data (*e.g.*, `canData`), and cryptography parameters (*e.g.*, `key`). One use case of these symbols would be identifying privacy sensitive data leakage through taint analysis [23, 27]. In addition, we show the top 10 most frequent symbols of properties, parameters, and functions in Table 9, Table 10, and Table 11, respectively, in Appendix.

6.2 Efficiency

On average, it takes QTRE 1.7 minutes to analyze a binary. The average time to analyze KDE and Tesla binaries is 1.5 and 3.6 minutes, as Tesla binaries have more classes and functions. In the worst case, QTRE spent 77 minutes analyzing the most complicated binary `libQtCarGUI`, which has almost 2,000 classes and more than 40K functions. To better evaluate QTRE's efficiency on each binary, we present the analysis time of each binary in Table 12 and Table 13 in Appendix for KDE and Tesla binaries, respectively.

6.3 Comparison with Other RE Tools

We compare QTRE with two state-of-the-art open-source RE tools: GHIDRA (v9.2.2) [4] and ANGR (v9.1) [61] that can also be applied to Qt binary analysis. The comparison was conducted to evaluate each tool's capability of recovering CFG (including callbacks) and symbols. The results are summarized in Table 5 and the detailed statistics per binary are presented in Table 12 and Table 13 in Appendix for KDE and Tesla binaries, respectively. As shown in Table 5, we show the total number of function call graph edges and symbols recovered by ANGR, GHIDRA, and GHIDRA with QTRE. As expected, *neither* ANGR *nor* GHIDRA *can recover any of the callbacks and symbols identified by* QTRE, because they are not specifically tailored for Qt binary analysis (*i.e.*, they cannot recognize and interpret Qt-specific code generated by the MoC). To clearly show QTRE's contribution atop GHIDRA, we present the total number of callbacks and symbols QTRE have recovered, which are essentially the 10,867 callbacks and 24,973 symbols (as reported in §6.1.2) from two binary suites. Note that these callbacks and symbols cannot be identified by generic binary RE tools such as ANGR and GHIDRA.

7 Applications

Being an RE tool, QTRE can be useful for real-world security problems. In particular, we demonstrate using QTRE to perform input validation analysis and extract hidden commands from a Tesla Model S firmware, because Tesla vehicles are known to contain many Easter eggs [57]. To perform the analysis, we make use of the recovered callbacks and symbols in §6.1.2. These results are crucial to our analysis as we leverage the recovered symbols to identify the user input variables (e.g., the class members `text`). Additionally, as the hidden commands are triggered by UI operations, it requires the recovered callbacks to construct a complete CFG, and otherwise we cannot identify those hidden commands.

Input Validation Analysis. First, we define a set of rules to identify user-controllable input variables by using recovered symbols. Next, starting from the identified input variables, we perform an automated taint analysis [23] to extract the input validations and resolve the corresponding compared variables.

- **Symbol-guided input variable identification.** According to our observation, the input variables are members of the UI widget classes (*e.g.*, `QLineEdit`), and thus they can appear in (1) *return values of "get" functions* (*e.g.*, `getText()`) and (2) *class members variables* (*e.g.*, `QLineEdit.text`). In addition, they can be members of either (1) *standard Qt library classes* or (2) *programmer-defined classes*, and we need to focus on both. For standard Qt library classes, we manually investigate the official Qt documentation [11] to find the function and the class members that convey user inputs, resulting in six such elements in the Tesla Qt binaries, as shown in rows 1-6 in Table 6. There are one function and five class members, with the corresponding relative addresses shown in the 3rd column.

  However, for the programmer-defined classes, it is not straightforward to recognize the input variables, as there is no documentation to rely on. Thus, we define two criteria to identify the desired classes and locate the input variables by utilizing the recovered symbols. The first criterion is that the class of our interest should have implemented signals that can monitor the input status. For instance, the standard library class `QLineEdit` defines a signal `textChanged()` to notify the callback function to update the text on the screen. Note that all child classes that inherit from these classes will be of our interest since they also hold these variables. The second criterion is that the desired classes should have a member variable to hold the user input, which should

Class Name	Var./Func. Name	Symbolic Address	In Qt Lib?
QLineEdit	text()	N/A	✔
QLineEdit	text	*(*(λ+4)+300)+8)	✔
QAbstractSpinBox	text	*(*(λ+4)+452)	✔
QDoubleSpinBox	text	*(*(λ+4)+452)	✔
QSpinBox	text	*(*(λ+4)+452)	✔
QDateTimeEdit	text	*(*(λ+4)+452)	✔
TextField	text	*(*(λ+796))	✗
PasswordTextField	text	*(*(λ+796))	✗
WebEntryField	text	*(*(λ+796))	✗
NavigationSearchBox	text	*(*(λ+796))	✗
CompleterTextField	text	*(*(λ+796))	✗
ExtEntryField	text	*(*(λ+796))	✗

Table 6: Taint analysis sources and their computed symbolic variable addresses (λ denotes `this` pointer).

be of a string type (*e.g.*, `QString` and `char*`). In addition, these variables should have names inferring that they are of string type and represent certain `text` variable, such as `text`. As shown in row 7-12, we identify six such variables.

- **Taint analysis.** Finally, the standard taint analysis [23] is automatically performed to analyze the input validation, which is implemented based on Ghidra's P-Code IR [8]. Prior to the analysis, we need to define the sources and sinks, which determine where the analysis starts and ends. As shown in Table 6, the input variables (*i.e.*, function return values and the class properties) identified are the sources, and the comparison instructions (*e.g.*, `operator==` and `QString.compare()`) are the sinks. During the analysis, we also consider the control flow transition between the callback targets of the function identified by QTRE. When the taint analysis is completed, QTRE obtains a set of input validation instructions. Therefore, we need to further resolve the compared variables, as they may not be hardcoded and require computation. This problem can be solved in the same way as we resolve the function parameters as described in §4.1. Specifically, QTRE traverses the use-def chains of the compared variable to find the data definition, and further resolve their concrete values.

Experiment Result. By performing taint analysis, QTRE was able to identify seven Easter eggs from the Tesla firmware, four access tokens, and one master password, as presented in Table 7. Although the seven Easter eggs are already known to the public, the remaining five are actually new, to our knowledge. We detail each type of these hidden commands as follows.

- **Easter egg**. The seven Easter eggs can be entered from the CID screen to trigger hidden behaviors on the vehicle, such as changing the CID GUI. These Easter eggs are often benign and do not have much security implication, as they are intentionally designed to entertain users. For example, by entering a string "mars" in the `AccessPopup` UI, the navigation map will become the surface of Mars. Other Easter eggs such as "showroom" and "performance" can enable hidden modes such as service mode.
- **Access token**. The four access token can trigger developer or diagnostic mode, which contains security-sensitive

Category	Content	Description	Vehicle Agnostic
Easter Egg	"007"	Submarine Easter egg	✔
	"modelxmas"	Show holiday lights	✔
	"42"	Change car name	✔
	"mars"	Turn map into Mars surface	✔
	"transport"	Transport mode	✔
	"performance"	Performance mode	✔
	"showroom"	Showroom mode	✔
Access Token	SecurityToken1	Enable diagnostic mode	✗
	SecurityToken2	Enable diagnostic mode	✗
	crc(token)==0x18e5a977	Enable developer mode	✔
	crc(token)==0x73bbee22	Enable developer mode	✔
Master Pwd	"3500"	Exit valet mode	✔

Table 7: Extracted hidden commands in Tesla binaries.

options such as limiting the vehicle speed, resetting the vehicle modules, and reading system logs. Compared to Easter eggs, access tokens are not intentionally left over to users, but are possibly for developers and technicians to debug and diagnose. As shown in rows 8-11 of Table 7, there are four such tokens. The first two can trigger diagnostic mode, and are different security tokens in the local storage, which are unique for each vehicle. The other two tokens that can enable developer mode (with more security-critical functions) are slightly more complicated, as their `crc32` checksum needs to match specific hexadecimal values (*i.e.*, 0x18e5a977 and 0x73bbee22). To search for a feasible input, we wrote a simple brute-force script that only took 30 minutes to generate a valid string "987090324273775".

- **Master password**. QTRE also reported one master password (*i.e.*, "3500") that can exit the valet mode regardless of the password set by the vehicle owner. Specifically, the valet mode is designed to preserve user privacy and vehicle safety when the vehicle is parked by a valet driver. When this mode is enabled, normal functions are no longer accessible (*e.g.*, speed will be limited, and personal data will not be shown on the screen), unless the user enters the 4-digit PIN code to exit the valet mode. Apparently, the master password is also for testing and development purposes and should not be available to any users.

Exploitation. We have successfully validated *all* the hidden commands on a real CID device extracted from a Tesla Model S. To exploit them, we consider any attackers on board with physical access to the CID, such as a hotel valet, a repair shop technician, or a designated driver. For instance, one can leverage the master password to escape the valet mode, or use access tokens to manipulate critical settings (*e.g.*, speed limit). However, these hidden commands require some preconditions. The latter two access tokens and the master password require setting an environment variable `GUI_isDevelopmentCar` as true, and the former two access tokens require to dump the vehicle-specific tokens from file system. These preconditions can be satisfied by using prior exploits (*e.g.*, remote browser exploits [51, 52]) targeting unpatched Tesla firmware.

8 Discussions

False positive and false negative. As described in §6.1.1, we did not observe any FPs among the validated results. However, we did observe 15% FNs among the recovered callbacks and three FNs in symbol recovery, due to indirect calls and memory aliasing, which prevent QTRE from accurately recovering the callback target and symbol address. Another potential source of FP is that the source-aware class inference may not accurately infer the actual class type. More specifically, when QTRE infers a class type *A* from function parameters or return types, the actual class type at run-time can be any class *B* inherited from *A*. However, this does not frequently occur, and we did not observe such a case in our validation.

Future work. As a domain-specific RE tool, QTRE can be applied to many other security-critical Qt applications, such as medical and automotive systems where Qt has played an essential role. Meanwhile, the callback and symbol recovery techniques of QTRE can enable many other security applications. In addition to the symbol-guided input validation analysis demonstrated in this paper, an immediate future work is to integrate QTRE with state-of-the-art fuzzers for GUI-fuzzing [65] of Qt binaries, which can help identify vulnerabilities triggered by external user input.

Responsible disclosure. We reported our findings in §7 to Tesla in November 2021, and received a response in April 2022. The Tesla security team acknowledged our findings, and claimed that they have eliminated the feasible paths for exploiting these hidden commands, such as removing the HTTP API for setting the `GUI_isDevelopmentCar` variable since version 2021.44. For other commands that do not require this precondition, they require invasive physical access (*e.g.*, to leak access tokens from the file system) and thus are difficult to exploit in practice. In conclusion, the best security practice is to keep the firmware up-to-date.

9 Related Work

C++ binary analysis. MARX [53] uses vtables to recover the hierarchy of classes, and DeClassifier [31] extends it to optimized C++ binaries. OOAnalyzer [59] leverages coding patterns to recover classes and methods. VirtAnalyzer [32] reverse-engineers virtual inheritance among C++ classes. OBJDIGGER [40] and PhASAR [58] use inter-procedural data flow analysis for C++ binary analysis. Howard [62], TIE [44], Rewards [45], DSIBin [56], Lego [64] perform dynamic analysis to recover data structures. HexType [39] and TCD [77] are two static analysis tools for detecting type confusion bugs in C++ binaries (including Qt), but they do not make use of any Qt's unique mechanisms.

Qt binary analysis. Although not presented in formal publications, there have been some tools for Qt binary RE [9, 14, 18], in which two of them are open-source [14, 18]. Specifically, one provides an IDA-Python script to parse metadata tables of Qt classes as QTRE does [14], but it requires to identify metadata table entrances manually and does not attempt to recover the relative addresses. Another tool uses heuristics to extract the Qt class hierarchy from memory at run-time [18]. However, as it targets an ancient Qt version (v2), it cannot analyze binaries in our datasets based on Qt4 or Qt5, due to significant changes in class layout [11, 15]. The remaining one [9] briefly mentions some RE observations, including using the `qt_metacall` function to analyze how Qt signals are dispatched. QTRE instead makes use of the `qt_metacall` function logic to compute the relative symbol addresses. In summary, while these tools provide some insight for Qt binary RE, they fail to (1) make any attempts to take advantage of these insights to solve fundamental binary RE challenges and (2) target any newer Qt versions.

Hidden behavior detection. In relation to the extraction of hidden commands, some works also detect hidden behaviors in binary programs. For instance, SUPOR [34] and UiPicker [49] detect sensitive user inputs in the Android app UI. InputScope [76] and FirmAlice [60] reveal backdoors through input validation analysis. AsDroid [35] uncovers stealthy behaviors by contrasting the intended behavior with the descriptions of the user interface. Many other works detect malware by analyzing their hidden behaviors, such as TriggerScope [33], IntelliDroid [73], and MineSweeper [25].

Security analysis of Tesla vehicles. The credentials used in the early versions of Tesla firmware were vulnerable [47], and the over-the-air (OTA) firmware update could be intercepted [17]. In addition, the CID browser and kernel had vulnerabilities that could lead to a remote root shell [51], which enabled attacks on the CAN bus and the autopilot system [43, 52]. Most recently, it has also been shown that Model X's keyless entry system can be hacked for car theft [2], and the autopilot's autonomous driving system is vulnerable to phantom attacks [50]. Meanwhile, none of them attempted to reverse engineer the Qt binaries for the GUI attack surface.

10 Conclusion

In this paper, we make the first look at the reverse engineering of Qt binaries, and develop QTRE, a static binary analysis tool that is capable of recovering function callback targets and semantic symbols based on Qt's unique mechanisms including signal and slot, and dynamic introspection. We have tested QTRE with two suites of Qt binaries: Linux KDE and Tesla Model S firmware, from which QTRE additionally recovered 10,867 instances of callbacks and identified 24,973 semantic symbols among 123 binaries in total. We further demonstrate an application of using QTRE to extract hidden commands from the Tesla Model S firmware, in which 12 unique hidden commands are discovered with five new to the public.

Acknowledgment

We would like to thank the anonymous reviewers and shepherd for their constructive feedback. This research was supported in part by ARO award W911NF2110081, DARPA award N6600120C4020, and NSF award 2118491.

References

[1] Awesome c++. https://github.com/fffaraz/awesome-cpp.

[2] Belgian security researchers from ku leuven and imec demonstrate serious flaws in tesla model x keyless entry system. https://www.imec-int.com/en/press/belgian-security-researchers-ku-leuven-and-imec-demonstrate-serious-flaws-tesla-model-x.

[3] Binary ninja. https://binary.ninja/.

[4] Ghidra. https://ghidra-sre.org/.

[5] Ida pro - hex rays. https://www.hex-rays.com/idapro.

[6] Kde github mirror. https://github.com/KDE.

[7] Language bindings - qt wiki. https://wiki.qt.io/Language_Bindings.

[8] P-code. https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/package-summary.html.

[9] Picturoku: Qt 4 you. http://picturoku.blogspot.com/2011/08/qt-4-you.html.

[10] The property system | qt core 5.15.3. https://doc.qt.io/qt-5/properties.html.

[11] Qobject class | qt core 5.15.7. https://doc.qt.io/qt-5/qobject.html.

[12] Qt | cross-platform software development for embedded & desktop. https://www.qt.io/.

[13] The qt company. https://www.qt.io/company.

[14] Qt internals & reversing. http://www.ntcore.com/files/qtrev.htm.

[15] Qt toolkit - porting to qt 2.x. https://qt.developpez.com/doc/2.3/porting/.

[16] qt5/ source tree. https://code.woboq.org/qt5.

[17] Reverse engineering the tesla firmware update process. https://www.pentestpartners.com/security-blog/reverse-engineering-the-tesla-firmware-update-process/.

[18] Reversing qt applications - and part ii | reversing.org. https://web.archive.org/web/20080703162127/http://www.reversing.org/node/view/7.

[19] Signals & slots | qt core 5.15.7. https://doc.qt.io/qt-5/signalsandslots.html.

[20] Using the meta-object compiler (moc) | qt 4.8. https://doc.qt.io/archives/qt-4.8/moc.html.

[21] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[22] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[23] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[24] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[25] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[26] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[27] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[28] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[29] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[30] Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. Type-based alias analysis. *ACM Sigplan Notices*, 33(5):106–117, 1998.

[31] Rukayat Ayomide Erinfolami and Aravind Prakash. Declassifier: Class-inheritance inference engine for optimized c++ binaries. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 28–40, 2019.

[32] Rukayat Ayomide Erinfolami and Aravind Prakash. Devil is virtual: Reversing virtual inheritance in c++ binaries. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 133–148, 2020.

[33] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE, 2016.

[34] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 977–992, 2015.

[35] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046, 2014.

[36] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87. IEEE, 2012.

[37] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014.

[38] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.

[39] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387, 2017.

[40] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, pages 1–11, 2014.

[41] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[42] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[43] Tencent Keen Security Lab. Experimental security research of tesla autopilot, 2019.

[44] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *18th Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[45] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.

[46] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[47] Kevin Mahaffey. Hacking a tesla model s: What we found and what we learned. *Lookout Blog*, 2015.

[48] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. Re-mind: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.

[49] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 993–1008, 2015.

[50] Ben Nassi, Dudi Nassi, Raz Ben-Netanel, Yisroel Mirsky, Oleg Drokin, and Yuval Elovici. Phantom of the adas: Phantom attacks on driver-assistance systems. *IACR Cryptol. ePrint Arch.*, 2020:85, 2020.

[51] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA*, 25:1–16, 2017.

[52] Sen Nie, Ling Liu, Yuefeng Du, and Wenkai Zhang. Over-the-air: How we remotely compromised the gateway, bcm, and autopilot ecus of tesla cars. *Briefing, Black Hat USA*, 2018.

[53] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *NDSS*, 2017.

[54] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. Razor: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1733–1750, 2019.

[55] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, 2018.

[56] Thomas Rupprecht, Xi Chen, David H White, Jan H Boockmann, Gerald Lüttgen, and Herbert Bos. Dsibin: Identifying dynamic data structures in c/c++ binaries. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 331–341. IEEE, 2017.

[57] David Schmidt. 20 clever easter eggs in tesla cars people don't know about. https://www.hotcars.com/clever-easter-eggs-in-tesla-cars-people-dont-know-about/.

[58] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410. Springer, 2019.

[59] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441, 2018.

[60] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Network and Distributed Systems Security Symposium (NDSS)*, 2015.

[61] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[62] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *18th Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[63] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25. Springer, 2008.

[64] Venkatesh Srinivasan and Thomas Reps. Recovery of class hierarchies and composition relationships from machine code. In *International Conference on Compiler Construction*, pages 61–84. Springer, 2014.

[65] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[66] Hieu Tran, Ngoc Tran, Son Nguyen, Hoan Nguyen, and Tien N Nguyen. Recovering variable names for minified code with usage contexts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1165–1175. IEEE, 2019.

[67] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[68] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.

[69] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[70] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[71] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[72] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 299–308, 2012.

[73] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[74] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.

[75] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, 2013.

[76] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. Automatic uncovering of hidden behaviors from input validation in mobile apps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1106–1120. IEEE, 2020.

[77] Changwei Zou, Yulei Sui, Hua Yan, and Jingling Xue. Tcd: Statically detecting type confusion errors in c++ programs. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 292–302. IEEE, 2019.

[78] Chaoshun Zuo and Zhiqiang Lin. Playing without paying: Detecting vulnerable payment verification in native binaries of unity mobile games. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3093–3110, 2022.

A Empirical Measurement of the Prevalence of C++ Frameworks

We detail how we conducted an empirical measurement study to understand the prevalence of C++ frameworks. First, we obtained a list of C++ frameworks from a comprehensive list [1]. As this list also contains a huge number of utility libraries which cannot be directly used to develop C++ applications (e.g., the STL libraries), we exclude them from the list. Eventually, we have 106 frameworks under 5 categories, namely framework, game engine, GUI, robotics, and web, according to the list. In addition, we also manually added a few popular C++ frameworks (*e.g.*, MFC) missing from the list (as they are not available on Github). Next, we counted the C++ repositories after searching with the framework's name as a keyword on Github, and the top 20 results are shown in Table 8 (as of Jan 2022). Our results show that Qt is dominant among all the studied C++ frameworks, as it has 45,635 C++ repositories on Github, which is nearly 3X of the second place ROS.

Name	Category	# Repository	%
Qt	Framework	45,635	35.70%
ROS	Robotics	16,796	13.14%
Boost	Framework	6,205	4.85%
MFC	Framework	4,409	3.45%
Cocos2d	Game Engine	3,587	2.81%
OpenFrameworks	Framework	3,264	2.55%
JUCE	Framework	2,204	1.72%
PCL	Robotics	1,719	1.34%
imgui	GUI	1,557	1.22%
wxWidgets	GUI	1,076	0.84%
Cinder	Framework	1,042	0.82%
Allegro	Game Engine	958	0.75%
Godot	Game Engine	682	0.53%
GamePlay	Game Engine	561	0.44%
dlib	Framework	547	0.43%
FLTK	GUI	518	0.41%
GTK++	GUI	436	0.34%
LibU	Framework	425	0.33%
raylib	Game Engine	376	0.29%
gtkmm	GUI	349	0.27%

Table 8: Top 20 C++ frameworks from our empirical measurement study.

B Recovered Symbols from QTRE

In this section, we show the top 10 most frequent symbol names of properties and function parameters in Table 9 and Table 10. As shown, the top 3 most frequent properties are `position`, `pressed`, and `title`, and the top 3 parameter names are `index`, `current`, and `text`. In addition, we show the top 10 most frequent names of signals and slots in Table 11 from KDE and Tesla firmware. As shown, the function names are often ended with `changed` and `completed`, which stand for callbacks to monitor the state of a specific variable. For example, whenever a URL gets changed, the corresponding slot `urlChanged` with be invoked to perform specific updates, allowing one to quickly locate the handling logic of the variable.

KDE			Tesla		
Name	**Type**	**Count**	**Name**	**Type**	**Count**
position	double	10	currentIndex	int	8
pressed	bool	9	text	QString	8
title	QString	9	orientation	Qt::Orientation	8
count	int	8	count	int	7
palette	QPalette	7	readOnly	bool	7
visualPosition	double	7	icon	QIcon	7
contentHeight	double	6	alignment	Qt::alignment	7
hovered	bool	6	enabled	bool	7
font	QFont	6	title	QString	6
horizontal	bool	6	iconSize	QSize	6

Table 9: Top 10 property names with types recovered.

KDE			Tesla		
Name	**Type**	**Count**	**Name**	**Type**	**Count**
index	int	55	ctx	ServiceCallContext*	3,114
current	int	43	_rval_	int&	691
text	QString	42	result	bool	144
url	QUrl	37	reason	QString	136
previous	int	37	index	int	116
printerName	QString	32	status	QString	49
printerUri	QString	23	routeID	int	41
role	QByteArray	21	id	int	38
item	QVariant	20	text	QString	35
msg	QByteArray	18	success	bool	34

Table 10: Top 10 parameter names with types recovered.

KDE		Tesla	
Name	**Count**	**Name**	**Count**
positionChanged	14	invokeObjectMethodCompleted	22
orientationChanged	9	findViewCompleted	22
pressedChanged	9	takeScreenshotOfViewCompleted	22
changed	9	flashViewCompleted	22
countChanged	8	changed	20
urlChanged	7	set_valet_modeCompleted	17
visualPositionChanged	7	set_tds_modeCompleted	17
selectionChanged	7	pop_questionCompleted	17
activated	7	reset_valet_pinCompleted	17
iconChanged	7	auto_conditioning_stopCompleted	17

Table 11: Top 10 function (signal & slot) names recovered.

C Detailed Results of Callback and Symbol Recovery in KDE and Tesla Binaries

We present the detailed experiment results of callback and symbol recovery for all KDE and Tesla binaries in Table 12 and Table 13, including the analysis time, number of symbols, call graph edges recovered by ANGR, GHIDRA, and GHIDRA with QTRE. Note that ANGR raised exceptions when analyzing five Tesla binaries for call graph generation, and we use *N/A* to denote the results as in the tables. In addition, we also show the detailed statistics of the callbacks and symbols contributed from QTRE, which cannot be identified by either ANGR or GHIDRA. For callbacks, we show the number of callbacks recovered by QTRE as well as the total number of callbacks (by counting the `connect` call sites). For symbols, we further present the statistics for each category including signals, slots, parameters, and function arguments. As shown in the tables, a Tesla binary has approximately 480 Qt symbols and 206 callbacks on average, which also indicates that the Tesla developers tend to use Qt's callback and meta object system more frequently than KDE.

Binary Name	Time(s)	# Symbol	# CGE	# CGE	# CGE	Callback From QTRE			Symbols From QTRE				
			ANGR	GHIDRA	GHIDRA+QTRE	# Recovered	%	# Total	# Prop.	# Signal	# Slot	# Param.	# Total
libKF5KHtml	965	5,931	140,805	66,208	66,282	74	38.5%	192	9	14	96	68	187
libQt5MultimediaWidgets	37	434	1,888	869	869	0	0%	22	14	11	16	22	63
libkonsoleprivate	269	3,118	24,702	15,322	15,715	393	99.2%	396	0	0	0	0	0
libkbolt	4	275	2,571	1,200	1,205	5	100%	5	0	0	0	0	0
libkworkspace5	181	391	4,119	2,494	2,511	17	100%	17	9	20	12	3	44
libKF5Plasma	48	1,539	12,445	7,474	7,533	59	78.7%	75	0	0	0	0	0
libKF5GlobalAccel	244	271	2,355	1,514	1,515	1	25%	4	8	1	3	10	22
libQt5Svg	73	1,092	5,722	3,103	3,107	4	57.1%	7	6	1	7	11	25
libpolkit-qt5-core-1	4	370	2,145	1,073	1,073	0	0%	1	0	0	0	0	0
libQt5TextToSpeech	62	192	758	367	369	2	100%	2	6	8	9	14	37
libpackagekitqt5	12	416	5,166	2,230	2,236	6	50%	12	0	0	0	0	0
libsignon-extension	2	268	934	414	414	0	0%	3	0	0	0	0	0
libplasmacomicprovidercore	2	151	554	279	285	6	100%	6	0	2	0	2	4
libkdeinit5_klipper	39	996	6,251	3,693	3,764	71	94.7%	75	0	0	0	0	0
liboxygenstyleconfig5	4	281	834	452	457	5	33.3%	15	0	0	0	0	0
libKF5KIOCore	96	2,828	27,104	17,570	17,707	137	93.8%	146	6	42	9	46	103
libkcupslib	62	1,029	5,770	3,564	3,622	58	89.2%	65	11	39	43	230	323
libKScreenLocker	20	544	4,733	2,873	2,919	46	86.8%	53	0	0	0	0	0
libgwenviewlib	433	4,084	24,933	14,486	14,729	243	89.7%	271	0	0	0	0	0
libKF5ModemManagerQt	27	818	8,597	5,464	5,478	14	46.7%	30	0	0	0	0	0
libKF5Torrent	84	3,524	34,338	16,785	16,890	105	98.1%	107	0	0	0	0	0
libKF5ItemModels	26	640	4,368	2,521	2,563	42	82.4%	51	13	19	29	61	122
libKF5IconThemes	43	796	4,212	2,445	2,453	8	38.1%	21	3	4	13	9	29
libqaccessibilityclient-qt5	81	337	3,881	2,638	2,641	3	9.4%	32	0	0	0	0	0
libKF5Style	1	138	459	178	179	1	100%	1	0	0	0	0	0
libpowerdevilconfigcommonprivate	3	279	989	488	492	4	100%	4	0	1	3	1	5
libKF5CompactDisc	4	283	2,263	1,136	1,137	1	33.3%	3	0	9	17	17	43
libKF5WaylandClient	48	2,672	12,625	6,659	6,784	125	100%	125	0	0	0	0	0
libKF5Activities	16	397	3,379	2,036	2,068	32	72.7%	44	0	0	0	0	0
libQt5QuickTemplates2	1,778	4,196	25,336	14,735	14,871	136	92.5%	147	477	498	105	102	1,182
libKF5Sane	57	671	5,554	3,117	3,163	46	63.9%	72	0	0	0	0	0
libKF5Solid	44	828	13,389	8,434	8,475	41	50.6%	81	0	0	0	0	0
libkdeinit5_kcalc	84	728	11,607	5,159	5,328	169	99.4%	170	0	0	0	0	0
libKF5Cddb	10	466	3,991	2,153	2,168	15	93.8%	16	0	0	0	0	0
libKF5Su	3	254	1,364	826	826	0	0%	1	0	0	0	0	0
libKF5KDEGamesPrivate	34	1,586	9,134	5,130	5,167	37	80.4%	46	0	44	25	101	170
libkcardgame	9	800	3,531	1,942	1,951	9	100%	9	0	13	8	11	32
libKF5UnitConversion	9	220	1,834	1,134	1,134	0	0%	0	0	0	0	0	0
libQt5WebKitWidgets	104	1,154	5,268	2,419	2,428	9	31%	29	40	58	22	64	184
libKF5Baloo	97	532	3,244	1,812	1,815	3	75%	4	20	10	0	0	30
libkImageAnnotator	55	3,386	13,386	6,876	7,002	126	100%	126	0	0	0	0	0
libKF5Purpose	4	412	1,924	1,037	1,044	7	100%	7	0	0	0	0	0
libKF5KDEGames	52	1,341	6,411	3,444	3,468	24	85.7%	28	25	17	16	20	78
libksgrd	5	208	1,146	585	592	7	77.8%	9	0	0	0	0	0
libkdsoap	11	905	5,185	2,477	2,477	0	0%	12	0	5	7	16	28
libvclplug_qt5lo	25	1,526	12,692	6,270	6,282	12	60%	20	0	2	2	6	10
libOkular5Core	42	2,531	17,767	10,400	10,435	35	100%	35	0	0	0	0	0
libKF5PlasmaQuick	33	982	6,101	3,334	3,369	35	59.3%	59	0	0	0	0	0
libKF5Parts	16	1,335	5,914	3,392	3,412	20	87%	23	0	0	0	0	0
libQt5MultimediaQuick	35	344	1,976	1,045	1,046	1	16.7%	6	9	7	14	10	40
libKF5People	10	444	2,672	1,564	1,570	6	66.7%	9	0	2	2	5	9
libQt5Gui	862	9,964	85,017	45,416	45,423	7	38.9%	18	108	132	58	150	448
libdolphinprivate	156	3,480	20,525	12,644	12,871	227	100%	227	16	119	147	281	563
libmilou	8	555	2,387	1,284	1,302	18	100%	18	0	0	0	0	0
libKF5PulseAudioQt	25	697	5,062	2,838	2,870	32	97%	33	0	0	0	0	0
libplasma-geolocation-interface	1	88	344	168	169	1	100%	1	0	2	2	4	8
libkdeinit5_ksysguard	65	1,139	9,514	5,882	5,969	87	84.5%	103	0	0	0	0	0
libFcitxQt5DBusAddons	34	314	2,537	1,019	1,022	3	75%	4	6	10	13	35	64
libKF5ItemViews	13	787	3,625	2,011	2,022	11	40.7%	27	7	6	17	25	55
libKF5Package	6	387	2,861	1,626	1,629	3	100%	3	0	0	0	0	0
libKF5DBusAddons	4	277	1,407	750	755	5	83.3%	6	0	8	2	11	21
libplasmanm_internal	33	892	5,662	3,443	3,488	45	97.8%	46	12	5	22	32	71
libphonon4qt5	36	1,201	8,808	5,039	5,056	17	25.8%	66	0	0	0	0	0
libReviewboardHelpers	3	224	1,159	633	639	6	100%	6	0	0	0	0	0
libKF5IdleTime	2	170	722	341	342	1	33.3%	3	0	5	14	9	28
libKF5KrossUi	8	623	1,957	1,033	1,044	11	50%	22	0	0	0	0	0
libKUserFeedbackWidgets	31	329	1,167	632	647	15	78.9%	19	0	0	0	0	0
libkf5be1lo	3	134	912	460	461	1	100%	1	0	0	0	0	0
libdebconf-kde	8	343	2,508	1,467	1,475	8	66.7%	12	0	0	0	0	0
libKWaylandServer	272	2,725	30,498	18,326	18,422	96	100%	96	0	0	0	0	0
libQt5XcbQpa	55	2,180	22,374	13,190	13,201	11	42.3%	26	0	1	3	4	8
libKF5GuiAddons	6	481	2,334	1,141	1,149	8	100%	8	5	18	0	21	44
libprocessui	93	1,049	5,220	2,931	2,966	35	89.7%	39	7	2	23	27	59
libKF5Bookmarks	17	956	5,225	3,216	3,254	38	86.4%	44	0	10	6	16	32
libqca-qt5	44	2,438	20,023	9,898	9,965	67	100%	67	0	0	0	0	0
libKF5WaylandServer	64	3,061	17,154	10,029	10,186	157	100%	157	0	0	0	0	0
libpolkit-qt5-gui-1	2	146	635	277	281	4	66.7%	6	0	0	0	0	0
libkhotkeysprivate	30	963	4,796	2,762	2,768	6	40%	15	0	0	0	0	0
libKF5KIOFileWidgets	199	2,297	17,111	10,499	10,701	202	99%	204	0	37	76	78	191
libQt5HunspellInputMethod	4	266	2,037	1,068	1,069	1	100%	1	0	0	0	0	0

Table 12: Detailed results of callback and semantics recovery in KDE binaries (CGE stands for call graph edges).

Binary Name	Time(s)	# Symbol	# CGE	# CGE	# CGE	Callback From QTRE			Symbols From QTRE				
			ANGR	GHIDRA	GHIDRA+QTRE	# Recovered	%	# Total	# Prop.	# Signal	# Slot	# Param.	# Total
libQtSql	0	671	4,206	2,207	2,207	0	0	0	0	5	9	13	27
QtCarCluster	337	40,712	69,025	23,503	23,636	133	84.2%	158	0	373	192	357	922
QtCarParrot	661	58,175	72,324	21,296	21,302	6	15.8%	38	0	673	185	378	1,236
QtCarNavServer	248	15,154	40,223	17,981	18,262	281	96.2%	292	0	160	82	223	465
libQtCarSim	298	45,882	57,169	21,430	21,464	34	100%	34	0	528	3	280	811
libQtCarServiceMgr	22	3,232	7,124	3,449	3,491	42	89.4%	47	0	14	12	11	37
libQtXmlPatterns	168	703	72,277	17,893	17,897	4	28.6%	14	0	0	0	0	0
libQtCore	200	4,191	34,100	18,773	18,783	10	24.4%	41	52	75	50	85	262
libQtGui	1,726	14,534	*N/A*	88,111	88,413	302	48.6%	622	649	349	602	546	2,146
QtCarMonitor	30	8,500	11,106	6,660	6,660	0	0%	3	0	114	0	87	201
QtCarNetManager	440	53,842	69,943	23,493	23,559	66	69.5%	95	0	626	49	355	1,030
libQtCarGUI	4,636	138,461	*N/A*	146,878	149,833	2,955	90%	3,282	15	845	1,160	1,022	3,042
QtCarSpeechRecognizer	410	51,547	68,072	24,277	24,322	45	95.7%	47	0	585	50	347	982
libQtCarCANData	0	12,712	*N/A*	1,539	1,539	0	0	0	0	12	0	18	30
libQtSvg	12	714	5,680	2,197	2,201	4	36.4%	11	5	1	7	11	24
QtCarEbServerIC	486	43,050	54,939	25,909	26,106	197	90.4%	218	0	449	37	228	714
libQtCarMediaV2	267	35,362	75,291	20,901	21,252	351	72.4%	485	0	64	18	78	160
QtCarEVLogService	194	34,420	38,690	15,775	15,781	6	75%	8	0	382	9	188	579
libQtNetwork	102	1,816	*N/A*	10,273	10,364	91	56.5%	161	1	96	78	73	248
QtCarScreenshot	0	465	421	219	219	0	0	0	0	0	0	0	0
QtCarMediaServerV2	64	13,811	20,655	11,705	11,758	53	84.1%	63	0	150	0	124	274
libQtCarUtils	99	17,161	32,439	12,650	12,695	45	67.2%	67	0	63	55	98	216
QtCarVehicle	1,567	41,296	48,564	29,204	30,779	1,575	100%	1,575	0	437	88	283	808
libQtDBus	17	726	9,858	4,769	4,783	14	51.9%	27	3	12	12	26	53
libQtOpenGL	16	1,477	8,479	4,716	4,720	4	66.7%	6	0	1	0	1	2
libQtCarAlerts	5	1,096	1,220	711	715	4	50%	8	0	12	5	13	30
libQtDesigner	43	6,724	73,650	36,919	36,919	0	0	0	7	31	29	51	118
libQtHelp	40	1,015	10,180	3,874	3,901	27	32.9%	82	3	20	16	20	59
libQtMultimedia	5	419	1,289	615	631	16	100%	16	0	11	0	2	13
libQtCarPower	18	4,594	8,060	4,914	4,918	4	100%	4	0	65	3	55	123
libQtWebKit	1,301	214,219	*N/A*	159,019	159,059	40	61.5%	65	38	57	23	46	164
QtCarGpsManager	359	48,125	63,677	23,226	23,242	16	84.2%	19	0	507	9	281	797
QtCarBrowser	59	14,966	23,097	10,580	10,606	26	92.9%	28	0	186	4	146	336
libQtCarUIFramework	653	61,679	80,410	49,863	49,991	128	84.8%	151	0	450	215	290	955
libQtMultimediaKit	193	13,970	12,977	6,835	6,835	0	0%	1	78	201	123	194	596
libQtLocation	38	2,192	11,654	6,649	6,661	12	12.9%	93	0	0	0	0	0
libQtDeclarative	64	3,652	57,123	29,555	29,555	0	0	0	100	83	38	47	268
libQtTest	0	276	1,773	940	940	0	0	0	0	0	0	0	0
QtCarServer	940	75,816	126,496	38,481	38,567	86	85.1%	101	0	878	117	744	1,739
libQtCarVAPI	308	46,571	54,498	21,917	21,936	19	95%	20	0	369	28	184	581
QtCarAudiod	305	35,108	41,079	25,310	25,405	95	88%	108	0	376	17	162	555
QtCarSimService	2,169	2,076	1,247	701	1,553	852	100%	852	0	5	0	0	5
libQtScript	131	878	26,078	11,346	11,347	1	33.3%	3	0	1	1	1	3

Table 13: Detailed results of callback and semantics recovery in Tesla binaries (CGE stands for call graph edges, N/A indicates unavailable results due to exceptions when analyzing the binary).