Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis

Kangjie Lu
University of Minnesota, Twin Cities

Hong Hu
Georgia Institute of Technology

Abstract

System software commonly uses indirect calls to realize dynamic program behaviors. However, indirect-calls also bring challenges to constructing a precise control-flow graph that is a standard prerequisite for many static program-analysis and system-hardening techniques. Unfortunately, identifying indirect-call targets is a hard problem. In particular, modern compilers do not recognize indirect-call targets by default. Existing approaches identify indirect-call targets based on type analysis that matches the types of function pointers and the ones of address-taken functions. Such approaches, however, suffer from a high false-positive rate as many irrelevant functions may share the same types.

In this paper, we propose a new approach, namely *Multi-Layer Type Analysis* (MLTA), to effectively refine indirect-call targets for C/C++ programs. MLTA relies on an observation that function pointers are commonly stored into objects whose types have a multi-layer type hierarchy; before indirect calls, function pointers will be loaded from objects with the same type hierarchy "layer by layer". By matching the *multi-layer types* of function pointers and functions, MLTA can dramatically refine indirect-call targets. MLTA is effective because multi-layer types are more *restrictive* than single-layer types. It does not introduce false negatives by conservatively tracking targets propagation between multi-layer types, and the layered design allows MLTA to safely fall back whenever the analysis for a layer becomes infeasible. We have implemented MLTA in a system, namely TypeDive, based on LLVM and extensively evaluated it with the Linux kernel, the FreeBSD kernel, and the Firefox browser. Evaluation results show that TypeDive can eliminate 86% to 98% more indirect-call targets than existing approaches do, without introducing new false negatives. We also demonstrate that TypeDive not only improves the scalability of static analysis but also benefits semantic-bug detection. With TypeDive, we have found 35 new deep semantic bugs in the Linux kernel.

CCS Concepts

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

Keywords

Layered type analysis; Indirect-call targets; Function pointers; CFI

ACM Reference Format:
Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354244

1 Introduction

Function pointers are commonly used in C/C++ programs to support dynamic program behaviors. For example, the Linux kernel provides unified APIs for common file operations such as `open()`. Internally, different file systems have their own implementations of these APIs, and the kernel uses function pointers to decide which concrete implementation to invoke at runtime. Such an invocation is known as an indirect call (*icall* for short). Common icall targets include callback functions, jump-table entries, and virtual functions.

While icalls are common and useful, by their dynamic nature, icall targets cannot be precisely decided through static analysis. This leads to inherent challenges in constructing precise global Control-Flow Graph (CFG) that connects icalls to their targets. In particular, compilers such as `GCC` and `LLVM` do not recognize icall targets by default. Users of CFG have two options: stopping an analysis when encountering icalls or continuing an analysis by taking all address-taken functions as potential targets. Both options have apparent drawbacks. The former limits the coverage of the analysis, while the latter limits the scalability and precision of the analysis, and hurts the strength of system hardening techniques. More specifically, many bug detection tools using inter-procedural analysis choose to skip icalls [24, 26, 40, 50], and thus will miss bugs hidden behind icalls. Including massive irrelevant functions, on the other hand, will lead to significant false positives to bug detection techniques based on cross-checking [34, 52], and will likely cause path explosion to symbolic executions [5, 47], impeding precise analyses. Furthermore, Control-Flow Integrity (CFI) [1, 6, 35, 46, 53, 55] prevents control-flow hijacking attacks by restricting control transfers to predefined icall targets. The inaccuracy in finding icall targets will result in a permissive CFI enforcement, rendering the protection ineffective [8, 12, 14, 17, 42].

Given the importance of identifying icall targets, researchers have attempted to achieve it in two general ways: pointer analysis and type analysis. In theory, pointer analysis can find out all possible icall targets through whole-program analysis [1, 2, 4, 33, 45]. However, since pointer analysis itself requires a CFG from the beginning, the analysis must be recursive and thus computational expensive. More importantly, realizing a precise pointer analysis is hard or typically does not guarantee the soundness [3, 14]. Given the limitations with pointer analysis, recent practical CFI techniques opt for function-type analysis [14, 35, 46]. These approaches identify icall targets by matching the type of function pointers with the ones

CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354244

of potential target functions. While such approaches have been practically applied to harden programs, they still suffer significant false positives—indirect calls with a small number of general-type parameters (*e.g.*, `void (*)(char *)`) will match a large number of unrelated function targets.

In this paper, we propose a new approach, *Multi-Layer Type Analysis (MLTA)*, to effectively refine icall targets without introducing false negatives to existing type analysis–based approaches. The intuition behind MLTA is that a function address is often stored into objects whose types belong to a multi-layer type hierarchy; when the function pointer is used in an icall, its value is loaded layer by layer from the same type hierarchy. For example, in the statement, `b.a.fptr=&f`, the function address of `f` is stored into the function-pointer field (`fptr`) of a type-`A` object (`a`) which is stored in a type-`B` object (`b`). Correspondingly, to invoke the function through an icall, the function address will be loaded layer by layer: object `a` will be first loaded from object `b`, and the function pointer, `fptr`, will be further loaded from object `a`. By matching the *multi-layer* types (*i.e.*, `B.A.fptr_t`) instead of only the *first-layer* type (*i.e.*, `fptr_t`) between address-taken functions and function pointers, we can dramatically refine icall targets. Accordingly, we denote the existing type analyses focusing on the "first-layer" as *First-Layer Type Analysis (FLTA)*.

MLTA has two unique advantages. First, given the fact that multi-layer types are more restrictive than first-layer types, it can significantly reduce false positives. Since the first-layer type P is the inner-most layer of all multi-layer types for an icall, the target set provided by FLTA for the first-layer type is the union of that provided by MLTA for all related multi-layer types. In other words, for the same icall, MLTA *always* provides a subset of the one given by FLTA. Second, the multi-layer type matching can be *elastic* to avoid potential false negatives. In general, types with more layers provide stronger restrictions in confining icall targets. However, when the complete multi-layer type is not available, *e.g.*, due to type escaping (§4.1.3), MLTA can fall back to a more permissive sub-type to find icall targets, without introducing false negatives. We provide a formal analysis in §6 to show that MLTA guarantees the effectiveness and does not introduce false negatives.

There are however two challenges in implementing the MLTA analysis. First, maintaining the map between multi-layer types and address-taken functions can be expensive in both storage and computation. In the aforementioned example, MLTA has to maintain the map for `fptr_t`, `A.fptr_t`, and `B.A.fptr_t`, if they are used separately in the program. Given that the map must be maintained globally for the target program, it is potentially large. Second, the maintaining may become complicated when casting occurs frequently between these multi-layer types; types must be recursively maintained for each casting. More importantly, the multi-layer types in the source and sink of a cast should be extracted, which can be challenging when complicated data flows are involved. MLTA must address this issue carefully to avoid potential false negatives.

Our solution to these challenges is to break a multi-layer type into a series of two-layer types and map each of them with its associated icall targets. For example, given `b.a.fptr=&f`, we will maintain the mapping for only `B.A` and `A.fptr_t`. Any cast can thus be simply recorded for the two-layer type instead of all involved multi-layer types. Each two-layer type is *independent* of others. Given an icall, based on where the function pointer is loaded from, we can assemble the two-layer types into a multi-layer type back and resolve the final icall targets. This way, we can restrict icall targets both effectively and efficiently.

Applying MLTA to C++ programs requires additional design efforts because the pointer to virtual-function tables (VTable) is frequently cast to general types such as `char*`, rendering the type matching ineffective. To address this problem, we develop a mechanism to precisely connect VTables to the corresponding classes and to keep track of class casting. A few recent works [25, 54] have attempted to enforce CFI for C++ programs in a similar way that a virtual function call can only invoke the virtual functions implemented in the current class or its derived classes, but not others. Such an analysis is realized by extracting the class hierarchy information from the C++ programs. We find that MLTA can outperform these works for two reasons. First, MLTA can further refine icall targets when an object pointer is recursively stored into an object of a different class. Second, MLTA precisely tracks type casting. Therefore, virtual functions of a derived class become valid icall targets of a base class only when an actual cast exists.

We have implemented our design of MLTA in a system called TypeDive. TypeDive can not only identify how function addresses are stored into and loaded from objects layer by layer, but also conservatively capture type-escaping cases. We have also evaluated the effectiveness and potential false-negative cases of TypeDive with three widely used large system programs—the Linux kernel, the FreeBSD kernel, and the Firefox browser. The evaluation results show that, compared to existing approaches using FLTA, TypeDive can additionally eliminate 86% to 98% icall targets for these large and complex programs. Our empirical false-negative evaluation also shows that TypeDive does not introduce any false negatives to FLTA. Further, we show how TypeDive can improve the scalability and precision of traditional static analyses. At last, we leveraged TypeDive to find peer functions and cross-check them to identify semantic bugs that either miss security checks or use initialized variables. From the results, we have manually confirmed 25 new missing-check bugs and 10 new missing-initialization bugs in the Linux kernel. All these bugs are hidden behind icalls where existing detection approaches either simply ignore or have significant false positives.

In summary, we make the following contributions in this paper.

- **A new approach.** We propose the multi-layer type analysis to effectively refine indirect-call targets. MLTA is elastic and does not introduce false negatives to existing type analysis–based approaches.
- **New techniques.** We propose multiple techniques such as type confinement and propagation analysis, and type escaping analysis to ensure the effectiveness of MLTA and to conservatively eliminate potential false negatives. We also extend MLTA to support C++ programs.
- **Extensive evaluation and new bugs.** We have implemented MLTA in a prototype system, TypeDive. We extensively evaluated its effectiveness, scalability, and false negatives by applying it to three large and complex real-world system programs. With TypeDive, we found 35 new semantic bugs in the Linux kernel.

The rest of this paper is organized as follows. We present the problem of icall-target resolving in §2; the overview of TypeDive in §3; the design of TypeDive in §4; the implementation of TypeDive in §5. We then formally analyze MLTA in §6. We present the evaluation of TypeDive in section §7, discuss its limitations in §8 and its related work in §9. We finally conclude in §10.

2 Problem Definition

In this section, we first provide the background of function pointers, including their usage and prevalence, and then show the limitations with existing approaches to motivate our approach.

2.1 Function Pointers and Indirect Calls

Programs use function pointers to realize dynamic features and to improve program performance. To achieve dynamic features, the icall target is determined by the value of a function pointer that originates from callbacks, exception handling, or C++ virtual functions. For example, C++ programs heavily use virtual functions to realize polymorphism, where the concrete behavior of a function is determined by the runtime type. C++ compilers save the addresses of all virtual functions implemented in a class into a table (VTable) and store the table address into the first entry of each class object. Following the pointer of a class object, the program can find the virtual function table, obtain the real function pointer by indexing the VTable and finally jump to the particular function through an icall. In OS kernels, function pointers are also extensively used to achieve polymorphism analogous to C++, e.g., which `open()` to invoke is determined by the specific file system in use.

Indirect calls can also help save CPU cycles. Considering a command dispatcher which supposes to invoke proper functions based on the input value. One way to implement the switch is to use a long list of comparisons followed by direct function calls. However, implementing such a switch with an icall is much more efficient because the program can just retrieve the proper function pointer using the input as an index and jump to that function through an icall.

Function pointers in memory. Function pointers can reside in two classes of memory objects, primitive-type variables or composite-type objects. Before a function pointer is stored into memory, its function type can be preserved or cast to general types, such as `char*`. To understand how common each case is, we conducted a study to statistically count each case in the Linux kernel by analyzing its LLVM bitcode files. In our results, among 212K instructions that store function pointers, 88% of them put function addresses into a composite-type object, and only 12% of them save function addresses into a primitive-type variable. In the former case, 91% of them do not cast the function type before the store instruction, while in the latter case, 80% of them cast the function pointer into a different type. The numbers show that storing function addresses in a composite-type object is quite common, and the function addresses will typically be loaded from objects of the same composite types before being dereferenced for icalls. MLTA will exploit such type information to refine icall targets.

A motivating example. We show a sample usage of function pointers in Figure 1. This code snippet defines one function pointer type, `fptr_t`, and three structures, `A`, `B` and `C`, where `A` contains a field (`handler`) of a function-pointer type, `fptr_t`, while both `B` and `C` have an instance of `A`. The code provides two string copy functions: `copy_with_check()` takes two `char *` arguments and performs the boundary check before copying a string; `copy_no_check()` is identical to `copy_with_check()` except that it does not do any boundary check. The code defines two global variables, `b` of type `B` and `c` of type `C`, and initializes these two variables accordingly with `copy_with_check()` and `copy_no_check()`. Function `handle_input()` processes the untrusted `user_input`, which could be of arbitrary length and contain malicious content. `handle_input()` first creates a stack buffer `buf` with the fixed size `MAX_LEN` and then retrieves function pointers from variables `b` and `c`, and uses them to copy `user_input` to the stack buffer `buf`, respectively. Next, We use this example to show how the existing FLTA and our MLTA identify the different numbers of icall targets.

```
 1 typedef void (*fptr_t)(char *, char *);
 2 struct A { fptr_t handler; };
 3 struct B { struct A a; };   // B is an outer layer of A
 4 struct C { struct A a; };   // C is an outer layer of A
 5
 6 void copy_with_check(char *dst, char *src) {
 7   if (strlen(src) < MAX_LEN) strcpy(dst, src);
 8 }
 9
10 void copy_no_check(char *dst, char *src) {
11   strcpy(dst, src);
12 }
13
14 // Store functions with initializers
15 struct B b = { .a = { .handler = &copy_with_check } };
16
17 // Store function with store instruction
18 struct C c; c.a.handler = &copy_no_check;
19
20 void handle_input(char *user_input) {
21   char buf[MAX_LEN];
22   ...
23   (*b.a.handler)(buf, user_input);    // safe
24   (*c.a.handler)(buf, user_input);    // buffer overflow !!
25   ...
26 }
```

Figure 1: Indirect function calls that can be confined by multi-layer type analysis. Function pointers used in line 23 and line 24 have type `fptr_t`, where traditional type-based matching will find two potential targets `copy_with_check()` and `copy_no_check()`. However, MLTA will identify that the pointer in line 23 can only be `copy_with_check()`, while the pointer in line 24 can only be `copy_no_check()`.

2.2 Existing Approaches and Limitations

Existing FLTA relies on type-based matching to infer the possible target(s) for an icall. Specifically, FLTA (1) identifies the function-pointer type in an icall (2) searches the whole program to find all address-taken functions of the same type. In the example of Figure 1, the icall in line 23 uses function pointer `b.a.handler` which takes type `fptr_t`. As both `copy_with_check()` and `copy_no_check()` have the matched type, and their addresses are taken for variable initialization (line 15, 18), FLTA will label both of them as valid targets. Similarly, FLTA will assign both functions to the icall in line 24. However, with manual checking, we can tell that the icall in line 23 can only target function `copy_with_check()`, while the icall in line 24 can only target function `copy_no_check()`. Thus, FLTA introduces false positives when identifying icall targets.

Triggering false alarms. The inaccuracy of FLTA will cause false alarms when we use static analysis to detect bugs. Let us consider

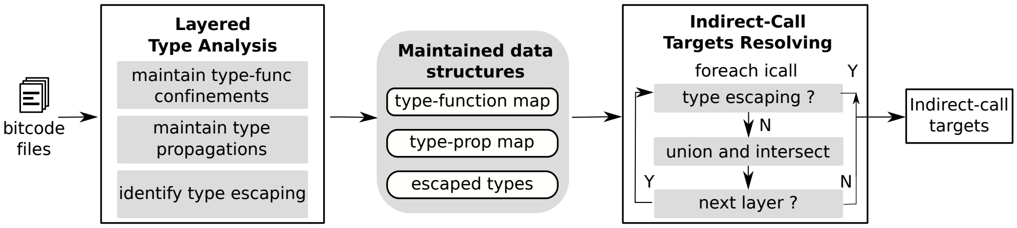

Figure 2: Overview of TypeDive. It takes the program LLVM bitcode as input and outputs the targets for each icall. TypeDive takes two phases to produce the icall targets. The first phase inspects the address-taken instructions to build the type-function map, analyzes all value store and casting instructions to collect the relationship between types, and checks each type to identify escaped ones. The second phase uses the built maps to resolve the targets for each icall.

a static analysis for identifying buffer overflows. From the example, the analysis first detects a buffer, `buf`, in `handle_input()`. Then it proceeds to check whether the memory access to this buffer is within its boundary. It follows the control flow and reaches the icall in line 23. FLTA tells that both `copy_with_check()` and `copy_no_check()` could be the targets. Therefore, the analysis inspects both functions and will report a buffer overflow in line 11 due to the missing of a boundary check. However, this is a false alarm because the icall in line 23 will never reach function `copy_no_check()`.

Crippling CFI protection. The inaccuracy of FLTA will also bring security issues to CFI protection. CFI aims to prevent control-flow hijacking attacks [1], where attackers maliciously change some memory variables like function pointers to divert the control flow for their bidding. CFI makes sure that each indirect control-flow transfer (*i.e.*, indirect call/jump and return) only goes to the predefined valid target(s). Therefore, an accurate indirect-call analysis is required for strong protection. In line 233 of Figure 1, attackers may have corrupted the function pointer, `b.a.handler`, to divert the control flow. With FLTA, the type analysis, as employed in recent CFI mechanisms [36, 46, 49], will allow both functions. Such approximation weakens the protection as attackers can divert the control flow to `copy_no_check()` to launch attacks.

FLTA fails to identify the accurate icall targets in the simple code of Figure 1. In real-world programs with millions of lines of code, such inaccuracy will lead to significant false positives, rendering the analysis results less meaningful. Therefore, it is necessary to develop a new approach to effectively refine icall targets.

2.3 Our Approach: Multi-Layer Type Analysis

The example in Figure 1 shows that FLTA uses only one-layer type information (*i.e.*, the function pointer type) to find icall targets, without considering more type layers. Such observation motivates us to propose a new approach—multi-layer type analysis (MLTA). The key insight of MLTA is that the target(s) of an icall can be confined through *multi-layer* types. We consider not only the function type but also the types of memory objects that hold the function pointers. Since memory objects can hold other memory objects recursively, we can further leverage the "layered" types to refine icall targets. For example, in Figure 1, MLTA will find that the function pointer, `b.a.handler`, in line 23 takes type `fptr_t`, and its value is read from `a`, an object of type `A`. In turn, `a` is retrieved from `b`, an object of type `B`. Therefore, we find a three-layer type to retrieve the function pointer: `B.A.fptr_t`. MLTA requires that the target(s) of the icall in line 23 must have its (theirs) address(s) taken and assigned to some pointers of the three-layer type `B.A.fptr_t`. By checking the program, we can find that the only function satisfying such a requirement is `copy_with_check()`, while `copy_no_check()` is assigned to a function pointer of type `C.A.fptr_t`. Therefore, in this example, the multi-layer type analysis helps remove false positives and find the correct, unique target.

3 Overview of TypeDive

MLTA refines icall targets through type analysis of multiple layers. In this section, we introduce our system, TypeDive, a practical implementation of multi-layer type analysis for refining icall targets.

Figure 2 shows the overview of TypeDive, which takes as input the LLVM bitcode files of the target program and identifies the targets for icalls as the outputs. TypeDive consists of two main phases, the type-analysis phase, and the target-resolving phase. The first phase thoroughly scans all bitcode files to collect type-related information. Note that, for simplicity, we refer to types as composite types by default in the following sections. TypeDive first collects all address-taken functions and identifies all address-taking operations, where the latter could be either a static initializer of a global variable or a store instruction. TypeDive then analyzes the address-taking operation to identify the multi-layer type of the memory object. For example, in line 15 of Figure 1, the address of `copy_with_check()` is taken and assigned to `b.a.handler`. TypeDive will identify its multi-layer type as `B.A.fptr_t`. Then TypeDive splits the multi-layer type into several two-layer types for efficient target propagation. In this case, `B.A.fptr_t` is split into `fptr_t` (the first layer), `A.fptr_t` (the second layer) and `B.A` (the third layer). TypeDive adds such information to the `type-function` map where the key is the hash of the two-layer type, and the value is the set of associated functions. Next, TypeDive identifies how typed objects are stored to other objects of different types, e.g., through `*p=a`, and adds the relationship between the types of two operands into the `type-propagation` map. The key of the `type-propagation` map is the type of the value object `a`, and the value is the type of the pointer object, `p`. TypeDive analyzes all casting operations and maintains the casting relationships between two types through the `type-propagation` map. Note that nested types will also be cast and maintained for casting operations. The last component of the first phase is to capture potential type escaping cases and add them into the `escaped-type` set. A type is escaping if we cannot decide all the icall targets it can confine. For example, when a primitive type is cast to a composite type, because we cannot decide the targets of the primitive type, the targets of the composite type also become undecidable. In this case, the composite type is escaped.

The second phase of TypeDive aims to resolve targets for each icall. Given an icall instruction, TypeDive identifies the multi-layer type of the function pointer and breaks it into a series of two-layer types. TypeDive initializes the target set with the first-layer type matching (*i.e.*, FLTA), and then iteratively resolves the targets layer by layer, from the first layer to the last layer. The target set of one layer will be intersected with the targets resolved from previous layers. At each layer, TypeDive first checks if the type at the layer has escaped or not, based on the maintained `escaped-type` set. If the type has escaped, TypeDive conservatively stops confining the targets for the icall and outputs the current target set (*i.e.*, an overestimation) as the final target set. Otherwise, TypeDive queries the `type-propagation` map to find all types that are ever cast to the current type. All targets of these types are recursively collected and combined as the targets set of the current layer, which are further intersected with the targets of previous layers. After each iteration, TypeDive continues to identify the next-layer type. If no further types are identified, TypeDive reports the existing target set as the final one for the icall. Otherwise, TypeDive continues the target resolving with the next-layer type.

As targets at each layer are intersected, TypeDive can effectively refine icall targets. TypeDive's analysis is conservative and elastic: (1) if TypeDive cannot find the next layer, it immediately returns the final target set; (2) if any type is escaping, TypeDive stops resolving the targets and falls back to a previous layer. Note that, TypeDive's analysis is field-sensitive, e.g., maintaining which field a store is targeting.

4 Design of TypeDive

TypeDive has two design goals: (1) to effectively refine the targets as much as possible and (2) to not introduce any false negatives (i.e., to not miss valid targets) to FLTA. In this section, we present how we design each component of TypeDive to achieve both goals.

4.1 Phase I: Layered Type Analysis

TypeDive includes two phases. As shown in Figure 2, the first phase identifies all stores and initializers of global variables that save a function address into memory, maintains how functions are confined by types, and how types are propagated to other types. More importantly, to avoid false negatives, this phase also identifies escaped types whose instances may hold type-undecided targets.

4.1.1 Maintaining Type-Function Confinements. For a function address to be a valid icall target, it must be first stored into memory (i.e., a variable or a field of an object). Given a store of a function address, TypeDive identifies the layered types of it and maintains the confinements in a global `type-function` map. While the key is a type, the value is a set of functions confined by the type. We call the stores of function addresses as *confinements* because the function addresses are expected to be loaded from the objects of the same types. The stores can be in a store instruction or a static initializer of a global variable (e.g., line 15 and line 18 in Figure 1). Our analysis is field-sensitive in that the type information in the map also includes the index of the field that holds the function.

Layered confinements. In an address-taking operation, if an object containing a function pointer is contained by another object of a different type, the confinement is layered. Such information enables TypeDive to match types layer by layer. Specifically, given a store of a function address, in either an initializer or a store instruction, TypeDive will recursively identify the layered types and also maintain the confinements in the map. As shown in Figure 1, both functions `copy_no_check()` and `copy_with_check()` are ever stored into objects of type `A`, which is further stored into objects of types `B` and `C`. In this case, we will identify that type `A` confines these functions at layer two, and that types `B` and `C` confine these functions at layer three. The function type is always at layer one, i.e., FLTA. The type matching at layer two (i.e., through type `A`) will identify both functions, `copy_no_check()` and `copy_with_check()`, as valid targets for icalls in lines 23 and 24. However, the type matching at layer three (i.e., through types `B` and `C`) can identify a unique function target for each icall. That is, the icall in line 23 can only call `copy_with_check()` because `copy_no_check()` is never confined by type `B`, and the icall at line 24 can only call `copy_no_check()`.

4.1.2 Maintaining Type Propagation. A multi-layer type in address-taking operations introduces layered confinement. However, if an object of one type is stored into an object of another type in non-address-taking operations, all functions confined by the first type should be propagated as possible targets to the second type. In this case, the targets of the second type could potentially be significantly expanded. We call such cases as *type propagation*.

To identify type propagation, we thoroughly analyze all store and cast operations. The store can be either value-based or reference-based (i.e., storing a pointer of an object into a field of another object). The cast operations can be in either a cast instruction or a cast operator in static initializers. For example, in LLVM IR, `unions` are implemented as one of its multiple types, and the IR relies on type casting to load and store the instance of other types. Given a store or cast operation, we identify the source type and the sink type, and use the `type-propagation` map to maintain the propagation. The key in the map is the sink type, and its value is a set of source types that are ever cast to the sink type.

Propagation for nested sub-types. When a source type propagates to a sink type, we will recursively perform the propagation for the nested sub-types from the source to the sink. This is necessary because TypeDive does not employ data-flow analysis, so we cannot guarantee that TypeDive always finds the base type where a function pointer is loaded from.

4.1.3 Identifying Escaping Types. The essence of TypeDive is to identify the confined targets for a type. However, this may become infeasible if the type contains undecidable targets. We call such cases *type escaping*. To specify the policy for identifying type-escaping cases, we first define *unsupported types* as follows:

(1) Non-composite types such as general pointer types (e.g., `char *`) and integer types.
(2) A type whose object pointers are *ever* arithmetically computed. Note this does not include field indexing for structures.

The first criterion is not mandatory; as long as the propagation of a type is thoroughly tracked, it can be included for layered type analysis. We exclude non-composite types based on the observation that such types can potentially contain a large number of function targets, rendering the layered analysis less effective. Further, including non-composite types will significantly enlarge the maintained data structures and impact the analysis efficiency. By contrast, the second criterion is required because, once a pointer is arithmetically computed, the type of the object it points to can be undecidable.

We then identify a composite-type as escaping if it has one of the following cases.

(1) The type is cast from an unsupported type;
(2) Its objects are stored to objects of an unsupported type;
(3) It is cast to an unsupported type.

The first policy is intuitive. Casting propagates icall targets from source to sink; because the target set of an unsupported type is undecidable, the target set of the composite type will become undecidable. The same reason applies to the second policy because a store also propagates icall targets. The third policy is also necessary. When a composite type is cast to an unsupported type, it may be used as a pointer to store a value with an unsupported type, like the destination pointer in `memory()`. In this case, the target set of the composite type will also become undecidable.

To identify escaping types, we analyze all store and cast operations targeting a composite type and extract the types of both the source and the sink. If either type is unsupported, we conservatively label the composite type as *escaping*. One thing to note is that, when we cannot decide if a composite type would be stored or cast to an unsupported type, e.g., a pointer of an object of the composite type is passed to or from other functions, and we cannot decide how this pointer is used in those functions, we will also treat the composite type as escaping.

4.1.4 *Supporting Field-Sensitive Analysis.*

TypeDive's analysis is field-sensitive. This is important to refining icall targets because a type may have multiple fields that can hold different function targets. Therefore, TypeDive computes the indexes of fields into the objects. TypeDive supports field-sensitive analysis by analyzing operations that control pointers for accessing elements of arrays and structs. The operations in LLVM IR are the `GetElementPtrInst` and `GEPOperator`; both are for *type-safe* pointer arithmetic for accessing fields in composite-type objects. When the indices in the operations are constants, analyzing the index of fields is straightforward. However, when the indices include non-constants, which is uncommon, TypeDive conservatively labels the type as escaping. All the types in the `type-function` confinement map, the `type-propagation` map, and the `type-escaping` set include indexes. With the field-sensitive analysis, `union` types can also be naturally supported—`union` types are treated as general composite types, and their fields are recognized based on the fields indexes.

4.2 Phase II: Targets Resolving for iCalls

Based on the collected information from the first phase, the second phase of TypeDive resolves possible targets for each icall. At a high level, TypeDive iteratively resolves confined targets for the type at each layer based on how the function pointer is loaded from memory. The targets of each layer are *intersected* to have the final target set for the icall. For each layer, TypeDive recursively resolves the targets based on the maintained type confinements and type propagations. In this section, we present TypeDive's target-resolving algorithm and the design of key components.

Algorithm 1: Iteratively resolve targets for an indirect call.

```
Input   : iCall: The given indirect call requires target resolving,
          type-function: The type-function confinement map,
          type-propagation: The type-propagation map,
          escaped-type: The set of escaped types
Output: TargetSet: The set of possible targets for the indirect call icall

1  Procedure ResolveTargets(iCall, type-function, type-propagation, escaped-type)
2      TargetSet ← all address-taken functions;
3      CurValue ← getCalledValue(iCall);
       // get next-layer value with a composite type
4      while CurValue ← getNextLayerValue(CurValue) do
           // index is also obtained along with the base type
5          CurType ← getBaseType(CurValue);
6          if isNotSupported(CurType) then break ;
           // ensure CurType has not escaped
7          if CurType ∈ escaped-type then break ;
8          LayerTargetSet ← type-function[CurType];
           // merge targets of all types propagating to CurType
9          for each PropType in type-propagation[CurType] do
               // recursively find targets of PropType
10             if PropTargetSet ← recurGetTargets(PropType, ...) then
11                 LayerTargetSet ← LayerTargetSet ⋃ PropTargetSet;
12             else
13                 return TargetSet;
14             end
15         end
           // intersect with targets of the current layer
16         TargetSet ← LayerTargetSet ∩ TargetSet;
17     end
18     return TargetSet;

19 Procedure recurGetTargets(PropType, type-function, type-propagation, escaped-type)
20     PropTargetSet ← type-function[PropType];
21     for each RecurPropType in type-propagation[PropType] do
           // ensure CurType has not escaped
22         if CurType ∈ escaped-type then return NULL;
           // recursively find targets for PropType
23         if RecurPropTargetSet ← recurGetTargets(RecurPropType) then
24             PropTargetSet ← PropTargetSet ⋃ RecurPropTargetSet;
25         else
26             return NULL;
27         end
28     end
29     return PropTargetSet;
```

4.2.1 *The Target-Resolving Algorithm.*

As shown in Algorithm 1, TypeDive uses function `ResolveTargets()` to iteratively resolve targets for an icall layer by layer. For simplicity, indices in types are omitted in the algorithm. The target set of the icall is initialized to contain all address-taken functions (line 2). Given an icall, TypeDive first gets the called value and obtains its type (line 3-5). The type must be either a function type or a composite type; it is the type of the object containing the value (line 6). Once the type is obtained, TypeDive ensures that it is not escaped by querying the `escaped-type` set (line 7). After that, TypeDive queries the `type-function` map to find the function targets confined by the type with the index (line 8). As explained in §4.1.2, the field with the index into the object of this type may be stored to or cast from other types' objects. TypeDive therefore also queries the `type-propagation` map to find all types propagating to the type (line 9), and employs the recursive function, `recurGetTargets()`, to conservatively collect all their function targets (line 10). All these targets are then "unioned" as the target set, `LayerTargetSet`, for the current layer (line 11), which is further "intersected" with the existing target set, `TargetSet` (line 16).

At this point, TypeDive finishes one iteration—resolving the targets for the given layer. Next, TypeDive tries to find the next-layer value and its base type, and starts another iteration (line 4). TypeDive stops the iterations when (1) it cannot get the next-layer base type, (2) the base type has escaped (line 7), or (3) the recursive function (`recurGetTargets()`) returns NULL (line 13). TypeDive's analysis is conservative: if any step cannot continue or fails, TypeDive immediately stops the resolving and returns the current target set. The returned target set by the algorithm represents the final icall targets.

4.2.2 Resolving Targets for a Layer. As shown in line 10 of Algorithm 1, TypeDive conservatively collects all targets of types that may propagate to the current type. Given a propagating type (`PropType`), `recurGetTargets()` first initializes the propagating target set (`PropTargetSet`) with the targets it confines by querying the `type-function` map (line 20). Next, TypeDive finds types that propagate to `PropType` by querying the `type-propagation` map (line 21). For each of them, TypeDive recursively finds the target set and *union*s all of them (line 23-24). If any type is escaped, TypeDive will return `NULL` (line 26) and terminate the target-resolving process in `ResolveTargets()`.

Resolving targets for function types (layer one). The first-layer type is the function type, including the types of parameters. In FLTA, parameter types are consolidated as a single type and matched. Such an approach will cause false negatives in MLTA because parameters may also have composite-types that have been propagated or escaped. To address this problem, in TypeDive, we treat the parameters of composite types as "fields" of the function type and apply the propagation and escaping policies to parameter types as well to resolve the targets of the function type.

4.3 Supporting C++

TypeDive's approach is general and can support both C and C++ programs in principle. As long as type propagation and escaping are captured, TypeDive can safely resolve icall targets without causing false negatives. To be conservative, TypeDive terminates the targets resolving whenever a type has escaped. As described in §4.1.3, a type that is cast to an unsupported type (*e.g.*, primitive type) will be identified as escaping. This introduces a problem for C++ programs because virtual-function table (VTable) pointers will always be cast to an unsupported-type pointer, specifically, a pointer referring to function pointers. This problem would make TypeDive meaningless because virtual-function pointers will be loaded through VTables. Therefore, we develop a technique to overcome this problem.

In C++ programs, each polymorphic class has a VTable, and each entry of the VTable contains the address of a virtual function. VTables are essentially global *arrays* with static initializers. In the constructor of a class, the VTable pointer is cast to a pointer of function pointers and stored to the first field of the constructed object. Before virtual functions are called, the VTable pointer is loaded from the object and looked up to load the correct virtual-function pointer.

Based on how VTable pointers are stored and loaded, we choose to "skip" the VTable pointers (unsupported types in TypeDive) and directly map the virtual functions to their class. Specifically, we analyze the constructor of a class to identify VTables and the contained virtual functions, and use the `type-function` map to map the virtual functions to the class. Correspondingly, given an icall for virtual functions, we ignore the layer for loading VTable pointers and directly resolve the targets for the next layer that loads the object pointer by querying the `type-function` map. This approach also supports `multiple inheritances`; virtual functions of a second base class will be mapped to the second field of the object, and so forth. This way, we avoid the type-escaping issues in C++ programs, and TypeDive can effectively support them as for C programs.

5 Implementation

We have implemented TypeDive based on LLVM of version 8.0.0 as an easy-to-use LLVM pass. The inputs are a list of unlinked LLVM bitcode files. TypeDive's first phase analyzes all bitcode files and generates the data structures for type information. Its second phase analyzes all the bitcode files again to resolve targets for each icall. The resulting icall targets are maintained in a map. The key of the map is a call instruction (either direct or indirect), and the value is a set of functions that can be targets of the call instruction. With the map, the results of TypeDive can be easily queried. In this section, we present important implementation details.

5.1 Generating Bitcode Files

Generating LLVM bitcode files for some system programs can be challenging if they use GCC-specific features such as `ASM goto`. Our strategy is to discard incompatible files because such cases are rare. We thus use the "-i" option (ignoring error) for compilation. To dump bitcode files, we implement an LLVM module pass that uses `WriteBitcodeToFile()` provided by LLVM to save the LLVM IR.

5.2 Analyzing and Storing Types

Supported types. We currently support function type (i.e., signature), `struct`, `array`, and `vector` for confining icall targets. Other types are conservatively excluded as unsupported types.

Storing types and indexes. LLVM has its own type system. Each type has an object in *each* module. That is, the same type will have different type objects and thus different type pointers, which makes type comparison challenging. In TypeDive, we thus choose to compare types through their strings; LLVM provides functions to easily get the string for a type. If the strings of two types match, we will say these types are the same. The string-based matching incurs a significant storage overhead because TypeDive maintains multiple maps for a large number of types. To address the problem, we choose the store the hashing value of types. In the current implementation, we used the default hash function in the C++ standard library. Integer indexes are first converted into strings and hashed together with type strings.

Virtual-function types. When virtual functions are compiled, a `this` pointer is automatically inserted as the first argument. When a virtual function is invoked through an icall, the `this` argument always has the type of the invoking object. For example, if an object pointer of type `base*` is cast from type `derived*`, and the object pointer is used to invoke a virtual function (the one in the derived class), the first argument will be `base*` instead of `derived*`, which will fail the matching of function types. To address this problem, we

	confinement	propagation	resolving
FLTA	$\frac{a = \&f}{M[t(a)] \cup= \{f\}}$	$\frac{y = cast\text{<}t(y)\text{>}\ x}{M[t(y)] \cup= M[t(x)]}$	$\frac{(*p)()}{M[t(p)]}$
MLTA	$\frac{a = \&f}{M[mlt(a)] \cup= \{f\}}$	$y = x$ $\forall\alpha \in comp(mlt(y))$, $\forall\beta \in comp(mlt(x))$, $M[mlt(y)] \cup= M[\beta]$ $M[\alpha] \cup= M[\beta]$	$(*p)()$ $\forall\gamma \in comp(mlt(p))$ $\bigcup \mathrm{M}[\gamma]$

Table 1: Formal representation of FLTA and MLTA. The confinement rule collects address-taken functions and initializes the map M. The propagation rule propagates functions between cast types; The resolving rule decides the icall targets. $t(x)$ gets the type of x; $mlt(x)$ gets the multi-layer type of x; $comp(\alpha)$ gets the compatible multi-layer type of α from M.

exclude the `this` pointer from the argument list when computing the hash value for the virtual-function type. Instead, TypeDive will correctly resolve the possible class types for the virtual-function call by maintaining the casting history.

5.3 Identifying Next-Layer Values and Types

TypeDive iteratively identifies the next layer for type matching. Identifying the next-layer value is required by both maintaining the `type-function` map and resolving icall targets. The next-layer value comes in two ways: (1) an outer layer and (2) a lower layer. The outer-layer value is the host object holding the *value* of the current layer object. In the example shown in Figure 1, `b` is the out-layer object of `a`. By contrast, the lower-layer value is the host object holding the *pointer* of the current layer object. Therefore if the field of `b` is `&a` instead of `a`, then `b` is a lower-type value of `a`.

With the definition of the outer layer and the lower layer, getting the next-layer value is easily implemented by recursively parsing the `GetElementPtrInst` (or `GEPOperator`) and the `LoadInst` instructions directly against the current-layer object. That is, the pointer operands in these instructions are identified as the next-layer value. To be conservative, TypeDive does not consider instructions targeting aliases of the current-layer object. The conservative analysis ensures to avoid potential false negatives.

Once the next-layer value is identified, TypeDive obtains its base type. Since the next-layer is always a pointer, its base type is the type of its element, obtained through `getPointerElementType()`. We also identify the indexes at this step. Specifically, if the next-layer value is obtained from a `LoadInst`, the index is always zero; however, if the next-layer value is either `GetElementPtrInst` or `GEPOperator`, the index is the accumulated constant indices. In case some indices are non-constant, TypeDive conservatively flags the type as escaping, which is rare.

6 Formal Analysis

To prove that (1) MLTA is effective in refining icall targets and (2) MLTA does not introduce more false negatives than FLTA does, we provide a formal analysis of MLTA and the state-of-the-art FLTA.

Assumption. The formal analysis focuses on operations related to FLTA or MLTA, like icalls and address-taking, and ignore others as they will not affect the type-analysis results. To simplify the description and proof, we make the following assumptions.

- Changing types is always through explicit type casting, e.g., `cast<Y> x` casts variable `x` to type `Y`.
- Type information is available for each variable, and all code is in the analysis scope.

The first assumption holds for most well-written programs. In case implicit type casting exists, previous work [35] demonstrates that it is not challenging to modify a program to eliminate all violations. For the second assumption, LLVM generates IR that has an explicit type for each variable as long as source code is available.

Table 1 shows the formal representation of FLTA and MLTA. Each rule in the table contains a code statement and the corresponding action to take. For example, for the confinement rule of FLTA, `a=&f` is the code, and $M[t(a)] \cup= \{f\}$ is the action. M is the type-confinement map. Its key is a type, and its value is the confined function set. $t(x)$ returns the first-layer type of variable x, while $mlt(x)$ returns the multi-layer type of x. For example, for the variable `c.a.handler` in Figure 1, $t()$ returns `fptr_t` while $mlt()$ returns `C.A.fptr_t`. $comp(\alpha)$ returns the set of all multi-layer types in M that is compatible with type α. We define multi-layer type α is compatible with another type β if α and β have overlapping types, and one of them is an instance of another. For example, "_:*A*:*B*:*ptr*" represents a multi-layer type in which (1) the function pointer has type *ptr*; (2) the function pointer is stored in an object of type *B*; (3) the object of *B* is stored in an object of type *A*; (4) the object of *A* escapes from the current function, and we represent its next layer as the wild-card "_". "_:*B*:*ptr*" is compatible with "_:*A*:*B*:*ptr*" as they share the same type *B* and the latter is an instance of the former. But "*A*:*B*:*ptr*" is not compatible with "_:*A*:*B*:*ptr*" as it cannot be an instance of the latter. "_:*C*:_" is not compatible with "_:*A*:*B*:*ptr*" due to lack of common types.

FLTA only cares about three types of operations: address-taking, casting involving function pointers, and icall. The confinement rule inspects each address-taking instruction that stores the address of f to the function pointer a. It adds function f to the function set in M corresponding to the type of a, *i.e.*, $M[t(a)]$. Storing a function address to memory has the same effect. The propagation rule handles the type casting from $t(x)$ to $t(y)$, where one of them is a function pointer type. With this cast, any function pointer with type $t(x)$ could be used as type $t(y)$. Therefore, all functions in $M[t(x)]$ should be added to $M[t(y)]$. FLTA resolves the targets of each icall $(*p)()$ using the resolving rule. Specifically, any function inside $M[t(p)]$ is considered a valid target.

MLTA instead considers the multi-layer type for each rule. For instruction `a=&f`, the confinement rule adds function f to the function set, $M[mlt(a)]$, where $mlt(a)$ is the multi-layer type for *a*. $mlt(a)$ is obtained through a conservative backward analysis: if we cannot find an outer-layer type, we set it to _ and finish the analysis. The propagation rule in MLTA handles casts, loads, and stores. If one variable x is assigned to another variable y, then all function targets reachable from $mlt(x)$ should propagate to $M[mlt(y)]$. Note that all reachable functions from $mlt(x)$ are not $M[mlt(x)]$, but a union of all $M[\alpha]$, where α is in M and compatible with $mlt(x)$. Further, any multi-layer type β in M that is compatible with $mlt(y)$ should receive the same update: if β is an instance of $mlt(y)$, the assignment could ultimately reach β; if $mlt(y)$ is an instance of β, $M[\beta]$ should cover targets from all its instances: MLTA resolves the icall targets with the new resolving rule. the reachable targets

```
1 typedef void (*fptr_t0)(char *);          typedef void (*fptr_t1)(int);
2 struct A { fptr_t0 handler; };            struct B { ftpr_t1 handler; };
3 void func_A(char *);                      void func_B(int);
4 struct A a = { .handler = &func_A };
5 struct B b = { .handler = &func_B } ;
6 struct B * a2b = (struct B *) &a;         (*a2b->handler)(0);
```

Figure 3: An example showing the FN of FLTA. The icall in line 6 takes type `ftpr_t1`, which only `func_B` matches. However, the real target is `func_A`. FLTA misses it as it does not consider the type casting between `A` and `B`.

of type $mlt(p)$ should be the union of all $M[\gamma]$, where γ is in M and compatible with $mlt(p)$. All functions in the union are valid targets of the icall.

Before we formally analyze MLTA and FLTA, we introduce the term *sensitive type* defined in the paper of Code Pointer Integrity recursively [27]: a sensitive type is a function-pointer type, a `void` type, a pointer type of another sensitive type, or a pointer type of a structure that has at least one member with a sensitive type.

Lemma 1. If the program does not have any type-cast of sensitive types, (1) FLTA has no FN; (2) MLTA has no FN; (3) MLTA introduces no extra FP than FLTA; (4) MLTA may have fewer FPs than FLTA.

Proof. Given any icall $(*p)()$, suppose f is one target function based on the ground-truth:
(1) FLTA has no FN. Suppose f has type F and its address is taken with $a{=}\&f$, a must have type $F*$. The confinement rule of FLTA will add f into $M[t(a)]$, *i.e.*, $f{\in}M[F*]$. Similarly, p must have type $F*$. The resolving rule of FLTA infers the target set of this icall is $M[t(p)]{=}{>}M[F*]$, which contains f.
(2) MLTA has no FN. Suppose the address of f is taken through $a{=}\&f$, and the complete multi-layer type of a is $A{:}\cdots{:}Z$, and the complete multi-layer type of p is $A_1{:}\cdots{:}Z_1$. As there is not cast allowed between sensitive types, for each involved basic byte X, X must be the same as X_1. Suppose the $mlt(a)$ in the confinement rule is $N{:}\cdots{:}Z$, where N is either A or _, then we have $f{\in}M[N{:}\cdots{:}Z]$. Similarly, in the resolving rule, suppose the $mlt(p)$ is $L{:}\cdots{:}Z$ where L is either A or _, the resolving result will be a super set of $M[N{:}\cdots{:}Z]$ as $N{:}\cdots{:}Z$ is always compatible with $L{:}\cdots{:}Z$. Therefore, the result must contain f.
(3) MLTA has no extra FPs than FLTA. Suppose f has type F, p must have type $F*$. Suppose the resolving rule of MLTA finds $mlt(p)$ is $N{:}\cdots{:}F*$, then the resolving result of MLTA must be a subset of that by resolving $\_{:}F*$, as any type compatible with $N{:}\cdots{:}F*$ is also compatible with $\_{:}F*$. Meanwhile, resolving $\_{:}F*$ with MLTA is equivalent to resolving $F*$ with FLTA. Thus, for one icall, the target set returned by MLTA is a subset of that returned by FLTA.
(4) MLTA may have fewer FPs than FLTA. The code in Figure 1 shows that MLTA introduces fewer FPs than FLTA. This is adequate to prove this predicate. In fact, we can generally view the map M in FLTA as a union of that in MLTA. As long as the propagation of MLTA does not merge all targets to their compatible first-layer type, like $\_{:}ptr$, MLTA will have fewer FPs than MLTA. □

Lemma 2. If the program has type-cast of sensitive types, (1) FLTA may have FNs; (2) MLTA has no FN; (3) MLTA may have fewer FPs than FLTA.

Proof. Given any indirect call instruction $(*p)()$, suppose f is one target function based on the ground-truth:
(1) FLTA may have FNs. Consider Figure 3: the function pointer `a2b->handler` has type `fptr_t1`, and FLTA will report the target is `func_B` as it is the only type-matched function. However, the real target is `func_A`. FLTA misses the real target as it does not consider the type casting in outer-layer types, *i.e.*, from `A*` to `B*`. In fact, FLTA will miss all implicit cast of function pointer types that are indicated by casts between outer-layer types.
(2) MLTA has no FN. The only way to introduce FNs here is through type casting. The propagation rule of MLTA handles type casting conservatively. $mlt(a)$ could be different from $mlt(p)$ at each layer, or even has a different number of basic types. The type casting may take several steps, each happening at a different layer. However, the propagation rule always conservatively propagates all functions from all compatible types of the source type to all compatible types of the destination type, including the longest type. Therefore, MLTA will not drop any function targets during the type casting. Therefore, it will not introduce any FN compared to the scenario where no cast of sensitive types is allowed. In summary, even with type casting, MLTA does not introduce any FN.
(3) MLTA may have fewer FPs than FLTA. With the multi-layer confinement, for each source type, MLTA may have fewer FPs. For programs with many type layers, MLTA can achieve significantly fewer FPs than FLTA, as we will show in §7.2. □

Theorem 1. (1) MLTA does not introduce any FN. (2) FLTA may have FNs. (3) MLTA can have fewer FPs than FLTA.

Proof. Lemma 1 and Lemma 2 indicate the theorem. □

7 Evaluation

We have provided our formal analysis of MLTA and implemented TypeDive based the LLVM compiler infrastructure. In this section, we evaluate TypeDive in the following aspects.

- **Scalability.** TypeDive excels in large programs where composite types are prevalent. The evaluation should first confirm that TypeDive can scale to extremely large programs (§7.1).
- **Effectiveness.** Reducing icall targets is the main goal of TypeDive. The evaluation should show to which extent TypeDive can reduce the number of icall targets (§7.2).
- **No false negatives.** Avoiding potential false negatives is another design goal of TypeDive. The evaluation should confirm that TypeDive does not miss valid icall targets (§7.3).
- **Important use cases.** As a foundational approach, we apply TypeDive to assist static bug-detection mechanisms to demonstrate its usefulness (§7.4).

Experimental setup. We apply TypeDive to real-world system programs, including the Linux kernel of version `5.1.0`, the FreeBSD kernel of version `12.0-RELEASE`, and the Firefox browser (C++) with the top commit number `f2cd91cb305f`. While the Linux kernel is compiled with the `allyesconfig` option (including as many modules as possible), the FreeBSD kernel and the Firefox browser are compiled with the default configuration. All programs are compiled with flags `-O0 -g -fno-inlining`. These flags make sure that the generated binary accurately contains all debug information, such as line numbers and function names, which will simplify our verification on false negatives (§7.3) and analysis on detected bugs (§7.4). However, TypeDive by design should work on any other

System	Modules	SLoC	Loading	Analysis
Linux	17,558	10,330K	2m 6s	1m 40s
FreeBSD	1,481	1,232K	6s	6s
Firefox	1,541	982K	27s	1m 25s

Table 2: Scalability of TypeDive.

compilation configurations, including higher optimization levels (e.g., -O2) and aggressive code inlining. Although the evaluation numbers could be different from what we report here, we believe the effectiveness of TypeDive should be similar for other compilation options. We leave the evaluation with other options as future work. The experimental machine is equipped with Ubuntu 18.04 LTS with LLVM version-8.0 installed. The machine has a 64GB RAM and an Intel CPU (Xeon R CPU E5-1660 v4, 3.20 GHz) with 8 cores.

7.1 Scalability of TypeDive

The results are shown in Table 2. TypeDive can finish constructing the global call-graph for 10 million lines of code in the Linux kernel within four minutes. In fact, more than two minutes are spent in loading bitcode files. The promising scalability benefits from that TypeDive avoids data-flow analysis or pointer analysis, but uses only type analysis. TypeDive's split of multi-layer types into two-layer types also helps reduce the storage and computation cost.

7.2 Reduction of Indirect-Call Targets

The main goal of TypeDive is to reduce false positives in finding icall targets. We evaluate the effectiveness of TypeDive by measuring to what extent TypeDive can reduce icall targets. In this evaluation, we take the total number of address-taken functions as the baseline and count how many false-positive targets can be removed. We report the average number of icall targets identified by TypeDive, for the three system programs, present the distribution of numbers of targets, and breakdown the reduction by layers.

Average number of targets. Table 3 shows the average numbers of icall targets reported by TypeDive. Column `iCall for TypeDive` denotes how many icalls benefit from TypeDive. If an icall does not load the function pointer from a composite-type object, or TypeDive cannot ensure zero false negatives (e.g., the composite type has escaped), the icall does not qualify TypeDive and thus is excluded in this column. The results show that most icalls can enjoy the reduction offered by TypeDive. In particular, 81% icalls in the Linux kernel have their targets refined by TypeDive. Column `&Func.` denotes the number of address-taken functions. All tested system programs have a large number of address-taken functions. Therefore, traditional coarse-grained CFI techniques [1] that conservatively take all these functions as valid icall targets will have weak protection. The column `Ave. target (signature)` shows the average number of targets after applying the signature-based (i.e., function type, first-layer) matching. The last column shows the final average number of targets after applying TypeDive, which is based on the icalls that qualify TypeDive.

The average numbers are calculated over icalls that can benefit from MLTA, i.e., the types have at least two layers. When calculating the average over all icalls (i.e., also including the ones cannot benefit from MLTA), the average numbers of indirect-call targets will be amortized: they are 31.6, 11.5, and 44.6 for Linux, FreeBSD, and Firefox, respectively.

System	Total iCall	iCall for TypeDive	&Func.	Ave. target (signature)	Ave. target (TypeDive)
Linux	58K	47K (81%)	180K	134	7.7 (94% ↓)
FreeBSD	6.3K	4.0K (64%)	8.7K	25.5	3.5 (86% ↓)
Firefox	37K	23K (63%)	58K	115	1.8 (98% ↓)

Table 3: Reduction of icall targets. `&Func` denotes the number of address-taken functions. The reduction percentage is based on the targets reported by the signature-based approach (i.e., the first-layer type analysis).

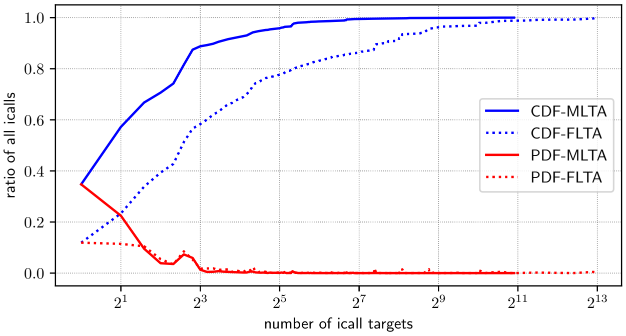

Figure 4: Distribution of the numbers of icall targets. MLTA identifies more icalls with fewer targets (less than four targets) while FLTA infers more icalls with more targets. Both of them have a long tail, where icalls under MLTA have at most 1,914 targets while icalls under FLTA have at most 7,983 targets.

The *average number of targets* reflects the complexity of static analysis to traverse the whole CFG for any particular analysis. As we treat TypeDive as a general tool for generating accurate CFG, the average number of targets is a reasonable metric. However, it may fail to measure the benefit of using TypeDive for defense mechanisms such as improving CFI [15, 55]—as will be shown as in the next evaluation of target distribution, some icalls still have a large number of targets.

Distribution of the numbers of targets. We show the distribution of the numbers of icall targets in Figure 4. The CDF (cumulative distribution function) of MLTA always has a larger value than CDF-FLTA, showing that MLTA consistently identifies fewer targets for icalls than FLTA. Based on the PDF (probability density function) graphs, MLTA identifies more icalls with less than four targets, while FLTA finds more icalls with at least four targets. All graphs have long tails, indicating that under static analyses, some icalls still permit a large number of targets. For example, under the MLTA analysis, icalls have at most 1,914 targets, while under FLTA, icalls could have up to 7,983 targets. For these icalls, we may seek help from runtime information to reduce the number of targets [22, 36, 48].

Reduction breakdowns. We further evaluated the extent to which the type matching at each layer can refine the icall targets. In this evaluation, we make TypeDive configurable to layers by specifying the maximum layers. Once TypeDive reaches the maximum number, it stops the analysis and reports the current target set. Table 4 shows the results. We found that the first couple of layers can dramatically reduce the icall targets. This is because types for each layer are largely independent. However, the reduction becomes steady after four layers. TypeDive can still slightly reduce the targets for the Linux kernel given the extremely complex code. Based on the results, we believe that a layer number of five should be able to remove most targets for general programs. Several recent works propose similar ideas for confining icall targets [15, 29], and most of them can be treated as special two-layer type analysis. We will discuss the difference between TypeDive and those works in §9.

False positives. Static analysis of icall resolution has to consider all possible invocations of one icall instruction, where each invocation may have different targets by design instead of a *unique code target* [22]. However, our evaluation shows that MLTA has dramatically reduced the average number even for extremely complicated programs like the Linux kernel. Especially for Firefox, the average number is less than 2. As shown in Table 3, MLTA significantly improves over the state-of-the-art FLTA techniques [35, 46], where the average number of icall targets is reduced by around 90%.

7.2.1 Comparisons with existing works. The work by Ge et al. [15] uses taint analysis and type analysis to find icall targets. The authors use the average number of icall targets to evaluate the analysis accuracy and provides the measurement on several operating systems. One common benchmark between our work and [15] is FreeBSD, where the latter reports that each icall has 6.64 targets on average. Although our result is better than that, we have to clarify two differences between the two evaluations. First, our result is calculated over icalls that MLTA can provide two-or-higher layer result, which is 64% of all icalls. When we calculate the average number over all icalls (i.e., also including the ones cannot benefit from MLTA), the number is about 12, higher than 6.64 reported in [15]. Second, our result does not count icalls in assembly code, which usually has a single target. [15] takes them into consideration; therefore, the average number should be higher if they exclude these single-target icalls. Other than the accuracy, [15] requires manual fixing when violations of assumptions are detected. In comparison, TypeDive automatically falls back to an upper layer to avoid false negatives. Further, since [15] uses static taint-analysis, it will take a longer time to finish.

Pointer analysis is an alternative approach to finding icall targets. Recent work [14] (section 5.4) compares the effectiveness of type-based CFI (*i.e.*, based on FLTA) and that of pointer analysis–based CFI (*i.e.*, using SVF [44]). The comparison indicates two results. First, there is no strong evidence showing that pointer analysis can lead to more accurate results in resolving icall targets. Among 14 evaluated programs, FLTA achieves more accurate results than SVF on eight programs but has worse results on the other four programs. Since MLTA performs much better than FLTA on kernels and a browser, we believe it will provide better results than pointer analysis–based approaches in general. Second, for two out of 14 programs, specifically, Nginx (139 kLoC) and Httpd (267 kLoC), SVF cannot finish in a reasonable time and crashes after five-hour running. This result shows the limited scalability of pointer analysis–based approaches. Our technique, MLTA, is able to finish the analysis for the Linux kernel (millions of lines of code) within four minutes. Therefore, compared to pointer analysis–based approaches, we believe that MLTA is able to find more accurate icall targets more efficiently.

System	Baseline	1-Layer	2-Layer	3-Layer	4-Layer	5-Layer
Linux	180K	134	9.12	8.03	7.91	7.78
FreeBSD	8.7K	25.5	3.53	3.50	3.49	3.49
Firefox	58K	115	1.86	1.84	1.82	1.82

Table 4: Breakdowns of target reduction by layers. Here we measure icalls that support multi-layer type analysis, which cover most icalls.

7.3 False-Negative Analysis

To understand the false negatives of MLTA and to compare with the FLTA, we empirically collect the icall traces of the Linux kernel and the Firefox browser as the partial ground-truth.

7.3.1 Collecting Traces We first present how we use Intel PT (Processor Tracing) and QEMU to accurately collect icall traces.

PT-based tracing. Intel PT is a hardware feature that records the control-flow of the execution with nearly no overhead. We use the most-recent feature of Intel PT `ptwrite`, to realize our trace collection[1]. A `ptwrite` instruction can dump any value from its operand into the trace, and Intel PT can generate another packet to log the address of the instruction. We instrument the program to insert a `ptwrite` before each icall, using the function pointer as the operand. At runtime, `ptwrite` will dump the target into PT trace and log the address of the `ptwrite` instruction, which is immediately followed by the real icall. We implement the instrumentation as an LLVM pass with 74 lines of C++ code, which can support any programs compatible with the LLVM-compiler.

QEMU-based tracing. Unfortunately, the Linux kernel does not fully support LLVM compilation, and collecting the PT trace for the kernel requires modification of both userspace code and the kernel [41]. Therefore, we instead modified QEMU to log the source and target of each icall. We find that Linux kernel implements most icall with thunks, where each thunk is a function containing only one indirect jump instruction that uses one specific register as the target, like `__x86_indirect_thunk_rax`. An icall instruction is realized with a direct call to one thunk. This simplifies our logging in QEMU, where we check whether the current code is within these thunks. If so, we log the target address in the specific register and the return address on the stack which immediately follows the original icall instruction. We implemented the QEMU-based tracing by adding 66 lines of code to the `cpu-exec.c` file of QEMU-2.9.0.

7.3.2 Results for False-Negative Analysis. We use the tracing tools to collect icall traces for evaluating false negatives. Since the traces include the source code information, the source file, and line number, we use such information for matching the callees. Specifically, given a trace, we use the caller to query the global call-graph constructed by TypeDive and obtain the icall targets. If the callee in the trace is contained in the icall targets, we say that TypeDive correctly identified the callee; otherwise, it is a false negative.

[1] Without `ptwrite`, we can perform the same evaluation with Intel PT. However, in that case, we have to sequentially decode all PT packages, which is time-consuming and suffers the data-loss problem [22].

Checking for Firefox. By manually searching google.com and youtube.com in the Firefox browser, our tracing tool in total collected 50k traces (i.e., indirect caller and caller pairs). However, the majority of them are repeating, or the corresponding bitcode files are not available with our experimental setup. After removing these cases, we finally obtained 1,595 unique icall traces. The evaluation results show that TypeDive missed only one callee. After investigating the cause, we found that a function pointer is loaded from a type other than the one confining the callee. Therefore, there is a cast between these two types. However, we could not find the casting in the analysis code. That is, the bitcode file that contains the type casting is not in our analysis scope. Thus, TypeDive missed the callee. We believe, once the bitcode file is included in the analysis, TypeDive will successfully catch it. As we will discuss in §8, TypeDive requires all source code to be available to comprehensively identify all type propagations.

Checking for Linux. To collect the traces for Linux, we run the kernel on QEMU and employ kAFL [41] to explore paths. Similarly, after removing cases that are repeating or do not have source code in our analysis scope, we in total collected 3,566 unique traces. Using the same way, we compared the traces with the icall targets reported by TypeDive. We found that TypeDive originally missed five callees in the recorded traces. We then investigated these cases. We found that these callees are missed by the first layer type analysis—function type matching. Specifically, the general types of parameters, such as `long int` and `void *`, are used in an interleaving way. Out of the missed cases, one is caused by the implicit casting between `long int` and `char *`, and four are caused by the implicit casting between `int` and `unsigned int`. Since the current implementation of TypeDive does not support the casting between primitive types, these cases are missed. This is a traditional problem in CFI works [27, 35, 46]. Current works solve the problem by equalizing certain primitive types, such as integers and pointers. TypeDive can also solve this problem using the same approach. However, given the small number of type violations, we will leave the integration of the approach for future work. The results show that although existing function-type matching may have false negatives, TypeDive does not introduce extra false negatives to it.

The empirical evaluation evidences that TypeDive does not introduce more false negatives to existing FLTA. However, due to the limited code coverage of dynamic executions, such evaluation can never be used as a complete proof. Applying TypeDive to more programs with diverse inputs (e.g., through fuzzing [18, 37, 41, 51] or symbolic execution [7, 11, 43]) would improve the reliability of the evaluation.

7.4 TypeDive for Semantic-Bug Detection

While the goal of TypeDive is to refine icall targets, we believe that TypeDive is also useful for finding semantic bugs that otherwise cannot be detected through shallow specifications such as no out-of-bound access. Our insight is that the targets of icalls are often peer functions that implement pre-defined interfaces, thus sharing similar semantics. By cross-checking peer functions, we can detect deviations or contradictions as potential bugs, which avoids the challenging problem of understanding semantics.

Detection approach. We also employ TypeDive to detect two classes of semantics errors, missing initialization and missing security check. Since TypeDive identifies icall targets that are typically semantically equivalent, we cross-check how parameters and functions are used in these peer targets. For example, if a parameter from an icall is commonly initialized in all peers except one function, we report a potential missing-initialization bug in this function. Also, if a function call or a parameter is commonly checked but not in one peer function, we also report it as a potential missing-check bug. We have realized the detection based on LLVM. In the detection, we implemented an intra-procedural data-flow analysis to reason about whether parameters are checked and initialized, and whether return values of function calls are checked. We then statistically rank the potential bug cases based on the ratio of the peers which do not have the issues.

Detection results. We have applied our detection to the Linux kernel. Since TypeDive reported thousands of ranked potential bugs, we chose the top 50 cases for each class of bugs and manually confirmed them. In total, we have confirmed 10 new missing-initialization bugs and 25 missing-check bugs. The details are shown in Table 5 and Table 6. Each shaded line contains one icall, including its location and the number of inferred targets by FLTA and MLTA. As we can see, MLTA significantly reduces the number of targets, which makes our bug detection efficient and reduces the manual effort for confirming the bugs. The lines following each shaded line show the bugs, including their subsystem, source file, function name, and the affected variable. We also provide the impact of each bug. For example, in Table 5 we show the number of bytes uninitialized (UI) or leaked (LK). The results confirm that TypeDive can assist semantic-bug detection, as it accurately identifies icall targets to allow effective cross-checking.

Existing detection methods on missing initialization [31] and missing security checks [32] either do not handle indirect calls, where they will miss all bugs we find here, or they use limited type information to infer icall targets, where we can expect a much higher false-positive rate. For example, to confirm the bugs shown in Table 5, we have to manually check *518* targets when using FLTA. However, the number is significantly reduced to *35* (6.8%) when using MLTA, confirming the usefulness of TypeDive in bug detection based on static analysis.

Case study: Bug hidden behind two indirect-calls. We use the first missing-initialization bug to further demonstrate the benefit of using MLTA for bug detection. This bug is an information leakage bug in the Linux kernel. We manually checked it and confirmed that it can leak a 4-byte memory region. To find this bug, one static bug detector will reach an icall in line 511 of file `oaktrail_crtc.c`. FLTA will identify 13 targets, and the bug-detector has to check them one by one, which may analyze 13 functions to get a chance to find this bug in the worst case. More importantly, if the detection employs cross-checking, including the 10 unrelated functions will likely bury the true bug. In comparison, MLTA only reports 3 targets, saving 77% of the analysis effort. More than that, the buggy function `cdv_intel_find_dp_pl` contains another icall, which the bug-detector has to analyze to confirm the bug. For this new icall, FLTA reports 54 targets while MLTA only permits only 3. By checking the latter set of targets, we quickly confirm this information

[Subsys] File	Function	Variable	Impact
drivers/gpu/drm/gma500/oaktrail_crtc.c:511 [13->3]			
[drm] cdv_intel_display.c	cdv_intel_find_dp_pll	clock	4B UI
[drm] oaktrail_crtc.c	mrst_sdvo_find_best_pll	clock	16B LK
[drm] oaktrail_crtc.c	mrst_lvds_find_best_pll	clock	16B LK
drivers/media/v4l2-core/v4l2-ioctl.c:1509 [438->5]			
[media] rcar_drif.c	rcar_drif_g_fmt_sdr_cap	f	24B UI
drivers/staging/rtl8188eu/core/rtw_security.c:229 [18->6]			
[crypto] lib80211_crypt_wep.c	lib80211_wep_set_key	wep	25B UI
[staging] rtllib_crypt_wep.c	prism2_wep_set_key	wep	25B UI
drivers/staging/media/davinci_vpfe/dm365_ipipe.c:1277 [36->18]			
[staging] dm365_ipipe.c	ipipe_set_wb_params	wbal	8B UI
[staging] dm365_ipipe.c	ipipe_set_rgb2rgb_params	rgb2rgb_ defaults	12B UI
[staging] dm365_ipipe.c	ipipe_set_rgb2yuv_params	rgb2yuv_ defaults	4B UI
crypto/af_alg.c:302 [13->3]			
[crypto] algif_hash.c	hash_accept_parent_nokey	ctx	680B UI

Table 5: New missing-initialization bugs found with TypeDive. Each shaded line shows the location of the icall, followed by the number of targets inferred by FLTA and MLTA. "UI" denotes that memory is not initialized properly, and "LK" denotes that the uninitialization causes information leakage, with "xB" indicates the size of the uninitialized memory.

leakage bug. Therefore, to find this bug, FLTA provides 702 possible paths while MLTA reduces the number to merely 9. The problem will be exaggerated when the paths include branches, leading to path explosion.

8 Discussion

Indirect jumps. Our current prototype of TypeDive does not identify indirect-jump targets. We observe that indirect jumps are mainly used for `switch` statements, where the compiler usually identifies all `cases` of the `switch` and puts them inside a read-only `jump table`. Therefore, the targets have been resolved by existing compilers, and we do not have to use MLTA to find them. Previous work also skips indirect jumps with the similar reason [15, 22]. However, if an indirect jump is used intentionally, we can definitely use the same algorithm (Algorithm 1) to find the function targets.

Complementing to data-flow analysis. Although MLTA is more scalable than data-flow analysis, they are complementary to each other rather than exclusive. In fact, MLTA can benefit from small-scale data-flow analysis (e.g., without aliases involved). For example, we can rely on taint analysis to find the multi-layer type for each variable within the same function. On the other hand, a data-flow analysis may use our MLTA to infer icall targets so that it can continue the analysis across function boundaries. A data-flow analysis equipped with MLTA will also be more scalable.

Indirect-call in assembly or binary. MLTA assumes that each function pointer has a well-defined type, which is violated in assembly and binary code. Fortunately, previous works show that icalls and jumps written in assembly code can be easily resolved with manual effort, in both user-space and system programs [15, 35]. A solution could be lifting assembly code into LLVM IR, and propagating the types of input operands and output operands. This solution may also work for icalls in binary. Further, we can infer the type information for each variable through reverse engineering techniques [28, 30], and use the inferred type for MLTA. Supporting MLTA in assembly or binary is out of the scope of this work.

Variable-argument functions. Variable-argument function reduces the accurate of type-based function matching. Our current prototype of MLTA uses the hash of the function type string as the function type, and therefore, it conservatively concludes a type matching as long as the fixed part of the arguments matches. However, this is not a new problem introduced in MLTA, which stems from function-type matching used in FLTA. Therefore, we leave the study of the problem as future work.

Data-only attacks. TypeDive provides an accurate control-flow graph (CFG) for CFI solutions to achieve stronger protections on control-flow transfers. However, data-only attacks will survive all CFI solutions, even the ones enhanced with TypeDive, as these attacks do not change any control-flow [10]. Recent works have shown that data-only attacks are expressive [21], and can be constructed automatically [20, 23, 38]. In fact, both attackers and defenders of data-only attacks can get benefit from the accurate CFG generated by TypeDive: while attackers can find reachable data-oriented gadgets with less false positives [21], which will simplify the attack construction process, defenders can calculate more accurate data flows, which helps realize more efficient and stronger data-flow integrity [9].

9 Related Work

Struct location vector. Li *et al.* propose FINE-CFI [29] which uses structure location vector (SLV) to confine icalls. The idea of SLV is to identify the location of function pointer in a struct, and to use location information to match icall targets. Compared with MLTA, SLV has two important limitations. First, when constructing the vector, SLV only considers struct members that are also structs (*i.e.*, nested structs), which does not include pointers of structs. This will cause false negatives because struct objects in OS kernels typically include an object as a field through pointers to avoid copying of the object memory. Second, SLV does not handle type casting generally. Instead, they treat these problems as corner-cases and try to find the real-type before casting. However, as we show in the paper, casting is a fundamental and common problem that will lead to false negatives if not properly handled.

Taint analysis. Ge *et al.* propose to use taint analysis to find icall targets for operating systems [15]. Their observation is that there are no complicated operations on function pointers inside the kernel and thus data-flow analysis will not face too many challenges. One of the taint propagation rules is that once a function pointer is assigned to a structure field, they taint the field for all memory objects of that structure's type. This policy is similar to our MLTA but is only applied to the first two layers: the function pointer and the structure field. MLTA can capture such relationships between any two layers. Moreover, [15] requires manual fixing when violations are detected, like data pointers pointing to function pointers. MLTA is elastic to automatically fall back to the inner layer to avoid false negatives.

CFG construction. Researchers have spent a large amount of effort in resolving icall targets for enforcing control-flow integrity (CFI) [1]. If program source code is not available, icall targets are conservatively set to all functions [53, 55]. That is, each icall is allowed to reach any valid function. When program source code is available, function-type information is used to infer fine-grained targets, where each icall can only go to one of the type-matched functions. Forward-edge CFI only uses partial function type—the number of parameters to confine icall [46] while modular-CFI uses the complete function type for type matching [35]. A mixed solution tries to infer the function type information from the program binary and uses argument number to match callers and callees [49]. Compared with these solutions, MLTA uses type information of multiple layers to dramatically refine icall targets. Recent CFI solutions leverage runtime information to restrict runtime control-flow transfer [13, 16, 22, 36, 48], which have a different problem scope—ensuring that a runtime target is valid. Instead, MLTA aims to infer a complete set of targets for all icalls statically.

Resolving C++ virtual calls. Most related work on resolving C++ virtual functions relies on class hierarchy analysis to build the class hierarchy tree [19, 25, 39, 46, 54]. That is, each virtual call can divert the control-flow to the function implemented in the current class, or those in derived classes. Essentially, such approaches use an expanded single-layer type for finding targets. In comparison, MLTA uses types of multiple layers to further refine the targets, and only virtual functions in a derived class that is cast to a base class are included as valid targets.

10 Conclusion

In this paper, we presented MLTA, a new approach that effectively refines indirect-call targets for both C and C++ programs. We implemented MLTA in a system called TypeDive. TypeDive uses independent types of different layers to dramatically refine the targets. TypeDive decouples types from data flows and is elastic to avoid false negatives. Evaluation results show that TypeDive can reduce 86%-98% more indirect-call targets than existing approaches based on function-type matching. We believe that TypeDive can significantly benefit existing static-analysis and system-hardening techniques. As an illustrating example, we also used TypeDive to effectively find 25 new missing-check and 10 new missing-initialization bugs in the Linux kernel.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. Kangjie Lu was supported in part by the NSF award CNS-1815621. Hong Hu was supported in part by the ONR under grants N00014-17-1-2895 and N00014-18-1-2662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[3] S. Biallas, M. C. Olesen, F. Cassez, and R. Huuck. PtrTracker: Pragmatic Pointer Analysis. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013.

[4] T. Bletsch, X. Jiang, and V. Freeh. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[5] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[6] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[8] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[9] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2005.

[11] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[12] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[13] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[14] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. On the Effectiveness of Type-Based Control Flow Integrity. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[15] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194. IEEE, 2016.

[16] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[17] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[18] Google. syzkaller - Kernel Fuzzer. https://github.com/google/syzkaller, 2019.

[19] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[20] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[21] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[22] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[23] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[24] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[25] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[26] Y. Kang, B. Ray, and S. Jana. APEx: Automated Inference of Error Specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on*

Automated Software Engineering, pages 472–482. ACM, 2016.

[27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[28] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[29] J. Li, X. Tong, F. Zhang, and J. Ma. FINE-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Transactions on Information Forensics and Security*, 13(6):1535–1550, 2018.

[30] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2010.

[31] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[32] K. Lu, A. Pakki, and Q. Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2018.

[33] A. Milanova, A. Rountev, and B. G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engg.*, 11(1):7–26, Jan. 2004. ISSN 0928-8910.

[34] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[35] B. Niu and G. Tan. Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[36] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[37] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[38] J. Pewny, P. Koppe, and T. Holz. STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming. In *Proceeding of the 4th IEEE European Symposium on Security*.

[39] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[40] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[41] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[42] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[43] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[44] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[45] Y. Sui and J. Xue. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++. *IEEE Transactions on Software Engineering*, PP, 09 2018.

[46] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, pages 941–955, 2014.

[47] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, 2018.

[48] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[49] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[50] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[51] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[52] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. APISan: Sanitizing API Usages through Semantic Cross-checking. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[53] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[54] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[55] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

A Appendix: New Missing-Check Bugs

[Subsys]	**File**	**Function**	**Variable**	**missed checks**	
[net]	cfg80211.c	mwifiex_cfg80211_connect	sme	args	
[firmware]	edd.c	edd_show_mbr_signature	edev	args	
[char]	tpm_infineon.c	static int tpm_inf_recv	count	args	
[dma]	omap-dma.c	omap_dma_prep_dma_cyclic	period_len	args	
[treewide]	se.c	st21nfca_connectivity_event_received	transaction	retval	devm_kzalloc()
[gpio]	gpio-aspeed.c	aspeed_gpio_probe	gpio->offset_timer	retval	devm_kzalloc()
[media]	tda18250.c	tda18250_set_params	ret	retval	regmap_write_bits()
[clk]	gcc-ipq4019.c	clk_cpu_div_set_rate	ret	retval	regmap_update_bits()
[ASoC]	cs35l34.c	cs35l34_sdin_event	ret	retval	regmap_update_bits()
[rtc]	rtc-rx8010.c	rx8010_set_time	ret	retval	i2c_smbus_write_byte_data()
[mfd]	tps65010.c	tps65010_work	status	retval	i2c_smbus_write_byte_data()
[treewide]	realtek_cr.c	rts51x_invoke_transport	ret	retval	usb_autopm_get_interface()
[nfp]	lag_conf.c	nfp_fl_lag_do_work	acti_netdevs	retval	kmalloc_array()
[scsi]	mptscsih.c	mptscsih_IssueTaskMgmt	timeleft	retval	wait_for_completion_timeout()
[misc]	tifm_7xx1.c	tifm_7xx1_resume	timeout	retval	wait_for_completion_timeout()
[Input]	usbtouchscreen.c	nexio_read_data	ret	retval	usb_submit_urb()
[USB]	iuu_phoenix.c	iuu_rxcmd	result	retval	usb_submit_urb()
[USB]	iuu_phoenix.c	read_rxcmd_callback	result	retval	usb_submit_urb()
[USB]	iuu_phoenix.c	iuu_status_callback	result	retval	usb_submit_urb()
[USB]	kobil_sct.c	kobil_open	result	retval	usb_submit_urb()
[USB]	kobil_sct.c	kobil_write	result	retval	usb_submit_urb()
[net/ncsi]	ncsi-netlink.c	ncsi_pkg_info_all_nl	attr	retval	nla_nest_start()
[netfilter]	conntrack.c	ovs_ct_limit_cmd_get	nla_reply	retval	nla_nest_start()
[dmaengine]	fsl-edma-common.c	fsl_edma_prep_slave_sg	fsl_chan->tcd_pool	retval	dma_pool_create()
[mtd]	generic.c	generic_onenand_probe	err	retval	mtd_device_register()

Table 6: New missing-check bugs found with TypeDive. Each row shows one missing-check bug, including the subsystem it belongs to, the file name, the function whether the missing check is in, and variable that requires extra check, and the source code the variable: it is from the arguments, or is a return value from another function.