Practical Program Modularization with Type-Based Dependence Analysis

Kangjie Lu
University of Minnesota

***Abstract*—Today's software programs are bloating and have become extremely complex. As there is typically no internal isolation among modules in a program, a vulnerability can be exploited to corrupt the memory and take control of the whole program. Program modularization is thus a promising security mechanism that splits a complex program into smaller modules, so that memory-access instructions can be constrained from corrupting irrelevant modules. A general approach to realizing program modularization is dependence analysis which determines if an instruction is independent of specific code or data; and if so, it can be modularized. Unfortunately, dependence analysis in complex programs is generally considered infeasible, due to problems in data-flow analysis, such as unknown indirect-call targets, pointer aliasing, and path explosion. As a result, we have not seen practical automated program modularization built on dependence analysis.**

This paper presents a breakthrough—Type-based dependence analysis for Program Modularization (TyPM). Its goal is to determine which modules in a program can never pass a type of object (including references) to a memory-access instruction; therefore, objects of this type that are created by these modules can never be valid targets of the instruction. The idea is to employ a type-based analysis to first determine which types of data flows can take place between two modules, and then transitively resolve all dependent modules of a memory-access instruction, with respect to the specific type. Such an approach avoids the data-flow analysis and can be practical. We develop two important security applications based on TyPM: refining indirect-call targets and protecting critical data structures. We extensively evaluate TyPM with various system software, including an OS kernel, a hypervisor, UEFI firmware, and a browser. Results show that on average TyPM additionally refines indirect-call targets produced by the state of the art by 31%-91%. TyPM can also remove 99.9% of modules for memory-write instructions to prevent them from corrupting critical data structures in the Linux kernel.

1. Introduction

The complexity of modern software keeps growing, as more and more features are introduced. For example, the Linux kernel and the Chromium browser both have exceeded over 30 million lines of code with more than 20 thousand modules. This trend is detrimental to security; there is typically no internal isolation in a program, so a vulnerability can be exploited to corrupt memory and take control of the control/data flow of the whole program. Naturally, program modularization can effectively mitigate the problem, which enforces the "least privilege" security principle. Program modularization splits a complex program into smaller modules, so that a vulnerability in one module can be constrained from affecting other irrelevant modules.

Program modularization generally requires dependence analysis which is to identify all possible control/data dependencies of an instruction; independent code/data thus can be safely excluded from the targets of the instruction. A standard approach is to first identify the control dependencies, and then perform data-flow analysis within the reachable control-flow graph to identify data dependencies. Note that soundness (i.e., identifying all valid targets) is a priority, as missing a target may cause a program to fail at runtime,

Dependence analysis for complex C/C++ programs has been generally considered "mission impossible". Given a memory-access instruction, determining which code/data in the program can never be valid targets requires keeping track of both control and data flows from sources (such as a memory allocation) to the memory-access instruction. Such an analysis suffers from hard problems, including unknown indirect-call targets, inaccurate pointer aliasing, and path explosion. Our experimental results on the Linux kernel and Firefox show that the aliasing problem alone would render the data-flow analysis impractical, due to overwhelming false results and scalability issues. Moreover, as we will explain in §2, existing dependence analyses would not work for a multi-entry program.

Dependence analysis has thus been largely limited to "detection" or offline analysis where inaccuracy and unsoundness can be tolerated, but it has not been applied to runtime defenses in practice. For instance, the state of the art on program compartmentalization still has to rely on humans to first annotate compartments [38]. We believe that if dependence analysis, as a basic technique, is made practical, it can potentially enable various security applications, such as control-flow integrity (CFI) [6, 59, 56, 10, 43, 17, 60, 44, 15] which limits control transfers to only valid code targets; data-flow integrity (DFI) [53, 7, 12] which identifies critical data structures and allows only valid memory read/write instructions to access them; program compartmentalization [33, 38, 39, 34, 28] which partitions programs into isolated regions to contain attack impacts; and software debloating [47, 8, 46] which tries to remove unrelated modules.

In this paper, we present a breakthrough in dependence analysis, namely, Type-based dependence analysis for Program Modularization (TyPM). Given a memory-

access instruction, its goal is to determine a set of modules (independent modules) in the program such that the object targeted by the instruction cannot have emanated from these modules. Therefore, objects created by independent modules can never be valid targets of the instruction. TYPM achieves this goal with a new type-based static analysis, instead of impractical data-flow analysis. The high-level idea of TYPM is to employ a type-based analysis to identify the types of data flows between two modules, and then to transitively identify dependent modules of a specific type based on the type-labeled cross-module data flows. TYPM analyzes and leverages the type information in the communication channels between two modules. As will be described in §3.2, the communication channels are clearly defined—there are actually only two channels, function-call arguments and global variables, both with type information. Therefore, TYPM's analysis can be practical and sound.

TYPM has three stages. The first stage is typecasting analysis, which is to comprehensively collect all typecasts in each module. The output is a cast map that maintains the cast-from and cast-to types for each module. The second stage is to identify the types of data-flows between two modules, by analyzing globals and function-call arguments. The analysis of this stage is general: It obtains types in both objects and their references, parses nested types, and handles typecasting by querying the cast map. The output is a flow map that maintains possible types of data flows between two modules. Given a <type, module> pair as the input, which can be obtained from a memory-access instruction, the last stage is to transitively resolve the dependent modules by querying the flow map. The last stage additionally uses *type elevation* to improve the precision.

We identify several challenges when implementing TYPM and propose new techniques to address them. (1) *Identification of types and directions of data flows between modules*. Global variables and function arguments can be complicated, as they often involve nested data structures. We recursively label the types of their elements (e.g., nested fields of structs), as well as directions of data flows through a minimal variable-use analysis. (2) *Typecasting analysis for soundness*. A major challenge in ensuring the soundness of TYPM is handling the prevalent typecasting. An element should be labeled with all possible types. We develop a typecasting analysis that is conservative and handles general types (e.g., `void *`). We also handle `unions` which may manifest various types. (3) *Iterative resolving for unknown indirect calls*. TYPM must analyze arguments of all function calls, including indirect ones whose targets are unknown. To address the problem, we propose to start with the existing imprecise type matching, but further use an iterative algorithm to optimize the precision. (4) *Optimization through type elevation*. Finally, we optimize the precision of dependence analysis by using elevated types together with original types to make the resolving much more restrictive.

A common question is whether TYPM is able to capture indirect data flows between two modules, e.g., a reference (pointer) to an object is passed to other modules through shared memory, heap, globals, or their combinations. TYPM generally handles such cases based on an important insight. For an object (e.g., a data structure) of module $M1$ to become a target of module $M2$, a reference to the object, i.e., the object itself or a pointer to it, must be passed from module $M1$ to $M2$ (directly or transitively), through function arguments and/or globals. Otherwise, module $M2$ cannot access the object. Such passing of references will be generally captured by TYPM's type analysis against function arguments and/or globals. This is regardless of where the object is stored: globals, the heap, shared memory, or the stack. An object itself and its references are treated equally in TYPM in such a way that TYPM always obtains the types in them. Note that references also contain type information, and TYPM handles nested types, no matter whether a field is an object or a reference. We provide an example in Figure 2 that illustrates how TYPM generally captures both direct and indirect data flows by inspecting the two channels.

We implement TYPM based on LLVM and demonstrate its power with two security applications. First, we show how TYPM can further significantly remove invalid indirect-call targets produced by existing type matching [56, 43, 37, 18, 61, 36]. Intuitively, the type matching should focus on only the dependent modules, but not the whole program. For each indirect call, we thus use TYPM to collect the dependent modules and remove the targets outside the modules. Essentially, we achieve a *scope-aware* type matching for indirect-call targets. Second, we apply TYPM to protect critical data structures (e.g., the ones related to access control) in the Linux kernel. We collect the critical data structures, and for each memory-write, we use TYPM to determine whether the types of the critical data structures are possibly passed to the memory-write; if not, the memory-write should not target the critical data structures and should be constrained from accessing them.

We extensively evaluate TYPM using an OS kernel (Linux), a hypervisor (Xen), UEFI firmware (OVMF), and a browser (Firefox). The results show that TYPM can typically finish the analysis of millions of lines of code within minutes or hours. TYPM can further dramatically remove the indirect-call targets produced by existing type matching by 31%-91%. Our evaluation results also show that TYPM does not introduce false negatives under its model. For the protection of critical data structures, on average TYPM is able to remove 99.9% of modules for memory-write instructions, so that these instructions can be effectively constrained from corrupting critical data structures we collected in the Linux kernel. We believe that TYPM is a practical and easy-to-use tool that would enable a variety of security applications. We release our implementation of TYPM at https://github.com/umnsec/typm.

To summarize, we make the following contributions in the paper.

- **A breakthrough in dependence analysis.** We propose a new concept—type-based dependence analysis for program modularization (TYPM). Given a type of data and a target module, TYPM determines modules that can never pass such a type of data to the target module, and thus can be safely excluded. Such an analysis can be generally

used to constrain targets of memory-access instructions (read/write/execute) to improve memory safety or precision of program analysis. TYPM avoids the challenging inter-procedural data-flow analysis, and its approach is sound in principle.

- **New techniques for practicality.** To implement a practical TYPM, we propose multiple new techniques to address challenges. In addition to identifying type- and direction-labeled cross-module data flows, we perform typecasting analysis and handle unions to ensure soundness, iteratively resolve unknown indirect-call targets, and significantly optimize results through type elevation.
- **Two security applications.** We implement TYPM based on LLVM and apply TYPM to further improve the precision of existing indirect-call analysis and to protect critical data structures. Evaluation results show that TYPM is scalable and precise, and can significantly refine indirect-call targets produced by existing techniques and remove the vast majority of modules for memory-write instructions in the monolithic Linux kernel.

2. Background and Motivation

In this section, we present the background of dependence analysis and type analysis, as well as the motivation.

2.1. Limitations with Dependence Analysis

Dependence analysis is a foundational technique for security applications, which determines if data can propagate from one code location to another. Existing techniques [33, 32] typically employ control- and data-flow analyses. A standard approach is to first identify the control dependencies, and then perform data-flow analysis within the reachable control-flow graph to identify data dependencies. Such an approach has important limitations.

Missing control dependence in multi-entry programs. System software tends to have multiple entry points to provide APIs or services to higher-level applications. For example, the Linux kernel has more than 300 system calls and many interrupt handlers; all of these are entry points. In such programs, control dependence will not be a prerequisite for forming data dependence. In particular, two functions that are control-independent can still pass data (e.g., function pointers) through global variables or shared memory. As a result, existing data-dependence analyses that rely on control dependences will suffer from false negatives when the target program has multiple entry points. We believe that an accurate dependence analysis should not limit data flows within reachable control-flow graphs. In TYPM, we will broadly recover data dependences, without relying on control dependences.

Challenges with data-flow analysis. In this paper, we study inter-procedural data dependences between modules. Inter-procedural data-flow analysis has been known to suffer from multiple open problems. First, point-to analysis or alias analysis is required in data-flow analysis. Based on our experiments, well-known inter-procedural pointer analysis tools (e.g., SVF [54] and Andersen [21]) cannot scale to programs like OS kernels. Second, when in the conservative or sound mode, their results generally suffer from overwhelming false positives, and the imprecision becomes worse in larger programs. Third, data-flow analysis itself requires a complete call-graph which is not available due to indirect calls.

2.2. Requirements for Defense Mechanisms

When using dependence analysis in defense mechanisms like control-flow integrity (CFI) [6] and data-flow integrity (DFI) [12], extra challenges exist.

Precision. Precision is a primary goal for defense mechanisms, as it relates to protection effectiveness, as well as runtime and memory overheads. A general defense mechanism (e.g., CFI and DFI) is to limit control/data flows to only valid targets. The most common approach for computing the targets is through data-flow analysis, and the limitations have been discussed in §2.1. Alternatively, type matching [43, 36] searches for potential targets based on types. As we will describe in §2.3, it is still far from being precise.

Minimal false negatives. False negatives are often unacceptable in security applications, as they cause runtime errors. It is typically impractical to guarantee zero false negatives in computing targets due to non-standard code. Even for the conservative function-type matching, it still has false negatives [43]. However, we should ensure that the false negatives are minimal (e.g., a handful), so that we can manually modify the code in an affordable way. To allow such modification, the analysis should be able to report such non-standard code or potential false negatives.

2.3. Existing Scope-Unaware Type Analysis

Type analysis has recently shown promising results in security applications. For example, researchers proposed to use function types to match targets for indirect calls to achieve CFI [56, 43]. Recently advances [37, 61, 36, 18, 31] further use struct types to match function targets. However, these techniques are imprecise because they *globally* scan the whole program for targets and would thus include massive irrelevant targets—most modules in the program may not be related at all. More importantly, the precision of such a global search will become worse and worse, as the size of the target program grows.

3. The Concept of TYPM

In this paper, we aim to address the limitations with dependence analysis to make it practical for security applications beyond detection. We propose a new concept, Type-based dependence analysis for Program Modularization (TYPM), that practically determines which modules in a program can never pass a type of object (the object or its references) to a target module. TYPM avoids the challenging

inter-procedural data-flow analysis and instead employs a type-based analysis. In this section, we provide related definitions and important insights behind TYPM.

3.1. Definitions

We observe that a module is relatively self-contained and has clear boundaries. A module is defined as follows.

> ***Definition 1.*** Module: A module is a compilation unit [5]. A compilation unit refers to source code that is compiled and treated as a single logical unit. Typically, a C/C++ file corresponds to a module. In LLVM, each bitcode file is a module.

If we perform the dependence analysis only between modules, the internal complexity of a module would not impact the dependence analysis, which may allow to achieve a practical dependence analysis. We thus propose Type-based Dependence Analysis which is defined as follows.

> ***Definition 2.*** Type-based Dependence Analysis: Given an object type and a target module, it determines which modules in the program may propagate (both directly and indirectly) objects (or their references) of this type to the target module.

3.2. Insights

In the following, we present our insights on why TYPM would work in practice.

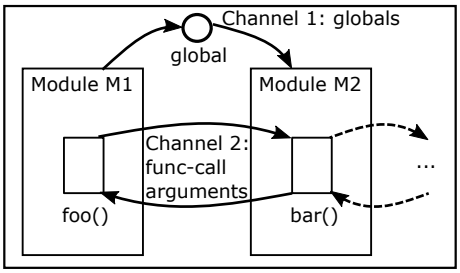

Figure 1: Two data-flow channels between modules

Only two data-flow channels between modules. Modules are relatively self-contained, as each of them is a separate compilation unit. The most important insight behind TYPM is that in a regular program, two modules have only two channels to pass references to memory objects: global variables (globals for short) and function-call arguments, as shown in Figure 1. In particular, to form a cross-module data flow through a global, one module writes to the global, and another module reads from it. To form a cross-module data flow through a function call, a module calls a function in another module and passes data through function arguments or return values. For simplicity, we use argument to represent both argument and return value. The data flows between two modules can be indirect (i.e., transitive): two modules without direct data flows can still be connected through other modules. Also, two channels can be used together for a cross-module data flow.

```
1  /* Global variable g */
2  struct global_t {
3      struct p_t *p;
4  } g;
5
6  /* Module M1: */
7  void foo() {
8      struct p_t *p = (struct p_t *)malloc(size);
9      bar(p);
10     g.p = p;
11     store_to_g(p); // store_to_g() is in M3
12 }
13
14 /* Module M2: */
15 void bar(struct p_t *p) {
16     use(p, g.p);
17     p = load_from_g(); // load_from_g() is in M4
18 }
```

Figure 2: A pseudo-code example that illustrates how module $M1$ can pass object reference p to module $M2$.

Figure 2 shows an example of how module $M1$ can pass object reference p to module $M2$ using the two channels. In the first case, on line 9, `foo()` in $M1$ calls `bar()` of $M2$ with argument p, so we identify a data flow of type `p_t` from $M1$ to $M2$. In the second case, line 10 stores p to global g in $M1$, and line 16 loads p from g in $M2$, so we identify two sub-flows: $M1$ –> g and g –> $M2$. By transitively connecting these two sub-flows, we can identify a data flow of type `p_t` from $M1$ to $M2$. The last case is an indirect data flow that involves both a function call and the global. Line 11 calls `store_to_g()` of module $M3$, which stores p to global g, while line 17 calls `load_from_g()` in module $M4$, which loads p from global g. Correspondingly, we will first identify two sub-flows: (1) $M1$ –> $M3$ –> g and (2) g –> $M4$ –> $M2$. By transitively connecting these two sub-flows, we can again identify the data flow of type `p_t` from $M1$ to $M2$. In all of these cases, the cross-module data flows of type `p_t` are identified by only inspecting the function calls and globals.

Such an observation indicates that by inspecting only the two channels between modules, without actually tracking the complicated data flows, we may still determine if two modules can pass data to each other. This shows a possibility of achieving dependence analysis without relying on data-flow analysis.

Types are restrictive to cross-module data flows. If we assume that two modules are dependent as long as there is a data-flow channel between them, the dependence analysis would be highly imprecise. Fortunately, we can leverage the type information (as well as directions) to restrict the data flows. Two modules may easily form a data flow, but the data flow is unlikely of a specific type (e.g., a specific field of a struct) and direction. More importantly, the likelihood is even lower when there are many other modules between

the two modules, as all these modules have to transitively form a data flow of this specific type.

Why TyPM can be effective and sound. The effectiveness of TyPM is based on the difficulty of forming a lengthy cross-module data flow that is of a specific type and direction. Typically there are many other modules between two modules, M_A and M_B. Given a possible path, $M_A, M_1, M_2, ..., M_n, M_B$, to form the dependence of type T through the path, there must exist a transitive and complete sequence of type-labeled and directional data flows, $DF_T(M_A, M_1)$, $DF_T(M_1, M_2)$, ..., $DF_T(M_n, M_B)$. Such conditions are restrictive, as all of the sub-flows must be labeled with type T and transitively connected. Missing any would not form the dependence.

The soundness of TyPM's approach is based on the fact that if a target object with type T in module M_A is indeed a target of a memory-access instruction, I, in M_B, then a reference to the object must be passed to I. The passing of the reference is essentially a data-flow process. That is, there must be a continuous data flow that can pass this specific type of object from M_A to M_B. This data flow will be identified by TyPM per its approach, so module M_A will be included as a dependent module of I.

4. Design of TyPM

TyPM is intended to be an effective and practical technique for program modularization and security mechanisms. In this section, we present how we design TyPM to achieve the goals.

4.1. An Overview

Assumptions. To realize TyPM, we make a few assumptions. (1) We assume that the target program does not intentionally access objects out of the boundary, which by itself is considered a violation of memory safety. (2) We assume that all source code of the target program is available and in our analysis scope. If a function is missing, TyPM may miss the data flows through the function and thus miss dependences. Hand-written assembly code is typically not a problem unless it involves typecasting, cross-module function calls, global accesses, or propagating data of target types. (3) We assume that the underlying compiler (e.g., LLVM) is correct in providing the type information of data.

The workflow. At a higher level, TyPM takes as input a target type and a module (a pair <type, module>), as well as the program, and automatically identifies all possible dependent modules. The <type, module> pair can be automatically obtained from a memory-access instruction: the type of accessed target and the module containing the instruction. Otherwise, a user can also manually specify the <type, module> pairs. Figure 3 shows the workflow of TyPM.

TyPM has three stages. The first stage is to comprehensively collect typecasts in each module. Its output is a casting map, $CastMap(M, T, \{T'\})$, which indicates that type T can be cast to types in set $\{T'\}$ in module M. The second stage is to label the types and directions of data flows between two modules. Its output is a flow map, $FlowMap(M_i, M_j, \{T\})$, which indicates that data flows of a type in set $\{T\}$) can take place from M_i to M_j. The analysis of this stage is general: It obtains types in both objects and their references, and parses nested types. The analysis also handles typecasting by recursively querying `CastMap`. For example, when we identify a data flow of type T from M_i to M_j, and types $\{T'\}$ are ever directly or transitively cast to T in M_i, then all the types in $\{T'\}$) will be added to the `FlowMap`, to conservatively indicate that data flows of these types may happen from M_i to M_j. Given any type-module pair $\langle T, M\rangle$ as an input, which can be obtained from any memory-access instruction, the final stage queries the `FlowMap` to compute dependent modules $\{M'\}$ that may have a direct or transitive data flow (of type T) to M. In particular, the stage determines the existence of such a data flow from M' to M by finding either $DF_T(M', M)$ or $M_1, M_2, ..., M_n$ such that $DF_T(M', M_1)$, $DF_T(M_1, M_2)$, ..., $DF_T(M_n, M)$ all exist. Other modules beyond $\{M'\}$ are considered independent ones. The last stage additionally uses *type elevation* to improve the precision.

4.2. Identifying Data-Flow Types and Directions

While the identification of data-flow types and directions is the second stage, we present it before typecasting analysis because typecasting analysis is to ensure the soundness of the identification. The identification parses both globals and function calls.

Type identification. The goal of the type identification is to conservatively collect all possible types of data in a global or a function argument. The general algorithm for the type identification is recursive. The algorithm iterates each field of the involved variable (either a global or an argument). If a field is a composite type (e.g., a struct), we recursively parse its fields. If a field is a pointer, we recursively parse the types of the pointed elements. Through such a recursive and exhaustive analysis, we can identify all possible types of data that can be passed through the global or argument. Note that this analysis does not require any data-flow analysis.

$$D_T(M, V, U) = \begin{cases} V \rightarrow M, & \text{if } U \text{ is a load,} \\ M \rightarrow V, & \text{if } U \text{ is a store, and } V \text{ is} \\ & \text{the pointer operand,} \\ Recursion(U), & \text{if } U \text{ is a cast or GEP,} \\ V \leftrightarrow M, & \text{otherwise.} \end{cases} \quad (1)$$

Direction identification. The goal of direction identification is to further restrict the data flows in a global or argument. Our design is to make the identification sound by conservatively performing a minimal intra-procedural data-flow analysis. We also enforce a fallback mechanism—If there is any uncertainty, we stop the analysis and consider the data flow bi-directional. The analysis parses each use (an instruction using the variable), U, of an argument or global, V, in module M, with the policy in Equation 1. $D_T(M, V, U)$ is the function that determines the direction of the data flow

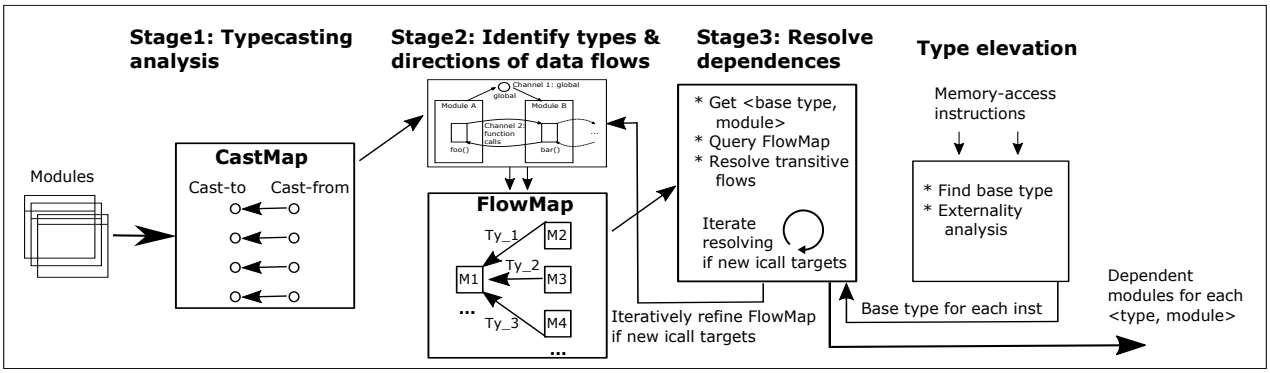

Figure 3: Overview of TYPM. It has three stages and two major maps. TYPM outputs dependent modules for a pair <type, module> which is obtained from a memory-access instruction.

(of type T) between M and V based on the use, U. `GEP` [3] is an instruction in LLVM that gets the pointer to a field based on the base pointer. In LLVM, a variable is a `value`, and a use is also a value; `Recursion(U)` is to recursively perform the identification on the use U of V.

Such a policy is conservative to ensure soundness. Basically, it only considers parsable `load` and `store` instructions for inferring directions. Whenever other cases are encountered, we assume that the data flow is bi-directional. Note that `memcpy` is also treated as a store operation.

Handling implicit data flows. Implicit data flows exist for globals and external functions. A global or an external function can be directly used in another module. For example, function `foo()` defined in module A can be directly called in module B without any explicit data flows that pass address of `foo()` from module A to module B. To capture such implicit data flows, we use a unique identifier to represent each global and external function, and use a map to link the unique identifier to the actual definition of the global or external function. This way, when module A calls `foo()`, we obtain its unique identifier and use the map to find its original definition, which essentially rebuilds the explicit data flows.

Building the FlowMap. With the results of $D_T(M, V, U)$, we build the `FlowMap` which maintains possible type- and direction-labeled data flows between two modules. We will identify a data flow $DF_T(M_i, M_j)$—a data flow of type T, from module M_i to module M_j—and add it to `FlowMap`, if:

1) Flow $V \rightarrow M_j$ of type T exists; V is an argument of function F in M_j; and M_i calls F; or
2) Flows $M_i \rightarrow V$ and $V \rightarrow M_j$ of type T exist; and V is a global.

4.3. Typecasting Analysis for Soundness

The identification of data-flow types and directions simply parses the globals and arguments. However, typecasting is prevalent. A type in a global or argument can be a cast-to or cast-from type, and its actual type can be others. In particular, casting a struct type to a general type (e.g., `void *` or `int64_t`) is common. Missing possible types would miss the type-based dependences. We thus propose a sound typecasting analysis to capture all possible types.

Given a module, for each cast operation, we obtain its cast-to and cast-from types, and use a map (`CastMap`) to record the typecasting relations. To achieve that, we conservatively analyze, (1) all globals (as well as their nested fields), (2) all instructions, and (3) all operands of instructions (an operand of an instruction can be a cast operator), to exhaustively capture all cast operations in the module.

In §4.2, whenever we encounter a type in parsing a global or argument, we query `CastMap` to recursively collect all related types. This way, we ensure that we collect all possible types that a global or an argument can potentially pass, in the presence of typecasting, which ensures the soundness. Note that `unions` may appear to be different types in modules, without an explicit casting operation. We handle `unions` by using only the names of `unions` but not the types (in LLVM, unions are treated as structs with a name). More details will be presented in §5.

4.4. Iterative Analysis for Indirect Calls and Dependences

Indirect calls have been an obstacle for inter-procedural program analysis. In TYPM, we also need to know the targets of indirect calls in the first place, so that we can perform the identification of type- and direction-labeled data flows through function arguments. Our strategy is to adopt the existing type matching (function-type matching and struct-type matching) to find indirect-call targets, which is sound (i.e., a target set is complete).

However, existing type matching is conservative in finding targets, and thus is imprecise; most targets are false positives. To address the problem, we propose an iterative algorithm to gradually refine the targets. The idea is that through one iteration, we can resolve the dependent modules of an indirect call; by limiting the type matching to the dependent modules, we refine its targets which are used in the next iteration. We stop the iterations until we cannot refine the targets. This way, we improve the precision of both indirect-call targets and dependence analysis.

Dependence resolving. With the type- and direction-labeled data flows in globals and arguments, we can build a large map (`FlowMap`) that maintains the type-labeled data flows between any two modules, which allows us to further resolve the dependence. Given a <type, module> pair, we simply query the map to recursively identify all dependent modules.

4.5. Type Elevation for Precision

In a real-world scenario, a typical case is to resolve the dependence for a specific memory-access instruction, such as indirect call and memory read/write. The type of the target of the memory-access instruction can be obtained through the type of the pointer operand of the instruction. For example, an indirect call tells the type of the function pointer (i.e., function type). Such a type can be generic. When we use the type to find dependences, the resolving can be very inclusive and returns a large set of dependent modules. To improve the precision, we propose *type elevation*.

```
1 /* Module: mm/mempolicy.c */
2 struct mempolicy *__get_vma_policy(struct vm_area_struct *vma) {
3     pol = vma->vm_ops->get_policy(vma, addr);
4 }
```

Figure 4: An example that qualifies type elevation.

The idea of type elevation is to use more complex types (thus more unique) to result in a smaller set of dependent modules. The pointer operand of a memory-access instruction is typically loaded from a struct, and oftentimes, a pointer of the struct is loaded from another more complex struct, and so forth. From the pointer operand of a memory-access instruction, we can get a complex and unique base type for it, i.e., type elevation. If we use both the base type and the original pointer-operand type to resolve the dependences, the resolving will certainly be much more restrictive, and the results would be much more precise. This is because the final results are essentially an *intersection* of the two sets obtained from the two types.

We use an example in Figure 4 to illustrate how type elevation works. In this case, we want to identify dependent modules of the indirect call on line 3. Initially, we obtain the type of the function pointer, `get_policy`, and use it to identify dependent modules, S_{T1}. Then, we elevate the type to the container of the function pointer, which is `struct vm_operations_struct` (from `vm_ops`) or even `struct vm_area_struct` (from `vma`). We can use the struct type to identify dependent modules, S_{T2}. The final dependent-module set is $S_{T1} \cap S_{T2}$, which would be smaller than the original S_{T1}.

However, such a simple design is not sound. When we elevate the type to a complex base type (e.g., struct), an object of this type may be created within the module, instead of being passed in from other modules. Therefore, if we still look for cross-module data flows of this base type, we may *not* find a dependence even if the original pointer type is indeed passed in from other modules, leading to false negatives or soundness issues. In other words, to use a base type for resolving dependences in a module, we must ensure that *this type of data is never internally created but is passed in from the external*.

Externality analysis for soundness. We thus propose *externality analysis* to avoid false negatives. The idea is to analyze the module to determine if an object of the elevated complex type is *ever* internally created. To conservatively identify any creation of the complex type, we analyze all initializers and store instructions that target the type. If any creation is found, we assume that data of this type may be internally generated, and the type should not be used as an elevated type, and we should fall back to the original type. Otherwise, it is safe to use it to restrictively resolve dependences. In the example in Figure 4, we found that `struct vm_area_struct` cannot be used as the base type because `vm_area_struct.vm_ops` is ever assigned in the module; that is, it may be created internally, and we have to *lower* the type. However, `struct vm_operations_struct` is indeed external, and we can use it for dependence resolving in this module. Note that once a type is lowered, we cannot elevate it again during a single resolving.

5. Implementation with LLVM

We implemented TYPM based on LLVM 15 as a pass. In this section, we present important considerations for the implementation.

Type representation and comparison. We found the type comparison in LLVM a challenging problem. First, in LLVM, a type is a memory object, and different modules have different memory objects even for the same type. We cannot simply use pointers to types for comparison. Second, a struct type may even have variant string representations. Therefore, a string-based comparison will not work either. Third, a struct may or may not have a name in different modules.

We use a strategy to deal with type comparison. First, if a type is not a struct type, we use its string. If a type is a struct, we use its name. However, we develop a name-recovering technique to identify missed names. For structs, we first record their names when they are available and collect type IDs [4] of their fields. When we encounter a struct without a name, we look up the records based on the field type IDs to find the name. If we still cannot find the name, we will use field type IDs for comparison. After obtaining the string and names, we perform hashing, so that the following frequent comparison can be efficient.

Handling union types. `union` types bring implementation challenges because a `union` type can appear as "different types", depending on the instantiation object. This can cause confusion to TYPM and result in soundness issues. We address this problem by only taking the name of a `union` type for comparison. This is based on the observation that regardless of the actual type of the instantiation object, the name of a `union` type in LLVM remains the same. Such an implementation decision slightly hurts precision but avoids soundness issues.

Type elevation. The idea of type elevation is to use as a complex type as possible when resolving the dependences. Given a memory-access instruction, its access target is specified by the pointer operand. In LLVM IR, we can get the type of the element pointed to by the pointer. We found that the element is typically (more than 90% in the Linux kernel) loaded from a base type, typically a struct. To get the base type, we parse the `GetElementPtr` (GEP) instructions [3] which contain both the base type and field type. If there is a sequence of `GEP` instructions, we will take the outermost (i.e., most complex) type as the base type for dependence resolving. Note that this analysis is also conservative; whenever we are unable to further parse the instructions, e.g., encountering an argument, we stop the analysis and take the current sequence of `GEP` instructions to infer the base type. When parsing the `GEP` instructions, we also record the indexes to achieve the field sensitivity which is required by the externality analysis.

It is worth noting that the externality analysis is the only part that requires field-sensitive analysis. For other parts, TYPM does not require a field-sensitive analysis, which significantly reduces the implementation complexity.

Specification of target types. We provide two ways for specifying the target types. One is to generally specify the class of memory-access instructions, such as indirect calls; TYPM will then automatically identify the pair <type, module> for each memory-access instruction. The other way is to specify the name of the type as a string in an input file, and TYPM will load the types from the file and use hash values to represent the type names. In the dependence-resolving stage, TYPM will use the hash values to match types.

LLVM opaque pointers. LLVM switched to opaque pointers on 04/12/2022 (commit `e758b77161a7`), and the type information of pointers is not immediately available. We identify two practical solutions. First, we can simply disable opaque pointers by using option `-no-opaque-pointers` or use an LLVM version that does not use opaque pointers. Second, we can build TYPM as a standalone binary based on LLVM before version `e758b77161a7`. In this work, we chose the second solution and developed a script to automate the process. In the future, we will also explore the possibility of making TYPM compatible with opaque pointers, as LLVM offers APIs for obtaining type information to migrate to opaque pointers [2].

6. Security Applications

Dependence analysis is an enabling technique for a variety of security applications. In this section, we use TYPM for two security applications: refining indirect-call targets produced by existing type matching and protecting critical data structures from independent writes. Both applications are security-critical, as they are used to prevent privilege-escalation attacks, and to achieve data integrity and control-flow integrity. We will evaluate both applications in §7.

6.1. Refining Indirect-Call Targets

Our first security application is to refine indirect-call targets produced by existing type matching. Precisely resolving indirect-call targets has been a prerequisite for effective control-flow integrity and precise inter-procedural static analysis. Our idea is that existing techniques *globally* match indirect-call targets based on the types of functions or structs in the whole program; if we use TYPM to first identify the dependent modules and apply type matching to only these modules, we would significantly improve the precision of the results. This is essentially achieving *scope-aware* type matching. In this application, the only input we need for TYPM is which type matching we plan to use and improve. We choose two schemes of type matching: function-type matching [43, 56] and struct-type matching [37, 61, 36, 31]. As function-type matching is straightforward, we have our own implementation in TYPM. For struct type matching, we use the implementation of MLTA [35].

Specifically, for an indirect call, TYPM identifies the dependent modules. After that, TYPM applies the existing type matching to only the dependent modules. That is, if a target returned by existing type matching does not belong to any dependent module, we discard it. Once we refine the targets, i.e., finishing one iteration, we go back to identify types and directions of data flows that involve indirect calls. We can perform the iterations until no more indirect-call targets are removed. In TYPM, we also provide the number of iterations as a configuration option in case users would like to get results sooner. When some target functions are not in the analysis scope, e.g., their address propagation involves assembly, we provide an allowlist to conservatively keep them in the resolving results.

6.2. Protecting Critical Data Structures

The motivation of the second application is that critical system attacks often corrupt critical data structures. For example, a pattern of privilege escalation is to overwrite the `UID` to zero. In system software like the Linux kernel, there are many centralized critical data structures that are used for access control.

The method. We apply TYPM to protect critical data structures in the Linux kernel. There are many memory-write instructions; each takes a pointer that specifies the writing target and stores data to it. When the pointer is controlled by an attacker, e.g., through a buffer overflow, the memory-write instruction becomes an arbitrary write and can overwrite critical data structures. TYPM can be naturally applied to mitigate the problem by identifying memory-write instructions where critical data structures can never be passed to them. We call such instructions *non-sensitive writes* which will be directly constrained from writing to any critical data structures. For each *sensitive write*, we further use TYPM to identify the specific dependent critical modules.

The method works as follows. First, we collect all modules that allocate critical data structures, including static and dynamic allocations. We refer to the module set as

critical modules. Second, for each memory-write instruction, we use TYPM to identify all possible target modules, which are referred to as *target modules*. Finally, we intersect *critical modules* with *target modules*. If there is any overlap, we mark the instruction as a sensitive write. All other instructions are marked as non-sensitive writes and may be constrained from writing to any critical data structures. For a sensitive write, we use TYPM to limit it to only the critical modules that create objects of the specific types it targets.

Identifying critical data structures. The very first task is to identify critical data structures. We take the Linux kernel as an example. In this project, we focus on permission- and access-control-related data structures. Previous works [25, 61] show that the Linux kernel mainly uses three kinds of access-control mechanisms, Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). Based on the kernel documentation and previous works [27, 26, 61], we manually collect the structures that are checked by the permission-check APIs (e.g., `capable(CAP_SYS_ADMIN)`) as critical data structures. As permission-check APIs are well-defined, the identification of such structures is straightforward. In total, we collected 37 data structures that require LSM checks, 5 data structures that require capability checks, and 2 data structures that require DAC checks.

7. Evaluation

We evaluate TYPM from the following perspectives:

- **Scalability and overall performance**. TYPM is expected to scale to programs as large as the Linux kernel and browsers.
- **Reducation rate of dependences**. Given a type and a module, we evaluate to what extent TYPM can remove independent modules and constrain memory-write instructions from overwriting critical data structures.
- **Refining rate on indirect-call targets**. Compared to existing type matching, we evaluate to what extent TYPM can further improve the precision.
- **False negatives or soundness**. Soundness is a design goal of TYPM. We evaluate if TYPM would introduce false negatives when refining indirect-call targets.
- **Mitigating existing exploits**. At last, we test if TYPM can defeat existing exploits (with CVE numbers) that escalate privileges by manipulating critical data structures.

Target programs. We have three criteria for selecting target programs: (1) the program should be security-sensitive and demand isolation or protection of critical data structures; (2) the program is written in an unsafe language like C/C++ and has indirect calls; (3) the program should be popular and have many users.

With the criteria, we select system software running on different privilege levels (rings 3, 0, and -1). Specifically, we selected the Linux kernel, the Xen hypervisor, OVMF UEFI firmware, as well as the Firefox browser. The complexity of these programs is summarized in Table 1.

Program	#Lines	#Modules	#Icalls	#Addrs	Time
Linux-default	1,401K	2,372	21K	32K	7m53s
Linux-allyes	10,318K	17,440	127K	210K	461m
Xen	365K	2,112	3.0K	4.2K	28s
OVMF	333K	1,262	8.0K	2.8K	21s
Firefox (C++)	6,475K	15,442	241K	189K	772m

TABLE 1: Target programs and their complexity. "Addrs" refers to the number of address-taken functions.

Experimental setup. We use LLVM 15.0 (`e758b77161a7`, 04/12/2022). We use the default compiler options to generate the bitcode files of the target programs, e.g., `-O2`. We use a desktop running on 64GB RAM and an Intel Xeon CPU 2.9 GHz with 8 cores. The OS is Ubuntu 20.04 LTS.

7.1. Overall Performance

Our evaluation results show that TYPM can easily scale to millions of lines of code. Table 1 also shows the analysis time for each program. In this evaluation, we use the basic function-type matching to get the original indirect-call targets, and the dependence resolving performs one iteration. We will evaluate multi-iteration in §7.2.

TYPM can finish the analysis for regular programs within a minute. However, for large programs with a massive number of indirect calls, the analysis can take much longer, but still within a day. We looked into the time distribution and found that the time for the first two stages is linear, which is less than half an hour for Linux (`allyes` config), but the time for the last stage (dependence resolving) is quadratic, with complexity $O(n^2)$, where n is the number of modules in TYPM's current design. For each indirect call, TYPM recursively traverses all modules and queries the `FlowMap`, which takes the majority of the time.

7.2. Refinement of Indirect-Call Targets

In this section, we evaluate the first security application.

7.2.1. Precision Improvements for Existing Techniques. We apply TYPM to identify indirect-call targets and compare the target reduction with existing type matching. We take existing type matching as a baseline and apply TYPM to improve the results. By its nature, TYPM's approach is orthogonal to existing type matching. We define the matching scope as the number of modules a type will be matched with. Previous type matching takes all modules of a program as the scope. Ideally the improvement (i.e., additional reduction of indirect-call targets) should be proportional to the reduction rate of the matching scope. Therefore, any existing type matching can be used. We first use the function-type matching to evaluate TYPM's effectiveness in reducing the matching scope and indirect-call targets. We run two sets of experiments, one with only one iteration and the other with multi-iterations until targets cannot be further removed. We do not apply multi-iterations to Linux-allyes or Firefox because the reduction rate is already good (87% and 71%),

and the running takes much time. We then switch to struct-type matching. As shown in existing works [36, 37, 61], the struct-type matching has significantly refined the indirect-call targets. However, by reducing the matching scope, we expect TYPM to further refine the targets effectively.

Table 2 shows the results. We first briefly describe the results based on function-type matching. Even with one iteration, TYPM is able to reduce the scope by up to 95% in OVMF and by at least 73% in Firefox. The target-reduction rates range from 35% (OVMF) to 87% (Linux-allyes). We found that the number of iterations is typically 3. With multi-iteration, the target-reduction rates are improved, while the scope-reduction rates are stable. For example, the target-reduction rate for OVMF is doubled, and the reduction rate for Linux-default is improved to 45% which is significant. We then describe the results based on struct-type matching. Overall, the scope reduction is almost the same. The target-reduction rates are a bit different but are still significant, which is expected as TYPM improves precision by reducing the matching scope, which is orthogonal to function-type matching or struct-type matching.

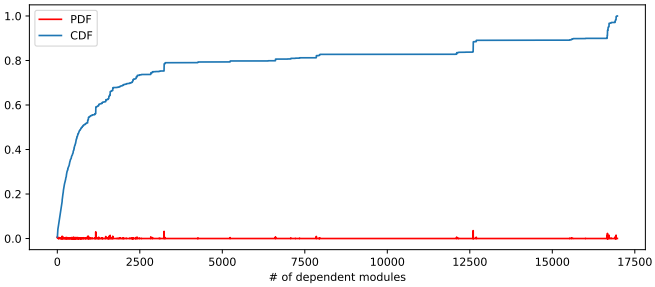

Figure 5: Distribution of scope reduction for func-type matching.

Distribution of scope reduction. We also study the distribution of matching-scope reduction. In this study, we take Linux-allyes (17K modules in total) as the target and function-type matching as the base. For each indirect call, we calculate the number of its dependent modules, and the scope-reduction rate is calculated as the number divided by the total number of modules. We draw the PDF and CDF curves in Figure 5. On average, TYPM achieves a scope-reduction rate of 88%. For most cases (about 80%), the number of dependent modules is less than 3,500 out of 17K, which is significant. There are about 10% of cases where the reduction rate is small (less than 5%).

Effectiveness breakdowns. We propose four techniques to make TYPM practical and effective. The typecasting analysis is a must, but the other three are optimizations, and readers may wonder how effective these optimizations are. The effectiveness of iterative resolving is already shown in Table 2, so here we evaluate the effectiveness of direction inference for data flows and type elevation. We take Linux-default as the target and compare the reduction rates by enabling and disabling the techniques. In this evaluation, we use one iteration.

When both techniques are enabled, as shown in Table 2, the target reduction is 39%, and the scope reduction is 81%. When only direction inference is disabled, the target reduction is 38%, and the scope reduction is 80%. When only type elevation is disabled, the target reduction is down to 9%, and the scope reduction is 81%. When both are disabled, the target reduction is 8%, and the scope reduction is 80%. From the results, we can see that type elevation contributes to the most reduction of targets, but not the reduction of scope. The limited contribution to scope reduction can be a result of that in type elevation, container types in globals and arguments are also included, so more type-labeled data flows are included, and the dependence-resolving results do not change much. The direction inference does not improve reduction significantly due to its conservative policy. As will be discussed in §8, improving the direction inference would be able to further improve the reduction.

7.2.2. False Negatives. Evaluating whether TYPM has false negatives is challenging because we lack ground truth. Therefore, in this work, we choose to use a tracer that collects ground-truth traces (i.e., runtime indirect-call targets) for soundness evaluation. When a runtime indirect-call target is not included in the results of TYPM, but included in the results of existing function-type matching, it is a false negative introduced by TYPM. We implement the tracer as an LLVM pass. In this pass, we add hook functions right before indirect-call instructions. To record indirect-call targets, we take the target address from an indirect-call instruction and send it to the hook function which invokes `sprint_symbol` provided by Linux to print out the target function. We finally build the Linux kernel with our LLVM pass and run LTP (Linux Test Project) to broadly collect traces.

With Linux LTP, we were able to collect 5,799 unique <indirect call, target> pairs using our tracer. The results show that TYPM indeed removes 7 true-positive pairs from the results of existing function-type matching. After looking into the causes, we found two reasons. The first one is about the `container_of` feature [1] in Linux, which results in 6 false negatives. The feature is to get a pointer of the container object based on the pointer of a field object. From the perspective of static analysis, this is an "out-of-bound" access and violates TYPM's assumption. At runtime, in case an object does not come with the corresponding container object, out-of-bound access may occur [23]. More specifically, in the second stage, when parsing nested element types, TYPM assumes that memory access can only be within the boundary of the current project. Note that not all such cases would cause false negatives in TYPM. For a `container_of`, there is a typecast from a field type to a container type. When both types are in the same module, TYPM will recognize it, and there will be no false negative. This only incurs false negatives when the two types are across modules and the cross-module data flows only reflect the field types. While we can still handle such cases by treating `container_of` the whole as a typecasting and expanding types of cross-module data flows, we do not plan to support it for now, as it is Linux-specific and violating the memory-safety policy from the static-analysis perspective.

The second cause is a compiler bug in LLVM 15.

Program	Func-type matching # of targets	TYPM based on function-type matching					TYPM based on struct-type matching				
		1-iteration		N-iteration			1-iteration		N-iteration		
		Target reduction	Scope reduction	N	Target reduction	Scope reduction	Target reduction*	Scope reduction	N	Target reduction*	Scope reduction
Linux-default	534,835	39% (324,534)	82%	3	45% (295,919)	82%	44% (176,852)	82%	4	44% (176,464)	83%
Xen	89,515	36% (57,449)	95%	2	43% (51,005)	95%	31% (42,218)	95%	2	31% (42,154)	95%
OVMF	51,701	35% (33,464)	95%	3	62% (19,476)	95%	42% (13,933)	96%	3	42% (13,837)	96%
Linux-allyes	50,709K	87% (6,545K)	88%	-	-	-	91% (4,524K)	89%	-	-	-
Firefox (C++)	308,426K	71% (87,320K)	73%	-	-	-	72% (84,862K)	75%	-	-	-

TABLE 2: Reduction of indirect-call targets and reduction of scope for type matching. Scope reduction is measured as the module-reduction rate. That is how many independent modules are excluded. Target reduction is measured as the additional removed indirect-call targets over the targets returned by existing techniques (either function-type matching or struct-type matching). *Here the target reduction is an additional reduction over existing struct-type matching. The number of total targets is listed in the parentheses.

A constant-size array is defined in one module with type `[4 x void (%struct.page*)*]`; however, in another module that dereferences its element, the type becomes `[0 x void (%struct.page*)*]`, which is incorrect. We compiled the same code with other LLVM versions and found that the bug does not exist in other versions.

7.3. Protection of Critical Data Structures

Dependence-reduction rate. In this section, we evaluate the dependence-reduction rate for memory-write instructions against critical data structures in the Linux kernel (Linux-default). As discussed in §6.2, we manually collected 44 critical data structures in the Linux kernel that are related to permission, capabilities, LSM, and DAC. To measure the effectiveness of TYPM in this application, we used the following strategy. For each memory-write instruction, we use TYPM to first identify if it may target a critical data structure. This is based on the type analysis in the type-elevation component of TYPM—checking if the pointer of a memory-access instruction refers to a critical-data type. If so, we go ahead to apply TYPM to resolve the dependent modules and report the number of dependent modules. By comparing it with the number of all modules, we can calculate a reduction rate. If a memory-write instruction does not target critical data structures, it may be directly constrained, and we do not further apply TYPM's dependence resolving.

In Linux-default, we identify in total 172,534 memory-write instructions. Out of them, we found that 16,739 may write data to at least one critical data structures based on the type analysis of type elevation; these are sensitive writes. The other memory writes (90.3%) are considered non-sensitive and may be directly constrained from writing to any of the critical data structures. The following results thus focus on only the sensitive writes which will further go through the dependence resolving. For sensitive writes, if we do not limit the write targets, they can target any of the 2,372 modules. Therefore, the total number of target modules would be 39,705K. However, with TYPM, we identify that only 51,244 modules can be their valid targets, on average removing 99.9% of modules. In other words, on average a sensitive memory-write instruction can only target three modules (as opposed to 2,372 modules); such a reduction is dramatic.

Case study with CVEs. We then evaluate whether TYPM is able to mitigate existing privilege-escalation attacks. We select the most recent public privilege-escalation attacks against Linux that corrupt critical data structures. In total, we collected 8 exploits with CVEs (2022-32250, 2022-34918, 2022-27666, 2021-41073, 2021-26708, 2022-29582, 2022-1015, 2022-25636). Three of them corrupt the critical `modprobe_path`, and the other five corrupt UID. We confirmed that TYPM can defeat the first five cases, as the involved writes are constrained from accessing the critical data based on its type-analysis results. TYPM cannot defeat the last three because they employ "confused-deputy" attacks which call the valid function `commit_cred()` to indirectly overwrite UID. To prevent them, TYPM must be enforced together with CFI. Note that this is a general problem shared by data-protection techniques [49, 53].

8. Discussion

Optimization for analysis time. We identify two major time-consuming analyses in TYPM; both involve recursion. The first one is in labeling the types and directions of data flows in globals and arguments. The process needs to recursively parse nested elements and cast-from types. The second is in resolving the dependences. The process needs to recursively traverse the data flows between modules to collect all dependences.

We can have two strategies to reduce the analysis time: caching and multi-threading. If a type in a module has been parsed, or its dependences have been resolved, we can cache it for reuse. The resolving is the most time-consuming process. Its current implementation is single-threaded. As it statically uses the `FlowMap`, multi-threading can be applied to speed up the resolving in the future.

Support for C++ and safe languages. We believe that the indirect-call targets with C++ programs can be significantly refined if we perform the class-hierarchy analysis [45, 24, 20] to directly map virtual functions to the corresponding classes, instead of using general type matching. This is an orthogonal approach to TYPM, and tools already exist, so we can integrate them with TYPM to achieve more precise indirect-call analysis for C++. TYPM can also be applied to type-safe languages to understand the dependences between modules. This can be useful in applications such as logic-error

containing, compartmentalization, or software debloating. When the language is type-safe, the typecasting and callback analyses would be simpler and have more-precise results. For example, when wild pointers (e.g., `void *`) do not exist, the conservative policy of assuming that wild pointers can target any type can be removed, which will significantly improve the precision of the analysis.

Potential improvements for precision. TYPM can be further improved for precision. We identify two potential efforts. First, direction inference: in the current implementation, TYPM conservatively performs a minimal intra-procedural analysis for only `load` and `store`. Any uncertain case would be treated bi-directional. As a result, in most cases, data flows are still considered bi-directional. A more thorough but sound data-flow analysis would effectively improve the precision in dependence resolving. Second, handling of general types: in TYPM, we assume that a general type (e.g., `void *`) can pass any type of data and can represent any other type that has transitive cast-to or cast-from relations with it. We believe that using an analysis against general types to more precisely infer which other types they can represent would effectively improve the precision in dependence resolving.

TyPM for runtime defenses. TYPM currently cannot be directly used for defenses due to false negatives as shown in §7.2.2. However, the false-negative results are encouraging and suggest that TYPM can also be applied for defense with additional analysis. First, we need to also perform typecasting against function arguments to make sure that the basic function-type matching does not have false negatives [43]. Second, by performing a cross-module downcasting analysis, we can detect dangerous casts that result in out-of-bound accesses and report them for manual validation. If a cast is valid, we can add it to an allowlist, so that TYPM can capture them. Besides runtime defense mechanisms, TYPM can be directly used to facilitate bug detection, reachability analysis, and any other static analyses that can tolerate a few false negatives.

Enforcement of write integrity. In this work, we focused on identifying irrelevant write instructions which should not target the critical data structures, but did not actually enforce the write integrity. To enforce the write integrity, one idea is to allocate critical structures in a dedicated memory region and use software-based fault isolation (SFI) [55] to constrain accesses from them. This requires the development of new memory allocators and an SFI mechanism. Given the complexity of the Linux kernel, the engineering efforts alone would deserve separate work. On the other hand, as will be mentioned in §9, researchers have proposed various techniques for the enforcement of write integrity [7, 30, 53]. It is worth noting that CFI should be enforced together with write integrity to avoid the "confused-deputy" problem—attackers may indirectly invoke valid write instructions to corrupt data structures via code-reuse attacks [48].

Function as dependence-resolving unit. TYPM's current implementation takes module as the unit. In principle, we can also take function as the unit. However, two potential issues should be addressed. First, the time of resolving is quadratic to the number of units. Having function as the unit would cost much more time. As an example, Linux-default has 2,372 modules, but 44,392 functions. Second, given a program, it is much easier to take out a module or isolate a module, as modules are relatively self-contained. However, functions may have complicated dependences with others, so removing or isolating them would be harder in practice. That said, function as a unit is still a valid choice and can be explored in the future.

9. Related work

In this section, we discuss the most related work from four perspectives: dependence analysis, type analysis, program modularization, and write integrity.

Dependence analysis. PtrSplit [33] is a recent and closely related work. It constructs program-dependence graph (PDG) for each function and then builds a global graph based on the PDGs, which is then used to resolve dependences for manually annotated sensitive data. TYPM distinguishes itself from PtrSplit in some important perspectives. First, PtrSplit does not handle globals or shared memory; as a result, it does not support multi-threaded programs such as the programs tested in the paper. Second, PtrSplit does not handle typecasting. As a result, the actual types of nested general pointers in function parameters cannot be resolved. Such cases are prevalent in system software. Third, the overhead of dependence resolving is quadratic to the number of units (e.g., module or function). Taking function as the unit would suffer from scalability issues. In comparison, TYPM does not construct a PDG, which itself is challenging to ensure precision and soundness. TYPM handles globals and shared memory, and can naturally support both single-threaded and multi-threaded programs. TYPM carefully handles typecasting and nested types in both parameters and globals. Last, our design choice of taking module as the unit allows TYPM to scale.

Other than that, ProgramCutter [57] is a graph-based approach to separating the privileged code of a program based on dynamic data dependency analysis. Bavota et al. [9] provided a software modularization technique based on the program structural dependencies analysis, which can help with program maintenance by modularizing programs. Decades ago, researchers studied program-modularization issues. Cardelli [11] and Glew et al. [19] tried to separate programs into self-contained and compilable modules, which is to ensure type-safe linking. General analysis tools such as SVF [54] and Andersen [21] can also be used to analyze dependences; however, they cannot scale to large programs and still suffer from accuracy issues.

Angr [52] and BPA [29] employ value-set analysis and point-to analysis based on binaries. In resolving indirect-call targets, besides scalability, their precision is not as good as source-level type-based analysis. For instance, BPA achieves a reduction rate of 37% over the original address-taken functions; in comparison, taking Linux-default as an example, function-type matching alone can reduce the average number of targets from 32,484 to 26 (with a reduction rate of more

than 99.9%), and TYPM is able to further reduce it to 8.5; therefore, although it is an unfair comparison (binary vs. source), in terms of the target-reduction rate, TYPM outperforms BPA by 3 orders of magnitude.

Type analysis. C/C++ is not type-safe. However, types provide rich semantics, sometimes even invariants, that are precious for security-property reasoning. In the security domain, the research on type-based analysis has thrived since its uses in CFI. Notably, modular control-flow integrity [43] and Google CFI [56] use function-type matching to resolve indirect-call targets, which is sound in principle but imprecise. τCFI also tries to use types to match targets of indirect calls. However, due to the missed type information in binaries, it only uses the size of each argument for the matching. Recent advances additionally use struct-type matching [36, 37, 61, 31, 18] to improve the precision. All such matching is global and will include targets from unrelated modules, which motivated us to perform scope-aware type matching in TYPM. VIP [16] uses types to improve the precision for pointer analysis related to virtual calls in C++. However, it is unsound and imprecise.

Zdancewic et al. proposed to partition programs based on security types [58, 62]. While they share a similar research goal—program partitioning, their approach is clearly different or even opposite. TYPM uses type-based analysis for data-flow dependences, while the work of Zdancewic et al. uses data-flow analysis for type propagation. More specifically, their work requires manual annotation of security types (classification labels) and pre-configured subprograms; it then employs data-flow analysis to associate fields and statements to subprograms. Such an approach is essentially data-flow analysis, instead of type-based analysis. As a result, the limitations with static data-flow analysis would still apply.

Program modularization and enforcement. Given its importance, program modularization has been actively studied. μSCOPE [49], which internally uses Memorizer [51], studies the separability of the monolithic Linux kernel. μSCOPE and TYPM use complementary approaches. μSCOPE uses dynamic tracing to understand the separability, while TYPM employs static type analysis to find the "CAN'T" set (independent modules). μSCOPE is precise but has false negatives, while TYPM is comprehensive but less precise. Note that comprehensiveness is important for runtime mechanisms to not crash the program.

Most recently, HAKC [38] supports kernel partitioning through a data-ownership mechanism. However, HAKC requires developers to annotate the compartments and specify policies. Similarly, other works [39, 13, 50] on compartmentalization typically assume that the compartments are provided. Isolation and privilege separation [14, 40, 42, 41, 22] in critical software such as OSes has also been extensively studied. They also assume that partitions are given. Therefore, TYPM can help make the works more usable by automatically identifying compartments or partitions.

Write integrity. To protect critical data structures, TYPM tries to constrain irrelevant write instructions from corrupting such data structures, which is essentially to provide write integrity. There are related works that also try to ensure write integrity for specific data objects. WIT [7] uses points-to analysis to compute the control-flow graph and the set of objects that can be targeted by each write instruction. Such an approach would not work for multi-entry programs as internal control dependence is not required to form data dependence, as shown in §2.1. Also, precise points-to analysis [54] is unscalable, and itself requires a global callgraph (with indirect-call targets resolved). DFI such as Kenali [53] also uses point-to analysis or data-flow analysis to identify valid targets of memory writes. Code-pointer integrity (CPI) [30] identifies sensitive pointers and stores them in safe regions. Sensitive pointers are labeled with types, but the propagation analysis still uses data-flow analysis. In comparison, TYPM supports multi-entry programs, avoids point-to analysis with the type-based analyses, and thus is practical. It is worth noting that TYPM can benefit existing mechanisms. For example, by further matching types of aliases, TYPM can improve the precision of alias (or point-to) analysis.

10. Conclusion

Dependence analysis is a foundational technique that enables security applications such as control-flow integrity, data integrity, program compartmentalization, and debloating. Dependence analysis has been known to be hard, and even infeasible for large C/C++ system programs. This paper presents a breakthrough in dependence analysis—type-based dependence analysis for program modularization (TYPM). Given a type and a module, TYPM conservatively determines all dependent modules in the program that may directly or indirectly pass data of this type to the module. Other modules are independent and can be excluded. We propose multiple techniques to make TYPM scalable, practical, and precise, including typecasting analysis, identification of type- and direction-labeled data flows, iterative dependence resolving, and type elevation. As a demonstration, we showed how to use TYPM to significantly refine indirect-call targets over the state of the art and to protect critical data structures from being overwritten. Extensive evaluation results on an OS kernel, a hypervisor, a firmware, and a browser confirm that TYPM is precise and practical, and does not introduce false negatives under its model. We open-source TYPM and hope it would enable more security applications.

11. Acknowledgment

The author thanks the anonymous reviewers and the shepherd for their valuable suggestions and comments. The author also thanks Qiushi Wu and Dinghao Liu for helping with the evaluation. This research was supported in part by the NSF awards CNS-1815621, CNS-1931208, CNS-2045478, CNS-2106771, and CNS-2154989. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of NSF.

References

[1] The linux container_of macro, 2022. https://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/067/6717/6717s2.html.

[2] Llvm opaque pointers, 2022. https://llvm.org/docs/OpaquePointers.html.

[3] llvm::gepoperator class reference, 2022. https://llvm.org/doxygen/classllvm_1_1GEPOperator.html.

[4] llvm::type class reference, 2022. https://llvm.org/doxygen/classllvm_1_1Type.html.

[5] Single compilation unit, 2022. https://en.wikipedia.org/wiki/Single_Compilation_Unit.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[7] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[8] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, Santa Clara, CA, August 2019.

[9] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):1–33, 2014.

[10] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[11] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, 1997.

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

[13] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303, 2017.

[14] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[16] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 329–340, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the Effectiveness of Type-Based Control Flow Integrity. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[18] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent 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.

[19] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 250–261, 1999.

[20] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[21] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[22] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application memory isolation on Ultra-Low-Power MCUs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 127–132, Boston, MA, July 2018. USENIX Association.

[23] Takashi Iwai. Fix oob access of mixer element list, 2020.

[24] Dongseok Jang, Zachary Tatlock, and Sorin 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, February 2014.

[25] Erik Karlsson. Evaluation of linux security frameworks, 2010.

[26] The kernel development community. Linux security module usage, 2022.

[27] The kernel development community. Security documentation, 2022.

[28] Douglas Kilpatrick. Privman: A library for partitioning applications. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association.

[29] Sun Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. 01 2021.

[30] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

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

[32] Donglin Liang and M.J. Harrold. Slicing objects using system dependence graphs. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 358–367, 1998.

[33] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[34] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[35] Kangjie Lu. Typedive: Multi-layer type analysis (mlta) for refining indirect-call targets, 2019.

[36] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[37] Kangjie Lu, Aditya Pakki, and Qiushi 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, August 2019.

[38] Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKCs. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, April 2022.

[39] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. D-Box: DMA-enabled Compartmentalization for Embedded Applications. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, April 2022.

[40] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association.

[41] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[42] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery.

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

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

[45] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

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

[47] Anh Quach, Aravind Prakash, and Lok Kwong Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[48] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), mar 2012.

[49] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. muscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 296–311, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 296–311, 2021.

[51] Nick Roessler, Yi Chien, Lucas Atayde, Peiru Yang, Imani Palmer, Lily Gray, and Nathan Dautenhahn. Lossless instruction-to-object memory tracing in the linux kernel. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR '21, New York, NY, USA, 2021. Association for Computing Machinery.

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

[53] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

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

[55] Gang Tan. *Principles and Implementation Techniques of Software-Based Fault Isolation*. Now Publishers Inc., Hanover, MA, USA, 2017.

[56] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, pages 941–955, 2014.

[57] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 323–333. IEEE, 2013.

[58] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. *SIGOPS Oper. Syst. Rev.*, 35(5):1–14, oct 2001.

[59] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei 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.

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

[61] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. {PeX}: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[62] Lantian Zheng, S. Chong, A.C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *2003 Symposium on Security and Privacy, 2003.*, pages 236–250, 2003.